ULTRIX

digital

Reference Pages
Section 2: System Calls

Order Number: AA–LY15B–TE

ULTRIX

Reference Pages Section 2: System Calls

Order Number: AA-LY15B-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

This manual defines system calls (entries into the ULTRIX kernel) that are used by all programmers for both RISC and VAX platforms.

digital equipment corporation
maynard, massachusetts

About Reference Pages

The *ULTRIX Reference Pages* describe commands, system calls, routines, file formats, and special files for RISC and VAX platforms.

Sections

The reference pages are divided into eight sections according to topic. Within each section, the reference pages are organized alphabetically by title, except Section 3, which is divided into subsections. Each section and most subsections have an introductory reference page called `intro` that describes the organization and anything unique to that section.

Some reference pages carry a one- to three-letter suffix after the section number, for example, `scan`(1mh). The suffix indicates that there is a "family" of reference pages for that utility or feature. The Section 3 subsections all use suffixes and other sections may also have suffixes.

Following are the sections that make up the *ULTRIX Reference Pages*.

Section 1: Commands

This section describes commands that are available to all ULTRIX users. Section 1 is split between two binders. The first binder contains reference pages for titles that fall between A and L. The second binder contains reference pages for titles that fall between M and Z.

Section 2: System Calls

This section defines system calls (entries into the ULTRIX kernel) that are used by all programmers. The introduction to Section 2, `intro`(2), lists error numbers with brief descriptions of their meanings. The introduction also defines many of the terms used in this section.

Section 3: Routines

This section describes the routines available in ULTRIX libraries. Routines are sometimes referred to as subroutines or functions.

Section 4: Special Files

This section describes special files, related device driver functions, databases, and network support.

Section 5: File Formats

This section describes the format of system files and how the files are used. The files described include assembler and link editor output, system accounting, and file system formats.

Section 6: Games

The reference pages in this section describe the games that are available in the unsupported software subset. The reference pages for games are in the document *Reference Pages for Unsupported Software.*

Section 7: Macro Packages and Conventions

This section contains miscellaneous information, including ASCII character codes, mail addressing formats, text formatting macros, and a description of the root file system.

Section 8: Maintenance

This section describes commands for system operation and maintenance.

Platform Labels

The *ULTRIX Reference Pages* contain entries for both RISC and VAX platforms. Pages that have no platform label beside the title apply to both platforms. Reference pages that apply only to RISC platforms have a "RISC" label beside the title and the VAX-only reference pages that apply only to VAX platforms are likewise labeled with "VAX." If each platform has the same command, system call, routine, file format, or special file, but functions differently on the different platforms, both reference pages are included, with the RISC page first.

Reference Page Format

Each reference page follows the same general format. Common to all reference pages is a title consisting of the name of a command or a descriptive title, followed by a section number; for example, `date(1)`. This title is used throughout the documentation set.

The headings in each reference page provide specific information. The standard headings are:

Name	Provides the name of the entry and gives a short description.
Syntax	Describes the command syntax or the routine definition. Section 5 reference pages do not use the Syntax heading.
Description	Provides a detailed description of the entry's features, usage, and syntax variations.
Options	Describes the command-line options.
Restrictions	Describes limitations or restrictions on the use of a command or routine.
Examples	Provides examples of how a command or routine is used.

Return Values	Describes the values returned by a system call or routine. Used in Sections 2 and 3 only.
Diagnostics	Describes diagnostic and error messages that can appear.
Files	Lists related files that are either a part of the command or used during execution.
Environment	Describes the operation of the system call or routine when compiled in the POSIX and SYSTEM V environments. If the environment has no effect on the operation, this heading is not used. Used in Sections 2 and 3 only.
See Also	Lists related reference pages and documents in the ULTRIX documentation set.

Conventions

The following documentation conventions are used in the reference pages.

%	The default user prompt is your system name followed by a right angle bracket. In this manual, a percent sign (%) is used to represent this prompt.
#	A number sign is the default superuser prompt.
`user input`	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in text to indicate the exact name of a command, routine, partition, pathname, directory, or file. This typeface is also used in interactive examples to indicate system output and in code examples and other screen displays.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
rlogin	This typeface is used for command names in the Syntax portion of the reference page to indicate that the command is entered exactly as shown. Options for commands are shown in bold wherever they appear.
filename	In examples, syntax descriptions, and routine definitions, italics are used to indicate variable values. In text, italics are used to give references to other documents.
[ ]	In syntax descriptions and routine definitions, brackets indicate items that are optional.
{ | }	In syntax descriptions and routine definitions, braces enclose lists from which one item must be chosen. Vertical bars are used to separate items.

. . .	In syntax descriptions and routine definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
`cat(1)`	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to `cat(1)` indicates that you can find the material on the `cat` command in Section 1 of the reference pages.

Online Reference Pages

The ULTRIX reference pages are available online if installed by your system administrator. The `man` command is used to display the reference pages as follows:

To display the `ls(1)` reference page:

```
% man ls
```

To display the `passwd(1)` reference page:

```
% man passwd
```

To display the `passwd(5)` reference page:

```
% man 5 passwd
```

To display the Name lines of all reference pages that contain the word "passwd":

```
% man -k passwd
```

To display the introductory reference page for the family of 3xti reference pages:

```
% man 3xti intro
```

Users on ULTRIX workstations can display the reference pages using the unsupported `xman` utility if installed. See the `xman(1X)` reference page for details.

Reference Pages for Unsupported Software

The reference pages for the optionally installed, unsupported ULTRIX software are in the document *Reference Pages for Unsupported Software*.

Name

intro – introduction to system calls

Syntax

#include <errno.h>

Description

Section 2 describes the ULTRIX system calls, which are the entries into the ULTRIX kernel. In this section, reference pages with the extension 2yp are specific to the Yellow Pages (YP) service. Those pages ending in 2nfs are specific to the Network File System (NFS) service.

Additionally, some Section 2 reference pages contain an ENVIRONMENT section that describes differences between the POSIX or SYSTEM V environment and the ULTRIX operating system.

Environmental Compatibility

Some system calls contain System V and POSIX features that are compatible with ULTRIX programs. These features are provided for applications that are being ported from System V or POSIX. Occasionally, the System V and POSIX features conflict with features present in the ULTRIX system. For example, a function performed under the ULTRIX operating system can produce different results in the System V or POSIX environment. If conflicts exist, the ENVIRONMENT section of the reference page highlights these differences.

Neither the System V compatibility features nor the POSIX compatibility features are not contained in the standard C runtime library. To use the compatibility features, you must set your programming environment to System V or POSIX when you compile or link your programs. To set the System V or POSIX environment, do either of the following:

1. Use the *–Y* option for the `cc` command. For example, the following demonstrates compiling a program in the System V environment first, and then in the POSIX environment:

```
% cc -YSYSTEM_FIVE program.c
% cc -YPOSIX program.c
```

2. Globally set the environment variable PROG_ENV to SYSTEM_FIVE or to POSIX.

   If you are using the C shell, execute the following line or include it in your `.login` file:

```
setenv PROG_ENV SYSTEM_FIVE
```

   Replace "SYSTEM_FIVE" with "POSIX" if you are using the POSIX environment.

   If you are using the Bourne or the System V shell, execute the following line or include it in your `.profile` file:

```
PROG_ENV=POSIX ; export PROG_ENV
```

Replace "POSIX" with "SYSTEM_FIVE" if you are using the System V environment.

In each instance, the `cc` command defines a preprocessor symbol, either SYSTEM_FIVE or POSIX. When the SYSTEM_FIVE symbol is defined, the C preprocessor, `cpp`, selects the System V data structures and symbol definitions. When the POSIX symbol is defined, `cpp` selects the POSIX data structures and symbol definitions.

In addition, if `cc` invokes the `ld` linker, it resolves references to routines by searching the System V version of the Standard C library (`libcV.a`) or the POSIX version of the Standard C library (`libcP.a`) before it searches `libc.a`. The linker searches `libcV.a` when the SYSTEM_FIVE symbol is defined. It searches `libcP.a` when POSIX is defined.

In the System V environment, if you specify the *–lm* option on either the `cc` or the `ld` command line, the linker includes the System V math library, instead of the ULTRIX math library, in your program.

Return Value

Most system calls have one or more return values. An error condition is indicated by an otherwise impossible return value. This value is usually –1. When a function returns an error condition, it also stores an error number in the external variable `errno`. This variable is not cleared on successful calls. Thus, you should test `errno` only after an error has occurred.

All return codes and values from functions are of type *int* unless otherwise noted.

For a list of the errors and their names as given in <errno.h>, see the `errno`(2) reference page.

Definitions

The following terms are used in Section 2:

Descriptor
: An integer assigned by the system when a file is referenced by `open`, `dup`, `pipe`, or a socket is referenced by `socket` or `socketpair` The descriptor uniquely identifies an access path to that file or socket from a given process or any of its children.

Directory
: A directory is a special type of file that contains references to other files, called links. By convention, a directory contains at least two links called dot (.) and dot-dot (..). Dot refers to the directory itself and dot-dot refers to its parent directory.

Effective User Id, Effective Group Id, and Access Groups
: Access to system resources is governed by the 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 can be modified through execution of a set-user-ID or set-group-ID file, or possibly by one of its ancestors. For more information, see `execve`(2).

