ChorusOS man pages section 2POSIX: POSIX System Calls

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3328
December 10, 1999

Contents

PREFACE

Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

NAME — This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS — This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ] The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.

. . . Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ‘ "`filename` . . ." .

| Separator. Only one of the arguments separated by this character can be specified at time.

{ } Braces. The options and / or arguments enclosed within braces are

	interdependent, such that everything enclosed must be treated as a unit.
FEATURES	This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.
OPTIONS	This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output - standard output, standard error, or output files - generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE	This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions.
FILES	This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
SEE ALSO	This section lists references to other man pages, in-house documentation and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

POSIX System Calls

14

NAME

Intro – introduction to POSIX compliant system calls

DESCRIPTION

This section describes the POSIX 1003.1 and 1003.1b compliant system calls within the ChorusOS API.

Each API function is associated with one or more system features. A given interface is available if, and only if, one of its associated features was configured in the target system when that system was built. See `intro`(2K) for a complete list of the features that make up the ChorusOS API.

System calls in this section follow POSIX 1003.1 / 1003.1b conventions for error conditions. If an error occurs, all calls return –1 and set the *errno* variable to indicate the specific error. POSIX error codes are listed in the file `<errno.h>`. For more detailed information on error returns, see `intro`(2K).

DEFINITIONS

Credentials

Each user allowed on the system is identified by a positive integer called a user ID (*uid*).

Each user is also member of one or more groups. One of these groups is distinguished from the others, and the positive integer corresponding to it is called the user's group ID. Positive integers corresponding to other groups are called supplementary group IDs.

An actor running under Extended Actor Management (see ACTOR_EXTENDED_MNGT(5FEA)) has a user ID and a set of group IDs. These IDs are the IDs of the user responsible for the creation of the c_actor and are collected into a data structure called the *credential structure* (see *acred*(2K)).

Super-user

If its user ID is 0, an actor is recognized as a *super-user* actor and is granted special privileges with regard to file operations.

File Descriptor

A file descriptor is a small integer used to do I / O on a file. The value of a file descriptor is from 0 to `OPEN_MAX`-1. This is a tunable configuration variable. An actor may have no more than `OPEN_MAX` file descriptors (0 - `OPEN_MAX`-1) open simultaneously. A file descriptor is returned by system calls such as *open*(2POSIX), *socket*(2POSIX), or shm_open(2POSIX). The file descriptor is used as an argument by calls such as `read`(2POSIX), `write`(2POSIX), *ioctl*(2POSIX), and *close*(2POSIX).

File Name

Names consisting of 1 to `NAME_MAX` characters may be used to identify an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding \:0 (null) and / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of file names because of the specific meanings attached to these characters by UNIX shells. Although permitted, it is advisable to avoid the use of unprintable characters in file names.

Path Name

A path name is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the *current working directory*.

A slash by itself names the root directory.

Unless specifically stated otherwise, a null path name is treated as if it named a non-existent file.

Posix Object Name

A posix object name is a null-terminated character string. Names consisting of 1 to `_SC_MQ_PATHMAX` or `_SC_SHM_PATHMAX` characters may be used to identify a message queue or a shared memory object.

Posix object names define a global name space.

Directory

Directories organize files into a hierarchical system where the directories are the nodes in the hierarchy. A directory is a file that catalogues the list of files, including sub-directories, that are directly below it in the hierarchy. Entries in a directory file are called links. A link associates a file indentifier with a file name. By convention, a directory contains at least two links, `.` and `..`, referred to as *dot* and *dot-dot*, respectively. The link called dot refers to the directory itself, while dot-dot refers to its parent directory. The root directory, which is the top-most node of the hierarchy, has itself as its parent directory. The path name of the root directory is `/` and the parent directory of the root directory is `/`.

Per Site Default Root Directory

When booted, each site is assigned a default root directory. Depending on the system configuration, this directory may be a directory of a local file system or a directory accessed through NFS. By default, each actor created on a site will have its own root directory set to the default root directory of the site.

Root Directory and Current Working Directory

Each actor has the concept of a root directory and a current working directory associated with it, for the purpose of resolving path name searches. The root directory of an actor does not have to be the default site root directory.

File Access Permissions

Read, write, and execute/search permissions on a file are granted to an actor if one or more of the following are true:

- The actor is a super-user actor.
- The user ID of the actor matches the user ID of the owner of the file, and the appropriate access bit of the “owner” portion (0700) of the file mode is set.
- The user ID of the actor does not match the user ID of the owner of the file, and either the group ID of the actor matches the group ID of the file, or the group ID of the file is in the supplementary group IDs, and the appropriate access bit of the “group” portion (070) of the file mode is set.

- Neither the user ID nor group ID and supplementary group IDs of the actor match the corresponding user ID and group ID of the file, but the appropriate access bit of the “other” portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`intro`(2K)

Name	Description
`accept`(2POSIX)	accept a connection on a socket
`access`(2POSIX)	check access permissions of a file or pathname
`bind`(2POSIX)	bind a name to a socket
`chdir`(2POSIX)	change current directory
`chmod`(2POSIX)	change mode of file
`chown`(2POSIX)	change owner and group of a file
`chroot`(2POSIX)	change the root directory
`close`(2POSIX)	close a file descriptor
`connect`(2POSIX)	initiate a connection on a socket
`dup`(2POSIX)	duplicate an open file descriptor
`dup2`(2POSIX)	See `dup`(2POSIX)
`fchdir`(2POSIX)	See `chdir`(2POSIX)
`fchmod`(2POSIX)	See `chmod`(2POSIX)
`fchown`(2POSIX)	See `chown`(2POSIX)
`fcntl`(2POSIX)	file control
`flock`(2POSIX)	apply or remove an advisory lock on an open file
`fpathconf`(2POSIX)	get configurable pathname variables
`fstat`(2POSIX)	See `stat`(2POSIX)
`fstatfs`(2POSIX)	See `statfs`(2POSIX)

fsync(2POSIX)	synchronize a file's in-memory state with that on the physical medium
ftruncate(2POSIX)	See truncate(2POSIX)
ftruncate(2POSIX)	See truncate(2POSIX)
getdirentries(2POSIX)	get directory entries in a filesystem\ |[mdash]\ |independent format
getdomainname(2POSIX)	get or set the domainname of the current host
getfh(2POSIX)	get file handle
getfsstat(2POSIX)	get list of all mounted filesystems
gethostname(2POSIX)	See hostname(2POSIX)
getpeername(2POSIX)	get name of connected peer
getrlimit(2POSIX)	control maximum system resource consumption
getsockname(2POSIX)	get socket name
getsockopt(2POSIX)	get and set options on sockets
gettimeofday(2POSIX)	get/set date and time
hostname(2POSIX)	get or set the name of the machine
ioctl(2POSIX)	control device
link(2POSIX)	make a hard file link
listen(2POSIX)	listen for connections on a socket
lseek(2POSIX)	move a read/write file pointer
lstat(2POSIX)	See stat(2POSIX)
mkdir(2POSIX)	make a directory file
mkfifo(2POSIX)	make a fifo file
mknod(2POSIX)	make a special file node
mmap(2POSIX)	map c_actor addresses to a shared memory object
mount(2POSIX)	mount or unmount a filesystem
mq_close(2POSIX)	close a message queue
mq_getattr(2POSIX)	retrieve message queue attributes
mq_open(2POSIX)	open a message queue

mq_receive(2POSIX)	receive a message from a message queue
mq_send(2POSIX)	send a message to a message queue
mq_setattr(2POSIX)	set message queue attributes
mq_unlink(2POSIX)	unlink a message queue
munmap(2POSIX)	unmap a previously mapped address
nfssvc(2POSIX)	NFS services
open(2POSIX)	open for reading or writing
pipe(2POSIX)	create descriptor a pair for interprocess communication
read(2POSIX)	read input
readlink(2POSIX)	read value of a symbolic link
readv(2POSIX)	See read(2POSIX)
recv(2POSIX)	receive a message from a socket
recvfrom(2POSIX)	See recv(2POSIX)
recvmsg(2POSIX)	See recv(2POSIX)
rename(2POSIX)	change the name of a file
rmdir(2POSIX)	remove a directory file
select(2POSIX)	synchronous I/O multiplexing
send(2POSIX)	send a message from a socket
sendmsg(2POSIX)	See send(2POSIX)
sendto(2POSIX)	See send(2POSIX)
sethostname(2POSIX)	See hostname(2POSIX)
sethostname(2POSIX)	See hostname(2POSIX)
setrlimit(2POSIX)	See getrlimit(2POSIX)
setsockopt(2POSIX)	See getsockopt(2POSIX)
settimeofday(2POSIX)	See gettimeofday(2POSIX)
shm_open(2POSIX)	open a shared memory object
shm_unlink(2POSIX)	unlink a shared memory object
shutdown(2POSIX)	shut down part of a full-duplex connection

`socket`(2POSIX)	create an endpoint for communication
`socketpair`(2POSIX)	create a pair of connected sockets
`stat`(2POSIX)	get file status
`statfs`(2POSIX)	get file system statistics
`swapon`(2POSIX)	add a swap device for swapping
`symlink`(2POSIX)	make a symbolic link to a file
`sync`(2POSIX)	synchronize the disk block in-core status with that on disk
`truncate`(2POSIX)	truncate a file to a specified length
`umask`(2POSIX)	set file creation mode mask
`unlink`(2POSIX)	remove a directory entry
`unmount`(2POSIX)	See `mount`(2POSIX)
`utimes`(2POSIX)	set file access and modification times
`write`(2POSIX)	write output
`writev`(2POSIX)	See `write`(2POSIX)

NAME

accept – accept a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, int *addrlen);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *s* argument is a socket that was created using *socket*(2POSIX), bound to an address using *bind*(2POSIX) and which listens for connections after a `listen`(2POSIX). The `accept` argument extracts the first connection on the queue of pending connections, creates a new socket with the same properties as *s*, and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `accept` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `accept` returns an error as described below. The accepted socket may not be used to accept more connections. The original socket *s* remains open.

The *addr* argument is a result parameter that is filled with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication takes place. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*. On return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to `select`(2POSIX) a socket in order to do an `accept` by selecting it for read.

RETURN VALUES:

Upon successful completion, `accept` returns a non-negative integer that is a descriptor for the accepted socket; otherwise it returns –1 and sets *errno* to indicate one of the following error conditions:

[EBADF]	The argument *s* is invalid.
[ENOTSOCK]	The argument *s* references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of the type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the c_actor's address space.
[EMFILE]	The c_actor descriptor table is full.
[ENFILE]	The system file table is full.

[ENOBUFS]	Insufficient buffer space is available. The operation cannot be completed until sufficient resources are freed.
[EWOULDBLOCK]	The socket is marked non-blocking, and no connections are present to be accepted.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`bind`(2POSIX), `connect`(2POSIX), `listen`(2POSIX), `select`(2POSIX), `socket`(2POSIX)

NAME

access – check access permissions of a file or pathname

SYNOPSIS

#include <unistd.h>
int **access**(const char **path*, int *mode*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *access* function checks the accessibility of the file named by *path* for the access permissions indicated by *mode*. The value of *mode* is the bitwise inclusive OR of the access permissions to be checked

`R_OK` for read permission,

`W_OK` for write permission, and

`X_OK` for execute/search permission, or an existence test.

`F_OK` All components of the pathname *path* are checked for access permissions (including *F_OK*).

The real user ID is used instead of the effective user ID, and the real group access list (including the real group ID) are used instead of the effective ID for checking permissions.

Even if a process has appropriate privileges and indicates success for *X_OK*,*R_OK* or *W_OK*, the file may not actually have execute permission bits set.

RETURN VALUES

If *path* cannot be found or if any of the desired access modes would not be granted, a value of -1 is returned; otherwise a 0 value is returned.

ERRORS

Access to the file is denied if:

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file currently being executed.
[EACCES]	Permission bits of the file mode do not permit the requested access, or search permission is denied

	on a component of the path prefix. The owner of a file has permissions checked with respect to the "owner" read, write, and execute mode bits. Members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.
[EFAULT]	*Path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chmod`(2POSIX), `stat`(2POSIX)

STANDARDS

The *access* call conforms to IEEE Std 1003.1-1988 *POSIX* .

CAVEAT

access is a potential security hole and should never be used.

NAME

bind – bind a name to a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int s, const struct sockaddr *name, int namelen);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *bind* system call assigns a name to an unnamed socket. When a socket is created using *socket*(2POSIX) it exists in a name space (address family) but has no name assigned to it. The *bind* system call requests that *name* be assigned to the socket.

RETURN VALUE

Upon successful completion, *bind* returns 0; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[EBADF]	The *s* argument is not a valid descriptor.
[ENOTSOCK]	The *s* argument is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the c_actor's address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`connect`(2POSIX), `getsockname`(2POSIX), `listen`(2POSIX), `socket`(2POSIX)

NAME

chdir, fchdir – change current directory

SYNOPSIS

```
#include <unistd.h>
int chdir(const char * path);

int fchdir(int fd);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

path points to the path name of a directory. *chdir* causes the named directory to become the current working directory of the calling c_actor, the starting point for path searches for pathnames not beginning with a slash (/).

The *fchdir* function causes the directory referenced by *fd* to become the current working directory, the starting point for path searches for pathnames not beginning with a slash (/).

In order for a directory to become the current directory, a process must have execute (search) access to the directory.

RETURN VALUES

Upon successful completion both *chdir* , and *fchdir* return 0; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[ENOTDIR]	A component of the *path* name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for a component of the *path* name.
[EFAULT]	*path* points outside the allocated address space of the c_actor.
[ENAMETOOLONG]	The length of a component of *path* exceeds NAME_MAX characters or the length of *path* exceeds PATH_MAX characters.
[ELOOP]	Too many symbolic links were encountered during analysis of *path* .
[EIO]	An I/O error occurred while reading from the file system.

fchdir will fail and the current working directory will be unchanged if one or more of the following are true:

[EACCES]	Search permission is denied for the directory referenced by the file descriptor.
[ENOTDIR]	The file descriptor does not reference a directory.
[EBADF]	The *fd* argument is not a valid file descriptor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chroot`(2POSIX)

STANDARDS

chdir is expected to conform to IEEE Std 1003.1-1988 *POSIX* .

HISTORY

The *fchdir* function call appeared in 4.2 BSD.

NAME

chmod, fchmod – change mode of file

SYNOPSIS

```
#include <sys/stat.h>
int chmod(const char * path, mode_t mode);

int fchmod(int fd, mode_t mode);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The `chmod()` function sets the file permission bits of the file specified by the pathname *path* to *mode*. `fchmod()` sets the permission bits of the specified file descriptor *fd*. `chmod()` verifies that the process owner (user) either owns the file specified by *path* (or *fd*), or is the super-user. A mode is created from `or`'d permission bit masks defined in `<sys/stat.h>`:

```
#define S_IRWXU       0000700    /* RWX mask for owner */
#define S_IRUSR       0000400    /* R for owner */
#define S_IWUSR       0000200    /* W for owner */
#define S_IXUSR       0000100    /* X for owner */

#define S_IRWXG       0000070    /* RWX mask for group */
#define S_IRGRP       0000040    /* R for group */
#define S_IWGRP       0000020    /* W for group */
#define S_IXGRP       0000010    /* X for group */

#define S_IRWXO       0000007    /* RWX mask for other */
#define S_IROTH       0000004    /* R for other */
#define S_IWOTH       0000002    /* W for other */
#define S_IXOTH       0000001    /* X for other */

#define S_ISUID       0004000    /* set user id on execution */
#define S_ISGID       0002000    /* set group id on execution */
#define S_ISVTX       0001000    /* save swapped text even after use */
```

The `ISVTX` (the *sticky* bit) indicates to the system which executable files are shareable (the default) and the system maintains the program text of the files in the swap area. The sticky bit may only be set by the super user on shareable executable files.

If mode `ISVTX` (the sticky bit) is set on a directory, an unprivileged user may not delete or rename files of other users in that directory. The sticky bit may be set by any user on a directory which the user owns or for which it has the appropriate permissions.

Writing or changing the owner of a file turns off the set-user-id and set-group-id bits unless the user is the super-user. This makes the system somewhat more secure by protecting set-user-id (set-group-id) files from remaining set-user-id (set-group-id) if they are modified, at the expense of a degree of compatibility.

RETURN VALUES

Upon successful completion, a value of `0` is returned. Otherwise, a value of `–1` is returned and `errno` is set to indicate the error.

ERRORS

`chmod()` will fail and the file mode will be unchanged if:

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*Path* points outside the process's allocated address space.
[EINVAL]	The pathname contains a character with the high-order bit set.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.

`fchmod()` will fail if:

[EBADF]	The descriptor is not valid.
[EINVAL]	Fd refers to a socket, not to a file.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EROFS]	The file resides on a read-only file system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX) , `chown`(2POSIX) , `stat`(2POSIX)

STANDARDS

`chmod` is expected to conform to IEEE Std 1003.1-1988 *POSIX* .

NAME

chown, fchown – change owner and group of a file

SYNOPSIS

```
#include <unistd.h>
int chown(const char * path, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The owner ID and group ID of the file named by *path* or referenced by *fd* is changed as specified by the arguments *owner* and *group* . The owner of a file may change the *group* to a group of which he or she is a member, but the change *owner* capability is restricted to the super-user.

The `chown()` function clears the set-user-id and set-group-id bits on the file to prevent accidental or mischievous creation of set-user-id and set-group-id programs.

`fchown()` is particularly useful when used in conjunction with the file locking primitives. See `flock`(2POSIX) .

RETURN VALUES

Upon successful completion, a value of `0` is returned. Otherwise, a value of `–1` is returned, and `errno` is set to indicate the error.

ERRORS

The `chown()` function will fail and the file will be unchanged if any of the following are true:

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*Path* points outside the process's allocated address space.
[EINVAL]	The pathname contains a character with the high-order bit set.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	Too many symbolic links were encountered when translating the pathname.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.
[ENOENT]	The file named does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID is not the super-user.
[EROFS]	The file named resides on a read-only file system.

`fchown( )` will fail if:

[EBADF]	*fd* does not refer to a valid descriptor.
[EINVAL]	*fd* refers to a socket, not a file.
[EPERM]	The effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.
[EIO]	An I/O error occurred while reading from or writing to the file system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chmod`(2POSIX) , `flock`(2POSIX)

STANDARDS

The `chown` function conforms to IEEE Std 1003.1-1988 *POSIX* . .

HISTORY

The `chown` function was changed to follow symbolic links in 4.4 BSD.

NAME

chroot – change the root directory

SYNOPSIS

#include <unistd.h>
int **chroot**(const char **dirname*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The `dirname` pointer indicates a pathname identifying a directory. The `chroot` system call causes the named directory to become the root directory of the calling c_actor. In other words, the named directory becomes the starting point for path searches for pathnames beginning with a slash, `/`. The c_actor's working directory is unaffected by the `chroot` system call.

The `.` (dot) entry in the root directory is interpreted to mean the root directory itself. Thus, `.` cannot be used to access files outside the subtree rooted at the root directory.

RETURN VALUE

Upon successful completion, `chroot` returns 0; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[ENOTDIR]	A component of the pathname is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the *path* name.
[EFAULT]	*path* points outside the allocated address space of the c_actor.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ELOOP]	Too many symbolic links or symbolic ports were encountered during analysis of *path*.
[EIO]	An I/O error occurred while reading from or writing to the file system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chdir`(2POSIX)

NAME

close – close a file descriptor

SYNOPSIS

```
#include <unistd.h>
int close(int fildes);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS, POSIX_SOCKETS, POSIX_SHM

DESCRIPTION

The *fildes* field contains a file descriptor obtained from an *open*(2POSIX), *dup*(2POSIX), *accept*(2POSIX), *socket*(2POSIX), or *shm_open*(2POSIX) system call. The *close* function closes the file descriptor indicated by *fildes*.

RETURN VALUES

Upon successful completion, *close* returns 0; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions.

ERRORS

[EBADF] *fildes* is not a valid open file descriptor.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

accept(2POSIX), afexec(2K), dup(2POSIX), open(2POSIX), socket(2POSIX), shm_open(2POSIX)

NAME

connect – initiate a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int s, const struct sockaddr *name, int namelen);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *s* parameter is a socket. If it is of the type SOCK_DGRAM, this call specifies the peer with which the socket is to be associated. Datagrams must be sent to this address; it is also the only address from which datagrams may be sent. If the socket is of the type SOCK_STREAM, this call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communication space of the socket. Each communication space interprets the *name* parameter in its own way. Generally, stream sockets may successfully *connect* only once; datagram sockets may use *connect* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUE

Upon successful completion, *connect* returns 0; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[EBADF]	The *s* argument is not a valid descriptor.
[ENOTSOCK]	The *s* argument is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the address family specified cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network isn't reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the c_actor's address space.
[EINPROGRESS]	The socket is non-blocking and the connection cannot be completed immediately. It is possible to `select`(2POSIX) for completion by selecting the socket for writing.

[EALREADY] The socket is non-blocking and a previous connection attempt has not yet been completed.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`accept`(2POSIX), `bind`(2POSIX), `getsockname`(2POSIX), `select`(2POSIX), `socket`(2POSIX)

NAME

dup, dup2 – duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>
int dup(int fildes);

int dup2(int fildes, int fildes2);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS, POSIX_SOCKETS, POSIX_SHM

DESCRIPTION

The *fildes* file contains a file descriptor obtained from an *open* (2POSIX), *dup* , *accept* (2POSIX), *socket* (2POSIX), or *shm_open* (2POSIX) system call. The *dup* function returns a new file descriptor which has the following in common with the original:

- Same open file (socket or shared memory object)
- Same file pointer (both file descriptors share one file pointer)
- Same access mode (read, write or read / write).

The new file descriptor is set to remain open across *afexec* (2K) system calls. See *fcntl* (2POSIX).

The file descriptor returned is the lowest one available.

The *dup2* system call performs a similar function. In this form of the call, *fildes* is a non-negative integer less than `OPEN_MAX` as returned by *aconf(2K)* or *sysconf(3POSIX)* . The *dup2* call causes *fildes2* to refer to the same file as *fildes* . If *fildes2* already refers to an open file, it is closed first.

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate one of the following error conditions.

