

NAME

opendir – open a directory / readdir – read a directory

SYNOPSIS

```
#include <sys/types.h>
```

```
#include <dirent.h>
```

```
DIR *opendir(const char *name);
```

```
struct dirent *readdir(DIR *dir);
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
```

DESCRIPTION opendir

The **opendir()** function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

RETURN VALUE

The **opendir()** function returns a pointer to the directory stream or NULL if an error occurred.

DESCRIPTION readdir

The **readdir()** function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by *dir*. It returns NULL on reaching the end-of-file or if an error occurred.

DESCRIPTION readdir_r

The **readdir_r()** function initializes the structure referenced by *entry* and stores a pointer to this structure in *result*. On successful return, the pointer returned at *result* will have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer will have the value NULL.

The data returned by **readdir()** is overwritten by subsequent calls to **readdir()** for the **same** directory stream.

The *dirent* structure is defined as follows:

```
struct dirent {
    long    d_ino;           /* inode number */
    off_t   d_off;          /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type;    /* type of file */
    char     d_name[256];    /* filename */
};
```

RETURN VALUE

The **readdir()** function returns a pointer to a dirent structure, or NULL if an error occurs or end-of-file is reached.

readdir_r() returns 0 if successful or an error number to indicate failure.

ERRORS**EACCES**

Permission denied.

ENOENT

Directory does not exist, or *name* is an empty string.

ENOTDIR

name is not a directory.

NAME

exec, execl, execl, execl, execl, execl, execlp, execlp – execute a file

SYNOPSIS

```
#include <unistd.h>
```

```
int execl(const char *path, const char *arg0, ..., const char *argn, char **/NULL*);
```

```
int execl(const char *path, char *const argv[]);
```

```
int execl(const char *path, char *const argv[], ..., const char *argn,
          char **/NULL*/, char *const envp[]);
```

```
int execlp(const char *path, char *const argv[] char *const envp[]);
```

```
int execlp(const char *file, const char *arg0, ..., const char *argn, char **/NULL*);
```

```
int execlp(const char *file, char *const argv[]);
```

DESCRIPTION

Each of the functions in the **exec** family overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

When a C program is executed, it is called as follows:

```
int main (int argc, char *argv[], char *envp[]);
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is at least one, and the first member of the array points to a string containing the name of the file.

The arguments *arg0*, ..., *argn* point to null-terminated character strings. These strings constitute the argument list available to the new process image. Conventionally at least *arg0* should be present. The *arg0* argument points to a string that is the same as *path* (or the last component of *path*). The list of argument strings is terminated by a (**char** *)0 argument.

The *argv* argument is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process image. By convention, *argv* must have at least one member, and it should point to a string that is the same as *path* (or its last component). The *argv* argument is terminated by a null pointer.

The *path* argument points to a path name that identifies the new process file.

The *file* argument points to the new process file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the **PATH** environment variable (see **environ(5)**).

File descriptors open in the calling process remain open in the new process.

Signals that are being caught by the calling process are set to the default disposition in the new process image (see **signal(3C)**). Otherwise, the new process image inherits the signal dispositions of the calling process.

RETURN VALUES

If a function in the **exec** family returns to the calling process, an error has occurred; the return value is **-1** and **errno** is set to indicate the error.

NAME

calloc, malloc, free, realloc – Allocate and free dynamic memory

SYNOPSIS

```
#include <stdlib.h>
```

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

DESCRIPTION

calloc() allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates *size* bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc()**, **calloc()** or **realloc()**. Otherwise, or if **free(ptr)** has already been called before, undefined behaviour occurs. If *ptr* is **NULL**, no operation is performed.

realloc() changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is **NULL**, the call is equivalent to **malloc(size)**; if *size* is equal to zero, the call is equivalent to **free(ptr)**. Unless *ptr* is **NULL**, it must have been returned by an earlier call to **malloc()**, **calloc()** or **realloc()**.

RETURN VALUE

For **calloc()** and **malloc()**, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or **NULL** if the request fails.

free() returns no value.

realloc() returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from *ptr*, or **NULL** if the request fails. If *size* was equal to 0, either **NULL** or a pointer suitable to be passed to **free()** is returned. If **realloc()** fails the original block is left untouched - it is not freed or moved.

CONFORMING TO

ANSI-C

SEE ALSO

brk(2), **posix_memalign(3)**

NAME

sigaction – POSIX signal handling functions.

SYNOPSIS