  The group access list is an additional set of group IDs used only in determining

resource accessibility. Access checks are performed as defined under the term File Access Permissions.

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 can be changed with the `chmod` call.

File access is separated into three types: read, write, and execute. Directory files use the execute permission to control whether or not the directory can 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, and 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 in the following instances:

- The process's effective user ID is that of the superuser.
- 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, but 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 the 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.

Read, write, and execute/search permissions on a file are not granted, as follows:

- If the process is trying to execute an image and the file system is mounted no execute, execute permission is denied.
- If the process's effective UID is not root, the process is attempting to access a character or block special device, and the file system is mounted with nodev, access is denied.
- If the process's effective UID is not root, the process is trying to execute an image with the *setuid* or *setgid* bit set in the file's permissions, and the file system is mounted nosuid, execute permission is denied.

File Name

Names consisting of up to {FILENAME_MAX} characters can be used to name an ordinary file, special file, or directory.

These characters can be selected from the set of all ASCII characters excluding null (0) and the ASCII code for backslash (\). The parity bit (bit 8) must be 0.

Avoid using asterisks (*), question marks (?), or brackets ([ ]) as part of filenames because of the special meaning attached to these characters by the shell.

Message Operation Permissions

In the `msgop(2)` and `msgctl(2)` system call descriptions, the permission required for an operation is specified by a token. The token argument is the type of permission needed and it is interpreted as follows:

```
00400    Read by user
00200    Write by user
00060    Read, Write by group
00006    Read, Write by others
```

Read and write permissions are granted to a process if one or more of the following are true:

- The effective user ID of the process is superuser.
- The effective user ID of the process matches *msg_perm.[c]uid* in the data structure associated with msqid and the appropriate bit of the user portion (0600) of *msg_perm.mode* is set.
- The effective user ID of the process does not match *msg_perm.[c]uid,* but the effective group ID of the process matches *msg_perm.[c]gid* and the appropriate bit of the group portion (060) of *msg_perm.mode* is set.
- The effective user ID of the process does not match *msg_perm.[c]uid* and the effective group ID of the process does not match *msg_perm.[c]gid,* but the appropriate bit of the other portion (06) of *msg_perm.mode* is set.

If none of the previous conditions are true, the read and write permissions are denied.

Message Queue Identifier

A message queue identifier (msqid) is a unique positive integer created by a `msgget` system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct  ipc_perm msg_perm; /*operation permission struct*/
ushort  msg_qnum;          /*number of msgs on q*/
ushort  msg_qbytes;        /*max number of bytes on q*/
ushort  msg_lspid;         /*pid of last msgsnd operation*/
ushort  msg_lrpid;         /*pid of last msgrcv operation*/
time_t  msg_stime;         /*last msgsnd time*/
time_t  msg_rtime;         /*last msgrcv time*/
time_t  msg_ctime;         /*last change time*/
                           /*Times measured in secs since*/
                           /*00:00:00 GMT, Jan.1, 1970*/
```

The *msg_perm* structure is an *ipc_perm* structure that specifies the message operation permission. The *msg_perm* structure includes the following members:

```
ushort  cuid;     /*creator user id*/
```

```
ushort  cgid;     /*creator group id*/
ushort  uid;      /*user id*/
ushort  gid;      /*group id*/
ushort  mode;     /*r/w permission*/
```

The *msg_qnum* member is the number of message currently on the queue. The *msg_qbytes* member is the maximum number of bytes allowed on the queue. The *msg_lspid* member is the process ID of the last process that performed a `msgrcv` operation. The *msg_lrpid* member is the process ID of the last process that performed a `msgop` operation. The *msg_stime* member is the time of the last `msgop` operation, *msg_rtime* is the time of the last `msgrcv` operation, and *msg_ctime* is the time of the last `msgctl` operation that changed a member of the above structure.

Parent process ID

A new process is created by a currently active process. For further information, see `fork`(2). The parent process ID of a process is the process ID of its creator.

Pathname

A pathname is a null-terminated character string containing an optional slash (/), followed by zero or more directory names separated by slashes. This sequence can optionally be followed by another slash and a filename. The total length of a pathname must be less than {PATHNAME_MAX} characters.

If a pathname 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 refers to the current directory.

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 0 to {PROC_MAX}.

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. For more information, see `killpg`(2) and the job control mechanisms described in `csh`(1).

Real User ID and Real Group ID

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

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

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 does not need to be the root directory of the root file system.

Semaphore Identifier

A semaphore identifier (semid) is a unique positive integer created by a `semget` system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct  ipc_perm sem_perm; /*operation permission struct*/
ushort  sem_nsems;         /*number of sems in set */
time_t  sem_otime;         /*last operation time*/
time_t  sem_ctime;         /*last change time*/
                           /*Times measured in secs since*/
                           /*00:00:00 GMT, Jan. 1, 1970*/
```

The *sem_perm* is an *ipc_perm* structure that specifies the semaphore operation permission. This structure includes the following members:

```
ushort cuid;  /*creator user id*/
ushort cgid;  /*creator group id*/
ushort uid;   /*user id*/
ushort gid;   /*group id*/
ushort mode;  /*r/a permission*/
```

The value of *sem_nsems* is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. The *sem_num* values run sequentially from 0 to the value of sem_nsems minus 1. The *sem_otime* member is the time of the last `semop` operation, and *sem_ctime* is the time of the last `semctl` operation that changed a member of the above structure.

A semaphore is a data structure that contains the following members:

```
ushort  semval;  /*semaphore value*/
short   sempid;  /*pid of last operation*/
ushort  semncnt; /*# awaiting semval > cval*/
ushort  semzcnt; /*# awaiting semval = 0*/
```

The *semval* member is a non-negative integer. The *sempid* member is equal to the process ID of the last process that performed a semaphore operation on this semaphore. The *semncnt* member is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value. The *semzcnt* member is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions

In the `semop`(2) and `semctl`(2) system call descriptions, the permission required for an operation is specified as {token}. The token argument is the type of permission needed and it is interpreted as follows:

```
00400   Read by user
00200   Alter by user
00060   Read, Alter by group
00006   Read, Alter by others
```

Read and alter permissions on a semid are granted to a process if one or more of the following are true:

- The effective user ID of the process is superuser.
- The effective user ID of the process matches *sem_perm.[c]uid* in

the data structure associated with semid and the appropriate bit of the user portion (0600) of *sem_perm.mode* is set.

- The effective user ID of the process does not match *sem_perm.[c]uid,* but the effective group ID of the process matches *sem_perm.[c]gid* and the appropriate bit of the group portion (060) of *sem_perm.mode* is set.
- The effective user ID of the process does not match *sem_perm.[c]uid* and the effective group ID of the process does not match *sem_perm.[c]gid,* but the appropriate bit of the other portion (06) of *sem_perm.mode* is set.

If none of the previous conditions are true, the read and alter permissions are denied.

Session

Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. Typically there is one session per login.

Shared Memory Identifier

A shared memory identifier (shmid) is a unique positive integer created by a `shmget` system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as `shmid_ds` and contains the following members:

```
struct  ipc_perm shm_perm;  /*operation permission struct*/
int     shm_segsz;          /*size of segment*/
ushort  shm_cpid;           /*creator pid*/
ushort  shm_lpid;           /*pid of last operation*/
short   shm_nattch;         /*number of current attaches*/
time_t  shm_atime;          /*last attach time*/
time_t  shm_dtime;          /*last detach time*/
time_t  shm_ctime;          /*last change time*/
                            /*Times measured in secs since*/
                            /*00:00:00 GMT, Jan. 1, 1970*/
```

The *shm_perm* member is an *ipc_perm* structure that specifies the shared memory operation permission. This structure includes the following members:

```
ushort  cuid;  /*creator user id*/
ushort  cgid;  /*creator group id*/
ushort  uid;   /*user id*/
ushort  gid;   /*group id*/
ushort  mode;  /*r/w permission*/
```

The *shm_segz* member specifies the size of the shared memory segment. The *shm_cpid* member is the process ID of the process that created the shared memory identifier. The *shm_lpid* member is the process ID of the last process that performed a `shmop` operation. The *shm_nattch* member is the number of processes that currently have this segment attached. The *shm_atime* member is the time of the last `shmat` operation, *shm_dtime* is the time of the last `shmdt` operation, and *shm_ctime* is the time of the last `shmctl` operation that changed one of the members of the above structure.

Shared Memory Operation Permissions

In the `shmop`(2) and `shmctl`(2) system call descriptions, the permission

required for an operation is given as {token}. The token argument is the type of permission needed and it is interpreted as follows:

```
00400   Read by user
00200   Write by user
00060   Read, Write by group
00006   Read, Write by others
```

Read and write permissions on a shmid are granted to a process if one or more of the following are true:

- The effective user ID of the process is superuser.
- The effective user ID of the process matches *shm_perm.[c]uid* in the data structure associated with shmid and the appropriate bit of the user portion (0600) of *shm_perm.mode* is set.
- The effective user ID of the process does not match *shm_perm.[c]uid,* but the effective group ID of the process matches *shm_perm.[c]gid* and the appropriate bit of the group portion (060) of *shm_perm.mode* is set.
- The effective user ID of the process does not match *shm_perm.[c]uid* and the effective group ID of the process does not match *shm_perm.[c]gid,* but the appropriate bit of the other portion (06) of *shm_perm.mode* is set.