ERRORS

[EBADF]	*fildes* is not a valid open file descriptor.
[EMFILE]	`OPEN_MAX` file descriptors are currently open.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`accept`(2POSIX) , `afexec`(2K) , `close`(2POSIX) , `fcntl`(2POSIX) , `open`(2POSIX) , `socket`(2POSIX) , `shm_open`(2POSIX) , `aconf`(2K) , `sysconf`(3POSIX)

NAME

dup, dup2 – duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>
int dup(int fildes);

int dup2(int fildes, int fildes2);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS, POSIX_SOCKETS, POSIX_SHM

DESCRIPTION

The *fildes* file contains a file descriptor obtained from an *open* (2POSIX), *dup* , *accept* (2POSIX), *socket* (2POSIX), or *shm_open* (2POSIX) system call. The *dup* function returns a new file descriptor which has the following in common with the original:

- Same open file (socket or shared memory object)
- Same file pointer (both file descriptors share one file pointer)
- Same access mode (read, write or read / write).

The new file descriptor is set to remain open across *afexec* (2K) system calls. See *fcntl* (2POSIX).

The file descriptor returned is the lowest one available.

The *dup2* system call performs a similar function. In this form of the call, *fildes* is a non-negative integer less than `OPEN_MAX` as returned by *aconf(2K)* or *sysconf(3POSIX)* . The *dup2* call causes *fildes2* to refer to the same file as *fildes* . If *fildes2* already refers to an open file, it is closed first.

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate one of the following error conditions.

ERRORS

[EBADF]	*fildes* is not a valid open file descriptor.
[EMFILE]	`OPEN_MAX` file descriptors are currently open.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`accept`(2POSIX) , `afexec`(2K) , `close`(2POSIX) , `fcntl`(2POSIX) , `open`(2POSIX) , `socket`(2POSIX) , `shm_open`(2POSIX) , `aconf`(2K) , `sysconf`(3POSIX)

NAME

chdir, fchdir – change current directory

SYNOPSIS

```
#include <unistd.h>
int chdir(const char * path);

int fchdir(int fd);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

path points to the path name of a directory. *chdir* causes the named directory to become the current working directory of the calling c_actor, the starting point for path searches for pathnames not beginning with a slash (/).

The *fchdir* function causes the directory referenced by *fd* to become the current working directory, the starting point for path searches for pathnames not beginning with a slash (/).

In order for a directory to become the current directory, a process must have execute (search) access to the directory.

RETURN VALUES

Upon successful completion both *chdir* , and *fchdir* return 0; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[ENOTDIR]	A component of the *path* name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for a component of the *path* name.
[EFAULT]	*path* points outside the allocated address space of the c_actor.
[ENAMETOOLONG]	The length of a component of *path* exceeds NAME_MAX characters or the length of *path* exceeds PATH_MAX characters.
[ELOOP]	Too many symbolic links were encountered during analysis of *path* .
[EIO]	An I/O error occurred while reading from the file system.

fchdir will fail and the current working directory will be unchanged if one or more of the following are true:

[EACCES]	Search permission is denied for the directory referenced by the file descriptor.
[ENOTDIR]	The file descriptor does not reference a directory.
[EBADF]	The *fd* argument is not a valid file descriptor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chroot`(2POSIX)

STANDARDS

chdir is expected to conform to IEEE Std 1003.1-1988 *POSIX* .

HISTORY

The *fchdir* function call appeared in 4.2 BSD.

NAME

chmod, fchmod – change mode of file

SYNOPSIS

```
#include <sys/stat.h>
int chmod(const char * path, mode_t mode);

int fchmod(int fd, mode_t mode);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The `chmod()` function sets the file permission bits of the file specified by the pathname *path* to *mode*. `fchmod()` sets the permission bits of the specified file descriptor *fd*. `chmod()` verifies that the process owner (user) either owns the file specified by *path* (or *fd*), or is the super-user. A mode is created from `or`'d permission bit masks defined in `<sys/stat.h>`:

```
#define S_IRWXU       0000700    /* RWX mask for owner */
#define S_IRUSR       0000400    /* R for owner */
#define S_IWUSR       0000200    /* W for owner */
#define S_IXUSR       0000100    /* X for owner */

#define S_IRWXG       0000070    /* RWX mask for group */
#define S_IRGRP       0000040    /* R for group */
#define S_IWGRP       0000020    /* W for group */
#define S_IXGRP       0000010    /* X for group */

#define S_IRWXO       0000007    /* RWX mask for other */
#define S_IROTH       0000004    /* R for other */
#define S_IWOTH       0000002    /* W for other */
#define S_IXOTH       0000001    /* X for other */

#define S_ISUID       0004000    /* set user id on execution */
#define S_ISGID       0002000    /* set group id on execution */
#define S_ISVTX       0001000    /* save swapped text even after use */
```

The `ISVTX` (the *sticky* bit) indicates to the system which executable files are shareable (the default) and the system maintains the program text of the files in the swap area. The sticky bit may only be set by the super user on shareable executable files.

If mode `ISVTX` (the sticky bit) is set on a directory, an unprivileged user may not delete or rename files of other users in that directory. The sticky bit may be set by any user on a directory which the user owns or for which it has the appropriate permissions.

Writing or changing the owner of a file turns off the set-user-id and set-group-id bits unless the user is the super-user. This makes the system somewhat more secure by protecting set-user-id (set-group-id) files from remaining set-user-id (set-group-id) if they are modified, at the expense of a degree of compatibility.

RETURN VALUES

Upon successful completion, a value of `0` is returned. Otherwise, a value of `–1` is returned and `errno` is set to indicate the error.

ERRORS

`chmod()` will fail and the file mode will be unchanged if:

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*Path* points outside the process's allocated address space.
[EINVAL]	The pathname contains a character with the high-order bit set.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.

`fchmod()` will fail if:

[EBADF]	The descriptor is not valid.
[EINVAL]	Fd refers to a socket, not to a file.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EROFS]	The file resides on a read-only file system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX) , `chown`(2POSIX) , `stat`(2POSIX)

STANDARDS

`chmod` is expected to conform to IEEE Std 1003.1-1988 *POSIX* .

NAME

chown, fchown – change owner and group of a file

SYNOPSIS

```
#include <unistd.h>
int chown(const char * path, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The owner ID and group ID of the file named by *path* or referenced by *fd* is changed as specified by the arguments *owner* and *group* . The owner of a file may change the *group* to a group of which he or she is a member, but the change *owner* capability is restricted to the super-user.

The `chown()` function clears the set-user-id and set-group-id bits on the file to prevent accidental or mischievous creation of set-user-id and set-group-id programs.

`fchown()` is particularly useful when used in conjunction with the file locking primitives. See `flock`(2POSIX) .

RETURN VALUES

Upon successful completion, a value of `0` is returned. Otherwise, a value of `–1` is returned, and `errno` is set to indicate the error.

ERRORS

The `chown()` function will fail and the file will be unchanged if any of the following are true:

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*Path* points outside the process's allocated address space.
[EINVAL]	The pathname contains a character with the high-order bit set.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	Too many symbolic links were encountered when translating the pathname.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.
[ENOENT]	The file named does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID is not the super-user.
[EROFS]	The file named resides on a read-only file system.

`fchown()` will fail if:

[EBADF]	*fd* does not refer to a valid descriptor.
[EINVAL]	*fd* refers to a socket, not a file.
[EPERM]	The effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.
[EIO]	An I/O error occurred while reading from or writing to the file system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chmod`(2POSIX) , `flock`(2POSIX)

STANDARDS

The `chown` function conforms to IEEE Std 1003.1-1988 *POSIX* . .

HISTORY

The `chown` function was changed to follow symbolic links in 4.4 BSD.

NAME

fcntl – file control

SYNOPSIS

```
#include <fcntl.h>
int fcntl(int fildes, int cmd, ... /*arg */);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS, POSIX_SOCKETS, POSIX_SHM

DESCRIPTION

The *fcntl* system call provides a control mechanism for open files. The *fildes* filed contains a file descriptor obtained from an *open*(2POSIX), *dup*(2POSIX), `accept`(2POSIX), *socket*(2POSIX) or *shm_open*(2POSIX) system call.

The *commands* available are:

F_DUPFD

- Lowest numbered available descriptor greater than or equal to *arg*
- Same object references as the original descriptor
- New descriptor shares the same file offset if the object was a file
- Same access mode (read, write or read/write)
- Same file status flags (i.e., both file descriptors share the same file status flags)
- The close-on-exec flag associated with the new file descriptor is set to remain open across *afexec*(2POSIX) system calls.

F_GETFD — Get the close-on-afexec flag associated with the file descriptor *fildes*. If the low-order bit is `0` the file will remain open across a local *afexec*(2POSIX), otherwise the file will be closed upon execution of *afexec*(2POSIX).

F_SETFD — Set the close-on-afexec flag associated with *fildes* to the low-order bit of *arg* (`0` or `1` as above).

F_GETFL — Get *fildes* status flags. This flag is not supported by shared memory objects.

F_SETFL — Set *fildes* status flags to the integer value given as the third argument. Only the O_APPEND and O_NONBLOCK flags can be set. This flag is not supported by shared memory objects.

RETURN VALUES

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD — A new file descriptor

F_GETFD	Value of the flag (only the low-order bit is defined)
F_SETFD	Value other than –1
F_GETFL	Value of file status flags. The return value will not be negative.
F_SETFL	Value other than –1

Otherwise, a value of –1 is returned and *errno* is set to indicate one of the following error conditions.

ERRORS

[EBADF]	*fildes* is not a valid open file descriptor.
[EINVAL]	*cmd* is not a valid command.
[EINVAL]	*fildes* defines a shared memory object or a message queue object, and either the F_GETLK or F_SETLK or F_SETLKW command was specified.
[ENOTTY]	The F_GETOWN or F_SETOWN command was specified.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`accept`(2POSIX), `afexec`(2K), `close`(2POSIX), `dup`(2POSIX), `open`(2POSIX), `socket`(2POSIX), `shm_open`(2POSIX)

NAME

flock – apply or remove an advisory lock on an open file

SYNOPSIS

```
#include <sys/file.h>
#define LOCK_SH   0x01   /* shared file lock */
#define LOCK_EX   0x02   /* exclusive file lock */
#define LOCK_NB   0x04   /* do not block when locking */
#define LOCK_UN   0x08   /* unlock file */
int flock(int fd, int operation);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

`flock()` applies or removes an *advisory* lock on the file associated with the file descriptor *fd*. A lock is applied by specifying an *operation* parameter that is one of `LOCK_SH` or `LOCK_EX` with the optional addition of `LOCK_NB`. An existing lock is removed using the `LOCK_UN` *operation*.

PARAMETERS

`flock()` takes the following parameters:

fd	File descriptor of the file on which to perform *operation*
operation	Operation to be performed on *fd*

EXTENDED DESCRIPTION

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency. Processes may, for example, access files without using advisory locks, which may result in inconsistencies.

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. At any time, multiple shared locks may be applied to a file, but at no time may multiple exclusive locks or both shared and exclusive locks be applied simultaneously to a file.

A shared lock may be *upgraded* to an exclusive lock and vice versa simply by specifying the appropriate lock type. Upgrading a lock involves releasing the previous lock and applying the new lock, possibly after other processes have obtained and released the lock.

Requesting a lock on an object that is already locked normally causes the caller to be blocked until the lock can be applied. If `LOCK_NB` is included in *operation*, the caller is not blocked. Instead, the call fails and `flock()` returns the error `EWOULDBLOCK`.

RETURN VALUES

`0` is returned if the operation was successful. If unsuccessful, `-1` is returned and `errno`.is set to indicate one of the following error conditions.

ERRORS

The `flock()` call fails if:

`EWOULDBLOCK`	The file is locked and the `LOCK_NB` *operation* was specified.
`EBADF`	The *fd* argument is an invalid descriptor.

EINVAL	The *fd* argument refers to an object other than a file.
EOPNOTSUPP	The *fd* argument refers to an object that does not support file locking.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

close(2POSIX), dup(2POSIX), open(2POSIX)

NOTES

Locks apply to files, not file descriptors. That is, file descriptors duplicated through dup( ) or afexec( ) do not result in multiple instances of a lock, but rather multiple references to a single lock. If, for example, an actor holding a lock on a file spawns and the child explicitly unlocks the file, the parent loses its lock.

NAME

fpathconf – get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>
long fpathconf(int fd, int name);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

`fpathconf()` provides a method for applications to determine the current value of a configurable system limit or option variable associated with a pathname or file descriptor.

PARAMETERS

`fpathconf()` takes the parameters described in the list below:

fd	An open file descriptor.
name	A system variable to be queried.
path	A file or directory name. Symbolic constants for each *name* value are found in the include file `<unistd.h>`.

name may be one of:

`_PC_CHOWN_RESTRICTED`	Returns `1` if appropriate privileges are required for the `chown` system call, otherwise returns `0`.
`_PC_LINK_MAX`	The maximum file link count.
`_PC_MAX_CANON`	The maximum number of bytes in a terminal canonical input line.
`_PC_MAX_INPUT`	The maximum number of bytes for which space is available in a terminal input queue.
`_PC_NAME_MAX`	The maximum number of bytes in a file name.
`_PC_NO_TRUNC`	Returns `1` if file names longer than `KERN_NAME_MAX` are truncated.
`_PC_PATH_MAX`	The maximum number of bytes in a pathname.
`_PC_PIPE_BUF`	The maximum number of bytes written automatically to a pipe.
`_PC_VDISABLE`	The terminal character disabling value.

RETURN VALUES

`fpathconf()` returns the following values:

`–1`	The call was not successful and `errno` was set appropriately, or the variable is associated with functionality that does not have a limit in the system and `errno` was not modified.
other	The current variable value.

ERRORS

If any of the following conditions occur, the `fpathconf( )` function returns `–1` and set `errno` to one of the following error conditions.

`[EINVAL]`	The value of the *name* argument is invalid.
`[EINVAL]`	The implementation does not support an association of the variable name with the associated file.

`fpathconf( )` will fail if:

`[EBADF]`	*fd* is not a valid open file descriptor.
`[EIO]`	An I/O error occurred while reading from or writing to the file system.

EXAMPLES

EXAMPLE 1

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysctl`(2POSIX)

NAME

stat, lstat, fstat – get file status

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
int **stat**(const char * *path*, struct stat * *buf*);

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

int **fstat**(int *fildes*, struct stat * *buf*);

FEATURES

MSDOSFS, NFS_CLIENT

DESCRIPTION

The *path* parameter points to the pathname of a file. Read, write, or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable. The *lstat* function is similar to *stat* except when the named file is a symbolic link, in which case *lstat* returns information about the link, while *stat* returns information about the file's link references. Unlike other filesystem objects, symbolic links do not have an owner, group, access mode, or times. Instead, these attributes are taken from the directory that contains the link. The only attributes returned from an *lstat* that refer to the symbolic link itself are the file type (S_IFLNK), size, blocks, and link count (always 1).

Similarly, *fstat* obtains information about an open file referenced by the file descriptor *fildes* , obtained from a successful *open* (2POSIX), or *dup* (2POSIX) system call.

The *buf* pointer indicates a *stat* structure into which information concerning the file is placed.

The contents of the structure pointed to by *buf* include the following members:

```
mode_t st_mode;          /* File mode */
ino_t st_ino;            /* Inode number */
dev_t st_dev;            /* ID of device containing (special file only) */
                         /* a directory entry for this file */
dev_t st_rdev;           /* ID of device (special files only) */
nlink_t st_nlink;        /* Number of links */
uid_t st_uid;            /* User ID of the file's owner */
gid_t st_gid;            /* Group ID of the file's group */
size_t st_size;          /* File size in bytes */
timestruct_t st_atime;   /* Time of last access */
timestruct_t st_mtime;   /* Time of last data modification */
timestruct_t st_ctime;   /* Time of last file status change */
                         /* Time is measured since 00:00:00 GMT, Jan. 1, 1970 */
```

RETURN VALUE

Upon successful completion, *stat* , *lstat* and *fstat* return 0; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*buf* or *path* points to an invalid address.
[ENAMETOOLONG]	The length of a component of *name* exceeds NAME_MAX characters, or the length of *name* exceeds PATH_MAX characters.
[ELOOP]	Too many symbolic links or symbolic ports were encountered during analysis of *name* .
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EBADF]	*fildes* is not a valid open file descriptor.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

dup(2POSIX) , open(2POSIX)

NAME

statfs, fstatfs – get file system statistics

SYNOPSIS

#include <sys/param.h>
#include <sys/mount.h>
int **statfs**(const char * *path*, struct statfs * *buf*);

int **fstatfs**(int *fd*, struct statfs * *buf*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *statfs* function call returns information about a mounted file system. The path name of any file in the mounted filesystem is defined by *path* . The *buf* pointer indicates a *statfs* structure defined as follows:

```
typedef     quad     fsid_t;

#define MNAMELEN 90     /* length of buffer for returned name */

struct statfs {
    short   f_type;                 /* type of filesystem (see below) */
    short   f_flags;                /* copy of mount flags */
    long    f_bsize;                /* fundamental file system block size */
    long    f_iosize;               /* optimal transfer block size */
    long    f_blocks;               /* total data blocks in file system */
    long    f_bfree;                /* free blocks in fs */
    long    f_bavail;               /* free blocks avail to non-superuser */
    long    f_files;                /* total file nodes in file system */
    long    f_ffree;                /* free file nodes in fs */
    fsid_t  f_fsid;                 /* file system id */
    long    f_spare[9];             /* spare for later */
    char    f_mntonname[MNAMELEN];   /* mount point */
    char    f_mntfromname[MNAMELEN]; /* mounted filesystem */
};
/*
* File system types.
*/
#define MOUNT_UFS        1     /* Fast Filesystem */
#define MOUNT_NFS        2     /* Sun-compatible Network Filesystem */
#define MOUNT_MFS        3     /* Memory-based Filesystem */
#define MOUNT_MSDOS      4     /* MS/DOS Filesystem */
#define MOUNT_LFS        5     /* Log-based Filesystem */
#define MOUNT_LOFS       6     /* Loopback Filesystem */
#define MOUNT_FDESC      7     /* File Descriptor Filesystem */
#define MOUNT_PORTAL     8     /* Portal Filesystem */
#define MOUNT_NULL       9     /* Minimal Filesystem Layer */
#define MOUNT_UMAP      10     /* Uid/Gid Remapping Filesystem */
#define MOUNT_KERNFS    11     /* Kernel Information Filesystem */
#define MOUNT_PROCFS    12     /* /proc Filesystem */
#define MOUNT_AFS       13     /* Andrew Filesystem */
#define MOUNT_CD9660    14     /* ISO9660 (aka CDROM) Filesystem */
#define MOUNT_UNION     15     /* Union (translucent) Filesystem */
```

Fields that are undefined for a particular file system are set to undefined values. The *fstatfs* system call returns the same information about an open file referenced by the *fd* descriptor.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error condition.

ERRORS

The error messages for *statfs* are the following:

[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[EINVAL]	*path* contains a character with the high-order bit set.
[ENAMETOOLONG]	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
[ENOENT]	The file referred to by *path* does not exist.
[EACCES]	Search permission is denied for a component of the path prefix of *path* .
[ELOOP]	Too many symbolic links were encountered in translating *path* .
[EFAULT]	In user mode, *buf* or *path* points to an invalid address. In supervisor mode, this is not detected and the state of the target is unknown.
[EIO]	An *I/O* error occurred while reading from or writing to the file system.

The error messages for *fstatfs* are the following:

[EBADF]	*fd* is not a valid open file descriptor.
[EFAULT]	*buf* points to an invalid address.
[EIO]	An *I/O* error occurred while reading from or writing to the file system.

HISTORY

The *statfs* function first appeared in 4.4BSD.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

fsync – synchronize a file's in-memory state with that on the physical medium

SYNOPSIS

```
#include <unistd.h>
int fsync(int fd);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

`fsync()` causes all modified data and attributes of *fd* to be moved to a permanent storage device. This usually results in all in-core modified copies of buffers for the associated file to be written to a disk.

`fsync()` should be used by programs that require a file to be in a known state, for example, in building a simple transaction facility.

PARAMETERS

`fsync()` takes the following parameter:

fd	File descriptor of the file to be synchronized

RETURN VALUES

On success, `0` is returned, if an error occurs, `-1` is returned and an error code is written to the global location `errno`.

ERRORS

The `fsync()` call fails if:

`EBADF`	*fd* is not a valid file descriptor.
`EINVAL`	The *fd* argument refers to an object other than a file.
`EIO`	An I/O error occurred while reading from or writing to the file system.
`ENOSYS`	IOM is not configured.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sync`(2POSIX)

NAME

truncate, ftruncate – truncate a file to a specified length

SYNOPSIS

#include <unistd.h>
int **truncate**(const char * *path*, off_t *length*);

int **ftruncate**(int *fd*, off_t *length*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *truncate* call causes the file named by *path* or referenced by *fd* to be truncated to a maximum of *length* bytes in size. If the file had previously been larger than the size specified, the extra data is lost. It must be possible to write to the file in order to use *ftruncate* .

RESTRICTION

An *ftruncate* operation performed after the first *mmap(2POSIX)* on the object will fail.

RETURN VALUES

If successful, *truncate* returns a value of 0; otherwise a value of –1 is returned, and *errno* is set to indicate one of the following error conditions.

ERRORS

The error messages for *truncate* are the following:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.
[ENOENT]	The file named does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	The file named is not writable by the user.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EISDIR]	The file named is a directory.
[EROFS]	The file named resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EIO]	An I/O error occurred updating the inode.
[EFAULT]	In user mode, *path* points outside the process' allocated address space. In supervisor mode, this is not detected, and the target's state is unknown.

The error messages for *ftruncate* are the following:

[EBADF]	The *fd* is not a valid descriptor.
[EINVAL]	The *fd* references a socket, not a file.
[EINVAL]	The *fd* is not open for writing.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mmap`(2POSIX) , `open`(2POSIX)

BUGS

These calls should be kept general to allow ranges of bytes in a file to be discarded.

HISTORY

This function call appeared in 4.2BSD.

NAME

getdirentries – get directory entries in a filesystem—independent format

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
int **getdirentries**(int *fd*, char **buf*, int *nbytes*, long **basep*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *getdirentries* function reads directory entries from the directory referenced by the file descriptor *fd* into the buffer pointed to by *buf*, in a filesystem—independent format. Up to *nbytes* of data will be transferred. The *nbytes* parameter must be greater than or equal to the block size associated with the file,)(see *stat(2POSIX)*) . Some filesystems may not support *getdirentries* with buffers smaller than this size.

The data in the buffer is a series of *dirent* structures, each containing the following entries:

```
unsigned longR     d_fileno;
unsigned short     d_reclen;
unsigned short     d_namlen;
char               d_name[MAXNAMELEN + 1]; /* see below */
```

The *d_fileno* entry is a number which is unique for each distinct file in the filesystem. Files that are linked by hard links (see *link(2POSIX)*) have the same *d_fileno*. The *d_reclen* entry is the length, in bytes, of the directory record. The *d_name* entry contains a null terminated file name. The *d_namlen* entry specifies the length of the file name excluding the null byte. The size of *d_name* may vary from 1 to *MAXNAMELEN + 1*.

