NAME

access − determine accessibility of file

SYNOPSIS

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

```
#define R_OK    4    /* test for read permission */
#define W_OK    2    /* test for write permission */
#define X_OK    1    /* test for execute (search) permission */
#define F_OK    0    /* test for presence of file */
```

int access(const char ****path*, mode_t** *mode***)**

DESCRIPTION

Access checks the given file *path* for accessibility according to *mode*, which is an inclusive or of the bits **R_OK**, **W_OK** and **X_OK**. Specifying *mode* as **F_OK** (i.e., 0) tests whether the directories leading to the file can be searched and the file exists.

The real user ID and the group access list (including the real group ID) are used in verifying permission, so this call is useful to set-UID programs.

Notice that only access bits are checked. A directory may be indicated as writable by **access**, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but **execve** will fail unless it is in proper format.

RETURN VALUE

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

ERRORS

Access to the file is denied if one or more of the following are true:

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

[ENAMETOOLONG]
The path name exceeds PATH_MAX characters.

[ENOENT] The named file 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. (Minix-vmd)

[EROFS] Write access is requested for a file on a read-only file system.

[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 permission checked with respect to the ‘‘owner’’ read, write, and execute mode bits, members of the file’s group other than the owner have permission 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.

SEE ALSO

chmod(2), **stat**(2).

NAME

alarm − schedule signal after specified time

SYNOPSIS

#include <unistd.h>

unsigned int alarm(unsigned int *seconds***)**

DESCRIPTION

Alarm causes signal SIGALRM, see **sigaction**(2), to be sent to the invoking process in a number of seconds given by the argument. Unless caught or ignored, the signal terminates the process.

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is canceled. Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

pause(2), **sigsuspend**(2), **sigaction**(2), **sleep**(3).

NAME

brk, sbrk − change data segment size

SYNOPSIS

#include <unistd.h>

char *brk(char ***addr)**

char *sbrk(int *incr***)**

DESCRIPTION

Brk sets the system’s idea of the lowest data segment location not used by the program (called the break) to *addr*. Locations greater than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In the alternate function **sbrk**, *incr* more bytes are added to the program’s data space and a pointer to the start of the new area is returned.

When a program begins execution via **execve** the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use **sbrk**.

RETURN VALUE

The address of the new break is returned if **brk** succeeds; **−1** if the program requests more memory than the system limit. **Sbrk** returns **−1** if the break could not be set.

ERRORS

Sbrk will fail and no additional memory will be allocated if one of the following are true:

[ENOMEM] The maximum possible size of a data segment (as set by **chmem**(1)) was exceeded.

[ENOMEM] Insufficient virtual memory space existed to support the expansion. (Minix-vmd)

SEE ALSO

chmem(1), **execve**(2), **malloc**(3), **end**(3).

NOTES

Minix-vmd rounds a small data segment limit up to 3 megabytes.

BUGS

Setting the break may fail due to a temporary lack of virtual memory under Minix-vmd. It is not possible to distinguish this from a failure caused by exceeding the maximum size of the data segment.

NAME

chdir − change current working directory

SYNOPSIS

#include <unistd.h>

int chdir(const char ****path*)**

DESCRIPTION

Path is the pathname of a directory. **Chdir** causes this directory to become the current working directory, the starting point for path names not beginning with ‘‘/’’.

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

ERRORS

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

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	The path name exceeds PATH_MAX characters.
[ENOENT]	The named directory does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname. (Minix-vmd)
[EACCES]	Search permission is denied for any component of the path name.
[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.

SEE ALSO

chroot(2).

NAME

chmod − change mode of file

SYNOPSIS

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

int chmod(const char *path, mode_t mode)
```

DESCRIPTION

The file whose name is given by *path* has its mode changed to *mode*. Modes are constructed by *or*'ing together some combination of the following, defined in *<sys/stat.h>*:

S_ISUID	04000	set user ID on execution
S_ISGID	02000	set group ID on execution
S_ISVTX	01000	'sticky bit' (see below)
S_IRWXU	00700	read, write, execute by owner
S_IRUSR	00400	read by owner
S_IWUSR	00200	write by owner
S_IXUSR	00100	execute (search on directory) by owner
S_IRWXG	00070	read, write, execute by group
S_IRGRP	00040	read by group
S_IWGRP	00020	write by group
S_IXGRP	00010	execute (search on directory) by group
S_IRWXO	00007	read, write, execute by others
S_IROTH	00004	read by others
S_IWOTH	00002	write by others
S_IXOTH	00001	execute (search on directory) by others

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. (Minix-vmd)

Only the owner of a file (or the super-user) may change the mode.

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 VALUE

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:

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

[ENAMETOOLONG]
The path name exceeds PATH_MAX characters.

[ENOENT] The named file 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. (Minix-vmd)

[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.

[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.

SEE ALSO

chmod(1), **open**(2), **chown**(2), **stat**(2).

NOTES

The sticky bit was historically used to lock important executables into memory.

NAME

chown − change owner and group of a file

SYNOPSIS

int chown(const char ****path*, **int *owner*, **int** *group***)**

DESCRIPTION

The file that is named by *path* has its *owner* and *group* changed as specified. Only the super-user may change the owner of the file, because if users were able to give files away, they could defeat file-space accounting procedures. The owner of the file may change the group to a group of which he is a member.

On some systems, *chown* clears the set-user-id and set-group-id bits on the file to prevent accidental creation of set-user-id and set-group-id programs.

RETURN VALUE

Zero is returned if the operation was successful; −1 is returned if an error occurs, with a more specific error code being placed in the global variable **errno**.

ERRORS

Chown will fail and the file will be unchanged if:

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

[ENAMETOOLONG]
The path name exceeds PATH_MAX characters.

[ENOENT] The named file 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. (Minix-vmd)

[EPERM] The effective user ID is not the super-user.

[EROFS] The named file resides on a read-only file system.

[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.

SEE ALSO

chown(8), **chgrp**(1), **chmod**(2).

NAME

chroot − change root directory

SYNOPSIS

#include <unistd.h>

int chroot(const char ***_dirname_)**

DESCRIPTION

Dirname is the address of the pathname of a directory, terminated by a null byte. **Chroot** causes this directory to become the root directory, the starting point for path names beginning with ‘‘/’’.

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

This call is restricted to the super-user.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and **errno** is set to indicate an error.

ERRORS

Chroot will fail and the root directory will be unchanged if one or more of the following are true:

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

[ENAMETOOLONG]
The path name exceeds PATH_MAX characters.

[ENOENT] The named directory does not exist.

[EACCES] Search permission is denied for any component of the path name.

[ELOOP] Too many symbolic links were encountered in translating the pathname. (Minix-vmd)

[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.

SEE ALSO

chdir(2).

NAME

close − delete a descriptor

SYNOPSIS

#include <unistd.h>

int close(int *d***)**

DESCRIPTION

The **close** call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, then it will be deactivated. For example, on the last close of a file the current *seek* pointer associated with the file is lost; on the last close of a TCP/IP descriptor associated naming information and queued data are discarded; on the last close of a file holding an advisory lock the lock is released (see further **fcntl**(2)).

A close of all of a process's descriptors is automatic on *exit*, but since there is a limit on the number of active descriptors per process, **close** is necessary for programs that deal with many descriptors.

When a process forks (see **fork**(2)), all descriptors for the new child process reference the same objects as they did in the parent before the fork. If a new process is then to be run using **execve**(2), the process would normally inherit these descriptors. Most of the descriptors can be rearranged with **dup2**(2) or deleted with **close** before the **execve** is attempted, but if some of these descriptors will still be needed if the **execve** fails, it is necessary to arrange for them to be closed if the **execve** succeeds. For this reason, the call ‘‘fcntl(d, F_SETFD, *flags*)’’ is provided, that can be used to mark a descriptor "close on exec" by setting the **FD_CLOEXEC** flag:

```
fcntl(d, F_SETFD, fcntl(d, F_GETFD) | FD_CLOEXEC);
```

RETURN VALUE

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

ERRORS

Close will fail if:

[EBADF] *D* is not an active descriptor.

SEE ALSO

open(2), **pipe**(2), **execve**(2), **fcntl**(2).

NAME

creat − create a new file

SYNOPSIS

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

int creat(const char ****name*, mode_t** *mode***)**

DESCRIPTION

This interface is made obsolete by open(2), it is equivalent to

open(*name*, O_WRONLY | O_CREAT | O_TRUNC, *mode*)

Creat creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*, as modified by the process's mode mask (see **umask**(2)). Also see **chmod**(2) for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned.

NOTES

The *mode* given is arbitrary; it need not allow writing. This feature has been used in the past by programs to construct a simple, exclusive locking mechanism. It is replaced by the O_EXCL open mode, or the advisory locking of the **fcntl**(2) facility.

RETURN VALUE

The value −1 is returned if an error occurs. Otherwise, the call returns a non-negative descriptor that only permits writing.

ERRORS

Creat will fail and the file will not be created or truncated if one of the following occur:

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

[ENAMETOOLONG]
The path name exceeds PATH_MAX characters.

[ENOENT] The named file does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname. (Minix-vmd)

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] The file does not exist and the directory in which it is to be created is not writable.

[EACCES] The file exists, but it is unwritable.

[EISDIR] The file is a directory.

[EMFILE] There are already too many files open.

[ENFILE] The system file table is full.

[ENOSPC] The directory in which the entry for the new file 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 file is being created.

[EROFS] The named file resides on a read-only file system.

[ENXIO] The file is a character special or block special file, and the associated device does not exist.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[EFAULT] *Name* points outside the process's allocated address space.

SEE ALSO

open(2), **write**(2), **close**(2), **chmod**(2), **umask**(2).

NAME

dup, dup2 − duplicate a descriptor

SYNOPSIS

#include <unistd.h>

int dup(int *oldd***)**
int dup2(int *oldd***, int** *newd***)**

DESCRIPTION

Dup duplicates an existing descriptor. The argument *oldd* is a small non-negative integer index in the per-process descriptor table. The value must be less than OPEN_MAX, the size of the table. The new descriptor returned by the call, let's name it *newd,* is the lowest numbered descriptor that is not currently in use by the process.

The object referenced by the descriptor does not distinguish between references using *oldd* and *newd* in any way. Thus if *newd* and *oldd* are duplicate references to an open file, **read**(2), **write**(2) and **lseek**(2) calls all move a single pointer into the file, and append mode, non-blocking I/O and asynchronous I/O options are shared between the references. If a separate pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional **open**(2) call. The close-on-exec flag on the new file descriptor is unset.

In the second form of the call, the value of *newd* desired is specified. If this descriptor is already in use, the descriptor is first deallocated as if a **close**(2) call had been done first. *Newd* is not closed if it equals *oldd*.

RETURN VALUE

The value −1 is returned if an error occurs in either call. The external variable **errno** indicates the cause of the error.

ERRORS

Dup and **dup2** fail if:

[EBADF] *Oldd* or *newd* is not a valid active descriptor

[EMFILE] Too many descriptors are active.

NOTES

Dup and **dup2** are now implemented using the **F_DUPFD** function of **fcntl**(2), although the old system call interfaces still exist to support old programs.

SEE ALSO

open(2), **close**(2), **fcntl**(2), **pipe**(2).

NAME

execve − execute a file

SYNOPSIS

#include <unistd.h>

int execve(const char ****name*, char *const** *argv***[], char *const** *envp***[])**

DESCRIPTION

Execve transforms the calling process into a new process. The new process is constructed from an ordinary file called the *new process file*. This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialized data pages. Additional pages may be specified by the header to be initialized with zero data. See **a.out**(5).

An interpreter file begins with a line of the form ‘‘#! *interpreter*’’. (Minix-vmd only.) When an interpreter file is **execve**’d, the system **execve**’s the specified *interpreter*, giving it the name of the originally exec’d file as an argument and shifting over the rest of the original arguments.

There can be no return from a successful **execve** because the calling core image is lost. This is the mechanism whereby different process images become active.

The argument *argv* is a null-terminated array of character pointers to null-terminated character strings. These strings constitute the argument list to be made available to the new process. By convention, at least one argument must be present in this array, and the first element of this array should be the name of the executed program (i.e., the last component of *name*).

The argument *envp* is also a null-terminated array of character pointers to null-terminated strings. These strings pass information to the new process that is not directly an argument to the command (see **environ**(7)).

Descriptors open in the calling process remain open in the new process, except for those for which the close-on-exec flag is set (see **close**(2)). Descriptors that remain open are unaffected by **execve**.

Ignored signals remain ignored across an **execve**, but signals that are caught are reset to their default values. Blocked signals remain blocked regardless of changes to the signal action. The signal stack is reset to be undefined (see **sigaction**(2) for more information).

Each process has *real* user and group IDs and an *effective* user and group IDs. The *real* ID identifies the person using the system; the *effective* ID determines his access privileges. **Execve** changes the effective user and group ID to the owner of the executed file if the file has the set-user-ID or set-group-ID modes. The *real* user ID is not affected.

The new process also inherits the following attributes from the calling process:

process ID	see **getpid**(2)
parent process ID	see **getppid**(2)
process group ID	see **getpgrp**(2)
access groups	see **getgroups**(2)
working directory	see **chdir**(2)
root directory	see **chroot**(2)
control terminal	see **tty**(4)
alarm timer	see **alarm**(2)
file mode mask	see **umask**(2)
signal mask	see **sigaction**(2), **sigprocmask**(2)

When the executed program begins, it is called as follows:

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

exit(main(argc, argv, envp));
```

where *argc* is the number of elements in *argv* (the ‘‘arg count’’) and *argv* is the array of character pointers to the arguments themselves.

Envp is a pointer to an array of strings that constitute the *environment* of the process. A pointer to this array is also stored in the global variable ‘‘environ’’. Each string consists of a name, an =, and a null-terminated value. The array of pointers is terminated by a null pointer. The shell **sh**(1) passes an environment entry for each global shell variable defined when the program is called. See **environ**(7) for some conventionally used names.

RETURN VALUE

If **execve** returns to the calling process an error has occurred; the return value will be −1 and the global variable **errno** will contain an error code.

ERRORS

Execve will fail and return to the calling process if one or more of the following are true:

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

[ENAMETOOLONG] The path name exceeds PATH_MAX characters.

[ENOENT] The new process file does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname. (Minix-vmd)

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] The new process file is not an ordinary file.

[EACCES] The new process file mode denies execute permission.

[ENOEXEC] The new process file has the appropriate access permission, but has an invalid magic number in its header.

[ENOMEM] The new process requires more (virtual) memory than is currently available.

[E2BIG] The number of bytes in the new process’s argument list is larger than the system-imposed limit ARG_MAX. The limit in the system as released is 4096 bytes for 16-bit Minix, 16384 bytes for 32-bit Minix, and unlimited for Minix-vmd.

[EFAULT] *Path* , *argv* , or *envp* point to an illegal address.

[EIO] An I/O error occurred while reading from the file system.

CAVEATS

If a program is *setuid* to a non-super-user, but is executed when the real **uid** is ‘‘root’’, then the program has some of the powers of a super-user as well.

SEE ALSO

exit(2), **fork**(2), **execl**(3), **environ**(7).

NAME

exit, _exit − terminate a process

SYNOPSIS

void _exit(int *status***)**

DESCRIPTION

_exit terminates a process with the following consequences:

> All of the descriptors open in the calling process are closed. This may entail delays, for example, waiting for output to drain; a process in this state may not be killed, as it is already dying.
>
> If the parent process of the calling process is executing a **wait** or is interested in the SIGCHLD signal (Minix-vmd), then it is notified of the calling process's termination and the low-order eight bits of *status* are made available to it; see **wait**(2).
>
> The parent process ID of all of the calling process's existing child processes are also set to 1. This means that the initialization process (see **intro**(2)) inherits each of these processes as well.

Most C programs call the library routine **exit**(3), which performs cleanup actions in the standard I/O library before calling *_exit* .

RETURN VALUE

This call never returns.

SEE ALSO

fork(2), **sigaction**(2), **wait**(2), **exit**(3).

NAME

fcntl − miscellaneous file descriptor control functions

SYNOPSIS

#include <fcntl.h>

int fcntl(int *fd***, int ***cmd***,** [*data*]**)**

DESCRIPTION

Fcntl() performs several file descriptor related functions, like duplicating a file descriptor, setting the "close on exec" attribute, etc. The *fd* argument is the file descriptor to operate on, *cmd* is the command code of the operation to perform, and *data* is an optional argument to give or receive parameters. The command codes and other symbols and types are declared in <fcntl.h>. The commands are:

fcntl(*fd***, F_DUPFD, int** *fd2***)**
: Returns a new file descriptor that is a duplicate of file descriptor *fd*. It shares the same file pointer and the same file status flags, but has separate file descriptor flags that are initially off. The value of the duplicate file descriptor is the first free file descriptor greater than or equal to *fd2*.

fcntl(*fd***, F_GETFD)**
: Returns the file descriptor flags associated with file descriptor *fd*. The flags are the "close on exec" flag **FD_CLOEXEC** that, when set, causes the file descriptor to be closed when the process executes another program. The Minix-vmd specific **FD_ASYNCHIO** flag marks a file descriptor for asynchronous I/O operation.

fcntl(*fd***, F_SETFD, int** *flags***)**
: Set the file descriptor flags of *fd* to *flags*.

fcntl(*fd***, F_GETFL)**
: Return the file status flags and file access modes associated with the file associated with file descriptor *fd*. The file status flags are **O_NONBLOCK** (non blocking I/O) and **O_APPEND** (append mode). The file access modes are **O_RDONLY** (read-only), **O_WRONLY** (write-only) and **O_RDWR** (read-write). These flags are also used in the second argument of **open**(2).

fcntl(*fd***, F_SETFL, int** *flags***)**
: Set the file status flags of the file referenced by *fd* to *flags*. Only **O_NONBLOCK** and **O_APPEND** may be changed. Access mode flags are ignored.