If none of the previous conditions are true, the read and write permissions are denied.

Sockets and Address Families

A socket is an endpoint for communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties determine whether messages sent and received at a socket require the name of the partner, if communication is reliable, and if the format is used in naming message recipients.

Each instance of the system supports some collection of socket types. See `socket`(2) for more information about the types available and their properties.

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

Special Processes

Those processes that have a process ID of 0, 1, and 2 are considered special processes. Process 0 is the scheduler. Process 1 is the initialization process *init*, and is the ancestor of every other process in the system. It controls the process structure. Process 2 is the paging daemon.

Superuser

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

tty Group ID

Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to arbitrate between multiple jobs contending for the same terminal. For more information, see `csh`(1) and `tty`(4).

See Also

cc(1), csh(1), tty(4), intro(3), perror(3)

accept(2)

Name

accept – accept a connection on a socket

Syntax

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

accept(s, addr, addrlen)
int ns, s;
struct sockaddr *addr;
int *addrlen;

Description

The `accept` system call accepts a connection on a socket. The argument *s* is a socket that has been created with the `socket`, call, bound to an address with the `bind`, call and is listening for connections after a `listen` call. The `accept` system call extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue, and the socket is not marked as nonblocking, `accept` blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, `accept` returns an error. The accepted socket, *ns*, cannot be used to accept more connections. The original socket **s** remains open.

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

You can use the `select` call for the purposes of doing an `accept` call by selecting the socket for reading.

Return Value

The call returns –1 on error. If the call succeeds, it returns a non-negative integer which is a descriptor for the accepted socket.

Diagnostics

The `accept` call fails if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.

[EWOULDBLOCK]
The socket is marked nonblocking and no connections are present to be accepted.

See Also

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

access (2)

Name

access – determine the accessibility of file

Syntax

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

Description

The system call, `access`, checks the given file *path* for accessibility according to *mode*. The argument *mode* is an inclusive OR of the bits R_OK, W_OK, and X_OK. Specifying the argument *mode* as F_OK tests whether the directories leading to the file can be searched and whether the file exists.

The real user ID and the group access list (including the real group ID) are used to verify permissions. This call is useful to set-UID programs.

Note that only access bits are checked. The `access` call may indicate that a directory is writeable, but an attempt to open the directory fails, although files are present in the directory. Additionally, a file may appear to be executable, but `execve` fails unless the file is in proper format.

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

Diagnostics

Access to the file is denied if any of the following is true:

[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's read, write, and execute mode bits. Members of the file's group, other than the owner, have permission checked with respect to the group's mode bits. All others have permissions checked with respect to the other mode bits.

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

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG] A *path* component length exceeds 255 characters or the length of *path* exceeds 1023 characters.

[ENOENT] The file referred to by *path* does not exist or the *path* points to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	Write access is requested for a file on a read-only file system.
[ESTALE]	The file handle given in the argument was invalid. The file referred to by that file handle no longer exists or has been revoked.
[ETIMEDOUT]	A connect request or remote file operation fails because the connected party did not respond after a period of time determined by the communications protocol.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.

See Also

chmod(2), stat(2)

acct(2)

Name

acct – turn accounting on or off

Syntax

acct(file)
char *file;

Description

The system is prepared to write a record in an accounting *file* for each process as it terminates. This call, with a null-terminated string naming an existing file as argument, turns on accounting; records for each terminating process are appended to *file*. An argument of 0 causes accounting to be turned off.

The accounting file format is given in `acct`(5).

This call is permitted only to the superuser. Accounting is automatically disabled when the file system the accounting file resides on runs out of space. It is enabled when space once again becomes available.

Return Value

On error, –1 is returned. The file must exist and the call may be exercised only by the superuser. It is erroneous to try to turn on accounting when it is already on. If successful, 0 is returned.

Diagnostics

The `acct` system call will fail if one of the following is true:

[EPERM] The caller is not the superuser.

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

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] The path name is not a regular file.

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

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

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

Restrictions

No accounting is produced for programs running when a crash occurs. In particular, nonterminating programs are never accounted for.

See Also

acct(5), sa(8)

adjtime (2)

Name

adjtime – correct the time to allow synchronization of the system clock

Syntax

#include <sys/time.h>

adjtime(delta, olddelta)
struct timeval *delta;
struct timeval *olddelta;

Description

The `adjtime` system call changes the system time, as returned by `gettimeofday`, moving it backward or forward by the number of microseconds corresponding to the timeval *delta*.

The time is maintained by incrementing it with a machine-dependent tick every clock interrupt. If *delta* is negative, the clock is slowed down by incrementing it in smaller ticks until the correction is made. If *delta* is positive, a larger tick is used. Thus, the time is always a monotonically increasing function. A time correction from an earlier call to `adjtime` may not be finished when `adjtime` is called again. If *olddelta* is nonzero, then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

This call can be used in time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

The `adjtime` call is restricted to the superuser.

Note

Time is incremented in 3.906ms ticks on MIPS and 10ms ticks on VAX. When `adjtime` is called with an argument other than zero, ticks of 9ms or 11ms are used until the time is corrected. A *delta* of less than 1ms has no effect.

Return Value

A return value of 0 indicates that the call succeeded. A return value of –1 indicates that an error occurred, and in this case an error code is stored in the global variable *errno*.

Diagnostics

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

[EFAULT]	An argument points outside the process's allocated address space.
[EPERM]	The process's effective user ID is not that of the super-user.

See Also

date(1), gettimeofday(2)

atomic_op (2)

Name

atomic_op – perform test and set operation.

Syntax

#include <sys/lock.h>

int atomic_op(*op, addr*)
int *op*;
int **addr*;

Arguments

op	This argument is the operation type. If the operation type is ATOMIC_SET, this call specifies the test and set operation on location *addr*. If the operation type is ATOMIC_CLEAR, this call specifies the clear operation on location *addr*.
addr	This is the target address of the operation.

Description

The `atomic_op` call provides test and set operation at a user address.

For RISC systems, `atomic_op` is executed as a system call. For VAX systems, a system call is not executed for this library function.

Return Value

If the atomic_op operation succeeds, then 0 is returned. Otherwise a –1 is returned, and a more specific error code is stored in *errno*.

Diagnostics

[EBUSY]	The location specified by *addr* is already set.
[EINVAL]	The *op* is not a valid operation type.
[EACCES]	The address specified in *addr* is not write accessible.
[EALIGN]	The *addr* is not on an integer boundary.

Name

audcntl – audit control

Syntax

#include <sys/audit.h>

audcntl(request, argp, len, cntl, audit_id)
int request;
char *argp;
int len;
char cntl;
audit_ID_t audit_id;

Description

The `audcntl` system call provides control over options offered by the audit subsystem. All requests are privileged. The following list describes the requests:

GET_SYS_AMASK and SET_SYS_AMASK
The system audit mask determines which system events are logged. GET_SYS_AMASK places the values of the system audit mask into a buffer pointed at by *argp*. SET_SYS_AMASK takes the values from a buffer pointed at by *argp* and assigns them to the system audit mask. Getting or setting the system mask returns the number of bytes transferred between the user's buffer and the audit mask. *Len* is the size of the user's buffer. The amount of data moved between the audit mask and the user's buffer is the smaller of the audit mask size and the buffer size.

GET_TRUSTED_AMASK and SET_TRUSTED_AMASK
The trusted audit mask determines which trusted events are logged. GET_TRUSTED_AMASK places the values of the trusted audit mask into a buffer pointed at by *argp*. SET_TRUSTED_AMASK takes the values from a buffer pointed at by *argp* and assigns them to the trusted audit mask. Getting or setting the trusted events mask returns the number of bytes transferred between the user's buffer and the audit mask. *Len* is the size of the user's buffer. The amount of data moved between the audit mask and the user's buffer is the smaller of the audit mask size and the buffer size.

GET_PROC_AMASK and SET_PROC_AMASK
The process audit mask determines which system events are logged for the current process. GET_PROC_AMASK places the values of the process audit mask into a buffer pointed at by *argp*. SET_PROC_AMASK takes the values from a buffer pointed at by *argp* and assigns them to the process audit mask. Getting or setting the process mask returns the number of bytes transferred between the user's buffer and the audit mask. *Len* is the size of the user's buffer. The amount of data moved between the audit mask and the user's buffer is the smaller of the audit mask size and the buffer size.

GET_PROC_ACNTL and SET_PROC_ACNTL
GET_PROC_ACNTL returns the audit control flags (the audcntl byte) of the current process (see `audit.h`). Audit control flags determine

whether auditing for the process is ON or OFF, and if ON, whether the process audit mask is logically ANDed or ORed with the system audit mask. SET_PROC_ACNTL assigns the values of the audit control flags from *cntl* and returns the previous values of the flags.

GET_AUDSWITCH and SET_AUDSWITCH
GET_AUDSWITCH returns the value of the system audit switch. A return value of 1 indicates auditing is turned on. A value of zero indicates auditing is turned off. SET_AUDSWITCH assigns the value of *cntl* to the system audit switch and returns the previous audit switch value. A value of 1 turns auditing on. A value of zero turns auditing off.

FLUSH_AUD_BUF
Flushes kernel audit buffer out to `/dev/audit`.

GETPAID and SETPAID
GETPAID returns the audit ID of the calling process. SETPAID assigns the value of *audit_id* to the process audit ID. SETPAID is effective only if *audit_id* is greater than 0.