Entries may be separated by extra spaces. The *d_reclen* entry may be used as an offset from the start of a *dirent* structure to the next structure, if any.

The actual number of bytes transferred is returned. The current position pointer associated with *fd* is set to point to the next block of entries. The pointer may not advance by the number of bytes returned by *getdirentries*. A value of zero is returned when the end of the directory has been reached.

The *getdirentries* function writes the position of the block read into the location pointed to by *basep*. Alternatively, the current position pointer may be set and retrieved using *lseek(2POSIX)*. The current position pointer should only be set to a value returned by *lseek(2POSIX)*, a value returned in the location pointed to by *basep*, or zero.

RETURN VALUES

If successful, the number of bytes actually transferred is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate one of the following error conditions.

ERRORS

[EBADF] *fd* is not a valid file descriptor open for reading.

[EFAULT]	In user mode, either *buf* or *basep* points outside the allocated address space. In supervisor mode, this is not detected and the target's state is unknown.
[EIO]	An I/O error occurred while reading from or writing to the file system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX), `lseek`(2POSIX), `stat`(2POSIX)

HISTORY

The getdirentries function first appeared in 4.4 BSD.

NAME

getdomainname, sethostname – get or set the domainname of the current host

SYNOPSIS

```
#include <unistd.h>
int getdomainname(char * name, int namelen);

int setdomainname(const char * name, int namelen);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

`getdomainname()` returns the standard domain name for the current host, as previously set by `setdomainname()`. The parameter *namelen* specifies the size of the *name* array. The returned *name* is null-terminated unless insufficient space is provided.

`setdomainname()` sets the domain name of the host system to be *name*, which has length *namelen*. This call is restricted to the superuser and is normally used only when the system is bootstrapped.

RETURN VALUES

If the call succeeds a value of `0` is returned. If the call fails, a value of `-1` is returned and an error code is placed in the global location `errno`.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller tried to set the hostname and was not the superuser.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`gethostname`(2POSIX), `sethostname`(2POSIX), `sysctl`(3POSIX)

NAME

getfh – get file handle

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mount.h>
int getfh(char *path, fhandle_t *fhp);
```

FEATURES

NFS_SERVER

DESCRIPTION

The *getfh* function returns a file handle for the specified file or directory into the file handle pointed to by *fhp*. This system call is restricted to the superuser.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate one of the following error conditions.

ERRORS

[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[EINVAL]	*path* contains a character with the high-order bit set.
[ENAMETOOLONG]	The length of a component of *path* exceeds NAME_MAX characters, or the length of *path* exceeds PATH_MAX characters.
[ENOENT]	The file referred to by *path* does not exist.
[EACCES]	Search permission is denied for a component of the path prefix of *path*.
[ELOOP]	Too many symbolic links were encountered in translating *path*.
[EFAULT]	In user mode,*fhp* points to an invalid address. In supervisor mode, this is not detected, and the state of the target is unknown.
[EIO]	An I/O error occurred while reading from or writing to the file system.

HISTORY

The *getfh* function first appeared in 4.4 BSD.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getfsstat – get list of all mounted filesystems

SYNOPSIS

#include <sys/param.h>
#include <sys/ucred.h>
#include <sys/mount.h>
int **getfsstat**(struct statfs *_buf_, long _bufsize_, int _flags_);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *getfsstat* function returns information about all mounted filesystems. The *buf* parameter is a pointer to *statfs* structures defined as follows:

```
typedef quad fsid_t;

#define MNAMELEN 90 /* length of buffer for returned name */
                    /* final '\0' is always provided */

struct statfs {
    short   f_type;     /* type of filesystem (see below) */
    short   f_flags;    /* copy of mount flags */
    long    f_bsize;    /* fundamental filesystem block size */
    long    f_iosize;   /* optimal transfer block size */
    long    f_blocks;   /* total data blocks in filesystem */
    long    f_bfree;    /* free blocks in fs */
    long    f_bavail;   /* free blocks avail to non-superuser */
    long    f_files;    /* total file nodes in filesystem */
    long    f_ffree;    /* free file nodes in fs */
    fsid_t  f_fsid;     /* filesystem id */
    long    f_spare[6]; /* spare for later */
    char    f_mntonname[MNAMELEN];   /* directory on which mounted */
    char    f_mntfromname[MNAMELEN]; /* mounted filesystem */
};
/*
 * File system types.
 */
#define   MOUNT_UFS   1
#define   MOUNT_NFS   2
#define   MOUNT_MSDOS 4   /* MS/DOS Filesystem */
```

Fields that are undefined for a particular filesystem are set to -1. The buffer is filled with an array of *fsstat* structures, one for each mounted filesystem up to the size specified by *bufsize*.

If *buf* is given as NULL, *getfsstat* only returns the number of mounted filesystems.

Normally *flags* should be specified as MNT_WAIT. If *flags* is set to MNT_NOWAIT, *getfsstat* will return the information it has available without requesting an update from each filesystem. Thus, some of the information will be out of date, but *getfsstat* will not block waiting for information from a filesystem that is unable to respond.

RETURN VALUES

Upon successful completion, the number of *fsstat* structures is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

getfsstat fails if one or more of the following conditions are true:

[EFAULT] Buf points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the filesystem.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`statfs`(2POSIX), `fstab`(4CC), `mountd`(1M)

RETURN VALUES

Upon successful completion, the number of *fsstat* structures is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

getfsstat fails if one or more of the following are true:

[EFAULT] In user mode, the pointer indicates an invalid address. In supervisor mode, this is not detected, and the state of the target is unknown.

[EIO] An I/O error occurred while reading from or writing to the filesystem.

SEE ALSO

`statfs`(2POSIX), `fstab`(4CC), `mountd`(1M)

HISTORY

The getfsstat function first appeared in 4.4 BSD.

NAME

hostname, gethostname, sethostname – get or set the name of the machine

SYNOPSIS

```
#include <unistd.h>
int gethostname(char * hostname, size_t len);

int sethostname(char * hostname, size_t len);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The pair of primitives *gethostname* and *sethostname* are used to get and set the name of the machine, respectively. The value of the *len* field defines the length of the name, and is limited to MAXHOSTNAMELEN (from <sys/param.h>).

Note - Currently, the value of MAXHOSTNAMELEN is 64.

RETURN VALUES

Upon successful completion, these primitives return 0; otherwise -1 is returned, and the global variable *errno* is set to indicate one of the following error conditions.

ERRORS

The *gethostname* primitive fails if:

[EFAULT] *len* is less than the current length of the machine name.

[EFAULT] The *name* or *namelen* gave an invalid address.

The *sethostname* primitive fails if either of the following are true:

[EFAULT] The new length of the machine name is greater than MAXHOSTNAMELEN.

Note - Currently, the value of MAXHOSTNAMELEN is 64

[EPERM] The caller is not the superuser.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getpeername – get name of connected peer

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int getpeername(int s, struct sockaddr *name, int *namelen);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *getpeername* function returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return, it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

RETURN VALUE

Upon successful completion, *getpeername* returns 0; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[EBADF]	The *s* argument is not a valid descriptor.
[ENOTSOCK]	The *s* argument is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available within the system to perform the operation.
[EFAULT]	The *name* parameter points to memory which is not in a valid part of the c_actor's address space.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

accept(2POSIX), bind(2POSIX), connect(2POSIX), getsockname(2POSIX), socket(2POSIX)

NAME

getrlimit, setrlimit – control maximum system resource consumption

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/types.h>
int getrlimit(int resource, struct rlimit * rlp);

int setrlimit(int resource, const struct rlimit * rlp);
```

DESCRIPTION

Limits on the consumption of system resources by the current actor may be obtained using the `getrlimit()` call, and set with the `setrlimit()` call.

PARAMETERS

`getrlimit()` and `setrlimit()` take the following *resource* parameters:

`RLIMIT_FSIZE`	The largest file size, in bytes, that can be created.
`RLIMIT_NOFILE`	The maximum number of open files for the process.

EXTENDED DESCRIPTION

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded, an actor is allowed to continue execution until it reaches the hard limit or modifies its resource limit. The `rlimit` structure is used to specify the hard and soft limits on a resource. For example:

```
struct rlimit {
        rlim_t rlim_cur; /* current (soft) limit */
        rlim_t rlim_max; /* maximum value for rlim_cur */
};
```

Only trusted actors may raise the maximum limits. Other actors may only alter `rlim_cur` within the range from `0` to `rlim_max` , or lower `rlim_max` irreversibly.

An “infinite” value for a limit is defined as `RLIM_INFINITY` .

RETURN VALUES

A return value of `0` indicates that the call succeeded in changing or returning the resource limit. A return value of `–1` means that an error occurred, and an error code is stored in the global location `errno` inficating one of the following error conditons.

ERRORS

`getrlimit()` and `setrlimit()` fail if:

`[EFAULT]`	The address specified for *rlp* is invalid.
`[EPERM]`	The limit specified for `setrlimit()` would raise the maximum value, and the caller is not a trusted actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getsockname – get socket name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockname(int s, struct sockaddr *name, int *namelen);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *getsockname* function returns the current *name* for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

RETURN VALUE

Upon successful completion, *getsockname* returns 0; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[EBADF]	The *s* argument is not a valid descriptor.
[ENOTSOCK]	The *s* argument is a file, not a socket.
[ENOBUFS]	Insufficient resources were available within the system to perform the operation.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`bind`(2POSIX), `socket`(2POSIX)

NAME

getsockopt, setsockopt – get and set options on sockets

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
int **getsockopt**(int *s*, int *level*, int *optname*, void * *optval*, int * *optlen*);

int **setsockopt**(int *s*, int *level*, int *optname*, const void * *optval*, int *optlen*);

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *getsockopt* and *setsockopt* functions manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost “socket” level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the “socket” level, *level* is specified as SOL_SOCKET. To manipulate options at any other level, the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol mumber of TCP ; see `getprotoent`(3POSIX) .

The *optval* and *optlen* parameters are used to access option values for *setsockopt* . For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt* , *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval* , and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as 0.

The *optname* parameter and any options specified are passed uninterpreted to the appropriate protocol module for interpretation. The include file < *sys/socket.h* > contains definitions for “socket” level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (7P).

Most socket-level options take an *int* parameter for *optval* . For *setsockopt* , the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. The SO_LINGER option uses a *struct linger* parameter, defined in < *sys/socket.h* >, which specifies the desired state of the option and the linger interval (see below). `SO_SNDTIMEO` and `SO_RCVTIMEO` use a `struct timeval` parameter, defined in `<sys/time.h>` .

The following options are recognized at the socket level. Except as noted, each may be examined using *getsockopt* and set using *setsockopt* .

SO_DEBUG	toggle recording of debugging information.
SO_REUSEADDR	toggle local address reuse.
SO_REUSEPORT	enable duplicate address and port bindings.

SO_KEEPALIVE	toggle keep connections alive.
SO_DONTROUTE	toggle routing bypass for outgoing messages.
SO_LINGER	linger on close if data present.
SO_BROADCAST	toggle permission to transmit broadcast messages.
SO_OOBINLINE	toggle reception of out-of-band data in band.
SO_SNDBUF	set buffer size for output.
SO_RCVBUF	set buffer size for input.
SO_SNDLOWAT	set minimum count for output.
SO_RECVLOWAT	set minimum count for input.
SO_SNDTIMEO	set timeout value for output.
SO_RCVTIMEO	set timout value for input.
SO_TYPE	get the type of the socket (get only).
SO_ERROR	get and clear error on the socket (get only).

The SO_DEBUG option enables debugging in the underlying protocol modules. The SO_REUSEADDR option indicates that the rules used in validating addresses supplied in a *bind* (2POSIX) call should allow reuse of local addresses. The SO_REUSEPORT option allows completely duplicate bindings by multiple processes if they all set SO_REUSEPORT before binding the port. This option permits multiple instances of a program, each to receive multicast or broadcast datagrams destined for the bound port. The SO_KEEPALIVE option enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken. The SO_DONTROUTE option indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

The SO_LINGER option controls the action taken when unsent messages are queued on a socket and a *close* (2POSIX) is performed. If the socket confirms reliable delivery of data and SO_LINGER is set, the system will block the c_actor on the *close (2POSIX)* attempt until it is able to transmit the data or until it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close (2POSIX)* is issued, the system will process the close in a manner that allows the c_actor to continue as quickly as possible.

The SO_BROADCAST option requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue

as received; it will then be accessible to *recv (2POSIX)* or `read` *(2POSIX)* calls without the MSG_OOB flag. The SO_SNDBUF and SO_RCVBUF options allow you to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values, which can be accessed through the `sysctl`(1M) MIB variable, `kern.maxsockbuf` .

SO_SNDLOWAT is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted, subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A select(2POSIX) operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for SO_SNDLOWAT is set to a convenient size for network efficiency, often 1024.

SO_RCVLOWAT is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amounts available or the amount requested. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark values or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that which was returned.

SO_SNDTIMEO is an option to set a timeout value for output operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this amount of time, it returns with a partial count or with the error EWOULDBLOCK if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output.

SO_RCVTIMEO is an option to set a timeout value for input operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error EWOULDBLOCK if no data were received.

Finally, SO_TYPE and SO_ERROR are options used only with *setsockopt* . The SO_TYPE option returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. The SO_ERROR option returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets, or any other asynchronous errors.

RETURN VALUE

Upon successful completion, *getsockopt* and *setsockopt* return 0; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[EBADF] The *s* argument is not a valid descriptor.

[ENOTSOCK] The *s* argument is a file, not a socket.

[ENOPROTOOPT] The option is unknown at the level indicated.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysctl`(1M) , `bind`(2POSIX) , `close`(2POSIX) , `ioctl`(2POSIX) , `read`(2POSIX) , `recv`(2POSIX) , `socket`(2POSIX) , `getprotoent`(3POSIX) , `protocols`(4CC)

NAME

gettimeofday, settimeofday – get/set date and time

SYNOPSIS

```
#include <sys/time.h>
int gettimeofday(struct timeval * tp, struct timezone * tzp);

int settimeofday(struct timeval * tp, struct timezone * tzp);
```

DESCRIPTION

The current Greenwich time and the current time zone on the system are obtained using the *gettimeofday* call, and set with the *settimeofday* call. The time is expressed as the number of seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware–dependent, and the time may be updated continuously or in "ticks". If *tp* or *tzp* is NULL, the associated time information will not be returned or set.

The structures pointed to by *tp* and *tzp* are defined in sys/time.h as:

```
struct timeval {
   long   tv_sec;    /* seconds since Jan. 1, 1970 */
   long   tv_usec;   /* and microseconds */
};
struct timezone {
   int   tz_minuteswest;   /* of Greenwich */
   int   tz_dsttime;       /* type of dst correction to apply */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if non zero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

Only the super-user may set the time of day or time zone.

RETURN

If successful, *gettimeofday* returns 0, otherwise it returns –1, and sets *errno* to indicate one of the following error conditions.

ERRORS

[EFAULT]	The address of an argument referenced invalid memory.
[EPERM]	A user other than the super-user attempted to set the time.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`date`(1CC) , `ctime`(3STDC) ,

NAME

hostname, gethostname, sethostname – get or set the name of the machine

SYNOPSIS

```
#include <unistd.h>
int gethostname(char * hostname, size_t len);

int sethostname(char * hostname, size_t len);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The pair of primitives *gethostname* and *sethostname* are used to get and set the name of the machine, respectively. The value of the *len* field defines the length of the name, and is limited to MAXHOSTNAMELEN (from <sys/param.h>).

Note - Currently, the value of MAXHOSTNAMELEN is 64.

RETURN VALUES

Upon successful completion, these primitives return 0; otherwise -1 is returned, and the global variable *errno* is set to indicate one of the following error conditions.

ERRORS

The *gethostname* primitive fails if:

[EFAULT] *len* is less than the current length of the machine name.

[EFAULT] The *name* or *namelen* gave an invalid address.

The *sethostname* primitive fails if either of the following are true:

[EFAULT] The new length of the machine name is greater than MAXHOSTNAMELEN.

Note - Currently, the value of MAXHOSTNAMELEN is 64

[EPERM] The caller is not the superuser.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioctl – control device

SYNOPSIS

```
#include <unistd.h>
int ioctl(int fildes, int request, ... /*arg */);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS, POSIX_SOCKETS

DESCRIPTION

ioctl performs the specified *request* on the object referred to by the open file descriptor *fildes*.

RETURN VALUE

Upon successful completion, *ioctl* returns 0; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[EBADF]	*fildes* is not a valid open file descriptor.
[ENOTTY]	*fildes* is not associated with a character special device or a socket
[EINVAL]	*request* or *arg* is not valid.
[EINTR]	The calling thread was aborted during the *ioctl* system call.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

open(2POSIX), socket(2POSIX)

NAME

link – make a hard file link

SYNOPSIS

#include <unistd.h>
int **link**(const char **name1*, const char **name2*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The `link` function call atomically creates the directory entry (hard link) specified by*name2* with the attributes of the underlying object pointed at by *name1*. If the link is successful, the link count of the underlying object is incremented; *name1* and *name2* share equal access and rights to the underlying object.

If *name1* is removed, the file *name2* is not deleted and the link count of the underlying object is decremented.

The *nme1* pointer must exist for the hard link to succeed, and both *name1* and *name2* must be in the same file system. Unless the caller is the super-user, *name1* cannot be a directory.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate one of the following error conditions.

ERRORS

[ENOTDIR]	A component of one of the path prefixes is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded `NAME_MAX` characters, or an entire pathname exceeded `PATH_MAX` characters.
[ENOENT]	A component of one of the path prefixes does not exist. The file named by *name1* does not exist.
[EACCES]	A component of one of the path prefixes denies search permission. The link requested requires writing to a directory that denies write permission.
[ELOOP]	Too many symbolic links were encountered in translating one of the pathnames.
[EEXIST]	The link named by *name2* does not t.
[EPERM]	The file named by *name1* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *name2* and the file named by *name1* are on different file systems.
[ENOSPC]	The directory in which the entry for the new link is being placed cannot be extended because there

	is no space left on the file system containing the directory.
[EIO]	An I/O error occurred while reading from or writing to the file system when making the directory entry.
[EROFS]	The requested link requires writing to a directory on a read-only file system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`unlink`(2POSIX)

NAME

listen – listen for connections on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int s, int backlog);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

To accept connections, a socket is first created using *socket*(2POSIX). The ability to accept incoming connections and a queue limit for incoming connections are specified using `listen`. The connections are accepted using `accept`(2POSIX). The `listen` call applies only to sockets of type SOCK_STREAM.

The *backlog* parameter defines the maximum length of the queue of pending connections. If a connection request arrives when the queue is full, the client may receive an error message (ECONNREFUSED), or, if the underlying protocol supports retransmission, the request may be ignored to enable retries.

RETURN VALUE

Upon successful completion, `listen` returns 0; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[EBADF]	The *s* argument is not a valid descriptor.
[ENOTSOCK]	The *s* argument is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the `listen` operation.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`accept`(2POSIX), `connect`(2POSIX), `socket`(2POSIX)

RESTRICTIONS

The *backlog* is currently limited (silently) to 5.

NAME

lseek – move a read / write file pointer

SYNOPSIS

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *fildes* field contains a file descriptor returned from an *open*(2POSIX), or *dup(2POSIX)* system call. The *lseek* function sets the file pointer associated with *fildes* as follows:

If `whence` is SEEK_SET (0), the pointer is set to *offset* bytes.

If `whence` is SEEK_CUR (1), the pointer is set to its current location plus *offset*.

If `whence` is SEEK_END (2), the pointer is set to the size of the file plus *offset*.

RETURN VALUE

Upon successful completion, *lseek* returns the resulting pointer location, as measured in bytes from the beginning of the file; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[EBADF]	*fildes* is not an open file descriptor.
[EINVAL]	`whence` is not SEEK_SET, SEEK_CUR or SEEK_END.
[EINVAL]	The resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with this type of device is undefined.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dup`(2POSIX), `open`(2POSIX)

NAME

stat, lstat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char * path, struct stat * buf);

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

int fstat(int fildes, struct stat * buf);
```

FEATURES

MSDOSFS, NFS_CLIENT

DESCRIPTION

The *path* parameter points to the pathname of a file. Read, write, or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable. The *lstat* function is similar to *stat* except when the named file is a symbolic link, in which case *lstat* returns information about the link, while *stat* returns information about the file's link references. Unlike other filesystem objects, symbolic links do not have an owner, group, access mode, or times. Instead, these attributes are taken from the directory that contains the link. The only attributes returned from an *lstat* that refer to the symbolic link itself are the file type (S_IFLNK), size, blocks, and link count (always 1).

Similarly, *fstat* obtains information about an open file referenced by the file descriptor *fildes* , obtained from a successful *open* (2POSIX), or *dup* (2POSIX) system call.

The *buf* pointer indicates a *stat* structure into which information concerning the file is placed.

The contents of the structure pointed to by *buf* include the following members:

```
mode_t st_mode;          /* File mode */
ino_t st_ino;            /* Inode number */
dev_t st_dev;            /* ID of device containing (special file only) */
                         /* a directory entry for this file */
dev_t st_rdev;           /* ID of device (special files only) */
nlink_t st_nlink;        /* Number of links */
uid_t st_uid;            /* User ID of the file's owner */
gid_t st_gid;            /* Group ID of the file's group */
size_t st_size;          /* File size in bytes */
timestruct_t st_atime;   /* Time of last access */
timestruct_t st_mtime;   /* Time of last data modification */
timestruct_t st_ctime;   /* Time of last file status change */
                         /* Time is measured since 00:00:00 GMT, Jan. 1, 1970 */
```

RETURN VALUE

Upon successful completion, *stat* , *lstat* and *fstat* return 0; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*buf* or *path* points to an invalid address.
[ENAMETOOLONG]	The length of a component of *name* exceeds NAME_MAX characters, or the length of *name* exceeds PATH_MAX characters.
[ELOOP]	Too many symbolic links or symbolic ports were encountered during analysis of *name* .
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EBADF]	*fildes* is not a valid open file descriptor.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

dup(2POSIX) , open(2POSIX)

NAME

mkdir – make a directory file

SYNOPSIS

#include <sys/stat.h>
int **mkdir**(const char **path*, mode_t *mode*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The directory *path* is created with the access permissions specified by *mode* and restricted by the *umask(2POSIX)* of the calling process.

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to that of the parent directory in which it is created.

RETURN VALUES

If succesful, *mkdir* returns 0; otherwise a value of –1 is returned, and *errno* is set to indicate one of the following error conditions.