The next four commands use a parameter of type **struct flock** that is defined in <fcntl.h> as:

```
struct flock {
    short   l_type;     /* F_RDLCK, F_WRLCK, or F_UNLCK */
    short   l_whence;   /* SEEK_SET, SEEK_CUR, or SEEK_END */
    off_t   l_start;    /* byte offset to start of segment */
    off_t   l_len;      /* length of segment */
    pid_t   l_pid;      /* process id of the locks' owner */
};
```

This structure describes a segment of a file. **L_type** is the lock operation performed on the file segment: **F_RDLCK** to set a read lock, **F_WRLCK** to set a write lock, and **F_UNLCK** to remove a lock. Several processes may have a read lock on a segment, but only one process can have a write lock. **L_whence** tells if the **l_start** offset must be interpreted from the start of the file (**SEEK_SET**), the current file position (**SEEK_CUR**), or the end of the file (**SEEK_END**). This is analogous to the third parameter of **lseek**(2). These **SEEK_*** symbols are declared in <unistd.h>. **L_start** is the starting offset of the segment of the file. **L_end** is the length of the segment. If zero then the segment extends until end of file. **L_pid** is the process-id of the process currently holding a lock on the segment. It is returned by **F_GETLK**.

fcntl(*fd***, F_GETLK, struct flock ***lkp***)**
: Find out if some other process has a lock on a segment of the file associated by file descriptor

fd that overlaps with the segment described by the **flock** structure pointed to by *lkp*. If the segment is not locked then **l_type** is set to **F_UNLCK**. Otherwise an **flock** structure is returned through *lkp* that describes the lock held by the other process. **L_start** is set relative to the start of the file.

fcntl(*fd***, F_SETLK, struct flock ****lkp***)**
Register a lock on a segment of the file associated with file descriptor *fd*. The file segment is described by the **struct flock** pointed to by *lkp*. This call returns an error if any part of the segment is already locked.

fcntl(*fd***, F_SETLKW, struct flock ****lkp***)**
Register a lock on a segment of the file associated with file descriptor *fd*. The file segment is described by the **struct flock** pointed to by *lkp*. This call blocks waiting for the lock to be released if any part of the segment is already locked.

fcntl(*fd***, F_FREESP, struct flock ****lkp***)**
Free a segment of disk space occupied by the file associated with file descriptor *fd*. The segment is described by the **struct flock** pointed to by *lkp*. The file is truncated in length to the byte position indicated by **l_start** if **l_len** is zero. If **l_len** is nonzero then the file keeps its size, but the freed bytes now read as zeros. (Other than sharing the flock structure, this call has nothing to do with locking.)

fcntl(*fd***, F_SEEK, u64_t** *pos***)**
This Minix-vmd specific call sets the file position of the file associated with file descriptor *fd* to the byte offset indicated by the 64-bit number *pos*. This is analogous to the call

lseek(*fd***,** *pos***, SEEK_SET)**

except that **F_SEEK** can be used on devices larger than 4 gigabyte.

SEE ALSO

open(2), **dup**(2), **lseek**(2), **ftruncate**(3), **int64**(3).

DIAGNOSTICS

Fcntl returns a file descriptor, flags, or **0** to indicate success. On error **−1** is returned, with **errno** set to the appropriate error code. The most notable errors are:

EINTR
If a blocked **F_SETLKW** operation is interrupted by a signal that is caught.

EAGAIN
By **F_SETLK** if a segment cannot be locked.

EBADF
A bad file descriptor in general, or an attempt to place a write lock on a file that is not open for writing, etc.

ENOLCK
No locks available, the file system code has run out of internal table space.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

fork − create a new process

SYNOPSIS

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

pid_t fork(void)

DESCRIPTION

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process except for the following:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an **lseek**(2) on a descriptor in the child process can affect a subsequent **read** or **write** by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.

The child starts with no pending signals and an inactive alarm timer.

RETURN VALUE

Upon successful completion, **fork** returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent process, no child process is created, and the global variable **errno** is set to indicate the error.

ERRORS

Fork will fail and no child process will be created if one or more of the following are true:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent. (The kernel variable NR_PROCS in <minix/config.h> (Minix), or <minix/const.h> (Minix-vmd).)

[ENOMEM] There is insufficient (virtual) memory for the new process.

SEE ALSO

execve(2), **wait**(2).

NAME

getgid, getegid − get group identity

SYNOPSIS

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

gid_t getgid(void)
gid_t getegid(void)

DESCRIPTION

Getgid returns the real group ID of the current process, **getegid** the effective group ID.

The real group ID is specified at login time.

The effective group ID is more transient, and determines additional access permission during execution of a ‘‘set-group-ID’’ process, and it is for such processes that **getgid** is most useful.

SEE ALSO

getuid(2), **setgid**(2).

NAME

getpid, getppid − get process identification

SYNOPSIS

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

pid_t getpid(void)
pid_t getppid(void)

DESCRIPTION

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

Getppid returns the process ID of the parent of the current process.

SEE ALSO

fork(2).

NAME

getuid, geteuid − get user identity

SYNOPSIS

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

uid_t getuid(void)
uid_t geteuid(void)

DESCRIPTION

Getuid returns the real user ID of the current process, **geteuid** the effective user ID.

The real user ID identifies the person who is logged in. The effective user ID gives the process additional permissions during execution of set-user-ID mode processes, which use **getuid** to determine the real-user-id of the process that invoked them.

SEE ALSO

getgid(2), **setuid**(2).

NAME

intro, errno − introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible return value. This is almost always −1; the individual descriptions specify the details. Note that a number of system calls overload the meanings of these error numbers, and that the meanings must be interpreted according to the type and circumstances of the call.

As with normal arguments, all return codes and values from functions are of type integer unless otherwise noted. An error number is also made available in the external variable **errno**, which is not cleared on successful calls. Thus **errno** should be tested only after an error has occurred.

The following is a complete list of the errors and their names as given in *<sys/errno.h>*:

0 OK Error 0

Unused. (The symbol "OK" is only used inside the kernel source.)

1 EPERM Not owner

Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory

This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process

The process or process group whose number was given does not exist, or any such process is already dead.

4 EINTR Interrupted system call

An asynchronous signal (such as interrupt or quit) that the user has elected to catch occurred during a system call. If execution is resumed after processing the signal and the system call is not restarted, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error

Some physical I/O error occurred during an I/O operation, usually **read** or **write**. Operations on file descriptors that refer to devices that are forcefully taken away or in a bad state will also provoke this error.

6 ENXIO No such device or address

I/O on a special file refers to a subdevice that does not exist, or beyond the limits of the device. It may also occur when, for example, an illegal tape drive unit number is selected or a disk pack is not loaded on a drive.

7 E2BIG Arg list too long

An argument list longer than ARG_MAX bytes is presented to **execve**. ARG_MAX is set to 4096 bytes for 16-bit Minix, 16384 bytes for 32-bit Minix, and unlimited for Minix-vmd as these systems are released.

8 ENOEXEC Exec format error

A request is made to execute a file that, although it has the appropriate permissions, does not start with a valid magic number, (see **a.out**(5)).

9 EBADF Bad file number

Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file that is open only for writing (resp. reading).

10 ECHILD No children
: **Wait** and the process has no living or unwaited-for children.

11 EAGAIN Resource temporarily unavailable
: In a **fork,** the system's process table is full or the user is not allowed to create any more processes, otherwise an operation that would cause a process to block was attempted on an object in non-blocking mode (see **fcntl**(2)).

12 ENOMEM Not enough core
: During an **execve** or **brk,** a program asks for more (virtual) memory than the system is able to supply, or a process size limit would be exceeded. The maximum size of the data+stack segment is set by the **chmem**(1) program. For Minix-vmd a small data+stack size is increased to 3 megabytes when a program is executed.

13 EACCES Permission denied
: An attempt was made to access a file in a way forbidden by the protection system. Also an attempt to open a device for writing that is physically write protected.

14 EFAULT Bad address
: An argument of a system call is outside the address space allocated to a process.

15 ENOTBLK Block device required
: A plain file was mentioned where a block device was required, e.g., in **mount**.

16 EBUSY Resource busy
: An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, or active text segment). A request was made to an exclusive access device that was already in use.

17 EEXIST File exists
: An existing file was mentioned in an inappropriate context, e.g., **link**.

18 EXDEV Cross-device link
: A hard link to a file on another device was attempted.

19 ENODEV No such device
: An attempt was made to access a device that is not configured by the system, i.e., there is no driver for the device.

20 ENOTDIR Not a directory
: A non-directory was specified where a directory is required, for example, in a path name or as an argument to **chdir**.

21 EISDIR Is a directory
: An attempt to write on a directory.

22 EINVAL Invalid argument
: Some invalid argument: dismounting a non-mounted device, mentioning an unknown signal in **signal,** or some other argument inappropriate for the call. Also set by math functions, (see **math**(3)).

23 ENFILE File table overflow
: The system's table of open files is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
: The limit on the number of open files per process, OPEN_MAX, is reached. As released, this limit is 20 for Minix, and 30 for Minix-vmd.

25 ENOTTY Not a typewriter
: The file mentioned in an **ioctl** is not a terminal or one of the devices to which this call applies. (Often seen error from programs with bugs in their error reporting code.)