```
#include <signal.h>
```

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

DESCRIPTION

The **sigaction** system call is used to change the action taken by a process on receipt of a specific signal.

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non-**NULL**, the new action for signal *signum* is installed from *act*. If *oldact* is non-**NULL**, the previous action is saved in *oldact*.

The **sigaction** structure is defined as something like

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

On some architectures a union is involved - do not assign to both *sa_handler* and *sa_sigaction*.

The *sa_restorer* element is obsolete and should not be used. POSIX does not specify a *sa_restorer* element.

sa_handler specifies the action to be associated with *signum* and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function.

sa_mask gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA_NODEFER** or **SA_NOMASK** flags are used.

sa_flags specifies a set of flags which modify the behaviour of the signal handling process. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP

If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when child processes receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN** or **SIGTTOU**).

SA_RESTART

Provide behaviour compatible with BSD signal semantics by making certain system calls restartable across signals.

RETURN VALUES

sigaction returns 0 on success and -1 on error.

ERRORS**EINVAL**

An invalid signal was specified. This will also be generated if an attempt is made to change the action for **SIGKILL** or **SIGSTOP**, which cannot be caught.

SEE ALSO

kill(1), **kill(2)**, **killpg(2)**, **pause(2)**, **sigsetops(3)**,

NAME

sigprocmask – change and/or examine caller’s signal mask
 sigsuspend – install a signal mask and suspend caller until signal

SYNOPSIS

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int sigsuspend(const sigset_t *set);
```

DESCRIPTION sigprocmask

The **sigprocmask()** function is used to examine and/or change the caller’s signal mask. If the value is **SIG_BLOCK**, the set pointed to by the argument *set* is added to the current signal mask. If the value is **SIG_UNBLOCK**, the set pointed by the argument *set* is removed from the current signal mask. If the value is **SIG_SETMASK**, the current signal mask is replaced by the set pointed to by the argument *set*. If the argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is NULL, the value *how* is not significant and the caller’s signal mask is unchanged; thus, the call can be used to inquire about currently blocked signals.

If there are any pending unblocked signals after the call to **sigprocmask()**, at least one of those signals will be delivered before the call to **sigprocmask()** returns.

It is not possible to block those signals that cannot be ignored this restriction is silently imposed by the system. See **sigaction(2)**.

If **sigprocmask()** fails, the caller’s signal mask is not changed.

RETURN VALUES

On success, **sigprocmask()** returns **0**. On failure, it returns **-1** and sets **errno** to indicate the error.

ERRORS

sigprocmask() fails if any of the following is true:

- EFAULT** *set* or *oset* points to an illegal address.
- EINVAL** The value of the *how* argument is not equal to one of the defined values.

DESCRIPTION sigsuspend

sigsuspend() replaces the caller’s signal mask with the set of signals pointed to by the argument *set* and then suspends the caller until delivery of a signal whose action is either to execute a signal catching function or to terminate the process.

If the action is to terminate the process, **sigsuspend()** does not return. If the action is to execute a signal catching function, **sigsuspend()** returns after the signal catching function returns. On return, the signal mask is restored to the set that existed before the call to **sigsuspend()**.

It is not possible to block those signals that cannot be ignored (see **signal(5)**); this restriction is silently imposed by the system.

RETURN VALUES

Since **sigsuspend()** suspends process execution indefinitely, there is no successful completion return value. On failure, it returns **-1** and sets **errno** to indicate the error.

ERRORS

sigsuspend() fails if either of the following is true:

- EFAULT** *set* points to an illegal address.
- EINTR** A signal is caught by the calling process and control is returned from the signal catching function.