ERRORS

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	The file named resides on a read-only file system.
[EEXIST]	The file named already exists.
[ENOSPC]	The new directory cannot be created because there is no space left on the file system to contain the directory.
[ENOSPC]	There are no free inodes on the file system on which the directory is being created.
[EDQUOT]	The new directory cannot be created because the user's quota of disk blocks on the file system that will contain the directory has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the directory is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.

[EIO]	An I/O error occurred while reading from or writing to the file system.
[EFAULT]	*path* points outside the process's allocated address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chmod`(2POSIX), `stat`(2POSIX)

STANDARDS

`mkdir` conforms to IEEE Std 1003.1-1988 *POSIX*.

RESTRICTIONS FOR ChorusOS

The current mask mode defaults to *rwxrwxrwx* on top of ChorusOS, see *security(4CC)* for further details.

NAME

mkfifo – make a fifo file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The `mkfifo` system call creates a new fifo file with the name *path*. The access permissions are specified by *mode*.

The fifo's owner ID is set to the process's effective user ID. The fifo's group ID is set to that of the parent directory in which it is created

If successful, *mkfifo* returns a value of 0, otherwise a value of –1 is returned, and *errno* is set to indicate one of the following error conditions.

ERRORS

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceeded `PATH_MAX` characters.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	The file named resides on a read-only file system.
[EEXIST]	The file named already exists.
[ENOSPC]	The directory in which the entry for the new fifo is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	There are no free inodes on the file system on which the fifo is being created.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode. An I/O error occurred while reading from or writing to the file system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `chmod`(2POSIX), `stat`(2POSIX)

NAME

mknod – make a special file node

SYNOPSIS

```
#include <unistd.h>
int mknod(const char *path, mode_t mode, dev_t dev);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The device special file *path* is created with the major and minor device numbers extracted from *mode*.

If *mode* indicates a block– or character– special file, *dev* is a configuration–dependent specification of a character or block I/O device and the superblock of the device. If *mode* does not indicate a block–special or character–special device, *dev* is ignored.

The use of `mknod` requires super-user privileges.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate one of the following error conditions.

ERRORS

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceeded `PATH_MAX` characters.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The process' effective user ID is not super-user.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[ENOSPC]	The directory in which the entry for the new node is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	There are no free inodes on the file system on which the node is being created.
[EROFS]	The file named resides on a read-only file system.
[EEXIST]	The file named already exists.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chmod`(2POSIX), `stat`(2POSIX)

NAME

mmap – map c_actor addresses to a shared memory object

SYNOPSIS

#include <sys/mman.h>
void *__mmap__(void **addr*, size_t *len*, int *prot*, int *flags*, int *fildes*, off_t *off*);

FEATURES

POSIX_SHM

DESCRIPTION

The *mmap* system call establishes a mapping between the c_actor's address space and a shared memory object. The format of the call is as follows:

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

It establishes a mapping between the c_actors's address space at an address *pa* for *len* bytes and the memory object represented by the file descriptor *fildes* at offset *off* for *len* bytes. The *fildes* parameter must have been obtained by a successful call to *shm_open*. The value of *pa* is a function of the parameter *addr* and the values of *flags*, described below. The *off* value must be a multiple of the page size (as returned by *vmPageSize(2K)*). A successful *mmap* call returns *pa* as its result. The address range starting at *pa* and continuing for *len* is legitimate for the current address space of the c_actor; this memory is locked. The range of bytes starting at *off* and continuing for *len* bytes is legitimate for the current offsets in the shared memory object represented by *fildes*.

The mapping established by *mmap* uses a free address range for those whole pages containing any part of the c_actor's address space starting at *pa* and continuing for *len* bytes.

The parameter *prot* determines the access permissions to the data being mapped. The *prot* parameter should be the bitwise inclusive OR of one or more of the other flags listed below and defined in the header `<sys/mman.h>`.

PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.

The behavior of PROT_WRITE is influenced by setting MAP_PRIVATE in the *flags* parameter as described below.

The *flags* parameter provides other information about the handling of the mapped data. The value of *flags* is the bitwise inclusive OR of these options as defined in the header `<sys/mman.h>`.

MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret *addr* exactly.

The MAP_SHARED and MAP_PRIVATE flags describe the disposition of write references to the memory object. If MAP_SHARED is specified, write references change the underlying object. If MAP_PRIVATE is specified, modifications to the mapped data by the calling c_actor are visible only to the calling c_actor and will not change the underlying object. Either MAP_SHARED or MAP_PRIVATE may be specified, but not both.

The MAP_FIXED flag tells the system that the value of *pa* is *addr* exactly.

When MAP_FIXED is not set, the *addr* parameter is ignored. The *pa* will be placed in an area of address space defined by the system for mapping of *len* bytes to the specified object.

If MAP_FIXED is set, the implementation requires that *addr* is a multiple of the page size (as returned by *vmPageSize(2K)).* The system performs mapping operations over whole pages. Thus, while the parameter *len* need not meet a size or alignment constraint, the system will include, in any mapping operation, any partial page specified by the address range starting at *pa* and continuing for *len* bytes.

The system always zero-fills any partial page at the end of an object.

RESTRICTION

If the size of the object is zero, *mmap* fails and returns the [EAGAIN] error code .

The PROT_NONE option is not supported.

RETURN VALUE

Upon successful completion, *mmap* returns the address at which the mapping was placed (*pa*); otherwise, it returns a value of MAP_FAILED and sets *errno* to indicate the error condition. The symbol MAP_FAILED is defined in the header `<mman.h>`.

ERRORS

[ENOSYS]	The *mmap* function is not supported.
	The POSIX_SHM feature is not configured.
[EINVAL]	The value in *flags* is invalid, neither MAP_PRIVATE nor MAP_SHARED is defined.
[EBADF]	*fildes* is not a valid open file descriptor.
[ENODEV]	The *fildes* argument refers to an object for which *mmap* is not supported, such as a file.
[EINVAL]	*off* is not a multiple of page size. Or MAP_FIXED was specified and *addr* is not a multiple of page size.
[EACCES]	The file descriptor *fildes* is not open for read access, regardless of the protection specified.

The file descriptor *fildes* is not open for write access and PROT_WRITE was specified for MAP_SHARED type mapping.

[ENXIO]	The addresses in the range starting at *off* and continuing for *len* bytes are invalid for the object specified by *fildes*.
[ENOMEM]	The mapping could not be locked in memory, because it would require more space than the system is able to supply.
[ENOMEM]	*len* exceeds the user address space size of a c_actor.
[ENOMEM]	MAP_FIXED was specified, and the address range starting at *addr* and continuing for *len* bytes exceeds that allowed for the address space of a c_actor; or MAP_FIXED was not specified and there is insufficient room in the address space to effect the mapping.
[EAGAIN]	The object has a size of 0.
[EAGAIN]	The mapping could not be locked in memory, due to a lack of resources.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`shm_open`(2POSIX), `shm_unlink`(2POSIX), `ftruncate`(2POSIX)

NAME

mount, unmount – mount or unmount a filesystem

SYNOPSIS

```
#include <sys/param.h>
#include <sys/mount.h>
int mount(int type, const char * dir, int flags, caddr_t data);

int unmount(const char * dir, int flags);
```

FEATURES

MSDOSFS, NFS_CLIENT, POSIX_SOCKETS, UFS

DESCRIPTION

The `mount` function grafts a filesystem object onto the system file tree at the point *dir* . The *data* argument describes the filesystem object to be mounted. The `type` argument defines how the kernel interprets *data* (See `type` below). The contents of the filesystem become available through the new mount point *dir* . Any files in *dir* at the time of a successful mount become hidden, and are unavailable until the filesystem is unmounted.

The following *flags* may be specified to suppress default semantics which affect filesystem access.

`MNT_RDONLY`	The filesystem should be treated as read-only; even the super-user may not write to it.
`MNT_NOEXEC`	Do not allow files to be executed from the filesystem.
`MNT_NOSUID`	Do not honor setuid or setgid bits on files when executing them.
`MNT_NODEV`	Do not interpret special files on the filesystem.
`MNT_SYNCHRONOUS`	All I/O to the filesystem should be done synchronously.
`MNT_NOATIME`	Disable update of file access times.
`MNT_ASYNC`	All I/O to the filesystem should be done asynchronously.
`MNT_WANTRDRW`	Upgrade a mounted read-only filesystem to read-write if MNT_UPDATE is also specified.
`MNT_FORCE`	Force a read-write mount even if the filesystem appears to be unclean. Use of this flag can be dangerous.

The `MNT_UPDATE` flag indicates that the mount command is being applied to a filesystem which is already mounted. This allows the mount flags to be changed without requiring that the filesystem be unmounted and remounted. Some

filesystems may not allow all flags to be changed. For example, most filesystems will not allow a change from read-write to read-only.

The *type* argument defines the type of the filesystem. The types of filesystems known to the system are defined in `sys/mount.h`. The *data* pointer indicates a structure that contains the type-specific arguments to mount. The types of filesystems currently supported and their type-specific data are:

```
Arguments for local filesystem mount calls

struct export_args {
    int     ex_flags;               /* export related flags */
    uid_t   ex_root;                /* mapping for root uid */
    struct  ucred ex_anon;          /* mapping for anonymous user */
    struct  sockaddr *ex_addr;      /* net address to which exported */
    int     ex_addrlen;             /* and the net address length */
    struct  sockaddr *ex_mask;      /* mask of valid bits in saddr */
    int     ex_masklen;             /* and the smask length */
     };


MOUNT_UFS

struct ufs_args {
    char      *fspec;               /* Block special file to mount */
    struct    export_args export;   /* network export information */
     };


MOUNT_NFS

struct nfs_args {
    struct sockaddr *addr;          /* file server address */
    int             addrlen;        /* length of address */
    int             sotype;         /* Socket type */
    int             proto;          /* and Protocol */
    u_char          *fh;            /* File handle to be mounted */
    int             fhsize;         /* Size, in bytes, of fh */
    int             flags;          /* flags */
    int             wsize;          /* write size in bytes */
    int             rsize;          /* read size in bytes */
    int             readdirsize;    /* readdir size in bytes */
    int             timeo;          /* initial timeout in .1 secs */
    int             retrans;        /* times to retry send */
    int             maxgrouplist;   /* Max. size of group list */
    int             readahead;      /* # of blocks to readahead */
    int             leaseterm;      /* Term (sec) of lease */
    int             deadthresh;     /* Retrans threshold */
    char            *hostname;      /* server's name */
     };


NFS mount option flags

     #define NFSMNT_SOFT         0x00000001  /* soft mount (hard is default) */
```

```
     #define NFSMNT_WSIZE        0x00000002  /* set write size */
     #define NFSMNT_RSIZE        0x00000004  /* set read size */
     #define NFSMNT_TIMEO        0x00000008  /* set initial timeout */
     #define NFSMNT_RETRANS      0x00000010  /* set number of request retries */
     #define NFSMNT_MAXGRPS      0x00000020  /* set maximum grouplist size */
     #define NFSMNT_INT          0x00000040  /* allow interrupts on hard mount */
     #define NFSMNT_NOCONN       0x00000080  /* Don't Connect the socket */
     #define NFSMNT_NQNFS        0x00000100  /* Use Nqnfs protocol */
     #define NFSMNT_NFSV3        0x00000200  /* Use NFS Version 3 protocol */
     #define NFSMNT_KERB         0x00000400  /* Use Kerberos authentication */
     #define NFSMNT_DUMBTIMR     0x00000800  /* Don't estimate rtt dynamically */
     #define NFSMNT_LEASETERM    0x00001000  /* set lease term (nqnfs) */
     #define NFSMNT_READAHEAD    0x00002000  /* set read ahead */
     #define NFSMNT_DEADTHRESH   0x00004000  /* set dead server retry thresh */
     #define NFSMNT_RESVPORT     0x00008000  /* Allocate a reserved port */
     #define NFSMNT_RDIRPLUS     0x00010000  /* Use Readdirplus for V3 */
     #define NFSMNT_READDIRSIZE  0x00020000  /* Set readdir size */
     #define NFSMNT_INTERNAL     0xfffc0000  /* Bits set internally */
     #define NFSMNT_HASWRITEVERF 0x00040000  /* Has write verifier for V3 */
     #define NFSMNT_GOTPATHCONF  0x00080000  /* Got the V3 pathconf info */
     #define NFSMNT_GOTFSINFO    0x00100000  /* Got the V3 fsinfo */
     #define NFSMNT_MNTD         0x00200000  /* Mnt server for mnt point */
     #define NFSMNT_DISMINPROG   0x00400000  /* Dismount in progress */
     #define NFSMNT_DISMNT       0x00800000  /* Dismounted */
     #define NFSMNT_SNDLOCK      0x01000000  /* Send socket lock */
     #define NFSMNT_WANTSND      0x02000000  /* Want above */
     #define NFSMNT_RCVLOCK      0x04000000  /* Rcv socket lock */
     #define NFSMNT_WANTRCV      0x08000000  /* Want above */
     #define NFSMNT_WAITAUTH     0x10000000  /* Wait for authentication */
     #define NFSMNT_HASAUTH      0x20000000  /* Has authenticator */
     #define NFSMNT_WANTAUTH     0x40000000  /* Wants an authenticator */
     #define NFSMNT_AUTHERR      0x80000000  /* Authentication error */
```

MOUNT_MSDOS

```
struct msdosfs_args {
     char    *fspec;       /* blocks special holding the fs to mount */
     struct  export_args export;     /* network export information */
     uid_t   uid;          /* uid that owns msdosfs files */
     gid_t   gid;          /* gid that owns msdosfs files */
     mode_t  mask;         /* mask to be applied for msdosfs perms */
     int     flags;        /* see below */
     int magic;            /* version number */
     u_int16_t u2w[128];   /* Local->Unicode table */
     u_int8_t  ul[128];    /* Local upper->lower table */
     u_int8_t  lu[128];    /* Local lower->upper table */
     u_int8_t  d2u[128];   /* DOS->local table */
     u_int8_t  u2d[128];   /* Local->DOS table */
     };
```

The umount function call dissociates the filesystem from the specified mount point *dir* .

The *flags* argument may specify *MNT_FORCE* to specify that the filesystem should be forcibly unmounted even if files are still active. Active special devices continue to work, but any further accesses to any other active files result in errors even if the filesystem is later remounted.

RETURN VALUES

If successful, `mount` returns a value of 0, otherwise -1 is returned and the *errno* variable is set to indicate the error.

If successful, `umount` returns a value of 0, otherwise -1 is returned and the variable *errno* is set to indicate the error.

ERRORS

`mount()` errors

`mount()` will fail when one of the following occurs:

[EPERM]	The caller is not the super-user.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or the entire length of a pathname exceeded 1023 characters.
[ELOOP]	Too many symbolic links were encountered in translating a pathname.
[ENOENT]	A component of *dir* does not exist.
[ENOTDIR]	A component of *dir* is not a directory, or a path prefix of *fspec* is not a directory.
[EBUSY]	Another process currently holds a reference to *dir* .
[EFAULT]	*dir* points outside the actor's allocated address space.

The following errors can occur for a UFS filesystem mount:

[ENODEV]	A component of `ufs_args` , *fspec* , does not exist.
[ENOTBLK]	*fspec* is not a block device.
[ENXIO]	The major device number of *fspec* is out of range (this indicates that no device driver exists for the associated hardware).
[EBUSY]	*fspec* is already mounted.
[EMFILE]	No space left in the mount table.
[EINVAL]	The superblock for the filesystem had a bad magic number or a block size that was out of range.

[ENOMEM]	Not enough memory was available to read the cylinder group information for the filesystem.
[EIO]	An I/O error occurred while reading the super block or cylinder group information.
[EFAULT]	*fspec* points outside the actor's allocated address space.

The following errors can occur for an NFS filesystem mount:

[ETIMEDOUT]	NFS timed out trying to contact the server.
[EFAULT]	Some part of the information described by *nfs_args* points outside the actor's allocated address space.

umount() errors

`umount()` may fail with one of the following errors:

[EPERM]	The caller is not the super-user.
[ENOTDIR]	A component of the path is not a directory.
[EINVAL]	A component of the path is not a directory.The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EINVAL]	The requested directory is not in the mount table.
[EBUSY]	A process is holding a reference to a file located on the filesystem.
[EIO]	An I/O error occurred while writing cached filesystem information.

A UFS mount can also fail if the maximum number of filesystems are currently mounted.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`C_INIT`(1M)

BUGS

Some of the error codes need translation to more obvious messages.

HISTORY

The `mount` and `umount` function calls appeared in Version 6 AT&T UNIX.

RESTRICTIONS FOR ChorusOS

Note that *mfs* filesystems are not supported.

NAME

mq_close – close a message queue

SYNOPSIS

```
#include <mqueue.h>
int mq_close(mqd_t mqdes);
```

FEATURES

POSIX_MQ

DESCRIPTION

The *mq_close* system call removes the association between the message queue descriptor, *mqdes* and its message queue. The *mqdes* message queue descriptor is obtained from a *mq_open(2POSIX)* system call. An *mq_close* system call is never blocked.

RETURN VALUE

Upon successful completion, *mq_close* returns a value of 0; otherwise, it will return -1 and set *errno* to indicate one of the following error conditions.

ERRORS

[ENOSYS]	The *mq_close* function is not supported.
	The MQ feature [_POSIX_MESSAGE_PASSING] option (see *sysconf(3POSIX)*) is not configured.
[EBADF]	The *mqdes* argument is not a valid message queue descriptor.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

mq_open(2POSIX), mq_unlink(2POSIX), sysconf(3POSIX)

NAME

mq_getattr – retrieve message queue attributes

SYNOPSIS

#include <mqueue.h>
int **mq_getattr**(mqd_t *mqdes*, struct mq_attr **mqstat*);

FEATURES

POSIX_MQ

DESCRIPTION

The *mq_getattr* system call is used to get status information and attributes associated with the message queue specified by *mqdes*.

The *mqstat* pointer indicates an *mq_attr* structure. This structure has the following form, as defined in the *mqueue.h* header:

```
long   mq_flags;    /* message queue flags  */
long   mq_maxmsg;   /* maximum number of messages */
long   mq_msgsize;  /* maximum message size */
long   mq_curmsgs;  /* number of mssages currently queued */
long   mq_sendwait; /* implementation extension: nb. send wait*/
long   mq_curmsgs;  /* implementation extension: nb. receive wait */
```

The following member represents an attribute that can be set and queried.

mq_flags Specifies actions and status for the message queue operations. If the O_NONBLOCK flag is set, neither the *mq_send(2POSIX)* nor the *mq_receive(2POSIX)* operations associated with this message queue will block.

The following members represent attributes that can be queried, but which can only be set at message queue creation.

mq_maxmsg Specifies the number of messages that can be held in the message queue without causing *mq_send* to fail or wait due to lack of resources.

mq_msgsize Specifies the maximum size of each message in the message queue.

The following members represent the current status of the dynamic attributes of the message queue. These can be queried, but they cannot be explicitly set.

mq_curmsgs Indicates the number of messages currently in the queue.

mq_sendwait Indicates the total number of threads waiting for *mq_send* to complete.

mq_rcvwait Indicates the total number of threads waiting for *mq_receive* to complete.

Upon successful return of *mq_getattr*, the *mq_flags* member of the *mq_attr* structure referenced by the *mqstat* argument will have the values that were set when the message queue was created, plus any modifications made by subsequent *mq_setattr* calls.

RETURN VALUE

Upon successful completion, *mq_getattr* returns 0. Otherwise, *mq_getattr* will return -1 and set *errno* to indicate one of the following error conditions.

ERRORS

[ENOSYS]	The *mq_getattr* function is not supported. The MQ feature [_POSIX_MESSAGE_PASSING] option (see *sysconf(3POSIX)*) is not configured.
[EBADF]	The *mqdes* argument is not a valid message queue descriptor.
[EFAULT]	*mqstat* points outside the allocated address space of the c_actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mq_open`(2POSIX), `mq_send`(2POSIX), `mq_receive`(2POSIX), `mq_setattr`(2POSIX), `sysconf`(3POSIX)

NAME

mq_open – open a message queue

SYNOPSIS

#include <mqueue.h>
mqd_t **mq_open**(const char **name*, int *oflag [*, mode_t *mode*, mq_attr **attr]*);

FEATURES

POSIX_MQ

DESCRIPTION

The *mq_open* system call establishes the connection between a c_actor and a message queue with a message queue descriptor. It creates a message queue descriptor that refers to that message queue. The message queue descriptor is used by other functions to refer to that message queue. The *name* argument points to a string naming a message queue. The name does not appear in the file system. The *name* argument points to a Posix object name. The message queue name cannot exceed MQ_PATHMAX characters as returned by *sysconf(3POSIX)*.

The *oflag* argument requests the desired receive and / or send access to the message queue. The requested access permission to receive messages or send messages is granted if the calling c_actor would be granted read or write access, respectively, to an equivalently protected file.

The value of *oflag* is the bitwise inclusive OR of values from the following list. Applications must specify precisely one of the first three values (access modes) below:

O_RDONLY	Open the message queue for receiving messages. The c_actor can use the returned message queue descriptor with *mq_receive*, but not *mq_send*. A message queue may be opened multiple times in the same or different c_actors for receiving messages.
O_WRONLY	Open the queue for sending messages. The c_actor can use the returned message queue descriptor with *mq_send* but not *mq_receive*. A message queue may be opened multiple times in the same or different c_actors for sending messages.
O_RDWR	Open the queue for both receiving and sending messages. The c_actor can use any of the functions allowed for O_RDONLY and O_WRONLY. A message queue may be opened multiple times in the same or different c_actors for sending and receiving messages.

Any combination of the remaining flags may be specified in the value of *oflag*:

O_CREAT	This option is used to create a message queue, and it requires two additional arguments: *mode* which is of type *mode_t*, and *attr*, which is a pointer to a *mq_attr* structure defined in *mq_getattr(2POSIX)*. If the posix object name, *name*, has already been used to create a message queue that still exists, this flag has no effect, except as noted under O_EXCL below. Otherwise, a message queue is created without any messages in it. The owner ID of the message queue is set to the user ID of the c_actor, the group ID of the message queue is set to the group ID of the c_actor. If *attr* in non-NULL, and the calling c_actor has the appropriate privilege on *name*, the message queue *mq_maxmsg* and *mq_msgsize* attributes are set to the values of the corresponding members in the *mq_attr* structure referred to by *attr*. If *attr* is NULL, the message queue is created with default message queue attributes as returned by *sysconf(3POSIX)*. If *attr* is non-NULL, but the calling c_actors does not have the appropriate privilege on *name*, *mq_open* will fail and return an error without creating the message queue.
O_EXCL	If O_EXCL and O_CREAT are set, *mq_open* will fail if the message queue *name* exists. The check for the existence of the message queue and the creation of the message queue if it does not exist is atomic with respect to other c_actors executing *mq_open* naming the same *name* with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set this flag is ignored.
O_NONBLOCK	The setting of this flag is associated with the open message queue descriptor, and determines whether a *mq_send* or *mq_receive* will wait for resources or messages that are not currently available, or fail with *errno* set to [EAGAIN].

Note that *mq_open* does not add or remove messages from the queue.

RETURN VALUE

Upon successful completion, *mq_open* returns a message queue descriptor. Otherwise, it returns *(mqd_t)* -1 and sets *errno* to indicate the error condition.

ERRORS

[ENOSYS]	The function *mq_open* is not supported. The MQ feature [_POSIX_MESSAGE_PASSING] option (see *sysconf(3POSIX)*) is not configured.
[EACCESS]	The message queue exists and the permissions specified by *oflag* are denied, or the message queue does not exist and permission to create the message queue is denied.
[EEXIST]	O_CREAT and O_EXCL are set and the named message queue already exists.
[ENAMETOOLONG]	The *name* string exceeds MQ_PATHMX.
[EINVAL]	The access modes specified by *oflag* are incompatible.
[EINVAL]	*attr* is non-NULL and the *mq_maxmsg* and *mq_msgsize* members in the *mq_attr* structure are invalid.
[EMFILE]	Too many message queue descriptors are currently in use by this c_actor.
[ENOSPC]	There is insufficient space for the creation of the new message queue.
[ENOENT]	O_CREAT is not set and the named message queue does not exist.
[EFAULT]	*name* or *attr* points outside the allocated address space of the c_actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mq_close`(2POSIX), `mq_receive`(2POSIX), `mq_send`(2POSIX), `mq_setattr`(2POSIX), `mq_getattr`(2POSIX), `mq_unlink`(2POSIX), `sysconf`(3POSIX)

NAME

mq_receive – receive a message from a message queue

SYNOPSIS

#include <mqueue.h>
ssize_t **mq_receive**(mqd_t *mqdes*, char **msg_ptr*, size_t *msg_len*, unsigned int **msg_prio*);

FEATURES

POSIX_MQ

DESCRIPTION

The *mq_receive* system call is used to receive the oldest of the highest priority message(s) from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msg_len* argument, is less than the *mq_msgsize* attribute of the message queue, *mq_receive* will fail and return an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by the *msg_ptr* argument.

If the *msg_prio* argument is not NULL, the priority of the selected message is stored at the location referenced by *msg_prio*.

If the specified message queue is empty and O_NONBLOCK is not set in the message queue description associated with *mqdes*, *mq_receive* will block until a message is enqueued on the message queue, or until *mq_receive* is interrupted by a signal, or until a *mq_close(2POSIX)* operation is performed using the same message queue descriptor. If more that one thread is waiting to receive a message when a message arrives at an empty queue, the thread that has been waiting the longest will be selected to receive the message. If the specified message queue is empty and O_NONBLOCK is set in the message queue description associated with *mqdes*, no message is removed from the queue, and *mq_receive* returns an error.

RETURN VALUE

Upon successful completion, *mq_receive* returns the length of the selected message in bytes and the message is removed from the queue. Otherwise, no message will be removed from the queue, *mq_receive* will return -1, and set *errno* to indicate one of the following error conditions.

ERRORS

[ENOSYS]	The *mq_receive* function is not supported. The MQ feature [_POSIX_MESSAGE_PASSING] option (see *sysconf(3POSIX)*) is not configured.
[EBADF]	The *mqdes* argument is not a valid message queue descriptor of a queue open for reading.
[EBADF]	*mq_receive* is blocked waiting for available messages and a *mq_close(2POSIX)* operation was performed with the same message queue descriptor.
[EAGAIN]	The O_NONBLOCK flag is set in the message queue description associated with *mqdes* and the message queue specified is empty.

[EFAULT]	*msg_ptr* or *msg_prio* points outside the allocated address space of the c_actor.
[EMSGSIZE]	The message buffer size specified, *msg_len*, is less than the message size attribute of the message queue.
[EINTR]	A signal interrupted the call to *mq_receive*.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mq_send`(2POSIX), `mq_setattr`(2POSIX), `mq_close`(2POSIX), `sysconf`(3POSIX)

NAME

mq_send – send a message to a message queue

SYNOPSIS

#include <mqueue.h>
int **mq_send**(mqd_t *mqdes*, const char **msg_ptr*, size_t *msg_len*, unsigned int *msg_prio*);

FEATURES

POSIX_MQ

DESCRIPTION

The *mq_send* system call adds the message pointed to by the argument *msg_ptr* to the message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message in bytes. The value of *msg_len* must be less than or equal to the *mq_msgsize* attribute of the message queue or *mq_send* will fail.

If the specified message queue is not full, *mq_send* will insert the message into the message queue at the position indicated by the *msg_prio* argument. A message with a larger numeric value of *msg_prio* is inserted before messages with lower values of *msg_prio*. A message will be inserted after other messages in the queue, if any, with equal *msg_prio* values. The value of *msg_prio* must be less than or equal to MQ_PRIO_MAX, as returned by *sysconf(3POSIX)*.

If the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, *mq_send* will block until space becomes available to enqueue the message, or until *mq_send* is interrupted by a signal or a *mq_close(2POSIX)* operation is performed using the same message queue descriptor. If more than one thread is waiting to send when space becomes available in the message queue, the thread that has been waiting the longest will be unblocked to send its message. If the specified message queue is full and O_NONBLOCK is set in the message queue description associated with *mqdes*, the message is not queued, and *mq_send* returns an error.

RETURN VALUE

Upon successful completion, *mq_send* returns a value of 0. Otherwise, no message will be enqueued, *mq_send* will return -1, and set *errno* to indicate one of the following error conditions.

ERRORS

[ENOSYS]	The *mq_send* function is not supported. The MQ feature [_POSIX_MESSAGE_PASSING] option (see *sysconf(3POSIX)*) is not configured.
[EBADF]	The *mqdes* argument is not a valid message queue descriptor of a queue open for writing.
[EBADF]	*mq_send* is blocked waiting for available space to enqueue the message and a *mq_close(2POSIX)* operation was performed using the same message queue descriptor.
[EINVAL]	The value of *msg_prio* was outside the valid range.

[EAGAIN]	The O_NONBLOCK flag is set in the message queue description associated with *mqdes* and the message queue specified is full.
[EFAULT]	*msg_ptr* points outside the allocated address space of the c_actor.
[EMSGSIZE]	The message length specified, *msg_len*, exceeds the message size attribute of the message queue.
[EINTR]	A signal interrupted the call to *mq_send*.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mq_receive`(2POSIX), `mq_setattr`(2POSIX), `mq_close`(2POSIX), `sysconf`(3POSIX)

NAME

mq_setattr – set message queue attributes

SYNOPSIS

#include <mqueue.h>
int **mq_setattr**(mqd_t *mqdes*, const struct mq_attr **mqstat*, struct mq_attr **omqstat*);

FEATURES

POSIX_MQ

DESCRIPTION

The *mq_setattr* system call is used to set attributes associated with the message queue specified by *mqdes* to the values specified by *mqstat*.

The message queue attributes corresponding to the following members defined in the *mq_attr* structure (defined in *mq_getattr(2POSIX))*, are set to the values specified upon successful completion of *mq_setattr*:

mq_flags The value of this member is the bitwise logical OR of zero and MQ_NONBLOCK.

The values of the *mq_maxmsg*, *mq_msgsize*, *mq_curmsgs*, *mq_sendwait*, and *mq_rcvwait* members of the *mq_attr* structure are ignored by *mq_setattr*.

If *omqstat* is non-NULL, *mq_setattr* stores, in the location referenced by *omqstat*, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to *mq_getattr()* at that point.

RETURN VALUE

Upon successful completion, *mq_setattr* returns 0 and the attributes of the message queue will have been changed as specified. Otherwise, the message queue attributes will not be changed, *mq_setattr* will return -1, and set *errno* to indicate the error condition.

ERRORS

[ENOSYS] The function *mq_setattr* is not supported.

The MQ feature [_POSIX_MESSAGE_PASSING] option (see *sysconf(3POSIX)*) is not configured.

[EBADF] The *mqdes* argument is not a valid message queue descriptor.

[EFAULT] *mqstat* or *mqostat* points outside the allocated address space of the c_actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mq_open`(2POSIX), `mq_send`(2POSIX), `mq_getattr`(2POSIX)

NAME

mq_unlink – unlink a message queue

SYNOPSIS

#include <mqueue.h>
int **mq_unlink**(const char *_name_);

FEATURES

POSIX_MQ

DESCRIPTION

The *mq_unlink* system call removes the message queue named by the pathname *name*. After a successful call to *mq_unlink* with *name*, a call to *mq_open* with *name* will fail if the flag O_CREAT is not set in *flags*. If one or more c_actors have the message queue open when *mq_unlink* is called, destruction of the message queue will be postponed until all references to the message queue have been closed. However, the *mq_unlink* call is not blocked and returns immediately.

RETURN VALUE

Upon successful completion, *mq_unlink* returns a value of 0. Otherwise, the named message queue will not be changed by this call, it will return -1, and set *errno* to indicate one of the following error conditions.

ERRORS

[ENOSYS]	The *mq_unlink* function is not supported. The MQ feature [_POSIX_MESSAGE_PASSING] option (see *sysconf(3POSIX)*) is not configured.
[ENAMETOOLONG]	The *name* string exceeds MQ_PATHMAX as returned by *sysconf(3POSIX)*.
[ENOENT]	The message queue referenced by *name* does not exist.
[EFAULT]	*name* points outside the allocated address space of the c_actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mq_open`(2POSIX), `mq_close`(2POSIX), `sysconf`(3POSIX)

NAME

munmap – unmap a previously mapped address

SYNOPSIS

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

FEATURES

POSIX_SHM

DESCRIPTION

The `munmap()` system call removes mappings for any entire pages containing part of the c_actor's address space, starting at *addr* and continuing for *len* bytes. Further references to these pages will result in a segmentation fault. If there are no mappings in the specified address range, `munmap()` will have no effect.

The implementation requires that *addr* be a multiple of the page size. If *len* is not a multiple of the page size, the system rounds it up to the next page boundary.

If a mapping to be removed was private, any modifications made in this address range will be discarded.

RETURN VALUES

Upon successful completion, `munmap()` returns a value of `0`; otherwise, it will return a value of `-1` and set `errno` to indicate the error condition.

ERRORS

[ENOSYS] The `munmap()` function is not supported.

The POSIX_SHM feature is not configured.

[EINVAL] Some of the addresses in the range starting at *addr* and continuing for *len* bytes are outside the range allowed for the address space of a c_actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mmap`(2POSIX)

NAME

nfssvc – NFS services

SYNOPSIS

```
#include <unistd.h>
#include <nfs/nfs.h>
int nfssvc(int flags, void *argstructp);
```

FEATURES

NFS_SERVER

DESCRIPTION

The `nfssvc()` function is used by the NFS daemons to pass information into and out of the kernel, as well as to enter the kernel as a server daemon. The *flags* argument consists of several bits that show what action is to be taken once in the kernel, The *argstructp* points to one of three structures depending on which bits are set in *flags*.

On the client side, `nfsd`(1M) calls `nfssvc()` with the *flags* argument set to `NFSSVC_BIOD` and *argstructp* set to `NULL` to enter the kernel as a block I/O server daemon. For `NQNFS`, `mount_nfs`(1M) calls `nfssvc()` with the `NFSSVC_MNTD` flag, optionally or'd with the flags `NFSSVC_GOTAUTH` and `NFSSVC_AUTHINFAIL` along with a pointer to the following strucure:

```
struct nfsd_cargs {
        char            *ncd_dirp;      /* Mount dir path */
        uid_t           ncd_authuid;    /* Effective uid */
        int             ncd_authtype;   /* Type of authenticator */
        int             ncd_authlen;    /* Length of authenticator string */
        u_char          *ncd_authstr;   /* Authenticator string */
        int             ncd_verflen;    /* and the verifier */
        u_char          *ncd_verfstr;
        NFSKERBKEY_T    ncd_key;        /* Session key */
};
```

The initial call only has the `NFSSVC_MNTD` flag set to specify service for the mount point. If the mount point is using Kerberos, the `mount_nfs`(1M) daemon will return from *nfssvc* with `errno` set to `ENEEDAUTH` whenever the client side requires an "rcmd" authentication ticket for the user.

The `mount_nfs`(1M) function will attempt to get the Kerberos ticket, and if successful, will call `nfssvc()` with the `NFSSVC_MNTD` and `NFSSVC_GOTAUTH` flags after putting the ticket information into the `ncd_authstr` field and setting the `ncd_authlen` and `ncd_authtype` fields of the `nfsd_cargs` structure. If `mount_nfs`(1M) failed to get the ticket, `nfssvc()` will be called with the flags `NFSSVC_MNTD`, `NFSSVC_GOTAUTH` and `NFSSVC_AUTHINFAIL` to denote a failed authentication attempt. See RESTRICTIONS for Kerberos filesystems with ChorusOS.

On the server side, `nfssvc()` is called with the flag `NFSSVC_NFSD` and a pointer to the following structure:

```
struct nfsd_srvargs {
        struct nfsd     *nsd_nfsd;      /* Pointer to in kernel nfsd struct */
        uid_t           nsd_uid;        /* Effective uid mapped to cred */
```

```
        u_long          nsd_haddr;      /* Ip address of client */
        struct ucred    nsd_cr;         /* Cred. uid maps to */
        int             nsd_authlen;    /* Length of auth string (ret) */
        u_char          *nsd_authstr;   /* Auth string (ret) */
        int             nsd_verflen;    /* and the verfier */
        u_char          *nsd_verfstr;
        struct timeval  nsd_timestamp;  /* timestamp from verifier */
        u_long          nsd_ttl;        /* credential ttl (sec) */
        NFSKERBKEY_T    nsd_key;        /* Session key */
};
```

This allows it to enter the kernel as an nfsd(1M) daemon. Whenever an nfsd(1M) daemon receives a Kerberos authentication ticket, it will return from nfssvc( ) with errno set to ENEEDAUTH. The nfsd(1M) will attempt to authenticate the ticket and generate a set of credentials on the server for the "user id" specified in the nsd_uid field. This is done by first authenticating the Kerberos ticket and then mapping the Kerberos principal to a local name and getting a set of credentials for that user via getpwnam( ) and getgrouplist( ). If successful, the nfsd(1M) will call nfssvc( ) with the NFSSVC_NFSD and NFSSVC_AUTHIN flags set to pass the credential mapping in nsd_crnto to the kernel to be cached on the server socket for that client. If the authentication failed, nfsd(1M) calls nfssvc( ) with the flags NFSSVC_NFSD and NFSSVC_AUTHINFAIL to denote an authentication failure. See RESTRICTIONS about authentication on top of ChorusOS.

The master nfsd(1M) server daemon calls *nfssvc* with the flag NFSSVC_ADDSOCK and a pointer to the following structure:

```
struct nfsd_args {
    int      sock;    /* Socket to serve */
    caddr_t  name;    /* Client address for connection based sockets */
    int      namelen; /* Length of name */
};
```

This allows it to pass a server side NFS socket into the kernel for servicing by the nfsd(1M) daemons.

RETURN VALUES

Under normal circumstances, *nfssvc* does not return. If the server is terminated by a signal, a value of 0 is returned. Otherwise, a value of -1 is returned and the global variable errno is set to indicate one of the following error conditions.

ERRORS

[ENEEDAUTH]	This special error value is used for authentication support, particularly Kerberos, as explained above.
[EPERM]	The caller is not the super-user.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `mount_nfs`(1M), `nfsd`(1M)

HISTORY The `nfssvc()` function first appeared in 4.4BSD.

BUGS The `nfssvc()` system call is designed specifically for the NFS support daemons and as such is specific to their requirements. It should return values to indicate the need for authentication support, as `ENEEDAUTH` is not really an error. Several fields of the argument structures are assumed to be valid and sometimes to be unchanged from a previous call, `nfssvc()` should therefore be used with extreme care.

RESTRICTIONS FOR ChorusOS The Kerberos filesystem is not supported.

Authentication on top of ChorusOS never calls `getpwnam()` or `getgrouplist()` because `/etc/passwd` and `/etc/group` files are not supported.

NAME

open – open for reading or writing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int oflag, ... /*mode_t mode */);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *path* parameter points to a path name naming a file. The *open* function opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. The *oflag* values are constructed by OR-ing flags from the following list (only one of the first three flags below may be used):

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only.
`O_RDWR`	Open for reading and writing.
`O_NONBLOCK`	This flag may affect subsequent reads and writes (see `read`(2POSIX) and `write`(2POSIX)). When opening a block special or character special file that supports nonblocking, this flags has the following effect: ■ If O_NONBLOCK is set, the open will return without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific. ■ If O_NONBLOCK is not set, the open will block until the device is ready or available.
`O_APPEND`	If set, the file pointer will be set to the end of the file prior to each write.
`O_CREAT`	If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the user ID of the c_actor, the group ID of the file is set to the group ID of the c_actor, and the low-order 12 bits of the file mode are set to the value of *mode*.
`O_TRUNC`	If the file exists, its length is truncated to 0 and the mode and owner are unchanged.
`O_EXCL`	If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *afexec (2K)* system calls. See *fcntl*(2POSIX).

RETURN VALUE

Upon successful completion, *open* returns the open file descriptor; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	O_CREAT is not set and the named file does not exist.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	*oflag* permission is denied for the named file.
[EISDIR]	The named file is a directory and *oflag* is write or read / write.
[EROFS]	The named file resides on a read-only file system and *oflag* is write or read / write.
[EMFILE]	`OPEN_MAX` file descriptors are currently open.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[EFAULT]	In user mode, *path* points outside the allocated address space of the c_actor. In supervisor mode, this is not detected, and the state of the target is unknown.
[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
[EINTR]	The calling thread has been aborted during the *open* system call.
[ENFILE]	The system file table is full.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters or the length of *path* exceeds `PATH_MAX` characters.
[ELOOP]	Too many symbolic links or symbolic ports were encountered during analysis of *path*.
[EIO]	An I/O error occurred while reading from or writing to the file system.

[ENOTIMPLEMENTED] One or more flags in *oflag* are not currently supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`close`(2POSIX), `dup`(2POSIX), `fcntl`(2POSIX), `lseek`(2POSIX), `read`(2POSIX), `write`(2POSIX), `stat`(2POSIX)

NAME

pipe – create descriptor a pair for interprocess communication

SYNOPSIS

#include <unistd.h>
int **pipe**(int *fildes*[2]);

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *pipe* function creates a *pipe*, which is an object allowing unidirectional data flow, and allocates a pair of file descriptors. The first descriptor connects to the `read` end of the pipe, and the second connects to the `write` end, so that data written to *fildes[1]* appears on (in other words, it can be read from) *fildes[0]* . This allows the output of one program to be sent to another program: the source's standard output is set up to be the write end of the pipe, and the sink's standard input is set up to be the read end of the pipe. The pipe itself persists until all its associated descriptors are closed.

A pipe whose read or write end has been closed is considered *widowed* . Writing to this type of pipe causes the writing process to receive a `SIGPIPE` signal (see RESTRICTIONS for ChorusOS). Widowing a pipe is the only way to deliver end-of-file to a reader:after the reader has consumed any buffered data; reading a widowed pipe returns a zero count.

Pipes are a special case of the *socketpair(2POSIX)* call and are implemented as such in the system.

RETURN VALUES

Upon successful creation of the pipe, `0` is returned. Otherwise, a value of `-1` is returned and the *errno* variable is set to indicate one of the following error conditions.

ERRORS

[EMFILE]	Too many descriptors are active.
[ENFILE]	The system file table is full.
[EFAULT]	The *fildes* buffer is in an invalid area of the process's address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`read`(2POSIX), `write`(2POSIX), `socketpair`(2POSIX)

HISTORY

This function call appeared in Version 6 AT&T UNIX.

RESTRICTIONS FOR ChorusOS

Signals are not handled with pipes on top of ChorusOS.

NAME

read, readv – read input

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
ssize_t read(int d, void * buf, size_t nbytes);

ssize_t readv(int d, const struct iovec * iov, int iovcnt);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS, POSIX_SOCKETS

DESCRIPTION

`read()` attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*. `readv()` performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the *iov* array: `iov[0]`, `iov[1]`, ..., `iov[iovcnt–1]`.

On objects capable of seeking, `read()` starts at a position given by the pointer associated with *d*. See `lseek`(2POSIX). Upon return from `read()`, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with this type of object is undefined.

PARAMETERS

`read()` takes the following parameters:

d	Descriptor of the object from which to read. This is a file descriptor from an `accept`(2POSIX), `dup`(2POSIX), `open`(2POSIX), `shm_open`(2POSIX) or `socket`(2POSIX) call.
buf	Buffer into which data is read.
nbytes	Number of bytes of data to read.

`readv()` takes the following parameters:

d	Descriptor of the object from which to read. This is a file descriptor from an `accept`(2POSIX), `dup`(2POSIX), `open`(2POSIX), `shm_open`(2POSIX) or `socket`(2POSIX) call.
iov	Array of buffers into which data is read.
iovcnt	Number of buffers into which data is read.

For `readv()`, the `iovec` structure is defined as:

```
struct iovec {
        char *iov_base; /* base address */
        size_t iov_len; /* length */
};
```

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. `readv( )` always fills an area completely before proceeding to the next one.

RETURN VALUES

Upon successful completion, `read( )` and `readv( )` return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file that contains that many bytes before the end-of-file, but in no other case. Upon end-of-file, `0` is returned. `read( )` and `readv( )` also return `0` if a non-blocking read is attempted on a socket that is no longer open. Otherwise, `–1` is returned, and the global variable `errno` is set to indicate the error.

ERRORS

`read( )` and `readv( )` succeed unless:

`[EAGAIN]`	The file was marked for non-blocking I/O , and no data were ready to be extracted..
`[EBADF]`	*d* is not a valid file or socket descriptor open for reading.
`[EFAULT]`	In user mode, *buf* points outside the allocated address space. In supervisor mode, this is not detected and the target may behave unpredictably.
`[EINTR]`	A read from a slow device was interrupted by the delivery of a signal before any data arrived.
`[EINVAL]`	The pointer associated with *d* was negative.
`[EIO]`	An I/O error occurred while reading from the file system.

In addition, `readv( )` may return one of the following errors:

`[EFAULT]`	Part of *iov* points outside the allocated address space.
`[EINVAL]`	*iovcnt* was less than or equal to `0` , or greater than `16` .
`[EINVAL]`	One of the *iov_len* values in the *iov* array was negative.
`[EINVAL]`	The sum of the *iov_len* values in the *iov* array overflowed a 32–bit integer.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dup`(2POSIX) , `fcntl`(2POSIX) , `open`(2POSIX) , `pipe`(2POSIX) , `select`(2POSIX) , `socket`(2POSIX) , `socketpair`(2POSIX)

NAME

readlink – read value of a symbolic link

SYNOPSIS

#include <unistd.h>
int **readlink**(const char **path*, char **buf*, int *bufsiz*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *readlink* system call places the contents of the symbolic link *path* in the buffer *buf*, which has the size *bufsiz*. The *readlink* system call does not append a *NULL* character to *buf*.

RETURN VALUES

If successful, the call returns the count of characters placed in the buffer, otherwise, it returns a value of –1 and sets *errno* to indicate one of the following error conditions.

ERRORS

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.
[ENOENT]	The file referred to by path does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EINVAL]	The file named is not a symbolic link.
[EIO]	An I/O error occurred while reading from the file system.
[EFAULT]	In user mode, *buf* extends outside the process' allocated address space. In supervisor mode, this is not detected and the state of the target is unknown.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`stat`(2POSIX), `lstat`(2POSIX)

HISTORY

The function call appeared in 4.2 BSD.

NAME

read, readv – read input

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
ssize_t read(int d, void * buf, size_t nbytes);

ssize_t readv(int d, const struct iovec * iov, int iovcnt);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS, POSIX_SOCKETS

DESCRIPTION

`read()` attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*. `readv()` performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the *iov* array: `iov[0]`, `iov[1]`, ..., `iov[iovcnt–1]`.

On objects capable of seeking, `read()` starts at a position given by the pointer associated with *d*. See `lseek`(2POSIX). Upon return from `read()`, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with this type of object is undefined.

PARAMETERS

`read()` takes the following parameters:

d	Descriptor of the object from which to read. This is a file descriptor from an `accept`(2POSIX), `dup`(2POSIX), `open`(2POSIX), `shm_open`(2POSIX) or `socket`(2POSIX) call.
buf	Buffer into which data is read.
nbytes	Number of bytes of data to read.

`readv()` takes the following parameters:

d	Descriptor of the object from which to read. This is a file descriptor from an `accept`(2POSIX), `dup`(2POSIX), `open`(2POSIX), `shm_open`(2POSIX) or `socket`(2POSIX) call.
iov	Array of buffers into which data is read.
iovcnt	Number of buffers into which data is read.

For `readv()`, the `iovec` structure is defined as:

```
struct iovec {
        char *iov_base; /* base address */
        size_t iov_len; /* length */
};
```

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. `readv()` always fills an area completely before proceeding to the next one.

RETURN VALUES

Upon successful completion, `read()` and `readv()` return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file that contains that many bytes before the end-of-file, but in no other case. Upon end-of-file, `0` is returned. `read()` and `readv()` also return `0` if a non-blocking read is attempted on a socket that is no longer open. Otherwise, `–1` is returned, and the global variable `errno` is set to indicate the error.

ERRORS

`read()` and `readv()` succeed unless:

`[EAGAIN]`	The file was marked for non-blocking I/O , and no data were ready to be extracted..
`[EBADF]`	*d* is not a valid file or socket descriptor open for reading.
`[EFAULT]`	In user mode, *buf* points outside the allocated address space. In supervisor mode, this is not detected and the target may behave unpredictably.
`[EINTR]`	A read from a slow device was interrupted by the delivery of a signal before any data arrived.
`[EINVAL]`	The pointer associated with *d* was negative.
`[EIO]`	An I/O error occurred while reading from the file system.

In addition, `readv()` may return one of the following errors:

`[EFAULT]`	Part of *iov* points outside the allocated address space.
`[EINVAL]`	*iovcnt* was less than or equal to `0` , or greater than `16` .
`[EINVAL]`	One of the *iov_len* values in the *iov* array was negative.
`[EINVAL]`	The sum of the *iov_len* values in the *iov* array overflowed a 32–bit integer.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dup`(2POSIX) , `fcntl`(2POSIX) , `open`(2POSIX) , `pipe`(2POSIX) , `select`(2POSIX) , `socket`(2POSIX) , `socketpair`(2POSIX)

NAME

recv, recvfrom, recvmsg – receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
int recv(int s, char * buf, int len, int flags);

int recvfrom(int s, char * buf, int len, int flags, struct sockaddr * from, int * fromlen);

int recvmsg(int s, struct msghdr * msg, int flags);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *recv* , *recvfrom* , and *recvmsg* system calls are used to receive messages from a socket.

The *recv* call is normally used only on a *connected* socket (see *connect* (2POSIX)), while *recvfrom* and *recvmsg* may be used to receive data on a socket whether or not it is connected.

If *from* is non-zero, the source address of the message is supplied. The *fromlen* field is a value-result parameter, initialized to the size of the buffer associated with *from* , and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the buffer supplied, excess bytes may be discarded depending on the type of socket the message is received from (see *socket* (2POSIX)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *ioctl* (2POSIX)) in which case –1 is returned with *errno* set to EWOULDBLOCK.

The `select` (2POSIX) call may be used to determine when more data arrives.

The *flags* argument to a recv call is formed by *or* 'ing one or more of the values,

```
#define MSG_OOB         0x1     /* process out-of-band data */
#define MSG_PEEK        0x2     /* peek at incoming message */
```

The *recvmsg* call uses a *msghdr* structure to minimize the number of parameters directly supplied. This structure has the following form, as defined in <*sys/socket.h*> :

```
struct msghdr {
     caddr_t     msg_name;           /* optional address */
     int         msg_namelen;        /* size of address */
     struct      iovec *msg_iov;     /* scatter/gather array */
     int         msg_iovlen;         /* # elements in msg_iov */
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *recvmsg* function scatters the input data into the *msg_iovlen* buffers specified by the members of the *msg_iov* array: msg_iov[0], msg_iov[1], ..., msg_iov[msg_iovcnt−1].

The *iovec* structure is defined as follows:

```
struct iovec {
     caddr_t     iov_base;
     int         iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The *recvmsg* function will always fill an area completely before proceeding to the next.

RETURN VALUE

Upon successful completion, these calls return the number of bytes received; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[EBADF]	The *s* argument is an invalid descriptor.
[ENOTSOCK]	The *s* argument is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EINTR]	The calling thread has been aborted before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the c_actor's address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsockopt`(2POSIX) , `read`(2POSIX) , `select`(2POSIX) , `send`(2POSIX) , `socket`(2POSIX)

NAME

recv, recvfrom, recvmsg – receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
int recv(int s, char * buf, int len, int flags);

int recvfrom(int s, char * buf, int len, int flags, struct sockaddr * from, int * fromlen);

int recvmsg(int s, struct msghdr * msg, int flags);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *recv* , *recvfrom* , and *recvmsg* system calls are used to receive messages from a socket.

The *recv* call is normally used only on a *connected* socket (see *connect* (2POSIX)), while *recvfrom* and *recvmsg* may be used to receive data on a socket whether or not it is connected.

If *from* is non-zero, the source address of the message is supplied. The *fromlen* field is a value-result parameter, initialized to the size of the buffer associated with *from* , and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the buffer supplied, excess bytes may be discarded depending on the type of socket the message is received from (see *socket* (2POSIX)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *ioctl* (2POSIX)) in which case –1 is returned with *errno* set to EWOULDBLOCK.

The `select` (2POSIX) call may be used to determine when more data arrives.

The *flags* argument to a recv call is formed by *or* 'ing one or more of the values,

```
#define MSG_OOB         0x1     /* process out-of-band data */
#define MSG_PEEK        0x2     /* peek at incoming message */
```

The *recvmsg* call uses a *msghdr* structure to minimize the number of parameters directly supplied. This structure has the following form, as defined in <*sys/socket.h*> :

```
struct msghdr {
     caddr_t     msg_name;           /* optional address */
     int         msg_namelen;        /* size of address */
     struct      iovec *msg_iov;     /* scatter/gather array */
     int         msg_iovlen;         /* # elements in msg_iov */
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *recvmsg* function scatters the input data into the *msg_iovlen* buffers specified by the members of the *msg_iov* array: msg_iov[0], msg_iov[1], ..., msg_iov[msg_iovcnt−1].

The *iovec* structure is defined as follows:

```
struct iovec {
     caddr_t     iov_base;
     int         iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The *recvmsg* function will always fill an area completely before proceeding to the next.

RETURN VALUE

Upon successful completion, these calls return the number of bytes received; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[EBADF]	The *s* argument is an invalid descriptor.
[ENOTSOCK]	The *s* argument is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EINTR]	The calling thread has been aborted before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the c_actor's address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsockopt`(2POSIX) , `read`(2POSIX) , `select`(2POSIX) , `send`(2POSIX) , `socket`(2POSIX)

NAME

recv, recvfrom, recvmsg – receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
int recv(int s, char * buf, int len, int flags);

int recvfrom(int s, char * buf, int len, int flags, struct sockaddr * from, int * fromlen);

int recvmsg(int s, struct msghdr * msg, int flags);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *recv* , *recvfrom* , and *recvmsg* system calls are used to receive messages from a socket.

The *recv* call is normally used only on a *connected* socket (see *connect* (2POSIX)), while *recvfrom* and *recvmsg* may be used to receive data on a socket whether or not it is connected.

If *from* is non-zero, the source address of the message is supplied. The *fromlen* field is a value-result parameter, initialized to the size of the buffer associated with *from* , and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the buffer supplied, excess bytes may be discarded depending on the type of socket the message is received from (see *socket* (2POSIX)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *ioctl* (2POSIX)) in which case –1 is returned with *errno* set to EWOULDBLOCK.

The `select` (2POSIX) call may be used to determine when more data arrives.

The *flags* argument to a recv call is formed by *or* 'ing one or more of the values,

```
#define MSG_OOB         0x1     /* process out-of-band data */
#define MSG_PEEK        0x2     /* peek at incoming message */
```

The *recvmsg* call uses a *msghdr* structure to minimize the number of parameters directly supplied. This structure has the following form, as defined in <*sys/socket.h*> :

```
struct msghdr {
     caddr_t     msg_name;           /* optional address */
     int         msg_namelen;        /* size of address */
     struct      iovec *msg_iov;     /* scatter/gather array */
     int         msg_iovlen;         /* # elements in msg_iov */
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *recvmsg* function scatters the input data into the *msg_iovlen* buffers specified by the members of the *msg_iov* array: msg_iov[0], msg_iov[1], ..., msg_iov[msg_iovcnt−1].

The *iovec* structure is defined as follows:

```
struct iovec {
     caddr_t     iov_base;
     int         iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The *recvmsg* function will always fill an area completely before proceeding to the next.

RETURN VALUE

Upon successful completion, these calls return the number of bytes received; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[EBADF]	The *s* argument is an invalid descriptor.
[ENOTSOCK]	The *s* argument is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EINTR]	The calling thread has been aborted before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the c_actor's address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsockopt`(2POSIX) , `read`(2POSIX) , `select`(2POSIX) , `send`(2POSIX) , `socket`(2POSIX)

NAME

rename – change the name of a file

SYNOPSIS

#include <stdio.h>
int **rename**(const char **from*, const char **to*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *rename* system call causes the link named *from* to be renamed as *to*. If *to* exists, it is first removed. Both *from* and *to* must be of the same type (that is, both directories or both non-directories), and must reside on the same file system.

The *rename* system call guarantees that an instance of *to* will always exist, even if the system should crash in the middle of the operation.

If the final component of *from* is a symbolic link, the symbolic link is renamed, not the file or directory to which it points.

CAVEAT

The system can deadlock if a loop in the file system graph is present. This loop takes the form of an entry in directory *a*,for example, *a/foo*, being a hard link to directory *b*, and an entry in directory *b*, for example,*b/bar*, being a hard link to directory *a*. When this type of loop exists and two separate processes attempt to perform *rename a/foo b/bar* and *rename b/bar a/foo*, respectively, the system may deadlock while attempting to lock both directories for modification. Hard links to directories should be replaced by symbolic links by the system administrator.

RETURN VALUES

If successful, *rename* returns a value of 0, otherwise a value of –1 is returned, and the global variable `errno` is set to indicate one of the following error conditions.

ERRORS

[ENAMETOOLONG]	A component of either pathname exceeded 255 characters, or the entire length of either pathname exceeded 1023 characters.
[ENOENT]	A component of the *from* path does not exist, or a path prefix of *to* does not exist.
[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing to a directory with a mode that denies write permission.
[EPERM]	The directory containing *from* is marked sticky, and neither the containing directory nor *from* are owned by the effective user ID.
[EPERM]	The file exists, the directory containing *to* is marked sticky, and neither the containing directory nor *to* are owned by the effective user ID.

[ELOOP]	Too many symbolic links were encountered in translating either pathname.
[ENOTDIR]	A component of either path prefix is not a directory.
[ENOTDIR]	*from* is a directory, but *to* is not a directory.
[EISDIR]	*to* is a directory, but *from* is not a directory.
[EXDEV]	The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.
[ENOSPC]	The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.
[EIO]	An I/O error occurred while making or updating a directory entry.
[EROFS]	The requested link requires writing to a directory on a read-only file system.
[EFAULT]	*from* or *to* point outside the process' allocated address space.
[EINVAL]	*from* is a parent directory of *to*, or an attempt has been made to rename.
[EEXIST]	*To* is a directory and is not empty.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX)

NAME

rmdir – remove a directory file

SYNOPSIS

#include <unistd.h>
int **rmdir**(const char **path*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The `rmdir` system call removes a directory file whose name is given by *path*. The directory must not have any entries other than . (dot) and .. (dot, dot)

RETURN VALUES

If successful, `rmdir` returns a value of 0, otherwie –1 is returned, and *errno* is set to indicate one of the following error conditions..

ERRORS

[ENOTDIR]	A component of the path is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.
[ENOENT]	The directory named does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENOTEMPTY]	The named directory contains files other than . and .. in it.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EPERM]	The directory containing the directory to be removed is marked sticky, and neither the containing directory nor the directory to be removed are owned by the effective user ID.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EIO]	An I/O error occurred while deleting the directory entry or deallocating the inode.
[EROFS]	The directory entry to be removed resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

mkdir(2POSIX), unlink(2POSIX)

HISTORY

This function call appeared in 4.2 BSD.

NAME

select – synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval
*timeout);

FD_SET(int fd, fd_set *fdset);

FD_CLR(int fd, fd_set *fdset);

FD_ISSET(int fd, fd_set *fdset);

FD_ZERO(fd_set *fdset);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The `select` system call examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to verify whether any of their descriptors are ready for reading, writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; in other words, the descriptors from 0 to *nfds*-1 inclusive in the descriptor sets are examined. On return, `select` replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating the descriptor sets:

FD_ZERO(&fdset)	Initializes a descriptor set *fdset* to the null set.
FD_SET(fd, &fdset)	Includes a particular descriptor *fd* in *fdset*.
FD_CLR(fd, &fdset)	Removes *fd* from *fdset*.
FD_ISSET(fd, &fdset)	Is nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is not the NULL pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is the NULL pointer, the select blocks indefinitely. To affect a poll, the *timeout* argument should be non NULL, pointing to a zero-valued timeval structure.

The *readfds*, *writefds*, and *exceptfds* parameters may be given as NULL pointers if no descriptors are required.

RETURN VALUE

The `select` system call returns the number of ready descriptors contained in the descriptor sets. If the time limit expires `select` returns 0. If `select` returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

An error return from `select` indicates:

[EBADF] One of the descriptor sets specified an invalid descriptor.

[EINTR] The calling thread has been aborted before the time limit expired and before any of the events selected occurred.

[EINVAL] The time limit specified is invalid. One of its components is negative or too large.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`accept`(2POSIX), `connect`(2POSIX), `read`(2POSIX), `recv`(2POSIX), `send`(2POSIX), `write`(2POSIX)

LIMITATIONS

It is possible that the timeout value will be modified by the `select` system call.

NAME

send, sendto, sendmsg – send a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
int send(int s, const char * msg, int len, int flags);

int sendto(int s, const char * msg, int len, int flags, const struct sockaddr * to, int tolen);

int sendmsg(int s, const struct msghdr * msg, int flags);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *send* , *sendto* , and *sendmsg* system calls are used to transmit a message to another socket. The *send* call may only be used when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* , with *tolen* specifying its size. The length of the message is given by *len* . If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send* . Return values of –1 indicate locally detected errors.

If no message space is available at the socket to hold the message to be transmitted, *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The `select` (2POSIX) call may be used to determine when it is possible to send more data.

The *flags* parameter may include one or more of the following:

```
#define MSG_OOB         0x1   /* process out-of-band data */
#define MSG_DONTROUTE   0x4   /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send “out-of-band” data on sockets that support this concept (for example, SOCK_STREAM);the underlying rotocol must also support “out-of-band” data. The MSG_DONTROUTE option is usually used only by diagnostic or routing programs.

See *recv* (2POSIX) for a description of the *msghdr* structure.

RETURN VALUE

Upon successful completion, these calls return the number of bytes sent; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The *s* argument is not a socket.

[EFAULT]	An invalid user address space was specified for a parameter.
[EMSGSIZE]	The socket requires messages to be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the operation requested would block.
[ENOBUFS]	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
[ENOBUFS]	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsockopt`(2POSIX) , `recv`(2POSIX) , `select`(2POSIX) , `socket`(2POSIX) , `write`(2POSIX)

NAME

send, sendto, sendmsg – send a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
int send(int s, const char * msg, int len, int flags);

int sendto(int s, const char * msg, int len, int flags, const struct sockaddr * to, int tolen);

int sendmsg(int s, const struct msghdr * msg, int flags);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *send* , *sendto* , and *sendmsg* system calls are used to transmit a message to another socket. The *send* call may only be used when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* , with *tolen* specifying its size. The length of the message is given by *len* . If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send* . Return values of –1 indicate locally detected errors.

If no message space is available at the socket to hold the message to be transmitted, *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The `select` (2POSIX) call may be used to determine when it is possible to send more data.

The *flags* parameter may include one or more of the following:

```
#define MSG_OOB         0x1   /* process out-of-band data */
#define MSG_DONTROUTE   0x4   /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send “out-of-band” data on sockets that support this concept (for example, SOCK_STREAM);the underlying rotocol must also support “out-of-band” data. The MSG_DONTROUTE option is usually used only by diagnostic or routing programs.

See *recv* (2POSIX) for a description of the *msghdr* structure.

RETURN VALUE

Upon successful completion, these calls return the number of bytes sent; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The *s* argument is not a socket.

[EFAULT]	An invalid user address space was specified for a parameter.
[EMSGSIZE]	The socket requires messages to be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the operation requested would block.
[ENOBUFS]	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
[ENOBUFS]	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsockopt`(2POSIX) , `recv`(2POSIX) , `select`(2POSIX) , `socket`(2POSIX) , `write`(2POSIX)

NAME

send, sendto, sendmsg – send a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
int send(int s, const char * msg, int len, int flags);

int sendto(int s, const char * msg, int len, int flags, const struct sockaddr * to, int tolen);

int sendmsg(int s, const struct msghdr * msg, int flags);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *send* , *sendto* , and *sendmsg* system calls are used to transmit a message to another socket. The *send* call may only be used when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* , with *tolen* specifying its size. The length of the message is given by *len* . If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send* . Return values of –1 indicate locally detected errors.

If no message space is available at the socket to hold the message to be transmitted, *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The `select` (2POSIX) call may be used to determine when it is possible to send more data.

The *flags* parameter may include one or more of the following:

```
#define MSG_OOB         0x1   /* process out-of-band data */
#define MSG_DONTROUTE   0x4   /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send “out-of-band” data on sockets that support this concept (for example, SOCK_STREAM);the underlying rotocol must also support “out-of-band” data. The MSG_DONTROUTE option is usually used only by diagnostic or routing programs.

See *recv* (2POSIX) for a description of the *msghdr* structure.

RETURN VALUE

Upon successful completion, these calls return the number of bytes sent; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The *s* argument is not a socket.

[EFAULT]	An invalid user address space was specified for a parameter.
[EMSGSIZE]	The socket requires messages to be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the operation requested would block.
[ENOBUFS]	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
[ENOBUFS]	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsockopt`(2POSIX) , `recv`(2POSIX) , `select`(2POSIX) , `socket`(2POSIX) , `write`(2POSIX)

NAME

hostname, gethostname, sethostname – get or set the name of the machine

SYNOPSIS

```
#include <unistd.h>
int gethostname(char * hostname, size_t len);

int sethostname(char * hostname, size_t len);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The pair of primitives *gethostname* and *sethostname* are used to get and set the name of the machine, respectively. The value of the *len* field defines the length of the name, and is limited to MAXHOSTNAMELEN (from <sys/param.h>).

Note - Currently, the value of MAXHOSTNAMELEN is 64.

RETURN VALUES

Upon successful completion, these primitives return 0; otherwise -1 is returned, and the global variable *errno* is set to indicate one of the following error conditions.

ERRORS

The *gethostname* primitive fails if:

[EFAULT] *len* is less than the current length of the machine name.

[EFAULT] The *name* or *namelen* gave an invalid address.

The *sethostname* primitive fails if either of the following are true:

[EFAULT] The new length of the machine name is greater than MAXHOSTNAMELEN.

Note - Currently, the value of MAXHOSTNAMELEN is 64

[EPERM] The caller is not the superuser.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getrlimit, setrlimit – control maximum system resource consumption

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/types.h>
int getrlimit(int resource, struct rlimit * rlp);

int setrlimit(int resource, const struct rlimit * rlp);
```

DESCRIPTION

Limits on the consumption of system resources by the current actor may be obtained using the `getrlimit()` call, and set with the `setrlimit()` call.

PARAMETERS

`getrlimit()` and `setrlimit()` take the following *resource* parameters:

`RLIMIT_FSIZE`	The largest file size, in bytes, that can be created.
`RLIMIT_NOFILE`	The maximum number of open files for the process.

EXTENDED DESCRIPTION

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded, an actor is allowed to continue execution until it reaches the hard limit or modifies its resource limit. The `rlimit` structure is used to specify the hard and soft limits on a resource. For example:

```
struct rlimit {
        rlim_t rlim_cur; /* current (soft) limit */
        rlim_t rlim_max; /* maximum value for rlim_cur */
};
```

Only trusted actors may raise the maximum limits. Other actors may only alter `rlim_cur` within the range from `0` to `rlim_max` , or lower `rlim_max` irreversibly.

An “infinite” value for a limit is defined as `RLIM_INFINITY` .