26 ETXTBSY Text file busy
: Attempt to execute a program that is open for writing. Obsolete under Minix.

27 EFBIG File too large
The size of a file exceeded the maximum (little over 64 megabytes for the V2 file system).

28 ENOSPC No space left on device
A **write** to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because no more disk blocks are available on the file system, or the allocation of an inode for a newly created file failed because no more inodes are available on the file system.

29 ESPIPE Illegal seek
An **lseek** was issued to a pipe or TCP/IP channel. This error may also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than a certain number of hard links to a file. The advertized maximum, LINK_MAX, is 127, but Minix-vmd uses a much larger maximum of 32767 for the V2 file system.

32 EPIPE Broken pipe
A write on a pipe or TCP/IP channel for which there is no process to read the data. This condition normally generates the signal SIGPIPE; the error is returned if the signal is caught or ignored.

33 EDOM Math argument
The argument of a function in the math package is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package is unrepresentable within machine precision.

35 EDEADLK Resource deadlock avoided
A process attempts to place a blocking lock on a file that is already locked by another process and that process is waiting for the first process to unlock a file that first process already has a lock on. (The classic "lock A, lock B" by process 1, and "lock B, lock A" by process 2.)

36 ENAMETOOLONG File name too long
The path name exceeds PATH_MAX characters. PATH_MAX equals 255 as distributed.

37 ENOLCK No locks available
The system's table of active locks is full.

38 ENOSYS Function not implemented
The system call is not supported. Either an old program uses an obsolete call, or a program for a more capable system is run on a less capable system.

39 ENOTEMPTY Directory not empty
A directory with entries other than . and .. was supplied to a remove directory or rename call.

40 ELOOP Too many symbolic links
A path name lookup involved more than SYMLOOP symbolic links. SYMLOOP equals 8 as distributed. (Minix-vmd)

50 EPACKSIZE Invalid packet size

51 EOUTOFBUFS Not enough buffers left

52 EBADIOCTL Illegal ioctl for device

53 EBADMODE Bad mode in ioctl

54 EWOULDBLOCK Would block

55 EBADDEST Bad destination address

56 EDSTNOTRCH Destination not reachable

57 EISCONN Already connected

58 EADDRINUSE Address in use

59 ECONNREFUSED Connection refused

60 ECONNRESET Connection reset

61 ETIMEDOUT Connection timed out

62 EURG Urgent data present

63 ENOURG No urgent data present

64 ENOTCONN No connection

65 ESHUTDOWN Already shutdown

66 ENOCONN No such connection

67 EINPROGRESS Operation now in progress

68 EALREADY Operation already in progress

DEFINITIONS

Process ID

Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to 29999. The special process with process ID 1 is **init**, the ancestor of all processes.

Parent process ID

A new process is created by a currently active process; (see **fork**(2)). The parent process ID of a process is the process ID of its creator, unless the creator dies, then **init** becomes the parent of the orphaned process.

Process Group ID

Each active process is a member of a process group that is identified by a positive integer called the process group ID. This is the process ID of the group leader. This grouping permits the signaling of related processes (see **kill**(2)).

Real User ID and Real Group ID

Each user on the system is identified by a positive integer termed the real user ID.

Each user is also a member of one or more groups. One of these groups is distinguished from others and used in implementing accounting facilities. The positive integer corresponding to this distinguished group is termed the real group ID. (Under standard Minix this is the only group a process can be a member of.)

All processes have a real user ID and real group ID. These are initialized from the equivalent attributes of the process that created it.

Effective User Id, Effective Group Id, and Access Groups

Access to system resources is governed by three values: the effective user ID, the effective group ID, and the group access list.

The effective user ID and effective group ID are initially the process's real user ID and real group ID respectively. Either may be modified through execution of a set-user-ID or set-group-ID file (possibly by one its ancestors) (see **execve**(2)).

The group access list is an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described below in ‘‘File Access Permissions’’. The maximum number of additional group ID's is NGROUPS_MAX. For Minix this is 0, but Minix-vmd supports a list of up to 16 additional group ID's. (Also known as ‘‘supplemental’’ group ID's.)

Super-user

A process is recognized as a *super-user* process and is granted special privileges if its effective user ID is 0.

Descriptor

An integer assigned by the system when a file or device is referenced by **open**(2), **dup**(2) or **fcntl**(2) which uniquely identifies an access path to that file or device from a given process or any of its children.

File Descriptor

Older, and often used name for a descriptor.

File Name

Names consisting of up to NAME_MAX characters may be used to name an ordinary file, special file, or directory. NAME_MAX is the maximum of the maximum file name lengths of the supported file systems. Excess characters are ignored when too long file names are used for files in a given file system. The maximum file name length of the V1 and V2 file systems is 14 characters. The Minix-vmd "flex" variants of V1 and V2 have a 60 character maximum.

The characters in a file name may assume any value representable in eight bits excluding 0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use one of \’"<>();~$^&*|{}[]? as part of file names because of the special meaning attached to these characters by the shell.

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. The total length of a path name must be less than PATH_MAX characters (255 as distributed.)

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. A null pathname is illegal, use "." to refer to the current working directory.

Directory

A directory is a special type of file that contains entries that are references to other files. Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. A process’s root directory need not be the root directory of the root file system.

File Access Permissions

Every file in the file system has a set of access permissions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access permissions are established at the time a file is created. They may be changed at some later time through the **chmod**(2) call.

File access is broken down according to whether a file may be: read, written, or executed. Directory files use the execute permission to control if the directory may be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, those users in the file’s group, anyone else. Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if:

The process’s effective user ID is that of the super-user.

The process's effective user ID matches the user ID of the owner of the file and the owner permissions allow the access.

The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's group access list, and the group permissions allow the access.

Neither the effective user ID nor effective group ID and group access list of the process match the corresponding user ID and group ID of the file, but the permissions for ''other users'' allow access.

Otherwise, permission is denied.

SEE ALSO

intro(3), **strerror**(3).

NAME

ioctl − control device

SYNOPSIS

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

int ioctl(int *d***, int** *request***, void ****argp***)** **(Minix)**
int ioctl(int *d***, ioreq_t** *request***, void ****argp***)** **(Minix-vmd)**

DESCRIPTION

Ioctl performs a variety of functions on open descriptors. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl** requests. The writeups of various devices in section 4 discuss how **ioctl** applies to them.

An ioctl *request* has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an ioctl *request* are located in the file *<sys/ioctl.h>*.

RETURN VALUE

If an error has occurred, a value of −1 is returned and **errno** is set to indicate the error.

ERRORS

Ioctl will fail if one or more of the following are true:

[EBADF]	*D* is not a valid descriptor.
[ENOTTY]	*D* is not associated with a character special device.
[ENOTTY]	The specified request does not apply to the kind of object that the descriptor *d* references.
[EINVAL]	*Request* or *argp* is not valid.

SEE ALSO

execve(2), **fcntl**(2), **mt**(4), **tty**(4), **intro**(4).

NAME

kill − send signal to a process

SYNOPSIS

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

int kill(pid_t *pid***, int** *sig***)**

DESCRIPTION

Kill sends the signal *sig* to a process, specified by the process number *pid*. *Sig* may be one of the signals specified in **sigaction**(2), or it may be 0, in which case error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all processes in the sender's process group.

If the process number is −1 and the user is the super-user, the signal is broadcast universally except to **init** and the process sending the signal. If the process number is −1 and the user is not the super-user, the signal is broadcast universally to all processes with the same uid as the user except the process sending the signal. No error is returned if any process could be signaled.

If the process number is negative but not −1, the signal is sent to all processes whose process group ID is equal to the absolute value of the process number.

Processes may send signals to themselves.

RETURN VALUE

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

Kill will fail and no signal will be sent if any of the following occur:

[EINVAL] *Sig* is not a valid signal number.

[ESRCH] No process can be found corresponding to that specified by *pid*.

[ESRCH] The process id was given as 0 but the sending process does not have a process group.

[EPERM] The sending process is not the super-user and its effective user id does not match the effective user-id of the receiving process. When signaling a process group, this error was returned if any members of the group could not be signaled.

SEE ALSO

getpid(2), **getpgrp**(2), **sigaction**(2), **raise**(3).

NAME

link − make a hard link to a file

SYNOPSIS

#include <unistd.h>

int link(const char ***_name1_, const char ***_name2_**)**

DESCRIPTION

A hard link to _name1_ is created; the link has the name _name2_. _Name1_ must exist.

With hard links, both _name1_ and _name2_ must be in the same file system. _Name1_ must not be a directory. Both the old and the new _link_ share equal access and rights to the underlying object.

RETURN VALUE

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

Link will fail and no link will be created if one or more of the following are true:

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

[ENAMETOOLONG] A path name exceeds PATH_MAX characters.

[ENOENT] A component of either path prefix does not exist.

[EACCES] A component of either path prefix denies search permission.

[EACCES] The requested link requires writing in a directory with a mode that denies write permission.

[ELOOP] Too many symbolic links were encountered in translating one of the pathnames. (Minix-vmd)

[ENOENT] The file named by _name1_ does not exist.

[EEXIST] The link named by _name2_ does exist.

[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 to make the directory entry.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EFAULT] One of the pathnames specified is outside the process's allocated address space.

SEE ALSO

symlink(2), **unlink**(2).