SEE ALSO

sigaction(2), **sigsetops(3C)**,

NAME

sigsetops, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate sets of signals

SYNOPSIS

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(sigset_t *set, int signo);
```

DESCRIPTION

These functions manipulate *sigset_t* data types, representing the set of signals supported by the implementation.

sigemptyset() initializes the set pointed to by *set* to exclude all signals defined by the system.

sigfillset() initializes the set pointed to by *set* to include all signals defined by the system.

sigaddset() adds the individual signal specified by the value of *signo* to the set pointed to by *set*.

sigdelset() deletes the individual signal specified by the value of *signo* from the set pointed to by *set*.

sigismember() checks whether the signal specified by the value of *signo* is a member of the set pointed to by *set*.

Any object of type *sigset_t* must be initialized by applying either **sigemptyset()** or **sigfillset()** before applying any other operation.

RETURN VALUES

Upon successful completion, the **sigismember()** function returns a value of one if the specified signal is a member of the specified set, or a value of 0 if it is not. Upon successful completion, the other functions return a value of 0. Otherwise a value of **-1** is returned and **errno** is set to indicate the error.

ERRORS

sigaddset(), **sigdelset()**, and **sigismember()** will fail if the following is true:

- EINVAL** The value of the *signo* argument is not a valid signal number.

sigfillset() will fail if the following is true:

- EFAULT** The *set* argument specifies an invalid address.

SEE ALSO

sigaction(2), **sigpending(2)**, **sigprocmask(2)**, **sigsuspend(2)**, **attributes(5)**, **signal(5)**

NAME

stat, lstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
int stat(const char *file_name, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

DESCRIPTION

These functions return information about the specified file. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file.

stat stats the file pointed to by *file_name* and fills in *buf*.

lstat is identical to **stat**, except in the case of a symbolic link, where the link itself is stat-ed, not the file that it refers to.

They all return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t    st_dev;    /* device */
    ino_t    st_ino;    /* inode */
    mode_t    st_mode;    /* protection */
    nlink_t    st_nlink; /* number of hard links */
    uid_t    st_uid;    /* user ID of owner */
    gid_t    st_gid;    /* group ID of owner */
    dev_t    st_rdev;    /* device type (if inode device) */
    off_t    st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks; /* number of blocks allocated */
    time_t    st_atime; /* time of last access */
    time_t    st_mtime; /* time of last modification */
    time_t    st_ctime; /* time of last status change */
};
```

The value *st_size* gives the size of the file (if it is a regular file or a symlink) in bytes. The size of a symlink is the length of the pathname it contains, without trailing NUL.

The field *st_atime* is changed by file accesses, e.g. by **execve(2)**, **mknod(2)**, **pipe(2)**, **utime(2)** and **read(2)** (of more than zero bytes). Other routines, like **mmap(2)**, may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, e.g. by **mknod(2)**, **truncate(2)**, **utime(2)** and **write(2)** (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

SEE ALSO

chmod(2), **chown(2)**, **readlink(2)**, **utime(2)**, **capabilities(7)**

NAME

waitpid – wait for child process to change state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

waitpid() suspends the calling process until one of its children changes state; if a child process changed state prior to the call to **waitpid()**, return is immediate. *pid* specifies a set of child processes for which status is requested.

If *pid* is equal to **(pid_t)-1**, status is requested for any child process.

If *pid* is greater than **(pid_t)0**, it specifies the process ID of the child process for which status is requested.

If *pid* is equal to **(pid_t)0** status is requested for any child process whose process group ID is equal to that of the calling process.

If *pid* is less than **(pid_t)-1**, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

If **waitpid()** returns because the status of a child process is available, then that status may be evaluated with the macros defined by **wstat(5)**. If the calling process had specified a non-zero value of *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*.

The *options* argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header **<sys/wait.h>**:

WCONTINUED The status of any continued child process specified by *pid*, whose status has not been reported since it continued, is also reported to the calling process.

WNOHANG **waitpid()** will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.

WNOWAIT Keep the process whose status is returned in *stat_loc* in a waitable state. The process may be waited for again with identical results.

RETURN VALUES

If **waitpid()** returns because the status of a child process is available, this function returns a value equal to the process ID of the child process for which status is reported. If **waitpid()** returns due to the delivery of a signal to the calling process, -1 is returned and **errno** is set to **EINTR**. If this function was invoked with **WNOHANG** set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, **0** is returned. Otherwise, -1 is returned, and **errno** is set to indicate the error.

ERRORS

waitpid() will fail if one or more of the following is true:

ECHILD The process or process group specified by *pid* does not exist or is not a child of the calling process or can never be in the states specified by *options*.

EINTR **waitpid()** was interrupted due to the receipt of a signal sent by the calling process.

EINVAL An invalid value was specified for *options*.

SEE ALSO

exec(2), **exit(2)**, **fork(2)**, **sigaction(2)**, **wstat(5)**