RETURN VALUES

A return value of `0` indicates that the call succeeded in changing or returning the resource limit. A return value of `–1` means that an error occurred, and an error code is stored in the global location `errno` inficating one of the following error conditons.

ERRORS

`getrlimit()` and `setrlimit()` fail if:

`[EFAULT]`	The address specified for *rlp* is invalid.
`[EPERM]`	The limit specified for `setrlimit()` would raise the maximum value, and the caller is not a trusted actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getsockopt, setsockopt – get and set options on sockets

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
int **getsockopt**(int *s*, int *level*, int *optname*, void * *optval*, int * *optlen*);

int **setsockopt**(int *s*, int *level*, int *optname*, const void * *optval*, int *optlen*);

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *getsockopt* and *setsockopt* functions manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost “socket” level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the “socket” level, *level* is specified as SOL_SOCKET. To manipulate options at any other level, the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol mumber of TCP ; see `getprotoent`(3POSIX) .

The *optval* and *optlen* parameters are used to access option values for *setsockopt* . For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt* , *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval* , and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as 0.

The *optname* parameter and any options specified are passed uninterpreted to the appropriate protocol module for interpretation. The include file < *sys/socket.h* > contains definitions for “socket” level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (7P).

Most socket-level options take an *int* parameter for *optval* . For *setsockopt* , the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. The SO_LINGER option uses a *struct linger* parameter, defined in < *sys/socket.h* >, which specifies the desired state of the option and the linger interval (see below). `SO_SNDTIMEO` and `SO_RCVTIMEO` use a `struct timeval` parameter, defined in `<sys/time.h>` .

The following options are recognized at the socket level. Except as noted, each may be examined using *getsockopt* and set using *setsockopt* .

SO_DEBUG	toggle recording of debugging information.
SO_REUSEADDR	toggle local address reuse.
SO_REUSEPORT	enable duplicate address and port bindings.

SO_KEEPALIVE	toggle keep connections alive.
SO_DONTROUTE	toggle routing bypass for outgoing messages.
SO_LINGER	linger on close if data present.
SO_BROADCAST	toggle permission to transmit broadcast messages.
SO_OOBINLINE	toggle reception of out-of-band data in band.
SO_SNDBUF	set buffer size for output.
SO_RCVBUF	set buffer size for input.
SO_SNDLOWAT	set minimum count for output.
SO_RECVLOWAT	set minimum count for input.
SO_SNDTIMEO	set timeout value for output.
SO_RCVTIMEO	set timout value for input.
SO_TYPE	get the type of the socket (get only).
SO_ERROR	get and clear error on the socket (get only).

The SO_DEBUG option enables debugging in the underlying protocol modules. The SO_REUSEADDR option indicates that the rules used in validating addresses supplied in a *bind* (2POSIX) call should allow reuse of local addresses. The SO_REUSEPORT option allows completely duplicate bindings by multiple processes if they all set SO_REUSEPORT before binding the port. This option permits multiple instances of a program, each to receive multicast or broadcast datagrams destined for the bound port. The SO_KEEPALIVE option enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken. The SO_DONTROUTE option indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

The SO_LINGER option controls the action taken when unsent messages are queued on a socket and a *close* (2POSIX) is performed. If the socket confirms reliable delivery of data and SO_LINGER is set, the system will block the c_actor on the *close (2POSIX)* attempt until it is able to transmit the data or until it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close (2POSIX)* is issued, the system will process the close in a manner that allows the c_actor to continue as quickly as possible.

The SO_BROADCAST option requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue

as received; it will then be accessible to *recv (2POSIX)* or `read` *(2POSIX)* calls without the MSG_OOB flag. The SO_SNDBUF and SO_RCVBUF options allow you to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values, which can be accessed through the `sysctl`(1M) MIB variable, `kern.maxsockbuf` .

SO_SNDLOWAT is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted, subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A select(2POSIX) operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for SO_SNDLOWAT is set to a convenient size for network efficiency, often 1024.

SO_RCVLOWAT is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amounts available or the amount requested. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark values or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that which was returned.

SO_SNDTIMEO is an option to set a timeout value for output operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this amount of time, it returns with a partial count or with the error EWOULDBLOCK if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output.

SO_RCVTIMEO is an option to set a timeout value for input operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error EWOULDBLOCK if no data were received.

Finally, SO_TYPE and SO_ERROR are options used only with *setsockopt* . The SO_TYPE option returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. The SO_ERROR option returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets, or any other asynchronous errors.

RETURN VALUE

Upon successful completion, *getsockopt* and *setsockopt* return 0; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[EBADF] The *s* argument is not a valid descriptor.

[ENOTSOCK] The *s* argument is a file, not a socket.

[ENOPROTOOPT] The option is unknown at the level indicated.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysctl`(1M) , `bind`(2POSIX) , `close`(2POSIX) , `ioctl`(2POSIX) , `read`(2POSIX) , `recv`(2POSIX) , `socket`(2POSIX) , `getprotoent`(3POSIX) , `protocols`(4CC)

NAME

gettimeofday, settimeofday – get/set date and time

SYNOPSIS

```
#include <sys/time.h>
int gettimeofday(struct timeval * tp, struct timezone * tzp);

int settimeofday(struct timeval * tp, struct timezone * tzp);
```

DESCRIPTION

The current Greenwich time and the current time zone on the system are obtained using the *gettimeofday* call, and set with the *settimeofday* call. The time is expressed as the number of seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware–dependent, and the time may be updated continuously or in "ticks". If *tp* or *tzp* is NULL, the associated time information will not be returned or set.

The structures pointed to by *tp* and *tzp* are defined in sys/time.h as:

```
struct timeval {
   long   tv_sec;    /* seconds since Jan. 1, 1970 */
   long   tv_usec;   /* and microseconds */
};
struct timezone {
   int   tz_minuteswest;   /* of Greenwich */
   int   tz_dsttime;       /* type of dst correction to apply */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if non zero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

Only the super-user may set the time of day or time zone.

RETURN

If successful, *gettimeofday* returns 0, otherwise it returns –1, and sets *errno* to indicate one of the following error conditions.

ERRORS

[EFAULT]	The address of an argument referenced invalid memory.
[EPERM]	A user other than the super-user attempted to set the time.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

date(1CC) , ctime(3STDC) ,

NAME

shm_open – open a shared memory object

SYNOPSIS

#include <posix/unistd.h>
#include <sys/mman.h>
int **shm_open**(const char **name*, int *oflag [*, mode_t *mode]*);

FEATURES

POSIX_SHM

DESCRIPTION

The *shm_open* system call establishes a connection between a shared memory object and a file descriptor. It creates a shared memory object description that refers to the shared memory object, and a file descriptor that refers to that shared memory object description. The file descriptor is used by other functions to refer to that shared memory object. The *name* argument points to a string naming a shared memory object. The name does not appear in the file system. The *name* argument points to a posix object name. The shared memory object name cannot exceed SHM_PATHMAX characters as returned by *aconf(2K)* or *sysconf(3POSIX)*.

If successful, *shm_open* returns a file descriptor for the shared memory object that is the lowest numbered file descriptor not currently open for that c_actor.

The new file descriptor is set to remain open across *afexec(2K)* system calls. See *fcntl*(2POSIX).

The file status flags and file access modes of the open file description are set according to the value of *oflag*. The *oflag* argument is the bitwise inclusive OR of the following flags defined in header `<fcntl.h>`. Applications should specify one of the following values (access modes) in *oflag*:

O_RDONLY — Open for read access only.

O_RDWR — Open for read or write access.

Any combination of the remaining flags may be specified in the value of *oflag*:

O_CREAT — If the shared memory object exists, this flag will have no effect, except as noted under O_EXCL below. Otherwise, the shared memory object is created. The owner ID of the shared memory object is set to the user ID of the c_actor, the group ID of the shared memory object is set to the group ID of the c_actor. The shared memory object's permission bits will be set to the value of the *mode* argument. When bits in *mode* other than the file permission bits are set, further access to the shared memory object may result in an access error. The *mode* argument does not affect whether the shared memory object is opened for reading,

	writing, or both. The shared memory object will have a size of 0.
O_EXCL	If O_EXCL and O_CREAT are set, *shm_open* will fail if the shared memory object exists. The check for the existence of the shared memory object (and the creation of the object if it does not exist) is atomic with respect to other threads executing *shm_open* naming the same shared memory object with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set this flag is ignored.

When a shared memory object is created, the state of the shared memory object, including all data associated with the shared memory object, persists until the shared memory object is unlinked and all other references are gone. The name and shared memory object state are invalid after a system reboot.

RETURN VALUE

Upon successful completion, *shm_open* returns a non-negative integer representing the lowest numbered unused file descriptor; otherwise, it returns -1 and sets *errno* to indicate the error condition.

ERRORS

[ENOSYS]	The function *shm_open* is not supported. The SHM feature [_POSIX_SHARED_MEMORY_OBJECTS] option (see *aconf(2K)*, *sysconf(3POSIX)*) is not configured.
[EACCES]	The shared memory object exists and the permissions specified by *oflag* are denied, or the shared memory object does not exist and permission to create the shared memory object is denied.
[EEXIST]	O_CREAT and O_EXCL are set and the named shared memory object already exists.
[EINVAL]	The *name* string exceeds _SC_SHM_PATHMAX.
[EINVAL]	The access modes specified by *oflag* are incompatible.
[EMFILE]	Too many shared file descritors are currently in use by this c_actor.
[ENOSPC]	There is insufficient space for the creation of the new shared memory object.

[ENOENT]	O_CREAT is not set and the named shared memory object does not exist.
[EFAULT]	*name* points outside the allocated address space of the c_actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`close`(2POSIX), `dup`(2POSIX), `afexec`(2K), `fcntl`(2POSIX), `shm_unlink`(2POSIX), `mmap`(2POSIX), `ftruncate`(2POSIX), `open`(2POSIX)

NAME

shm_unlink – unlink a shared memory object

SYNOPSIS

```
#include <posix/unistd.h>
#include <sys/mman.h>
int shm_unlink(const char *name);
```

FEATURES

POSIX_SHM

DESCRIPTION

The `shm_unlink()` system call removes the name of the shared memory object named by the string pointed to by *name*. If one or more references to the shared memory object exist when the object is unlinked, the name is removed before *shm_unlink* returns. Removal of the memory object contents will be postponed until all open and map references to the shared memory object have been removed.

RETURN VALUES

Upon successful completion, `shm_unlink()` returns a value of `0`. Otherwise, it returns `-1` and sets `errno` to indicate the error condition. If `-1` is returned, the named shared memory object will not be changed by this call.

ERRORS

[EACCES]	Permission to unlink the shared memory object, *name* is denied.
[ENOSYS]	The `shm_unlink()` function is not supported.
[ENAMETOOLONG]	The *name* string exceeds SHM_PATHMAX as returned by `aconf`(2K) or `sysconf`(3POSIX).
[ENOENT]	The *name* shared memory object does not exist.
[EFAULT]	*name* points outside the allocated address space of the c_actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`close`(2POSIX), `dup`(2POSIX), `fcntl`(2POSIX), `shm_open`(2POSIX)

NAME

shutdown – shut down part of a full-duplex connection

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int shutdown(int s, int how);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The `shutdown` call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, subsequent receives will be disallowed. If *how* is 1, subsequent sends will be disallowed. If *how* is 2, subsequent sends and receives will be disallowed.

RETURN VALUE

Upon successful completion, `shutdown` returns 0; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[EBADF]	*s* is not a valid descriptor.
[ENOTSOCK]	*s* is a file, not a socket.
[ENOTCONN]	The specified socket is not connected.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`connect`(2POSIX), `socket`(2POSIX)

NAME

socket – create an endpoint for communication

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
int **socket**(int *domain*, int *type*, int *protocol*);

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *socket* system call creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file <*sys/socket.h*>. The formats currently understood are:

```
AF_INET (ARPA Internet protocols),
```

The socket has the *type*indicated, which specifies the semantics of communication. The types currently defined are:

```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
```

A SOCK_STREAM type provides sequenced, reliable, two-way, connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). The SOCK_RAW sockets provide access to internal network protocols and interfaces. The SOCK_RAW type, which is available only to the super-user, is not described.

The *protocol* specifies a particular protocol to be used with the socket. However, as the current implementation only supports the SOCK_STREAM, SOCK_DGRAM and SOCK_RAW types within the PF_INET family, *protocol* may be supplied as 0, as only a single protocol exists to support a particular socket type within the PF_INET familly.

Sockets of the type SOCK_STREAM are full-duplex byte streams. A stream socket must be in a *connected* state before any data may be sent or received. A connection to another socket is created with a *connect*(2POSIX) call. Once connected, data may be transferred using the `read`(2POSIX) and `write`(2POSIX) calls or variants of the *send*(2POSIX) and *recv*(2POSIX) calls. When a session has been completed, a *close*(2POSIX) may be performed. Out-of-band data may

also be transmitted as described in *send*(2POSIX) and received as described in *recv*(2POSIX).

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be transmitted successfully within a reasonable length of time, the connection is considered broken. Calls will indicate an error by returning a value of –1, and setting *errno* to ETIMEOUT.. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period (for exmaple, 5 minutes).

The SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2POSIX) calls. Datagrams are generally received using *recvfrom*(2K), which returns the next datagram with its return address.

The operation of sockets is controlled by the socket level *options.* These options are defined in the file <*sys/socket.h*>. The *setsockopt*(2K) and *getsockopt*(2POSIX) functions are used to set and get options, respectively.

RETURN VALUE

Upon successful completion, *socket* returns a non-negative integer that is the descriptor of the new socket; otherwise it returns -1 and sets *errno* to indicate one of the following error conditions:

[EPROTONOSUPPORT]	The protocol type or the specified protocol is not supported within this domain.
[EMFILE]	The c_actor descriptor table is full.
[ENFILE]	The system file table is full.
[EACCESS]	Permission to create a socket of the specified type and / or protocol is denied.
[ENOBUFS]	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`accept`(2POSIX), `bind`(2POSIX), `connect`(2POSIX), `getsockname`(2POSIX), `getsockopt`(2POSIX), `ioctl`(2POSIX), `listen`(2POSIX), `read`(2POSIX), `recv`(2POSIX), `select`(2POSIX), `send`(2POSIX), `shutdown`(2POSIX), `write`(2POSIX)

NAME

socketpair – create a pair of connected sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int socketpair(int d, int type, int protocol, int *sv);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The *socketpair* call creates an unnamed pair of connected sockets in the domain specified by *d*, of the `type` specified, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv[0]* and *sv[1]*. The two sockets are indistinguishable.

DIAGNOSTICS

If successful, *socketpair* returns a value of 0; otherwise it returns a value of –1, and sets *errno* to indicate one of the following error conditions.

ERRORS

[EMFILE]	Too many descriptors are in use by this process.
[ENFILE]	The system file table is full.
[ENOBUFS]	Insufficient buffer space is available. The operation cannot be completed until sufficient resources are freed.
[EAFNOSUPPORT]	The address family specified is not supported on this machine.
[EPROTONOSUPPORT]	The protocol specified is not supported on this machine.
[EOPNOSUPPORT]	The protocol specified does not support creation of socket pairs.
[EFAULT]	The address *sv* does not specify a valid part of the process address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`read`(2POSIX), `write`(2POSIX), `pipe`(2POSIX)

BUGS

This call is only currently implemented for the UNIX domain.

HISTORY This function call appeared in 4.2 BSD.

NAME

stat, lstat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char * path, struct stat * buf);

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

int fstat(int fildes, struct stat * buf);
```

FEATURES

MSDOSFS, NFS_CLIENT

DESCRIPTION

The *path* parameter points to the pathname of a file. Read, write, or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable. The *lstat* function is similar to *stat* except when the named file is a symbolic link, in which case *lstat* returns information about the link, while *stat* returns information about the file's link references. Unlike other filesystem objects, symbolic links do not have an owner, group, access mode, or times. Instead, these attributes are taken from the directory that contains the link. The only attributes returned from an *lstat* that refer to the symbolic link itself are the file type (S_IFLNK), size, blocks, and link count (always 1).

Similarly, *fstat* obtains information about an open file referenced by the file descriptor *fildes* , obtained from a successful *open* (2POSIX), or *dup* (2POSIX) system call.

The *buf* pointer indicates a *stat* structure into which information concerning the file is placed.

The contents of the structure pointed to by *buf* include the following members:

```
mode_t st_mode;          /* File mode */
ino_t st_ino;            /* Inode number */
dev_t st_dev;            /* ID of device containing (special file only) */
                         /* a directory entry for this file */
dev_t st_rdev;           /* ID of device (special files only) */
nlink_t st_nlink;        /* Number of links */
uid_t st_uid;            /* User ID of the file's owner */
gid_t st_gid;            /* Group ID of the file's group */
size_t st_size;          /* File size in bytes */
timestruct_t st_atime;   /* Time of last access */
timestruct_t st_mtime;   /* Time of last data modification */
timestruct_t st_ctime;   /* Time of last file status change */
                         /* Time is measured since 00:00:00 GMT, Jan. 1, 1970 */
```

RETURN VALUE

Upon successful completion, *stat* , *lstat* and *fstat* return 0; otherwise they return -1 and set *errno* to indicate one of the following error conditions:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*buf* or *path* points to an invalid address.
[ENAMETOOLONG]	The length of a component of *name* exceeds NAME_MAX characters, or the length of *name* exceeds PATH_MAX characters.
[ELOOP]	Too many symbolic links or symbolic ports were encountered during analysis of *name* .
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EBADF]	*fildes* is not a valid open file descriptor.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

dup(2POSIX) , open(2POSIX)

NAME

statfs, fstatfs – get file system statistics

SYNOPSIS

```
#include <sys/param.h>
#include <sys/mount.h>
int statfs(const char * path, struct statfs * buf);

int fstatfs(int fd, struct statfs * buf);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *statfs* function call returns information about a mounted file system. The path name of any file in the mounted filesystem is defined by *path* . The *buf* pointer indicates a *statfs* structure defined as follows:

```
typedef     quad     fsid_t;

#define MNAMELEN 90     /* length of buffer for returned name */

struct statfs {
    short   f_type;                 /* type of filesystem (see below) */
    short   f_flags;                /* copy of mount flags */
    long    f_bsize;                /* fundamental file system block size */
    long    f_iosize;               /* optimal transfer block size */
    long    f_blocks;               /* total data blocks in file system */
    long    f_bfree;                /* free blocks in fs */
    long    f_bavail;               /* free blocks avail to non-superuser */
    long    f_files;                /* total file nodes in file system */
    long    f_ffree;                /* free file nodes in fs */
    fsid_t  f_fsid;                 /* file system id */
    long    f_spare[9];             /* spare for later */
    char    f_mntonname[MNAMELEN];   /* mount point */
    char    f_mntfromname[MNAMELEN]; /* mounted filesystem */
};
/*
* File system types.
*/
#define MOUNT_UFS        1     /* Fast Filesystem */
#define MOUNT_NFS        2     /* Sun-compatible Network Filesystem */
#define MOUNT_MFS        3     /* Memory-based Filesystem */
#define MOUNT_MSDOS      4     /* MS/DOS Filesystem */
#define MOUNT_LFS        5     /* Log-based Filesystem */
#define MOUNT_LOFS       6     /* Loopback Filesystem */
#define MOUNT_FDESC      7     /* File Descriptor Filesystem */
#define MOUNT_PORTAL     8     /* Portal Filesystem */
#define MOUNT_NULL       9     /* Minimal Filesystem Layer */
#define MOUNT_UMAP      10     /* Uid/Gid Remapping Filesystem */
#define MOUNT_KERNFS    11     /* Kernel Information Filesystem */
#define MOUNT_PROCFS    12     /* /proc Filesystem */
#define MOUNT_AFS       13     /* Andrew Filesystem */
#define MOUNT_CD9660    14     /* ISO9660 (aka CDROM) Filesystem */
#define MOUNT_UNION     15     /* Union (translucent) Filesystem */
```

Fields that are undefined for a particular file system are set to undefined values. The *fstatfs* system call returns the same information about an open file referenced by the *fd* descriptor.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error condition.

ERRORS

The error messages for *statfs* are the following:

[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[EINVAL]	*path* contains a character with the high-order bit set.
[ENAMETOOLONG]	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
[ENOENT]	The file referred to by *path* does not exist.
[EACCES]	Search permission is denied for a component of the path prefix of *path* .
[ELOOP]	Too many symbolic links were encountered in translating *path* .
[EFAULT]	In user mode, *buf* or *path* points to an invalid address. In supervisor mode, this is not detected and the state of the target is unknown.
[EIO]	An *I/O* error occurred while reading from or writing to the file system.

The error messages for *fstatfs* are the following:

[EBADF]	*fd* is not a valid open file descriptor.
[EFAULT]	*buf* points to an invalid address.
[EIO]	An *I/O* error occurred while reading from or writing to the file system.

HISTORY

The *statfs* function first appeared in 4.4BSD.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

swapon – add a swap device for swapping

SYNOPSIS

#include <unistd.h>
int **swapon**(const char **special*);

FEATURES

FS_MAPPER

DESCRIPTION

The `swapon()` system call makes the block device *special* available to the system for allocation for swapping. The names of potentially available devices are known to the system, and defined at system configuration time. The size of the swap area on *special* is calculated at the time the device is first made available for swapping.

RETURN VALUES

If an error occurs, a value of `-1` is returned and `errno` is set to indicate one of the following error conditions.

ERRORS

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.
[ENOENT]	The named device does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EPERM]	The caller is not the super-user.
[ENOTBLK]	*special* is not a block device.
[EBUSY]	The device specified by *special* has already been made available for swapping.
[EINVAL]	The device configured by *special* was not configured into the system as a swap device.
[ENXIO]	The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).
[EIO]	An I/O error occurred while opening the swap device.
[EFAULT]	*special* points outside the process's allocated address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO C_INIT(1M)

BUGS It is not possible to stop swapping on a disk in order to dismount the pack.

This call will be upgraded in future versions of the BSD system.

HISTORY This function call appeared in 4.0 BSD.

NAME

symlink – make a symbolic link to a file

SYNOPSIS

#include <unistd.h>
int **symlink**(const char **name1*, const char **name2*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

A symbolic link *name2* is created to *name1* (name2 is the name of the file created, name1 is the string used to create the symbolic link). Either name may be an arbitrary pathname, and the files need not be on the same file system.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise a value of –1 is returned, and *errno* is set to indicate one of the following error conditions.

ERRORS

[ENOTDIR]	A component of the *name2* prefix is not a directory.
[ENAMETOOLONG]	A component of either pathname exceeded NAME_MAX characters, or the entire length of either pathname exceeded PATH_MAX characters.
[ENOENT]	The file named does not exist.
[EACCES]	A component of the *name2* path prefix denies search permission.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EEXIST]	*name2* already exists.
[EIO]	An I/O error occurred while making the directory entry for *name2*, allocating the inode for *name2* or writing out the link contents of *name2*.
[EROFS]	The file *name2* would reside on a read-only file system.
[ENOSPC]	The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	The new symbolic link cannot be created because there is no space left on the file system which will contain the symbolic link.
[ENOSPC]	There are no free inodes on the file system on which the symbolic link is being created.

[EDQUOT]	The directory in which the entry for the new symbolic link is being placed cannot be extended; the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	The new symbolic link cannot be created because the user's quota of disk blocks on the file system that will contain the symbolic link has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the symbolic link is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EFAULT]	In user mode, *name1* or *name2* points outside the process's allocated address space. In supervisor mode, this is not detected, and the state of the target is unknown.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`link`(2POSIX), `unlink`(2POSIX)

NAME

sync – synchronize the disk block in-core status with that on disk

SYNOPSIS

```
#include <unistd.h>
void sync(void);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The sync function forces a write of dirty (modified) buffers in the block buffer cache to disk. The kernel keeps this information in core to reduce the number of disk I/O transfers required by the system. As information in the cache is lost after a system crash, a sync call is issued frequently by the user process *syncd(1M)* (about every 10 seconds).