NAME

lseek − move read/write pointer

SYNOPSIS

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

#define SEEK_SET    0    /* offset is absolute */
#define SEEK_CUR    1    /* relative to current position */
#define SEEK_END    2    /* relative to end of file */

off_t lseek(int d, off_t offset, int whence)
```

DESCRIPTION

The descriptor *d* refers to a file or device open for reading and/or writing. **Lseek** sets the file pointer of *d* as follows:

If *whence* is SEEK_SET, the pointer is set to *offset* bytes.

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

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

Upon successful completion, the resulting pointer location as measured in bytes from beginning of the file is returned. Some devices are incapable of seeking. The value of the pointer associated with such a device is undefined.

NOTES

Seeking far beyond the end of a file, then writing, creates a gap or hole, which occupies no physical space and reads as zeros.

RETURN VALUE

Upon successful completion, the current file pointer value is returned. Otherwise, a value of −1 is returned and **errno** is set to indicate the error.

ERRORS

Lseek will fail and the file pointer will remain unchanged if:

[EBADF] *Fildes* is not an open file descriptor.

[ESPIPE] *Fildes* is associated with a pipe or a socket.

[EINVAL] *Whence* is not a proper value.

SEE ALSO

fcntl(2), **open**(2).

BUGS

This document’s use of *whence* is incorrect English, but maintained for historical reasons.

NAME

mkdir − make a directory file

SYNOPSIS

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

int mkdir(const char ***_path_, mode_t** _mode_**)**

DESCRIPTION

Mkdir creates a new directory file with name _path_. The mode of the new file is initialized from _mode_. (The protection part of the mode is modified by the process's mode mask; see **umask**(2)).

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.

The low-order 9 bits of mode are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared. See **umask**(2).

RETURN VALUE

A 0 return value indicates success. A −1 return value indicates an error, and an error code is stored in **errno.**

ERRORS

Mkdir will fail and no directory will be created if:

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

[ENAMETOOLONG]
The path name exceeds 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. (Minix-vmd)

[EROFS] The named file resides on a read-only file system.

[EEXIST] The named file exists.

[ENOSPC] The directory in which the entry for the new directory is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] The new directory cannot be created because there there is no space left on the file system that will contain the directory.

[ENOSPC] There are no free inodes on the file system on which the directory is being created.

[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.

SEE ALSO

chmod(2), **stat**(2), **umask**(2).

NAME

mknod, mkfifo − make a special file

SYNOPSIS

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

int mknod(const char ***_path_, mode_t** _mode_**, dev_t** _dev_**)**
int mkfifo(const char ***_path_, mode_t** _mode_**)**

DESCRIPTION

Mknod creates a new file whose name is _path_. The mode of the new file (including special file bits) is initialized from _mode_, as defined in <_sys/stat.h_>. (The protection part of the mode is modified by the process's mode mask (see **umask**(2))). The first block pointer of the i-node is initialized from _dev_ and is used to specify which device the special file refers to.

If mode indicates a block or character special file, _dev_ is the device number of a character or block I/O device. The low eight bits of the device number hold the minor device number that selects a device among the devices governed by the same driver. The driver is selected by the major device number, the next eight bits of the device number.

If _mode_ does not indicate a block special or character special device, _dev_ is ignored. (For example, when creating a ‘‘fifo’’ special file.)

Mknod may be invoked only by the super-user, unless it is being used to create a fifo.

The call **mkfifo(**_path_**,** _mode_**)** is equivalent to

> **mknod(**_path_**, (**_mode_ **& 0777) | S_IFIFO, 0)**

RETURN VALUE

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

Mknod will fail and the file mode will be unchanged if:

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

[ENAMETOOLONG]
The path name exceeds 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. (Minix-vmd)

[EPERM] The process's 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 named file resides on a read-only file system.

[EEXIST] The named file exists.

[EFAULT] _Path_ points outside the process's allocated address space.

SEE ALSO

chmod(2), **stat**(2), **umask**(2).

NAME

mount, umount − mount or umount a file system

SYNOPSIS

#include <unistd.h>
#include <sys/mount.h>

int mount(char ****special*, char ****name***, int** *flag***)**
int umount(char ****name*)**

DESCRIPTION

Mount() tells the system that the file system *special* is to be mounted on the file *name*, effectively overlaying *name* with the file tree on *special*. *Name* may of any type, except that if the root of *special* is a directory, then *name* must also be a directory. *Special* must be a block special file, except for loopback mounts. For loopback mounts a normal file or directory is used for *special*, which must be seen as the root of a virtual device. *Flag* is 0 for a read-write mount, 1 for read-only.

Umount() removes the connection between a device and a mount point, *name* may refer to either of them. If more than one device is mounted on the same mount point then unmounting at the mount point removes the last mounted device, unmounting a device removes precisely that device. The unmount will only succeed if none of the files on the device are in use.

Both calls may only be executed by the super-user.

SEE ALSO

mount(1), **umount**(1).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

open − open a file for reading or writing, or create a new file

SYNOPSIS

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

int open(const char *path, int flags [, mode_t mode])
```

DESCRIPTION

Open opens the file *path* for reading and/or writing, as specified by the *flags* argument and returns a descriptor for that file. The *flags* argument may indicate the file is to be created if it does not already exist (by specifying the O_CREAT flag), in which case the file is created with mode *mode* as described in **chmod**(2) and modified by the process' umask value (see **umask**(2)).

Path is the address of a string of ASCII characters representing a path name, terminated by a null character. The flags specified are formed by *or*'ing the following values

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NONBLOCK	do not block on open
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to 0
O_EXCL	error if create and file exists

Opening a file with O_APPEND set causes each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, the file is truncated to zero length. If O_EXCL is set with O_CREAT, then if the file already exists, the open returns an error. This can be used to implement a simple exclusive access locking mechanism. If O_EXCL is set and the last component of the pathname is a symbolic link, the open will fail even if the symbolic link points to a non-existent name. If the O_NONBLOCK flag is specified and the open call would result in the process being blocked for some reason, the open returns immediately.

Upon successful completion a non-negative integer termed a file descriptor is returned. The file pointer used to mark the current position within the file is set to the beginning of the file.

The new descriptor is set to remain open across **execve** system calls; see **close**(2).

The system imposes a limit on the number of descriptors open simultaneously by one process.

ERRORS

The named file is opened unless one or more of the following are true:

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

[ENAMETOOLONG] The path name exceeds PATH_MAX characters.

[ENOENT] O_CREAT is not set and the named file does not exist.

[ENOENT] A component of the path name that must exist does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] The required permissions (for reading and/or writing) are denied for the named file.

[EACCES] O_CREAT is specified, the file does not exist, and the directory in which it is to be created does not permit writing.

[EACCES] A device to be opened for writing is physically write protected.

[ELOOP] Too many symbolic links were encountered in translating the pathname. (Minix-vmd)

[EISDIR] The named file is a directory, and the arguments specify it is to be opened for writing.

[EROFS] The named file resides on a read-only file system, and the file is to be modified.

[EMFILE] The system limit for open file descriptors per process has already been reached.

[ENFILE] The system file table is full.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[ENOSPC] O_CREAT is specified, the file does not exist, and the directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] O_CREAT is specified, the file does not exist, and there are no free inodes on the file system on which the file is being created.

[EIO] An I/O error occurred while making the directory entry or allocating the inode for O_CREAT.

[EFAULT] *Path* points outside the process's allocated address space.

[EEXIST] O_CREAT and O_EXCL were specified and the file exists.

SEE ALSO

chmod(2), **close**(2), **dup**(2), **fcntl**(2), **lseek**(2), **read**(2), **write**(2), **umask**(2).

NAME

pause − stop until signal

SYNOPSIS

#include <unistd.h>

int pause(void)

DESCRIPTION

Pause never returns normally. It is used to give up control while waiting for a signal from **kill**(2) or the alarm timer, see **alarm**(2). Upon termination of a signal handler started during a **pause,** the **pause** call will return.

RETURN VALUE

Always returns −1.

ERRORS

Pause always returns:

[EINTR] The call was interrupted.

SEE ALSO

alarm(2), **kill**(2), **sigsuspend**(2).

NAME

pipe − create an interprocess communication channel

SYNOPSIS

#include <unistd.h>

int pipe(int *fildes***[2])**

DESCRIPTION

The **pipe** system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor *fildes*[1] up to PIPE_MAX bytes of data are buffered before the writing process is suspended. A read using the descriptor *fildes*[0] will pick up the data.

PIPE_MAX equals 7168 under Minix, but note that most systems use 4096.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent **fork** calls) will pass data through the pipe with **read** and **write** calls.

The shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

The signal SIGPIPE is generated if a write on a pipe with only one end is attempted.

RETURN VALUE

The function value zero is returned if the pipe was created; −1 if an error occurred.

ERRORS

The **pipe** call will fail if:

[EMFILE]	Too many descriptors are active.
[ENFILE]	The system file table is full.
[ENOSPC]	The pipe file system (usually the root file system) has no free inodes.
[EFAULT]	The *fildes* buffer is in an invalid area of the process's address space.

SEE ALSO

sh(1), **read**(2), **write**(2), **fork**(2).