Return Value

The values returned for successful calls can be found under the description of the specific call request.

If a call fails, a –1 is returned.

Diagnostics

The `audcntl` call fails under the following conditions:

[EFAULT]	The *argp* argument contains an invalid address.
[EACCES]	The user does not have the privileges needed to perform this operation.
[EINVAL]	The value of the *len* or *request* argument is invalid or *audit_id* was previously set.
[EPERM]	The user is not privileged to get or set the audit ID, or the user attempted to get the audit ID when it was not set.
[EOPNOTSUPP]	The *request* argument contains an unsupported operation.

Name

audgen – generate an audit record

Syntax

audgen(event, tokenp, argv)
int event;
char *tokenp, *argv[];

Description

The `audgen` system call generates an audit record, which gets placed in the auditlog.

The argument *event* is an integer indicating the event type of the operation being audited (see `audit.h`). The value of *event* must be between MIN_TRUSTED_EVENT and MIN_TRUSTED_EVENT+N_TRUSTED_EVENTS.

The argument *tokenp* is a null-terminated array of token types (see `audit.h`), each of which represents the type of argument referenced by the corresponding **argv* argument.

The argument *argv* is a pointer to an array containing the actual arguments or pointers to those arguments that are to be recorded in the audit record. A pointer to the actual argument is placed in that array when the argument is a string, array, or other variable length structure. Arguments represented as int's or short's are placed directly in that array. Each member of the array must be word-aligned. You cannot change the values for the audit_id, uid, ruid, pid, ppid, device, IP address, or hostid (secondary tokens for these values are available).

Return Value

Upon successful completion, `audgen` returns a value of 0. Otherwise, it returns a value of –1 and sets the global integer variable *errno* to indicate the error.

Restrictions

The `audgen` call is a privileged system call. No record is generated if the specified *event* is not being audited for the current process. The maximum number of arguments referenced by *argv* is AUD_NPARAM (8).

Diagnostics

The `audgen` system call fails under the following conditions:

[EACCES]	The user is not privileged for this operation.
[EINVAL]	The value supplied for the *event, tokenp,* or *argv* argument is invalid.

bind(2)

Name

bind – bind a name to a socket

Syntax

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

bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

bind(s, name, namelen)
int s;
struct sockaddr_un *name;
int namelen;
```

Description

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

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed, using the `unlink` system call.

The *sockaddr* argument specifies a general address family. The *sockaddr_un* argument specifies an address family in the UNIX domain.

The rules used in name binding vary between communication domains. Consult the reference pages in the *ULTRIX Reference Pages Section 4: Special Files* for detailed information.

Return Value

If the `bind` is successful, the call returns a 0 value. A return value of –1 indicates an error, which is further specified in the global variable *errno*.

Diagnostics

The `bind` call fails under the following conditions:

[EBADF] *S* is an invalid descriptor.

[ENOTSOCK] *S* is not a socket.

[EADDRNOTAVAIL] The specified address is not available from the local machine.

[EADDRINUSE] The specified address is already in use.

[EINVAL] The socket is already bound to an address.

[EACCESS] The requested address is protected, and the current user has

inadequate permission to access it.

[EFAULT] The *name* parameter is not in a valid part of the user address space.

The following errors are specific to binding names in the UNIX domain:

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

[ENAMETOOLONG] A component of a pathname exceeds 255 characters, or an entire pathname exceeds 1023 characters.

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

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

[EROFS] The name would reside on a read-only file system.

[EISDIR] A null pathname was specified.

See Also

connect(2), getsockname(2), listen(2), socket(2), unlink(2)

Name

brk, sbrk – change data segment space allocation

Syntax

#include <sys/types.h>

char *brk(addr)
char *addr;

char *sbrk(incr)
int incr;

Description

The `brk` system call sets the system's idea of the lowest data segment location not used by the program (called the break) to *addr* (rounded up to the next multiple of the system's page size). Locations greater than *addr* and below the stack pointer are not in the address space and thus will 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 by `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 `brk`.

The `getrlimit`(2) system call may be used to determine the maximum permissible size of the data segment; it will not be possible to set the break beyond the *rlim_max* value returned from a call to `getrlimit`(2). For example:

```
0x10000000 + rlp -> rlim_max
```

Return Value

Upon successful completion, the `brk` system call returns a value of 0 or –1 if the program requests more memory than the system limit. The `sbrk` system call returns –1 if the break could not be set.

Restrictions

Setting the break may fail due to a temporary lack of swap space. It is not possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting `getrlimit`.

Diagnostics

The `sbrk` system call fails and no additionally memory is allocated if one of the following is true:

[ENOMEM]	The limit, as set by `setrlimit`(2) was exceeded.
[ENOMEM]	The maximum possible size a data segment (compiled into the system) was exceeded.
[ENOMEM]	Insufficient space existed in the swap area to support the expansion.

See Also

execve(2), getrlimit(2), setrlimit(2), ulimit(2)

brk(2)

Name

brk, sbrk – change core allocation

Syntax

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

caddr_t brk(addr)
caddr_t addr;

caddr_t sbrk(incr)
int incr;
```

Description

The `brk` system call sets the system's idea of the lowest data segment location not used by the program (called the break) to *addr* (rounded up to the next multiple of the system's page size). 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 using `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`.

The `getrlimit` system call may be used to determine the maximum permissible size of the *data* segment. It will not be possible to set the break beyond the *rlim_max* value returned from a call to `getrlimit`, for example, **etext + rlp –*rlim_max***. See `end`(3) for the definition of *etext*.

Return Value

If the call is successful, `brk` returns a 0 value. If the program requests more memory than the system limit, `brk` returns –1. If the break could not be set, `brk` returns –1.

Restrictions

Setting the break may fail due to a temporary lack of swap space. It is not possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting `getrlimit`.

Diagnostics

The `sbrk` call fails and no additional memory is allocated under the following conditions:

[ENOMEM]	The limit, as set by `setrlimit`, is exceeded.
[ENOMEM]	The maximum possible size of a data segment (compiled into the system) is exceeded.
[ENOMEM]	Insufficient space exists in the swap area to support the expansion.

See Also

execve(2), getrlimit(2), setrlimit(2), end(3), malloc(3)

Name

cachectl – mark pages cacheable or uncacheable

Syntax

#include <mips/cachectl.h>

cachectl(addr, nbytes, op)
char *addr;
int nbytes, op;

Description

The `cachectl` system call allows a process to make ranges of its address space cacheable or uncacheable. Initially, a process's entire address space is cacheable.

The *op* parameter is one of the following:

CACHEABLE	Make the indicated pages cacheable.
UNCACHEABLE	Make the indicated pages uncacheable.

The arguments CACHEABLE and UNCACHEABLE affect the address range indicated by the *addr* and *nbytes* parameters. The *addr* must be page aligned, and *nbytes* must be a multiple of the page size.

Changing a page from UNCACHEABLE state to CACHEABLE state causes both the instruction and data caches to be flushed.

Return Value

The `cachectl` system call returns 0 on success. If errors are detected, the `cachectl` system call returns –1 with the error cause indicated in *errno*.

Diagnostics

[EFAULT]	Some or all of the address range *addr* to (*addr*+*nbytes*–1) are not accessible.
[EINVAL]	The *op* parameter is not CACHEABLE or UNCACHEABLE.
[EINVAL]	The *addr* parameter is not page aligned, or the *nbytes* parameter is not a multiple of pagesize.

Name

cacheflush – flush the instruction cache, data cache, or both

Syntax

#include <mips/cachectl.h>

cacheflush(addr, nbytes, cache)
char *addr;
int nbytes, cache;

Description

Flushes contents of indicated caches for user addresses in the range of *addr* to (*addr+nbytes*–1). The *cache* parameter is one of the following:

ICACHE	Flush only the instruction cache.
DCACHE	Flush only the data cache.
BCACHE	Flush both the instruction and data caches.

Return Value

The `cacheflush` system call returns 0 when errors are not detected. If errors are detected, the `cacheflush` system call returns –1 with the error cause indicated in *errno*.

Diagnostics

[EFAULT]	Some or all of the address range in the *addr* to (*addr+nbytes*–1) are not accessible.
[EINVAL]	The *cache* parameter is not ICACHE, DCACHE, or BCACHE.

chdir(2)

Name

chdir – change working directory

Syntax

chdir(*path*)
char **path;*

Description

The *path* is the pathname of a directory. The `chdir` system call causes this directory to become the current working directory, which is the starting point for pathnames that do not begin at the root directory (/).

For a directory to become the current directory, the 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.

Environment

Differs from the System V definition in that ELOOP is a possible error condition.

Diagnostics

The `chdir` system call fails and the current working directory is unchanged under the following conditions:

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

[ENAMETOOLONG]
A component of a pathname exceeds 255 characters, or an entire path name exceeds 1023 characters.

[ENOENT] The named directory does not exist or the path points to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

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

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

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

[ESTALE] The file handle given in the argument was invalid. The file referred to by that file handle no longer exists or has been revoked.