The *fsync(2)* function may be used to synchronize individual file descriptor attributes.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

fsync(2POSIX), syncd(1M)

BUGS

The sync call may return before the buffers are completely flushed.

HISTORY

This function call appeared in Version 6 AT&T UNIX.

NAME

truncate, ftruncate – truncate a file to a specified length

SYNOPSIS

```
#include <unistd.h>
int truncate(const char * path, off_t length);

int ftruncate(int fd, off_t length);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *truncate* call causes the file named by *path* or referenced by *fd* to be truncated to a maximum of *length* bytes in size. If the file had previously been larger than the size specified, the extra data is lost. It must be possible to write to the file in order to use *ftruncate* .

RESTRICTION

An *ftruncate* operation performed after the first *mmap(2POSIX)* on the object will fail.

RETURN VALUES

If successful, *truncate* returns a value of 0; otherwise a value of –1 is returned, and *errno* is set to indicate one of the following error conditions.

ERRORS

The error messages for *truncate* are the following:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.
[ENOENT]	The file named does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	The file named is not writable by the user.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EISDIR]	The file named is a directory.
[EROFS]	The file named resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EIO]	An I/O error occurred updating the inode.
[EFAULT]	In user mode, *path* points outside the process' allocated address space. In supervisor mode, this is not detected, and the target's state is unknown.

The error messages for *ftruncate* are the following:

[EBADF]	The *fd* is not a valid descriptor.
[EINVAL]	The *fd* references a socket, not a file.
[EINVAL]	The *fd* is not open for writing.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mmap`(2POSIX) , `open`(2POSIX)

BUGS

These calls should be kept general to allow ranges of bytes in a file to be discarded.

HISTORY

This function call appeared in 4.2BSD.

NAME

umask – set file creation mode mask

SYNOPSIS

```
#include <sys/stat.h>
mode_t umask(mode_t numask);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

`umask()` sets the file mode creation mask for a process to *numask* and returns the previous value of the mask. The nine low-order access permission bits of *numask* are used by system calls, including `mkdir`(2POSIX), `mkfifo`(2POSIX) and `open`(2POSIX), to turn off corresponding bits requested in file mode. See `chmod`(2POSIX). This clearing allows users to restrict the default access to their own files.

The default mask value is `S_IWGRP|S_IWOTH`, or `022`, providing write access for the owner only. Child processes inherit the mask of the calling process.

PARAMETERS

`umask()` takes the parameter *numask*, which defines access permission bits.

RETURN VALUES

The previous value of the file mode mask is returned by the call.

ERRORS

The `umask()` function is always successful.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chmod`(2POSIX), `mkfifo`(2POSIX), `mkdir`(2POSIX), `mknod`(2POSIX), `open`(2POSIX)

NAME

unlink – remove a directory entry

SYNOPSIS

#include <unistd.h>
int **unlink**(const char **path*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The `unlink()` system call removes the directory entry named referred to by *path*.

RETURN VALUES

Upon successful completion, `unlink()` returns `0`. Otherwise it returns `-1` and sets `errno` to indicate one of the following error conditions:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EROFS]	The directory entry to be unlinked is part of a read-only file system.
[ENAMETOOLONG]	The length of a component of *path* exceeds `NAME_MAX` characters, or the length of *path* exceeds `PATH_MAX` characters.
[ELOOP]	Too many symbolic links or symbolic ports were encountered during analysis of *path*.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EFAULT]	*path* points outside the allocated address space of the actor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`close`(2POSIX), `open`(2POSIX)

BUGS

Weaknesses in the protocol underlying NFS may cause files in directories mounted over NFS to disappear while still in use. In order to avoid this problem, always call `close()` before calling `unlink()`.

NAME

mount, unmount – mount or unmount a filesystem

SYNOPSIS

```
#include <sys/param.h>
#include <sys/mount.h>
int mount(int type, const char * dir, int flags, caddr_t data);

int unmount(const char * dir, int flags);
```

FEATURES

MSDOSFS, NFS_CLIENT, POSIX_SOCKETS, UFS

DESCRIPTION

The `mount` function grafts a filesystem object onto the system file tree at the point *dir* . The *data* argument describes the filesystem object to be mounted. The `type` argument defines how the kernel interprets *data* (See `type` below). The contents of the filesystem become available through the new mount point *dir* . Any files in *dir* at the time of a successful mount become hidden, and are unavailable until the filesystem is unmounted.

The following *flags* may be specified to suppress default semantics which affect filesystem access.

`MNT_RDONLY`	The filesystem should be treated as read-only; even the super-user may not write to it.
`MNT_NOEXEC`	Do not allow files to be executed from the filesystem.
`MNT_NOSUID`	Do not honor setuid or setgid bits on files when executing them.
`MNT_NODEV`	Do not interpret special files on the filesystem.
`MNT_SYNCHRONOUS`	All I/O to the filesystem should be done synchronously.
`MNT_NOATIME`	Disable update of file access times.
`MNT_ASYNC`	All I/O to the filesystem should be done asynchronously.
`MNT_WANTRDRW`	Upgrade a mounted read-only filesystem to read-write if MNT_UPDATE is also specified.
`MNT_FORCE`	Force a read-write mount even if the filesystem appears to be unclean. Use of this flag can be dangerous.

The `MNT_UPDATE` flag indicates that the mount command is being applied to a filesystem which is already mounted. This allows the mount flags to be changed without requiring that the filesystem be unmounted and remounted. Some

filesystems may not allow all flags to be changed. For example, most filesystems will not allow a change from read-write to read-only.

The *type* argument defines the type of the filesystem. The types of filesystems known to the system are defined in `sys/mount.h`. The *data* pointer indicates a structure that contains the type-specific arguments to mount. The types of filesystems currently supported and their type-specific data are:

```
Arguments for local filesystem mount calls

struct export_args {
    int     ex_flags;               /* export related flags */
    uid_t   ex_root;                /* mapping for root uid */
    struct  ucred ex_anon;          /* mapping for anonymous user */
    struct  sockaddr *ex_addr;      /* net address to which exported */
    int     ex_addrlen;             /* and the net address length */
    struct  sockaddr *ex_mask;      /* mask of valid bits in saddr */
    int     ex_masklen;             /* and the smask length */
     };


MOUNT_UFS

struct ufs_args {
    char      *fspec;               /* Block special file to mount */
    struct    export_args export;   /* network export information */
     };


MOUNT_NFS

struct nfs_args {
    struct sockaddr *addr;          /* file server address */
    int             addrlen;        /* length of address */
    int             sotype;         /* Socket type */
    int             proto;          /* and Protocol */
    u_char          *fh;            /* File handle to be mounted */
    int             fhsize;         /* Size, in bytes, of fh */
    int             flags;          /* flags */
    int             wsize;          /* write size in bytes */
    int             rsize;          /* read size in bytes */
    int             readdirsize;    /* readdir size in bytes */
    int             timeo;          /* initial timeout in .1 secs */
    int             retrans;        /* times to retry send */
    int             maxgrouplist;   /* Max. size of group list */
    int             readahead;      /* # of blocks to readahead */
    int             leaseterm;      /* Term (sec) of lease */
    int             deadthresh;     /* Retrans threshold */
    char            *hostname;      /* server's name */
     };


NFS mount option flags

     #define NFSMNT_SOFT         0x00000001  /* soft mount (hard is default) */
```

```
     #define NFSMNT_WSIZE        0x00000002  /* set write size */
     #define NFSMNT_RSIZE        0x00000004  /* set read size */
     #define NFSMNT_TIMEO        0x00000008  /* set initial timeout */
     #define NFSMNT_RETRANS      0x00000010  /* set number of request retries */
     #define NFSMNT_MAXGRPS      0x00000020  /* set maximum grouplist size */
     #define NFSMNT_INT          0x00000040  /* allow interrupts on hard mount */
     #define NFSMNT_NOCONN       0x00000080  /* Don't Connect the socket */
     #define NFSMNT_NQNFS        0x00000100  /* Use Nqnfs protocol */
     #define NFSMNT_NFSV3        0x00000200  /* Use NFS Version 3 protocol */
     #define NFSMNT_KERB         0x00000400  /* Use Kerberos authentication */
     #define NFSMNT_DUMBTIMR     0x00000800  /* Don't estimate rtt dynamically */
     #define NFSMNT_LEASETERM    0x00001000  /* set lease term (nqnfs) */
     #define NFSMNT_READAHEAD    0x00002000  /* set read ahead */
     #define NFSMNT_DEADTHRESH   0x00004000  /* set dead server retry thresh */
     #define NFSMNT_RESVPORT     0x00008000  /* Allocate a reserved port */
     #define NFSMNT_RDIRPLUS     0x00010000  /* Use Readdirplus for V3 */
     #define NFSMNT_READDIRSIZE  0x00020000  /* Set readdir size */
     #define NFSMNT_INTERNAL     0xfffc0000  /* Bits set internally */
     #define NFSMNT_HASWRITEVERF 0x00040000  /* Has write verifier for V3 */
     #define NFSMNT_GOTPATHCONF  0x00080000  /* Got the V3 pathconf info */
     #define NFSMNT_GOTFSINFO    0x00100000  /* Got the V3 fsinfo */
     #define NFSMNT_MNTD         0x00200000  /* Mnt server for mnt point */
     #define NFSMNT_DISMINPROG   0x00400000  /* Dismount in progress */
     #define NFSMNT_DISMNT       0x00800000  /* Dismounted */
     #define NFSMNT_SNDLOCK      0x01000000  /* Send socket lock */
     #define NFSMNT_WANTSND      0x02000000  /* Want above */
     #define NFSMNT_RCVLOCK      0x04000000  /* Rcv socket lock */
     #define NFSMNT_WANTRCV      0x08000000  /* Want above */
     #define NFSMNT_WAITAUTH     0x10000000  /* Wait for authentication */
     #define NFSMNT_HASAUTH      0x20000000  /* Has authenticator */
     #define NFSMNT_WANTAUTH     0x40000000  /* Wants an authenticator */
     #define NFSMNT_AUTHERR      0x80000000  /* Authentication error */
```

MOUNT_MSDOS

```
struct msdosfs_args {
     char    *fspec;       /* blocks special holding the fs to mount */
     struct  export_args export;     /* network export information */
     uid_t   uid;          /* uid that owns msdosfs files */
     gid_t   gid;          /* gid that owns msdosfs files */
     mode_t  mask;         /* mask to be applied for msdosfs perms */
     int     flags;        /* see below */
     int magic;            /* version number */
     u_int16_t u2w[128];   /* Local->Unicode table */
     u_int8_t  ul[128];    /* Local upper->lower table */
     u_int8_t  lu[128];    /* Local lower->upper table */
     u_int8_t  d2u[128];   /* DOS->local table */
     u_int8_t  u2d[128];   /* Local->DOS table */
     };
```

The umount function call dissociates the filesystem from the specified mount point *dir* .

The *flags* argument may specify *MNT_FORCE* to specify that the filesystem should be forcibly unmounted even if files are still active. Active special devices continue to work, but any further accesses to any other active files result in errors even if the filesystem is later remounted.

RETURN VALUES

If successful, `mount` returns a value of 0, otherwise -1 is returned and the *errno* variable is set to indicate the error.

If successful, `umount` returns a value of 0, otherwise -1 is returned and the variable *errno* is set to indicate the error.

ERRORS

`mount()` errors

`mount()` will fail when one of the following occurs:

[EPERM]	The caller is not the super-user.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or the entire length of a pathname exceeded 1023 characters.
[ELOOP]	Too many symbolic links were encountered in translating a pathname.
[ENOENT]	A component of *dir* does not exist.
[ENOTDIR]	A component of *dir* is not a directory, or a path prefix of *fspec* is not a directory.
[EBUSY]	Another process currently holds a reference to *dir* .
[EFAULT]	*dir* points outside the actor's allocated address space.

The following errors can occur for a UFS filesystem mount:

[ENODEV]	A component of `ufs_args` , *fspec* , does not exist.
[ENOTBLK]	*fspec* is not a block device.
[ENXIO]	The major device number of *fspec* is out of range (this indicates that no device driver exists for the associated hardware).
[EBUSY]	*fspec* is already mounted.
[EMFILE]	No space left in the mount table.
[EINVAL]	The superblock for the filesystem had a bad magic number or a block size that was out of range.

[ENOMEM]	Not enough memory was available to read the cylinder group information for the filesystem.
[EIO]	An I/O error occurred while reading the super block or cylinder group information.
[EFAULT]	*fspec* points outside the actor's allocated address space.

The following errors can occur for an NFS filesystem mount:

[ETIMEDOUT]	NFS timed out trying to contact the server.
[EFAULT]	Some part of the information described by *nfs_args* points outside the actor's allocated address space.

umount() errors

`umount()` may fail with one of the following errors:

[EPERM]	The caller is not the super-user.
[ENOTDIR]	A component of the path is not a directory.
[EINVAL]	A component of the path is not a directory.The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EINVAL]	The requested directory is not in the mount table.
[EBUSY]	A process is holding a reference to a file located on the filesystem.
[EIO]	An I/O error occurred while writing cached filesystem information.

A UFS mount can also fail if the maximum number of filesystems are currently mounted.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`C_INIT`(1M)

BUGS

Some of the error codes need translation to more obvious messages.

HISTORY

The `mount` and `umount` function calls appeared in Version 6 AT&T UNIX.

RESTRICTIONS FOR ChorusOS

Note that *mfs* filesystems are not supported.

NAME

utimes – set file access and modification times

SYNOPSIS

```
#include <sys/time.h>
int utimes(const char *file, const struct timeval *times);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

`utimes()` sets the access and modification times of the named file from the structures in the argument array *times*.

The first structure is the access time. The second is the modification time.

If the times are specified, that is, if the *times* parameter is not `NULL`, then the caller must be either the owner of the file or the superuser.

If the times are not specified, that is, if the *times* parameter is `NULL`, then the caller must either be the owner of the file, have permission to write to the file or be the superuser.

PARAMETERS

`utimes()` supports the following parameters:

file	Name of the file for which to set access and modification times.
times	Array of structures containing the access and modification times to set.

RETURN VALUES

`utimes()` returns `0` on successful completion. Otherwise, `utimes()` returns `–1` and `errno` is set to indicate the error.

ERRORS

`utimes()` will fail if:

`EACCES`	Search permission is denied for a component of the path prefix; or the *times* parameter is `NULL` and the effective user ID of the process does not match the owner of the file, and is not the superuser, and write access is denied.
`EFAULT`	*file* or *times* points outside the address space allocated to the process.
`EIO`	An I/O error occurred while reading or writing the affected inode.
`ELOOP`	Too many symbolic links were encountered in translating the pathname.
`ENAMETOOLONG`	A component of a pathname exceeded `255` characters, or an entire path name exceeded `1023` characters.
`ENOENT`	The named file does not exist.

ENOTDIR	A component of the path prefix is not a directory.
EPERM	The *times* parameter is not NULL and the effective user ID does not match the owner of the file and is not the superuser.
EROFS	The file system containing the file is mounted read-only.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

stat(2POSIX)

NAME

write, writev – write output

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
ssize_t write(int d, const void * buf, size_t nbytes);

ssize_t writev(int d, const struct iovec * iov, int iovcnt);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS, POSIX_SOCKETS

DESCRIPTION

`write()` attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*. `writev()` performs the same action, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: `iov[0]`, `iov[1]`, ..., `iov[iovcnt–1]`.

On objects capable of seeking, the `write()` starts at a position given by the pointer associated with *d*. See `lseek`(2POSIX). Upon return from `write()`, the pointer is incremented by the number of bytes which were written.

Objects that are not capable of seeking always write from the current position. The value of the pointer associated with this type of object is undefined.

If the real user is not the superuser, then `write()` clears the set-user- ID bit on a file. This prevents penetration of system security by a user who “captures” a writable set-user- ID file owned by the super-user.

When using non-blocking I/O on objects such as sockets that are subject to flow control, `write()` and `writev()` may write fewer bytes than requested. The return value must be noted, and the remainder of the operation should be retried when possible.

PARAMETERS

`write()` takes the following parameters:

d	Descriptor of object to which to write.
buf	Buffer from which to write data.
nbytes	Number of bytes of data to write.

`writev()` takes the following parameters:

d	Descriptor of object to which to write.
iov	Array of buffers from which to write data.
iovcnt	Number of buffers of data to write.

For `writev()`, the `iovec` structure is defined as:

```
struct iovec {
        char *iov_base; /* base address */
        size_t iov_len; /* length */
};
```

Each `iovec` entry specifies the base address and length of an area in memory from which data should be written. `writev( )` always writes a complete area before proceeding to the next.

RETURN VALUES

Upon successful completion, `write( )` and `writev( )` return the number of bytes actually written. Otherwise, they return `–1` and set `errno` to indicate the error.

ERRORS

`write( )` and `writev( )` fail and the file pointer remains unchanged if:

`[EAGAIN]`	The file was marked for non-blocking I/O , and no data could be written immediately.
`[EBADF]`	*d* is not a valid descriptor of an object open for writing.
`[EFAULT]`	In user mode, part of *buf* or *iov* to be written to the file points outside the allocated address space for the process. In supervisor mode, this is not detected, and the state of the target is unknown.
`[EFBIG]`	An attempt was made to write a file that exceeds the maximum file size or the file size limit for the process.
`[EINVAL]`	The pointer associated with *d* was negative.
`[EIO]`	An I/O error occurred while reading from or writing to the file system.
`[ENOSPC]`	No free space remained on the file system containing the file.
`[EPIPE]`	An attempt was made to write to a pipe that is not open for reading by any process.
`[ESOCKET]`	An attempt was made to write to a socket of type `SOCK_STREAM` that is not connected to a peer socket.

In addition, `writev( )` may return one of the following errors:

`[EINVAL]`	*iovcnt* was less than or equal to `0` , or greater than `UIO_MAXIOV` .
`[EINVAL]`	One of the *iov_len* values in the *iov* array was negative.
`[EINVAL]`	The sum of the *iov_len* values in the *iov* array overflowed a 32–bit integer.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fcntl`(2POSIX) , `lseek`(2POSIX) , `open`(2POSIX) , `pipe`(2POSIX) , `select`(2POSIX)

NAME

write, writev – write output

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
ssize_t write(int d, const void * buf, size_t nbytes);

ssize_t writev(int d, const struct iovec * iov, int iovcnt);
```

FEATURES

MSDOSFS, NFS_CLIENT, UFS, POSIX_SOCKETS

DESCRIPTION

`write()` attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*. `writev()` performs the same action, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: `iov[0]`, `iov[1]`, ..., `iov[iovcnt–1]`.

On objects capable of seeking, the `write()` starts at a position given by the pointer associated with *d*. See `lseek`(2POSIX). Upon return from `write()`, the pointer is incremented by the number of bytes which were written.

Objects that are not capable of seeking always write from the current position. The value of the pointer associated with this type of object is undefined.

If the real user is not the superuser, then `write()` clears the set-user- ID bit on a file. This prevents penetration of system security by a user who “captures” a writable set-user- ID file owned by the super-user.

When using non-blocking I/O on objects such as sockets that are subject to flow control, `write()` and `writev()` may write fewer bytes than requested. The return value must be noted, and the remainder of the operation should be retried when possible.

PARAMETERS

`write()` takes the following parameters:

d	Descriptor of object to which to write.
buf	Buffer from which to write data.
nbytes	Number of bytes of data to write.

`writev()` takes the following parameters:

d	Descriptor of object to which to write.
iov	Array of buffers from which to write data.
iovcnt	Number of buffers of data to write.

For `writev()`, the `iovec` structure is defined as:

```
struct iovec {
        char *iov_base; /* base address */
        size_t iov_len; /* length */
};
```

Each `iovec` entry specifies the base address and length of an area in memory from which data should be written. `writev( )` always writes a complete area before proceeding to the next.

RETURN VALUES

Upon successful completion, `write( )` and `writev( )` return the number of bytes actually written. Otherwise, they return `–1` and set `errno` to indicate the error.

ERRORS

`write( )` and `writev( )` fail and the file pointer remains unchanged if:

`[EAGAIN]`	The file was marked for non-blocking I/O , and no data could be written immediately.
`[EBADF]`	*d* is not a valid descriptor of an object open for writing.
`[EFAULT]`	In user mode, part of *buf* or *iov* to be written to the file points outside the allocated address space for the process. In supervisor mode, this is not detected, and the state of the target is unknown.
`[EFBIG]`	An attempt was made to write a file that exceeds the maximum file size or the file size limit for the process.
`[EINVAL]`	The pointer associated with *d* was negative.
`[EIO]`	An I/O error occurred while reading from or writing to the file system.
`[ENOSPC]`	No free space remained on the file system containing the file.
`[EPIPE]`	An attempt was made to write to a pipe that is not open for reading by any process.
`[ESOCKET]`	An attempt was made to write to a socket of type `SOCK_STREAM` that is not connected to a peer socket.

In addition, `writev( )` may return one of the following errors:

`[EINVAL]`	*iovcnt* was less than or equal to `0` , or greater than `UIO_MAXIOV` .
`[EINVAL]`	One of the *iov_len* values in the *iov* array was negative.
`[EINVAL]`	The sum of the *iov_len* values in the *iov* array overflowed a 32–bit integer.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `fcntl`(2POSIX) , `lseek`(2POSIX) , `open`(2POSIX) , `pipe`(2POSIX) , `select`(2POSIX)

Index

T

U

W