NOTES

Writes may return ENOSPC errors if no pipe data can be buffered, because the pipe file system is full.

BUGS

Should more than PIPE_MAX bytes be necessary in any pipe among a loop of processes, deadlock will occur.

NAME

ptrace − process trace

SYNOPSIS

#include <sys/types.h>
#include <sys/signal.h>
#include <sys/ptrace.h>

int ptrace(int *request***, pid_t** *pid***, long** *addr***, long** *data***)**

DESCRIPTION

Note: This manual page has no relation to Minix. Someone who knows ptrace() has to check, or rewrite, this page. (kjb)

Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like illegal instruction or externally generated like interrupt. See **sigaction**(2) for the list. Then the traced process enters a stopped state and its parent is notified via **wait**(2). When the child is in the stopped state, its core image can be examined and modified using **ptrace**. If desired, another **ptrace** request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

PT_TRACE_ME
: This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

PT_READ_I, PT_READ_D
: The word in the child process's address space at *addr* is returned. If I and D space are separated (e.g. historically on a pdp-11), request PT_READ_I indicates I space, PT_READ_D D space. *Addr* must be even on some machines. The child must be stopped. The input *data* is ignored.

PT_READ_U
: The word of the system's per-process data area corresponding to *addr* is returned. *Addr* must be even on some machines and less than 512. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

PT_WRITE_I, PT_WRITE_D
: The given *data* is written at the word in the process's address space corresponding to *addr*, which must be even on some machines. No useful value is returned. If I and D space are separated, request PT_WRITE_I indicates I space, PT_WRITE_D D space. Attempts to write in pure procedure fail if another process is executing the same file.

PT_WRITE_U
: The process's system data is written, as it is read with request PT_READ_U. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

PT_CONTINUE
: The *data* argument is taken as a signal number and the child's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped.

PT_KILL
: The traced process terminates.

PT_STEP
Execution continues as in request PT_CONTINUE; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. (On the VAX-11 the T-bit is used and just one instruction is executed.) This is part of the mechanism for implementing breakpoints.

As indicated, these calls (except for request PT_TRACE_ME) can be used only when the subject process has stopped. The **wait** call is used to determine when a process stops; in such a case the termination status returned by **wait** has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, **ptrace** inhibits the set-user-id and set-group-id facilities on subsequent **execve**(2) calls. If a traced process calls **execve**, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

On a VAX-11, word also means a 32-bit integer, but the even restriction does not apply.

RETURN VALUE

A 0 value is returned if the call succeeds. If the call fails then a −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EIO]	The request code is invalid.
[ESRCH]	The specified process does not exist.
[EIO]	The given signal number is invalid.
[EIO]	The specified address is out of bounds.
[EPERM]	The specified process cannot be traced.

SEE ALSO

wait(2), **sigaction**(2), **mdb**(1).

BUGS

Ptrace is unique and arcane; it should be replaced with a special file that can be opened and read and written. The control functions could then be implemented with **ioctl**(2) calls on this file. This would be simpler to understand and have much higher performance.

The request PT_TRACE_ME call should be able to specify signals that are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use illegal instruction signals at a very high rate) could be efficiently debugged.

The error indication, −1, is a legitimate function value; **errno**, (see **intro**(2)), can be used to disambiguate.

It should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

NAME

read − read input

SYNOPSIS

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

ssize_t read(int *d*, **void ****buf*, **size_t** *nbytes***)**

DESCRIPTION

Read attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*.

On objects capable of seeking, the **read** starts at a position given by the pointer associated with *d* (see **lseek**(2)). 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 such an object is undefined.

Upon successful completion, **read** 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 has that many bytes left before the end-of-file, but in no other case.

If the returned value is 0, then end-of-file has been reached.

RETURN VALUE

If successful, the number of bytes actually read is returned. Otherwise, a −1 is returned and the global variable **errno** is set to indicate the error.

ERRORS

Read will fail if one or more of the following are true:

[EBADF]	*D* is not a valid descriptor open for reading.
[EFAULT]	*Buf* points outside the allocated address space.
[EIO]	An I/O error occurred while reading from the file system.
[EINTR]	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
[EAGAIN]	The file was marked for non-blocking I/O, and no data were ready to be read.

SEE ALSO

dup(2), **fcntl**(2), **open**(2), **pipe**(2), **write**(2).

NAME

reboot − close down the system or reboot

SYNTAX

#define _MINIX_SOURCE 1

#include <unistd.h>

int reboot(int *how***, ...)**

DESCRIPTION

Reboot() is used to close down the system. It allows several ways of shutting down depending on *how*:

reboot(RBT_HALT)
Halt the system and return to the monitor prompt.

reboot(RBT_REBOOT)
Reboot the system by letting the monitor execute the "boot" command.

reboot(RBT_PANIC)
Cause a system panic. This is not normally done from user mode, but by servers using the **sys_abort()** kernel call.

reboot(RBT_MONITOR, *code, length***)**
Halt the system and let the monitor execute the given code of the given length. (*code* is of type **char *** and *length* of type **size_t**.)

reboot(RBT_RESET)
Reboot the system with a hardware reset.

Reboot() may only be executed by the super-user.

DIAGNOSTICS

If the call succeeds, it never returns. If something went wrong, the return value is -1 and an error is indicated by **errno**.

SEE ALSO

shutdown(8), **reboot**(8), **halt**(8), **sync**(1).

NOTES

Minix can not return to the monitor if running in real mode, or if started from MS-DOS. This means that most of the reboot functions will change to a reset.

AUTHOR

Edvard Tuinder (v892231@si.hhs.NL)

NAME

rename − change the name of a file

SYNOPSIS

#include <stdio.h>

int rename(const char ****from*, const char ****to***)**

DESCRIPTION

Rename causes the link named *from* to be renamed as *to*. If *to* exists, then 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.

Rename 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.

RETURN VALUE

A 0 value is returned if the operation succeeds, otherwise **rename** returns −1 and the global variable **errno** indicates the reason for the failure.

ERRORS

Rename will fail and neither of the argument files will be affected if any of the following are true:

[ENAMETOOLONG]	A path name exceeds PATH_MAX 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 in 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 *to* 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. (Minix-vmd)
[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).
[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 in a directory on a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.
[EINVAL]	*From* is a parent directory of *to*, or an attempt is made to rename ''.'' or ''..''.
[ENOTEMPTY]	*To* is a directory and is not empty.

SEE ALSO

open(2)

NAME

rmdir − remove a directory file

SYNOPSIS

#include <unistd.h>

int rmdir(const char ***_path_)**

DESCRIPTION

Rmdir removes a directory file whose name is given by _path._ The directory must not have any entries other than . and ...

RETURN VALUE

A 0 is returned if the remove succeeds; otherwise a −1 is returned and an error code is stored in the global location _errno_ .

ERRORS

The named file is removed unless one or more of the following are true:

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

[ENAMETOOLONG] The path name exceeds PATH_MAX characters.

[ENOENT] The named directory does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname. (Minix-vmd)

[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.

SEE ALSO

mkdir(2), **unlink**(2).

NAME

setsid, getpgrp − create process group, get process group id

SYNOPSIS

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

pid_t setsid(void)
pid_t getpgrp(void)

DESCRIPTION

Setsid() creates a new session if the calling process is not already a session leader. The calling process becomes the session leader of a new process group and the process group ID of this new process group will be equal to the process ID of the new session leader. The process group ID is inherited on a **fork()**.

Getpgrp() returns the process group ID of the calling process.

SEE ALSO

kill(2), **termios**(3), **tty**(4).

DIAGNOSTICS

Setsid() returns the new process group ID on success, or −1 with **errno** set to **EPERM** if the process is already a session leader.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

setuid, setgid − set user or group ID’s

SYNOPSIS

#include <sys/types.h>

int setuid(uid_t *uid***)**
int setgid(gid_t *gid***)**

DESCRIPTION

Setuid sets the real and effective user ID’s of the current process to *uid*. Unprivileged users may only change both user ID’s to the real user ID; only the super-user may make other changes. **Setgid** does the same for the real and effective group ID’s.

Minix-vmd allows an unprivileged user to change ID’s to the original real or effective ID as they were at the time the process was executed. **Setgid** may also set the group ID’s to any of the additional group ID’s. If one of the remembered user ID’s was 0 then any user or group ID may be chosen.

RETURN VALUE

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

[EPERM] The current process is not the super-user and a change other than one of the allowed changes was attempted.

SEE ALSO

getuid(2), **getgid**(2).

NAME

sigaction, signal − manage signal state and handlers

SYNOPSIS

#include <signal.h>

int sigaction(int *sig*, **const struct sigaction ****act*, **struct sigaction ****oact*)
void (*signal(int *sig*, **void (****handler*)**(int)))(int);**

DESCRIPTION

Sigaction() is used to examine, set, or modify the attributes of a signal. The argument *sig* is the signal in question. The *act* argument points to a structure containing the new attributes of the signal, the structure pointed to by *oact* will receive the old attributes that were in effect before the call.

The *act* and *oact* arguments may be **NULL** to indicate that either no new attributes are to be set, or that the old attributes are not of interest.