[ETIMEDOUT] A connect request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

chroot(2)

chmod(2)

Name

chmod, fchmod – change mode of file

Syntax

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

chmod(path, mode)
char *path;
mode_t mode;

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

Description

The file whose name is provided by *path* or referenced by the descriptor *fd* has its mode changed to *mode*. Modes are constructed by ORing combinations of the following:

S_ISUID	– 04000	set user ID on execution
S_ISGID	– 02000	set group ID on execution
S_ISVTX	– 01000	save text image after execution
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 (search) by group
S_IRWXO	– 00007	read, write, execute (search) by others

If an executable file is set up for sharing (the default), the mode S_ISVTX prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. The ability to set this bit is restricted to the superuser.

If the mode S_ISVTX (sticky bit) is set on a directory, an unprivileged user cannot delete or the rename files of other users in that directory. For more information on the sticky bit, see `sticky`(8).

Only the owner of a file or the superuser can change the mode.

Writing a file or changing the owner of a file clears the set-user-id and set-group-id bits of that file. Turning off these bits when a file is written or its owner changed protects the file from remaining set-user-id or set-group-id after being modified. If a file, specifically a program, remained set-user-id or set-group-id after being modified, that file could allow unauthorized access to other files or accounts.

Environment

System Five

ELOOP is a possible error condition.

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.

Diagnostics

The `chmod` system call fails and the file mode remains unchanged under the following conditions:

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

[EFAULT] The *path* argument points outside the process's allocated address space.

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG] A pathname component exceeds 255 characters, or an entire pathname exceeds 1023 characters.

[ENOENT] The named file does not exist.

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

[EPERM] The effective user ID does not match the owner of the file and the effective user ID is not the superuser.

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

[ESTALE] The file handle given in the argument is invalid. Either the file referred to by that file handle no longer exists or it has been revoked.

The `fchmod` system call fails under the following conditions:

[EBADF] The descriptor is not valid.

[EINVAL] The *fd* refers to a socket, not to a file.

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

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

[ETIMEDOUT] A connect request or remote file operation failed because the connected party did not respond after a period of time determined by the communications protocol.

See Also

open(2), chown(2)

Name

chown, fchown – change owner and group of a file

Syntax

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

chown(path, owner, group)
char *path;
uid_t owner;
gid_t group;

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

Description

The `chown` and `fchown` system calls change the owner and group of the file named by *path* or referenced by *fd* . Only the superuser can change the owner of a file. Other users can change the group-id of a file that they own to another group to which they belong.

If you specify –1 in *owner* or *group,* the corresponding owner-id or group-id of the file is unchanged.

The `chown` system call clears the set-user-id and set-group-id bits on the file when it returns successfully, unless the call is made by the superuser. Clearing these bits when a file's owner is changed protects the file from remaining set-user-id or set-group-id after being modified. If a file, specifically a program, remained set-user-id or set-group-id after being modified, that file could allow unauthorized access to other files or accounts.

You should use the `fchown` system call with the file locking primitives because `fchown` preserves any locks you previously obtained with the `flock` system call. For more information about file locking, see the `flock`(2) reference page.

Return Value

The `chown` and `fchown` calls return zero if the operation is successful; if an error occurs they return –1 and store a more specific error code in the global variable *errno.*

Environment

System Five

Differs from the System V definition in that only the superuser can change the ownership of a file. In addition, ELOOP is a possible error condition.

POSIX

When your program is compiled in the POSIX environment, the *owner* argument is of type *uid_t,* and the *group* argument is of type *gid_t.*

Diagnostics

The `chown` system call fails and the file is unchanged under the following conditions:

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

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

[ENOENT] The named file does not exist.

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

[EPERM] The effective user-id is not the superuser.

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

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

[ELOOP] Too many symbolic links are encountered in translating the pathname.

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

[ESTALE] The *fd* argument is invalid because the file referred to by that file handle no longer exists or has been revoked.

The `fchown` system call fails if:

[EBADF] The *fd* argument does not refer to a valid descriptor.

[EINVAL] The *fd* argument refers to a socket, not a file.

[EPERM] The effective user-id is not the superuser.

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

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

[ETIMEDOUT] A connect request or remote file operation fails because the connected party does not properly respond after a period of time that is dependent on the communications protocol.

See Also

chmod(2), flock(2)

Name

chroot – change root directory

Syntax

chroot(*dirname*)
char **dirname;*

Description

The *dirname* is the address of the pathname of a directory, terminated by a null byte. The `chroot` system call causes this directory to become the root directory (/).

For a directory to become the root directory, a process must have execute (search) access to the directory.

This call is restricted to the superuser.

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.

Diagnostics

The `chroot` system call fails and the root directory is unchanged under the following conditions:

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

[ENAMETOOLONG] A component of a *dirname* exceeded 255 characters, or an entire *dirname* exceeded 1023 characters.

[ENOENT] The *dirname* argument points to the name of a directory which does not exist, or to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

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

[ESTALE] The file handle given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.

[ETIMEDOUT] A connect request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

[EPERM] The effective user ID is not that of superuser.

See Also

chdir(2)

Name

close – delete a descriptor

Syntax

close(*fd*)
int *fd*;

Description

The `close` call deletes a descriptor from the per-process object reference table. If the descriptor is the last reference to the underlying object, then the object is 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 socket, `close` discards associated naming information and queued data. On the last close of a file holding an advisory lock, the lock is released. For further information, see `flock`(2).

A process's descriptors are automatically closed when a process exits, but because each process can have a limited number of active descriptors, `close` is necessary for programs that deal with many descriptors.

When a process forks, all descriptors for the new child process reference the same objects as they did in the parent process before the fork. For further information, see `fork`(2). If a new process is then to be run using `execve`, the process would normally inherit these descriptors. Most of the descriptors can be rearranged with the `dup2` system call or deleted with `close` before `execve` is called. However, if any descriptors are needed if the `execve` fails, they must be closed if the execve succeeds. For this reason, the call, fcntl(d, F_SETFD, 1), is provided. This call arranges that a descriptor is closed after a successful `execve` call. The call, fcntl(d, F_SETFD, 0), restores the default, which is to not close the descriptor.

When `close` is used on a descriptor that refers to a remote file over NFS, and that file has been modified by using `write`(2), then any cached `write` data is flushed before `close` returns. If an asynchronous write error has occurred previously with this remote file, or occurred as part of the flush operation described above, then `close` returns –1 and errno will be set to the error code. The return code from `close`(2) should be inspected by any program that can `write` over NFS.

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.

Diagnostics

The `close` system call fails under the following conditions:

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

[EINTR] The `close` function was interrupted by a signal.

If an error occurs on an asynchronous write over NFS, the error cannot always be returned from a `write` system call. The error code is returned on `close` or `fsync`. The following are NFS-only error messages:

[EACCESS] The requested address is protected, and the current user has inadequate permission to access it.

[ENOSPC]	There is no free space remaining on the file system containing the file.
[EDQUOT]	The user's quota of disk blocks on the file system containing the file has been exhausted.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EROFS]	The file is on a read-only file system.
[ESTALE]	The *fd* argument is invalid because the file referred to by that file handle no longer exists or has been revoked.
[ETIMEDOUT]	A write operation failed because the server did not properly respond after a period of time that is dependent on the `mount`(8nfs) options.

See Also

accept(2), execve(2), fcntl(2), flock(2), fsync(2), open(2), pipe(2), socket(2), socketpair(2), write(2)

Name

connect – initiate a connection on a socket

Syntax

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

connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;

Description

The `connect` call initiates a connection on a socket. The parameter *s* is a socket. If the socket is of type SOCK_DGRAM, this call permanently specifies the peer to which datagrams are sent. If it is of type SOCK_STREAM, this call attempts to make a connection to another socket. The other socket is specified by *name,* which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way. The size of the structure *sockaddr* is *namelen.*

Return Value

If the connection or binding succeeds, then 0 is returned. Otherwise, a –1 is returned, and a more specific error code is stored in *errno.*

Diagnostics

The call fails under the following conditions:

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

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

[EADDRNOTAVAIL]
The specified address is not available on this machine.

[EAFNOSUPPORT]
Addresses in the specified address family cannot be used with this socket.

[EINPROGRESS]
The connection is requested on a socket with FNDELAY set (using `fcntl`(2)).

[EISCONN] The socket is already connected.

[ETIMEDOUT] Connection establishment timed out without establishing a connection.

[ECONNREFUSED]
The attempt to connect was forcefully rejected.

[ENETUNREACH]
The network is not reachable from this host.

[EADDRINUSE] The address is already in use.

[EFAULT] The *name* parameter specifies an area outside the process address space.

[EWOULDBLOCK] The socket is nonblocking, and the connection cannot be completed immediately. You can select the socket for writing by using the `select` system call while it is connecting.

The following errors are specific to connecting names in the ULTRIX domain:

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

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named socket 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.

