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 \’"<>();˜$ˆ&*|{}[]? 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).