The structure containing the signal attributes is defined in <signal.h> and looks like this:

```
struct sigaction {
    void        (*sa_handler)(int sig);
    sigset_t    sa_mask;
    int         sa_flags;
};
```

The **sa_handler** field contains the address of a signal handler, a function that is called when the process is signalled, or one of these special constants:

SIG_DFL Default signal handling is to be performed. This usually means that the process is killed, but some signals may be ignored by default.

SIG_IGN Ignore the signal.

The **sa_mask** field indicates a set of signals that must be blocked when the signal is being handled. Whether the signal *sig* itself is blocked when being handled is not controlled by this mask. The mask is of a "signal set" type that is to be manipulated by the **sigset**(3) functions.

How the signal is handled precisely is specified by bits in **sa_flags**. If none of the flags is set then the handler is called when the signal arrives. The signal is blocked during the call to the handler, and unblocked when the handler returns. A system call that is interrupted returns **−1** with **errno** set to **EINTR**. The following bit flags can be set to modify this behaviour:

SA_RESETHAND
Reset the signal handler to **SIG_DFL** when the signal is caught.

SA_NODEFER Do not block the signal on entry to the handler.

SA_COMPAT Handle the signal in a way that is compatible with the the old **signal()** call.

The old **signal()** signal system call sets a signal handler for a given signal and returns the old signal handler. No signals are blocked, the flags are **SA_RESETHAND | SA_NODEFER | SA_COMPAT**. New code should not use **signal()**. Note that **signal()** and all of the **SA_*** flags are Minix extensions.

Signal handlers are reset to **SIG_DFL** on an **execve**(2). Signals that are ignored stay ignored.

Signals

Minix knows about the following signals:

signal	num	notes	description
SIGHUP	1	k	Hangup
SIGINT	2	k	Interrupt (usually DEL or CTRL−C)
SIGQUIT	3	kc	Quit (usually CTRL−\)
SIGILL	4	kc	Illegal instruction
SIGTRAP	5	xkc	Trace trap
SIGABRT	6	kc	Abort program

SIGFPE	8	k	Floating point exception
SIGKILL	9	k	Kill
SIGUSR1	10	k	User defined signal #1
SIGSEGV	11	kc	Segmentation fault
SIGUSR2	12	k	User defined signal #2
SIGPIPE	13	k	Write to a pipe with no reader
SIGALRM	14	k	Alarm clock
SIGTERM	15	k	Terminate (default for kill(1))
SIGCHLD	17	pvi	Child process terminated
SIGCONT	18	p	Continue if stopped
SIGSTOP	19	ps	Stop signal
SIGTSTP	20	ps	Interactive stop signal
SIGTTIN	21	ps	Background read
SIGTTOU	22	ps	Background write
SIGWINCH	23	xvi	Window size change

The letters in the notes column indicate:

k The process is killed if the signal is not caught.

c The signal causes a core dump.

i The signal is ignored if not caught.

v Only Minix-vmd implements this signal.

x Minix extension, not defined by POSIX.

p These signals are not implemented, but POSIX requires that they are defined.

s The process should be stopped, but is killed instead.

The **SIGKILL** signal cannot be caught or ignored. The **SIGILL** and **SIGTRAP** signals cannot be automatically reset. The system silently enforces these restrictions. This may or may not be reflected by the attributes of these signals and the signal masks.

Types

POSIX prescribes that <sys/types.h> has the following definition:

typedef int (*sighandler_t)(int)

With this type the following declarations can be made:

sighandler_t sa_handler;
sighandler_t signal(int *sig***, sighandler_t** *handler***);**

This may help you to understand the earlier declarations better. The **sighandler_t** type is also very useful in old style C code that is compiled by a compiler for standard C.

SEE ALSO

kill(1), **kill**(2), **pause**(2), **sigprocmask**(2), **sigsuspend**(2), **sigpending**(2), **sigset**(3).

DIAGNOSTICS

Sigaction() returns **0** on success or **−1** on error. **Signal()** returns the old handler on success or **SIG_ERR** on error. The error code may be:

EINVAL Bad signal number.

EFAULT Bad *act* or *oact* addresses.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

sigpending − report pending signals

SYNOPSIS

#include <signal.h>

int sigpending(sigset_t ***set*)**

DESCRIPTION

Sigpending() returns the set of signals that are waiting to be delivered. They are currently blocked by the signal mask.

SEE ALSO

sigaction(2), **sigprocmask**(2), **sigsuspend**(2), **sigset**(3).

DIAGNOSTICS

Returns **0** on success and **−1** on error. The only possible error code is **EFAULT** for a bad *set* address.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

sigprocmask − manipulate the signal mask

SYNOPSIS

#include <signal.h>

int sigprocmask(int *how***, const sigset_t ****set***, sigset_t ****oset***)**

DESCRIPTION

Sigprocmask() examines or manipulates the signal mask. This mask is the set of signals that are currently blocked. The *how* argument determines the action that must be performed. In all cases the signal set referenced by *oset*, if not **NULL**, will be used to receive the old signal mask. The *set* argument, if not **NULL**, will be used to set or modify the current signal mask.

How can be one of:

SIG_BLOCK Add the signals referenced by *set* to the mask.

SIG_UNBLOCK
Remove the signals referenced by *set* from the mask.

SIG_SETMASK Set the signal mask to the set referenced by *set*.

The value of *how* is ignored if *set* is **NULL**.

SEE ALSO

sigaction(2), **sigpending**(2), **sigsuspend**(2), **sigset**(3).

DIAGNOSTICS

Returns **0** on success and **−1** on error. The error code is **EFAULT** for a bad *set* or *oset* address, or **EINVAL** for a bad *how* argument.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

sigsuspend − suspend until signalled

SYNOPSIS

#include <signal.h>

int sigsuspend(const sigset_t ****set*)**

DESCRIPTION

Sigsuspend() installs the signal mask referenced by *set* and suspends the process until signalled. The signal is handled, the signal mask is restored to the value it had before the **sigsuspend()** call and call returns.

SEE ALSO

pause(2), **sigaction**(2), **sigpending**(2), **sigprocmask**(2), **sigset**(3).

DIAGNOSTICS

Sigsuspend() never returns normally, so it always returns **−1**. The error code is either **EINTR** indicating that a signal has arrived, or **EFAULT** for a bad *set* address.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

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*)** **(Minix-vmd)**
int fstat(int *fd*, **struct stat ****buf***)**

DESCRIPTION

Stat obtains information about the file *path*. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be reachable.

Lstat is like **stat** except in the case where the named file is a symbolic link, in which case **lstat** returns information about the link, while **stat** returns information about the file the link references. (Minix-vmd)

Fstat obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an **open** call. Pipe descriptors look like named pipes with a link count of zero. The st_size field of pipes or named pipes shows the amount of bytes currently buffered in the pipe.

Buf is a pointer to a **stat** structure into which information is placed concerning the file. The contents of the structure pointed to by *buf* is as follows:

```
struct stat {
        dev_t       st_dev;       /* device inode resides on */
        ino_t       st_ino;       /* this inode's number */
        mode_t      st_mode;      /* file mode, protection bits, etc. */
        nlink_t     st_nlink;     /* number or hard links to the file */
        uid_t       st_uid;       /* user-id of the file's owner */
        gid_t       st_gid;       /* group-id of the file's owner */
        dev_t       st_rdev;      /* the device type, for inode that is device */
        off_t       st_size;      /* total size of file */
        time_t      st_atime;     /* time of last access */
        time_t      st_mtime;     /* time of last data modification */
        time_t      st_ctime;     /* time of last file status change */
};
```

st_atime   Time when file data was last read or modified. Changed by the following system calls: **mknod**(2), **utime**(2), **read**(2), and **write**(2). For reasons of efficiency, st_atime is not set when a directory is searched, although this would be more logical.

st_mtime   Time when data was last modified. It is not set by changes of owner, group, link count, or mode. Changed by the following system calls: **mknod**(2), **utime**(2), **write**(2).

st_ctime   Time when file status was last changed. It is set both both by writing and changing the i-node. Changed by the following system calls: **chmod**(2) **chown**(2), **link**(2), **mknod**(2), **rename**(2), **unlink**(2), **utime**(2), **write**(2).

The file type information in **st_mode** has bits:

```
#define S_IFMT        0170000     /* type of file */
#define   S_IFIFO     0010000     /* named pipe */
#define   S_IFCHR     0020000     /* character special */
#define   S_IFDIR     0040000     /* directory */
#define   S_IFBLK     0060000     /* block special */
#define   S_IFREG     0100000     /* regular */
#define   S_IFLNK     0120000     /* symbolic link (Minix-vmd) */
```

The mode bits 0007777 encode set-uid/gid bits and permission bits (see **chmod**(2)).

RETURN VALUE

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

Stat and **lstat** will fail if one or more of the following are true:

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

[ENAMETOOLONG] The path name exceeds PATH_MAX characters.

[ENOENT] The named file 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. (Minix-vmd)

[EFAULT] *Buf* or *name* points to an invalid address.

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

Fstat will fail if one or both of the following are true:

[EBADF] *Fildes* 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.

SEE ALSO

chmod(2), **chown**(2), **utime**(2).

NAME

sync − update super-block

SYNOPSIS

#include <unistd.h>

int sync(void)

DESCRIPTION

Sync causes all information in the file system buffers that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

SEE ALSO

sync(8), **update**(8).

NAME

time, stime − get/set date and time

SYNOPSIS

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

time_t time(time_t ***_tp_)**
int stime(time_t ***_tp_)**

DESCRIPTION

The system's notion of the current Greenwich time is obtained with the **time** call, and set with the **stime** call. The time is expressed in seconds since midnight (0 hour), January 1, 1970. The time is both returned by **time** and stored in the variable pointed to by *tp* unless *tp* is the null pointer.

Stime obtains the time to set from the variable pointed to by *tp*.

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

RETURN

A 0 return value from **stime** indicates that the call succeeded. **Time** returns the current time on success. A −1 return value indicates an error occurred, and in this case an error code is stored into the global variable **errno**.

ERRORS

The following error codes may be set in **errno**:

[EFAULT] The *tp* address referenced invalid memory.

[EPERM] A user other than the super-user attempted to set the time.

SEE ALSO

date(1), **ctime**(3).

NAME

times − get process times

SYNOPSIS

#include <sys/types.h>
#include <sys/times.h>
#include <time.h>

int times(struct tms ***_buffer_)**

DESCRIPTION

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/CLOCKS_PER_SEC seconds.

This is the structure returned by **times**:

```
struct tms {
        clock_t     tms_utime;      /* user time for this process */
        clock_t     tms_stime;      /* system time for this process */
        clock_t     tms_cutime;     /* children’s user time */
        clock_t     tms_cstime;     /* children’s system time */
};
```

The user time is the number of clock ticks used by a process on its own computations. The system time is the number of clock ticks spent inside the kernel on behalf of a process. This does not include time spent waiting for I/O to happen, only actual CPU instruction times.

The children times are the sum of the children’s process times and their children’s times.

RETURN

Times returns 0 on success, otherwise −1 with the error code stored into the global variable **errno**.

ERRORS

The following error code may be set in **errno**:

[EFAULT] The address specified by the _buffer_ parameter is not in a valid part of the process address space.

SEE ALSO

time(1), **wait**(2), **time**(2).

NAME

umask − set file creation mode mask

SYNOPSIS

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

mode_t umask(mode_t *mask***)**

DESCRIPTION

Umask sets the process's file mode creation mask to *mask* and returns the previous value of the mask. The low-order 9 bits of *mask* are used whenever a file is created, clearing corresponding bits in the file mode (see **chmod**(2)). This clearing allows each user to restrict the default access to his files.

The value is initially 022 (write access for owner only). The mask is inherited by child processes.

RETURN VALUE

The previous value of the file mode mask is returned by the call.

SEE ALSO

chmod(2), **mknod**(2), **open**(2).

NAME

uname − get system info

SYNOPSIS

#include <sys/utsname.h>

int uname(struct utsname *name)

DESCRIPTION

Uname() fills a struct utsname with system information. This structure is described in <sys/utsname.h> as follows:

```
struct utsname {
      char   sysname[15+1];          /* System name */
      char   nodename[255+1];        /* Node/Network name */
      char   release[11+1];          /* O.S. release */
      char   version[7+1];           /* O.S. version */
      char   machine[11+1];          /* Machine hardware */
      char   arch[11+1];             /* Architecture */
};
```

The strings are always null terminated, and may be of a different length then shown here. The first five are required by POSIX, the last is Minix specific.

SEE ALSO

uname(1).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

unlink − remove directory entry

SYNOPSIS

#include <unistd.h>

int unlink(const char *path)

DESCRIPTION

Unlink removes the entry for the file *path* from its directory. If this entry was the last link to the file, and no process has the file open, then all resources associated with the file are reclaimed. If, however, the file was open in any process, the actual resource reclamation is delayed until it is closed, even though the directory entry has disappeared.

RETURN VALUE

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 *unlink* succeeds unless:

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

[ENAMETOOLONG] The path name exceeds PATH_MAX characters.

[ENOENT] The named 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.

[ELOOP] Too many symbolic links were encountered in translating the pathname. (Minix-vmd)

[EPERM] The named file is a directory.

[EPERM] The directory containing the file is marked sticky, and neither the containing directory nor the file to be removed are owned by the effective user ID. (Minix-vmd)

[EBUSY] The entry to be unlinked 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 named file resides on a read-only file system.

[EFAULT] *Path* points outside the process's allocated address space.

SEE ALSO

close(2), **link**(2), **rmdir**(2).

NAME

utime − set file times

SYNOPSIS

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

int utime(const char ***_file_, struct utimbuf ***_times_**)**

DESCRIPTION

The **utime** call uses the accessed and updated times from the utimbuf structure pointed to by _times_ to set the corresponding recorded times for _file_.

Struct utimbuf is defined in <utime.h> as follows:

```
struct utimbuf {
        time_t      actime;       /* access time */
        time_t      modtime;      /* modification time */
};
```

The caller must be the owner of the file or the super-user. The inode-changed time of the file is set to the current time.

RETURN VALUE

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

Utime will fail if one or more of the following are true:

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

[EINVAL] The pathname contains a character with the high-order bit set.

[ENAMETOOLONG] The path name exceeds PATH_MAX characters.

[ENOENT] The named file does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname. (Minix-vmd)

[EPERM] The process is not super-user and not the owner of the file.

[EACCES] Search permission is denied for a component of the path prefix.

[EROFS] The file system containing the file is mounted read-only.

[EFAULT] _File_ or _times_ points outside the process’s allocated address space.

[EIO] An I/O error occurred while reading or writing the affected inode.

SEE ALSO

stat(2).

NAME

wait, waitpid − wait for process to terminate

SYNOPSIS

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

pid_t wait(int ****status*)**
pid_t waitpid(pid_t *pid***, int ****status***, int** *options***)**

DESCRIPTION

Wait causes its caller to delay until a signal is received or one of its child processes terminates. If any child has died since the last **wait**, return is immediate, returning the process id and exit status of one of the terminated children. If there are no children, return is immediate with the value −1 returned.

On return from a successful **wait** call, *status* is nonzero, and the high byte of *status* contains the low byte of the argument to **exit** supplied by the child process; the low byte of *status* contains the termination status of the process. A more precise definition of the *status* word is given in *<sys/wait.h>*. If **wait** can called with a null pointer argument to indicate that no status need be returned.

Waitpid provides an alternate interface for programs that must not block when collecting the status of child processes, or that wish to wait for one particular child. The pid parameter is the process ID of the child to wait for, −1 for any child. The *status* parameter is defined as above. The *options* parameter is used to indicate the call should not block if there are no processes that wish to report status (WNOHANG), and/or that children of the current process that are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal should also have their status reported (WUNTRACED). (Job control is not implemented for Minix, but these symbold and signals are.)

When the WNOHANG option is specified and no processes wish to report status, **waitpid** returns −1 with **errno** set to **EAGAIN**. The WNOHANG and WUNTRACED options may be combined by *or'*ing the two values.

NOTES

The call **wait(&***status***)** is equivalent to **waitpid(−1, &***status***, 0)**.

See **sigaction**(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process that has not terminated and can be restarted; see **ptrace**(2). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

<sys/wait.h> defines a number of macros that operate on a status word:

WIFEXITED(*status***)**
True if normal exit.

WEXITSTATUS(*status***)**
Exit status if the process returned by a normal exit, zero otherwise.

WTERMSIG(*status***)**
Signal number if the process died by a signal, zero otherwise.

WIFSIGNALED(*status***)**
True if the process died by a signal.

WIFSTOPPED(*status***)**
True if the process is stopped. (Never true under Minix.)

WSTOPSIG(*status***)**
Signal number of the signal that stopped the process.

RETURN VALUE

If **wait** returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of −1 is returned and **errno** is set to indicate the error.

Waitpid returns −1 if there are no children not previously waited for, if the process that it wants to wait for doesn't exist, or if WNOHANG is specified and there are no stopped or exited children.

ERRORS

Wait will fail and return immediately if one or more of the following are true:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EFAULT] The *status* argument points to an illegal address.

[EAGAIN] **Waitpid** is called with the WNOHANG option and no child has exited yet.

SEE ALSO

execve(2), **exit**(2), **sigaction**(2).

NAME

write − write output

SYNOPSIS

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

ssize_t write(int *d*, **const void** *****buf*, **size_t** *nbytes***)**

DESCRIPTION

Write attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*.

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

Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

When using non-blocking I/O on objects such as TCP/IP channels that are subject to flow control, **write** may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise a −1 is returned and the global variable **errno** is set to indicate the error.

ERRORS

Write will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF]	*D* is not a valid descriptor open for writing.
[EPIPE]	An attempt is made to write to a pipe that is not open for reading by any process.
[EPIPE]	An attempt is made to write to a TCP channel that is not connected to a peer socket.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
[EFAULT]	Part of the data to be written to the file points outside the process's allocated address space.
[ENOSPC]	There is no free space remaining on the file system containing the file.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EAGAIN]	The file was marked for non-blocking I/O, and no data could be written immediately.

SEE ALSO

fcntl(2), **lseek**(2), **open**(2), **pipe**(2), **read**(2).