See Also

accept(2), fcntl(2), getsockname(2), select(2), shutdown(2), socket(2)

Name

creat – create a new file

Syntax

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

#include <limits.h> /*Definition of OPEN_MAX*/
creat(name, mode)
char *name;
mode_t mode;
```

Description

The `creat` system call 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. For further information, 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 zero length.

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

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 `flock`(2) facility.

No process may have more than OPEN_MAX files simultaneously.

Return Value

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

Environment

Differs from the System V definition in that ELOOP and ENXIO are possible error conditions, but ENFILE and ENOSPC are not.

Diagnostics

The `creat` system call fails and the file is not created or truncated under the following conditions:

[ENOTDIR]	A component of the path prefix is not a directory.
[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]	Too many files are open.
[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.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EFAULT]	The *name* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EOPNOTSUPP]	The file is a socket, which is not implemented.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[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.
[EDQUOT]	The directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the file is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[ESTALE]	The ''file handle'' given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.
[ETIMEDOUT]	A connect request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

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

Name

dup, dup2 – duplicate an open file descriptor

Syntax

newd = dup(*oldd*)
int newd, *oldd*;

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

Description

The `dup` system call duplicates an existing object descriptor. The argument *oldd* is a small non-negative integer index in the per-process descriptor table. The value must be less than the size of the table, which is returned by `getdtablesize`. The new descriptor, *newd,* returned by the call 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`, `write`, and `lseek` calls all move a single pointer into the file. If a separate pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional `open` call.

In the second form of the call, specify the value of *newd* needed. If this descriptor is already in use, the descriptor is first deallocated as if a `close` call had been done.

Return Value

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

Diagnostics

The `dup` and `dup2` system calls fail under the following conditions:

[EBADF]	The *oldd* or *newd* is not a valid active descriptor.
[EMFILE]	Too many descriptors are active.
[EINTR]	The `dup()` or `dup2()` function was terminated prematurely by a signal.

See Also

accept(2), close(2), getdtablesize(2), lseek(2), open(2), pipe(2), read(2), socket(2), socketpair(2), write(2)

errno (2)

Name

errno – introduction error numbers

Syntax

#include <errno.h>

Description

The `errno` external variable is set when an error occurs in a system call. You can use the value stored in errno to obtain a more detailed description of the error than is given in the system call's return value. The `errno` variable is not cleared on successful system calls, so you should check its value only when an error is reported.

Return Value

Most system calls have one or more return values. An error condition is indicated by an otherwise impossible return value. This value is almost always –1. All return codes and values from system call are of type *int,* unless otherwise noted.

When an error occurs, most calls store one of the following values, as defined in <errno.h>, in the errno variable:

0 Unused.

1 EPERM Not owner
This error indicates an attempt to modify a file in some way forbidden except to its owner or the superuser. It is also returned for attempts by ordinary users to do things allowed only to the superuser.

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but does not, or when one of the directories in a pathname does not exist.

3 ESRCH No such process
The process whose number was given to `kill` and `ptrace` does not exist or is already dead.

4 EINTR Interrupted system call
An asynchronous signal (such as interrupt or quit) that the program catches occurred during a system call. If execution resumes after the asynchronous signal is processed, 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 a `read` or `write`. This error may occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice that does not exist or to an area beyond the limits of the device. This error might also occur when 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 10240 bytes is presented to `execve`.

8 ENOEXEC Exec format error
A request is made to execute a file that does not start with a valid magic number, although it has the appropriate permissions. For further information, see `a.out`(5).

9 EBADF Bad file number
Either a file descriptor refers to no open file or a read request is made for a file that is open only for writing. Likewise, a write request made to a file open only for reading causes this error.

10 ECHILD No children
The program issued a `wait` call and the process has no active or unwaited-for children.

11 EAGAIN No more processes
In a *fork,* the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough core
During an `execve` or `brk`, a program asks for more core or swap space than the system is able to supply. A lack of swap space is normally a temporary condition. However, a lack of core is not a temporary condition; the maximum size of the text, data, and stack segments is a system parameter.

13 EACCES Permission denied
The call attempts to access a file in some way forbidden by the protection system.

14 EFAULT Bad address
The system encountered a hardware fault in attempting to access the arguments of a system call.

15 ENOTBLK Block device required
The call specifies a plain file where a block device is required.

16 EBUSY Mount device busy
The call attempts to mount a device that was already mounted or to unmount a device on which there was an active file directory, an open file, current directory, mounted-on file, or active text segment. Or, the call attempts to modify a partition table incorrectly. See the restrictions in `chpt`(8).

17 EEXIST File exists
An existing file is mentioned in an inappropriate context.

18 EXDEV Cross-device link
The call attempts to form a hard link to a file on another device.

19 ENODEV No such device
The call attempts to perform an invalid operation on a device, such as write to a read-only device.

20 ENOTDIR Not a directory
A file that is not a directory is specified where a directory is required, for example, in a pathname or as an argument to `chdir`.

21 EISDIR Is a directory
The call attempts to write on a directory.

22 EINVAL Invalid argument
An invalid argument is specified. For example, the call might specify dismounting a device that is not mounted or reading or writing a file for which `seek` has generated a negative pointer. This error is also set by math functions, as described in the `intro`(3) reference page.

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more `open` calls can be processed.

24 EMFILE Too many open files
The process has opened too many files. The customary configuration limit is 20 files per process.

25 ENOTTY Not a typewriter
The file named in an `ioctl` call is not a terminal or one of the other devices to which the call applies.

26 ETXTBSY Text file busy
The call attempts to execute a pure-procedure program that is currently open for writing or reading. Or, the call attempts to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeds the maximum (about 10^9 bytes).

28 ENOSPC No space left on device
A device runs out of space during a write to an ordinary file.

29 ESPIPE Illegal seek
An `lseek` call specifies a pipe or other device that `lseek` does not support.

30 EROFS Restricted operation on a file system
The call attempts to access a file or directory on a mounted file system when that permission has been revoked. For example, the call attempts to write a file on a file system mounted read only.

31 EMLINK Too many links
The call attempts to make more than {LINK_MAX} hard links to a file.

32 EPIPE Broken pipe
The call attempts to write on a pipe or socket for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Argument too large
The argument of a function in the math package (which is described in the *ULTRIX Reference Pages, Section 3: Subroutines*) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (which is described in the *ULTRIX Reference Pages, Section 3: Subroutines*) is unrepresentable within machine precision.

35 EWOULDBLOCK Operation would block
The call attempts an operation that would cause a process to block on an object in nonblocking mode. For further information, see `ioctl`(2).

36 EINPROGRESS Operation now in progress
The call is performing an operation that takes a long time to complete, such as a `connect` call, on a nonblocking object. For further information, see `ioctl`(2).

37 EALREADY Operation already in progress
The call attempts an operation on a nonblocking object that already has an operation in progress.

38 ENOTSOCK Socket operation on non-socket
The call attepmts to perform a socket-specific operation on an entity that is not a socket.

39 EDESTADDRREQ Destination address required
A required address is omitted from an operation on a socket.

40 EMSGSIZE Message too long
A message sent on a socket is larger than the internal message buffer.

41 EPROTOTYPE Protocol wrong type for socket
A protocol is specified that does not support the semantics of the socket type requested. For example, you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

42 ENOPROTOOPT Protocol not available
A bad option was specified in a `getsockopt` or `setsockopt` call.

43 EPROTONOSUPPORT Protocol not supported
The protocol has not been configured into the system or an implementation for it does not exist.

44 ESOCKTNOSUPPORT Socket type not supported
The support for the socket type has not been configured into the system or an implementation for it does not exist.

45 EOPNOTSUPP Error–operation not supported
The call attempts an unsupported operation, such as trying to accept a connection on a datagram socket.

46 EPFNOSUPPORT Protocol family not supported
The protocol family has not been configured into the system or an implementation for it does not exist.

47 EAFNOSUPPORT Address family not supported by protocol family
An address incompatible with the requested protocol is specified. For example, you cannot use PUP Internet addresses with ARPA Internet protocols.

48 EADDRINUSE Address already in use
The call attempts to use an address that is already in use. Each address can be used only once.

49 EADDRNOTAVAIL Cannot assign requested address
The call attempts to create a socket with an address not on this machine.

50 ENETDOWN Network is down
A socket operation encountered a network that is not operating.

51 ENETUNREACH Network is unreachable
A socket operation attempts to reach an unreachable network.

52 ENETRESET Network dropped connection on reset
The host to which the program was connected to crashed and rebooted.

53 ECONNABORTED Software caused connection abort
A connection abort has occurred internal to your host machine.

54 ECONNRESET Connection reset by peer
A connection has been forcibly closed by a peer. This error usually results from the peer executing a `shutdown` call.

55 ENOBUFS No buffer space available
The system lacks sufficient buffer space to perform an operation on a socket or pipe.

56 EISCONN Socket is already connected
A `connect` request names an already connected socket, or a `sendto` or `sendmsg` request on a connected socket specifies a destination other than the connected party.

57 ENOTCONN Socket is not connected
A request to send or receive data could not complete because the socket is not connected.

58 ESHUTDOWN Cannot send after socket shutdown
A request to send data could not complete because the socket has already been shut down with a previous `shutdown` call.

59 ETOOMANYREFS Too many references: cannot splice

60 ETIMEDOUT Connection timed out
A `connect` request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.) For example, this error occurs when an NFS file system is mounted with the "soft," option and the server is not responding to file operation requests.

61 ECONNREFUSED Connection refused
No connection could be made because the target machine actively refused it. This error usually results from trying to connect to a service that is inactive on the remote host.

62 ELOOP Too many levels of symbolic links
A pathname lookup involves more than eight symbolic links.

63 ENAMETOOLONG File name too long
A component of a path name exceeds 255 characters, or an entire path name exceeds 1023 characters.

64 EHOSTDOWN Host is down
A socket operation has failed because the destination host is down.

65 EHOSTUNREACH No route to host
A socket operation attempts to reach an unreachable host.

66 ENOTEMPTY Directory not empty
A directory with entries other than dot (.) and dot-dot (..) is specified in a `rmdir` or `rename` call.

67 EPROCLIM Too many processes
Creating the process would cause the user to exceed the number of user processes that are available. The *maxuprc* option in the configuration file controls this limit.

68 EUSERS Too many users
A login process would exceed the maximum allowable login processes for which the system is licensed.

69 EDQUOT Disk quota exceeded
A `write` to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry has failed because the user's quota of disk blocks is exhausted. Or, the allocation of an inode for a newly created file has failed because the user's quota of inodes is exhausted.

70 ESTALE Stale NFS file handle
Information used by the operating system to identify a file in an NFS file system that is no longer valid. This error code results from operating on a remote file that no longer exists on the server or resides in a file system that has been moved to a different device on the server.

71 EREMOTE Too many levels of remote in path
A remote NFS client has requested an operation on a file that is remote to the server as well. An attempt has been made to mount an NFS remote file system that is not local to the specified server. This error code cannot occur except in response to a failed `mount` call.

72 ENOMSG No message of desired type
An attempt was made to receive a message of a type that does not exist on the specified message queue. For further information, see `msgop`(2).

73 EIDRM Identifier removed
In semaphores, shared memory, or message queues, the caller tried to access the identifier after it had been removed from the system.

74 EALIGN Alignment error
Alignment error of some type has occurred, for example, cluster, page, or block.

75 ENOLCK No locks available
A file locking request could not be fulfilled because a system limit on the number of active locks would have been exceeded.

76 ENOSYS Function not implemented
The requested function is not available in ULTRIX. Included for POSIX compatibility only.

See Also

`perror`(3)

execve (2)

Name

execve – execute a file

Syntax

execve(name, argv, envp)
char *name, *argv[], *envp[];

Description

The `execve` system call 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 can be specified by the header to be initialized with zero data. For further information, see `a.out`(5).

An interpreter file begins with a line of the form ''#! *interpreter*''. When an interpreter file is executed the system executes the specified interpreter, giving it the name of the originally executed file as an argument, 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 an 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, the last component of *name*.

The argument *envp* is also an array of character pointers to null-terminated strings. These strings pass information to the new process, but they are not directly arguments to the command. For further information, 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. For further information, see `close`(2). Descriptors which remain open are unaffected by `execve`.

Ignored signals remain ignored across an `execve`, but signals that are caught are reset to their default values. The signal stack is reset to be undefined. For further information, see `sigvec`(2).

Each process has *real* user and group IDs and *effective* user and group IDs. The *real* ID identifies the person using the system; the *effective* ID determines his access privileges. The `execve` system call 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 getpid(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)
Resource usages	See getrusage(2)
Interval timers	See getitimer(2)
Resource limits	See getrlimit(2)
File mode mask	See umask(2)
Signal mask	See sigvec(2)

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

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

The *argc* argument is the number of elements in *argv* (the ''arg count'') and *argv* is the array of character pointers to the arguments themselves.

The *envp* argument 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 environ variable. Each string consists of a name, an equal sign (=), 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.

If `execve` returns to the calling process, an error has occurred; the return value is –1 and the global variable errno contains an error code.

Environment

POSIX, System Five

When your program is compiled using the POSIX or System V environment, the effective user ID and effective group ID of the new process image are saved (as the saved-set-uid and saved-set-gid) for later use by the `setuid`, `setgid`, and `kill` functions.

Restrictions

If a program's effective user ID is not the superuser, but it is executed when the real user ID is root, then the program has the powers of the superuser.

Diagnostics

The `execve` system call fails and returns to the calling process under the following conditions:

[ENOENT]	The new process file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[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 it has an invalid magic number in its header.

[ETXTBSY] The new process file is a pure procedure (shared text) file that is currently open for writing or reading by some process.

[ENOMEM] The new process requires more virtual memory than is allowed by the imposed maximum. For further information, see `getrlimit`(2).

[E2BIG] The number of bytes in the new process's argument list is larger than the system-imposed limit of {ARG_MAX} bytes.

[EFAULT] The new process file is not as long as indicated by the size values in its header.

[EFAULT] The *path*, *argv*, or *envp* points to an illegal address.

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

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EROFS] If binaries cannot be executed from the file system.

[EROFS] If `setuid` and `setgid` programs cannot be executed from the file system.

[ESTALE] The file handle given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.

[ETIMEDOUT] A connect request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

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

Name

_exit – terminate a process

Syntax

#include <stdlib.h>
void _exit(*status*)
int *status*;

Description

The function, `_exit`, terminates a calling process with the following consequences:

- All of the file descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a `wait`, it is notified of the calling process's termination and the low-order eight bits of *status* are made available to it. For further information, see `wait`(2).
- The parent process ID of all of the calling process's existing child processes and zombie processes are also set to 1. This means that the initialization process inherits each of these processes as well. For further information, see `intro`(2),
- Each attached shared memory segment is detached and the value of *shm_nattach* in the data structure associated with its shared memory identifier is decremented by 1.
- For each semaphore for which the calling process has set a semadj value, (see `semop`(2),) that semadj value is added to the semval of the specified semaphore.
- If the process has a process, text, or data lock, an unlock is performed.
- An accounting record is written on the accounting file if the system's accounting routine is enabled. For more information, see `acct`(2).

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

Environment

POSIX, System V

The `_exit` function differs from the System V as well as POSIX definition in that even if the calling process is a process group leader, the SIGHUP signal is not sent to each process that has a process group ID equal to that of the calling process.

The `_exit` function also differs in that the `exit` routine is declared as type *int* instead of type *void.*

See Also

fork(2), wait(2), exit(3), signal(3).

exportfs (2nfs)

Name

exportfs – exports an NFS file system

Syntax

#include <sys/mount.h>
exportfs(name, rootuid, exflags)
char *name;
int rootuid, exflags;

Description

The `exportfs` system call allows the specified local file system to be mounted remotely by an NFS client. This system call is usually called from `mountd`. Security on the exported file systems can be improved by setting the root mapped user ID, *rootuid,* and two mount structure flags, *exflags* for the local file system, *name*.

The *name* argument is a pointer to a null-terminated string containing the path name of the file system being exported.

The *rootuid* argument is used to set the user ID that root maps to. By default, root maps to user id -2.

The *exflags* argument contains the flags that are to be set in the mount structure corresponding to *name*. The following flags are the only possible flags accepted by `exportfs`:

```
#define M_NOFH          0x1000          /* no fhandle flag */
#define M_EXRONLY       0x2000          /* export read-only */
```

Setting the M_NOFH flag does not allow access to the fhandle of the file system's root gnode. The M_EXRONLY flag exports a filesystem read only.

The `exportfs` system call returns a value of 0 upon successful completion of a operation, and –1 upon failure.

Diagnostics

[EPERM]	Not superuser.
[EIO]	Not enough memory in the system to service the request.
[EFAULT]	Bad address or bad length of *name*.
[ENOENT]	The *name* cannot be found.

See Also

exports(5nfs), mountd(8nfs)

Name

fcntl – file control

Syntax

#include <fcntl.h>

res = fcntl (*fd, request, arg*)
int res;
int *fd, request, arg*

Arguments

The following arguments can be used with `fcntl`:

fd Descriptor to be operated on. Depending on the function selected by the *request* argument, the *fd* argument can be a file descriptor returned by an `open` system call, or a socket descriptor returned by a `socket` system call.

request Defines what you want done. The possible values are defined in `<fcntl.h>`. See the Description section for more information.

arg Varies according to the *request* argument. See the Description section for more information.

Description

The `fcntl` system call provides for control over descriptors. The descriptors can be either file descriptors returned by the `open` system call or socket descriptors returned by the `socket` system call.

Possible *request* arguments are the following:

F_DUPFD – Return New Descriptor

The shell provides an example of when a new descriptor is useful. Suppose the shell receives a command such as:

```
cat > myfile
```

The shell needs to redirect the output of the `cat` command from the file descriptor 1 (standard output) to a new file named `myfile`. The `fcntl` call, using the old file descriptor of 1, to obtain a new file descriptor for the file `myfile`.

F_DUPFD When *request* is set for F_DUPFD:

The `fcntl` call returns a new descriptor. The new file descriptor returned has the following characteristics:

- The file descriptor returned is the lowest numbered available descriptor that is greater than or equal to the argument *arg*.
- The descriptor has the same object references as the original descriptor. That is, if the original file descriptor referred to a file, the new file descriptor refers to a file. If the original descriptor referred to a socket, the new file descriptor refers to a socket.

- The new descriptor shares the same file pointer if the object was a file. (A file pointer points to an inode, which in turn points to a file. Thus, the new descriptor refers to the same file as the old descriptor.)
- The new descriptor has the same access mode as the old descriptor (read, write, or read/write).
- The new descriptor shares the same file status flags as the old file descriptor. (See the discussion of F_GETFL and F_SETFL for a description of file status flags.)
- The close-on-exec flag associated with the new file descriptor is set to remain open across `execve` system calls. (See the discussion of F_GETFD and F_SETFD for a description of the close-on-exec flag.)

F_GETFD and F_SETFD – Close-on-exec Flag

Each file descriptor points to an entry in an array of file pointers that, among other things, define certain characteristics for the file. One such characteristic is the close-on-exec flag. This flag defines whether or not a file remains open across calls to `execve`. If cleared, the file descriptor remains open in the new image loaded by the call to `execve`. If set, the file descriptor is closed in the new image loaded by the call to `execve`.

F_GETFD When *request* is set to F_GETFD:

The `fcntl` call returns the close-on-exec flag associated with the file descriptor *fd*. If the low-order bit of the value returned by `fcntl` is 0, the file remains open across calls to `execve`. If the low-order bit of the value returned by `fcntl` is 1, the file descriptor is closed across calls to `execve`.

F_SETFD When *request* is set to F_SETFD:

The `fcntl` call sets the close-on-exec flag associated with *fd* to the low-order bit of *arg* (0 or 1).

F_GETFl and F_SETFL – Descriptor Status

Each file descriptor points to an entry in an array of file pointers that, among other things, define the file's current status. One such item of status, for example, is whether or not input/output operations to a file are currently blocked.

You might want to program your process to allow blocking so that a user who runs your process in the background, while doing other work in the foreground, need not see output from the background job displayed on the screen.

These and other status indicators are discussed in the list that follows. Some status indicators do not apply to all types of descriptors. The O_APPEND status, for example, is meaningless for sockets.

F_GETFL When *request* is set to F_GETFL:

The `fcntl` call returns the file's descriptor status flags. The following names have been defined in `<fcntl.h>` for these status

flags:

O_NDELAY Nonblocking I/O. If no data is available to a `read` call, or if a write operation would block, the call returns –1 with the error [EWOULDBLOCK]. The flag FNDELAY is an obsolete synonym for O_NDELAY.

O_FSYNC (O_SYNC) Synchronous write flag. Forces subsequent file writes to be done synchronously. For further information, see `write`(2.) The flag OFSYNCRON is an obsolete synonym for FSYNCRON.

O_APPEND Force each write to append at the end of file. This corresponds to the action taken with the O_APPEND flag of `open`. The flag FAPPEND is an obsolete synonym for O_APPEND.

FASYNC Enable the SIGIO signal to be sent to the process group when I/O is possible. For example, send SIGIO when data is available to be read.

O_NONBLOCK
POSIX environment, nonblocking I/O flag. See O_NDELAY request for description of operation. The flag FNBLOCK is an obsolete synonym for O_NONBLOCK.

F_SETFL When *request* is set to F_SETFL:

The `fcntl` call sets descriptor status flags specified in *arg* (see F_GETFL). Refer to the F_SETOWN section for more information.

F_GETOWN and F_SETOWN – Get Or

With these requests, your process can recognize the software interrupts SIGIO or SIGURG. As described in `sigvec`, SIGIO is a signal indicating that I/O is possible on a descriptor. SIGURG indicates an urgent condition present on a socket.

F_GETOWN When *request* is set to F_GETOWN:

The `fcntl` call returns the process ID or process group currently receiving SIGIO and SIGURG signals. Process groups are returned as negative values.

F_SETOWN When *request* is set to F_SETOWN:

The `fcntl` call sets the process or process group to receive SIGIO and SIGURG signals; process groups are specified by supplying *arg* as negative. Otherwise, *arg* is interpreted as a process ID. Refer to the F_SETFL section for more information.

F_GETLK, F_SETLK, and F_SETLKW – Locking

With these requests, your process can:

- Test a file for a region that might have been read-locked or write-locked by another process.
- Set or clear a file region read or write lock.
- Set a file region read or write lock, sleeping, if necessary, until locks previously set by other processes are unlocked.

When a read lock has been set on a segment of a file, other processes can also set read locks on that file segment or portions thereof.

A read lock prevents any other process from write locking the protected area. More than one read lock can exist for a given region of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any other process from read locking or write locking the protected region. Only one write lock can exist for a given region of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

Locks can start and extend beyond the current end of a file, but cannot be negative relative to the beginning of the file.

Changing or unlocking a region from the middle of a larger locked region leaves two smaller regions with the old setting at either end. Locking a region that is already locked by the calling process causes the old lock to be removed and the new lock type to take effect.

All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a `fork`(a)2 system call.

F_GETLK When *request* is set to F_GETLK:

The `fcntl` call gets the lock information for a read or write locked region. In the call, you pass a lock description in a variable of type *struct flock* pointed to by *arg*.

If the region defined in the *flock* structure is already locked by a process other than the caller, a description of the existing lock is returned in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

The *flock* structure is defined as follows:

```
struct flock {
        short    l_type;
        short    l_whence;
        long     l_start;
        long     l_len;
        int      l_pid;
};
```

Data Passed in *flock*:

In the data you pass in *flock*, the *l_type* value defines the lock type to be tested for: F_RDLCK for a read lock and F_WRLCK for a write lock.

The *l_whence* value defines the point from which the starting byte of the region is to be measured. If *l_whence* is 0, the value in *l_start* is taken as the starting byte of the region. If *l_whence* is 1, the current file offset plus the value of *l_start* is taken as the starting point. If *l_whence* is 2, the file size plus the value of *l_start* is taken as the starting point.

The *l_len* value is the length of the region to be tested, in bytes. If *l_len* is zero, the length to be tested extends to the end of file. If *l_len* is zero and *l_start* is zero, the whole file is to be tested. If *l_len* is negative, the area affected starts at *l_start* + *l_len* and ends at *l_start* - 1.

The *l_pid* value has no significance in the data passed.

Data Returned in *flock*:

The *l_type* value can be F_RDLCK if the region passed is under a read lock. F_WRLCK means that the region passed is under a write lock. F_UNLCK means that the region is not currently locked by any process that would prevent this lock from being created; for example, the region might be locked by the caller.

The *l_whence*, *l_start*, and *l_len* values have similar meanings as discussed under Data Passed, except that they define the region currently under read or write lock.

The *l_pid* value is only used with F_GETLK to return the value for a blocking lock. An example of a blocking lock is a write lock currently set by a process other than the calling process.

F_SETLK When *request* is set to F_SETLK:

You set or clear a file region lock according to the variable of *l_type* in the *struct flock* pointed to by *arg*. (The *flock* structure is shown under the description of F_GETLK, preceding.)

The *l_type* value is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set, `fcntl` will return immediately with an error value of –1.

F_SETLKW When *request* is set to F_SETLKW:

The `fcntl` call takes the same action as for F_SETLK, except that if a read or write lock is blocked by other locks, the process sleeps until the segment is free to be locked.

Files and region locking are supported over the Network File System (NFS) services if you have enabled the NFS locking service.

Return Values

Upon successful completion, the value returned depends upon the *request* argument as follows:

F_DUPFDA new file descriptor.
F_GETFDValue of flag (only the low-order bit is defined).
F_GETFLValue of flags.
F_GETOWNValue of file descriptor owner.
otherValue other than –1.

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

Diagnostics

The `fcntl` fails if under the following conditions:

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

[EBADF] The environment defined is POSIX, the *request* argument is F_SETLK or F_SETLKW, the type of lock, *l_type*, is a shared lock (F_RDLCK), and *fildes* is not a valid file descriptor open for reading, or the type of lock, *l_type*, is an exclusive lock (F_WRLCK), and *fildes* is not a valid file descriptor open for writing.

[EFAULT] The *arg* is pointing to an address outside the process's allocated space.

[EINVAL] The *request* argument is F_DUPFD, and *arg* is negative or greater than the maximum allowable number. For further information, see `getdtablesize`(2).

[EINVAL] The *request* argument is F_SETSYN, to change the write mode of a file to synchronous, and this operation is not valid for the file descriptor. For example, the file was opened for read-only operations.

[EINVAL] The *request* argument is F_GETLK,F_SETLK, or SETLKW and the data *arg* points to is not valid.

[EINVAL] The *request* argument is invalid.

[EINVAL] The *fildes* argument refers to a file that does not support locking.

[EACCES] The *request* argument is F_SETLK, the type of lock (*l_type*) is a read (F_RDLCK) or write (F_WRLCK) lock, and the region of the file to be locked is already write locked by another process. Or, the type is a write lock and the region of the file to be locked is already read or write locked by another process. Or, the file is remotely mounted and the NFS locking service is not enabled.

[EMFILE] The *request* argument is F_DUPFD, and the maximum allowed number of file descriptors is currently open, or no file descriptors greater than or equal to *arg* are available.

[ENOSPC] The *request* argument is F_SETLK or F_SETLKW, the type of

lock is a read or write lock, and there are no more file locking headers available (too many files have segments locked). Or, there are no more record locks available (too many file segments locked).

[EDEADLK] The *request* argument is F_SETLKW, and the lock is blocked by some lock from another process that is sleeping (waiting) for that lock to become free. This detection avoids a deadlock situation.

[EOPNOTSUPP] Attempting an operation that is not valid for the file descriptor. This can occur if the file descriptor argument, *fd*, points to a socket address, and the *request* argument is only valid for files.

[EINTR] The *request* argument is F_SETLKW and the function was interrupted by a signal.

Environment

The `fcntl` description differs from the POSIX and XPG3 definitions in that ENOLCK is not a possible error condition.

See Also

close(2), execve(2), getdtablesize(2), open(2), sigvec(2), lockd(8c)

flock(2)

Name

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

Syntax

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

#define LOCK_SH    1    /* shared lock */
#define LOCK_EX    2    /* exclusive lock */
#define LOCK_NB    4    /* don't block when locking */
#define LOCK_UN    8    /* unlock */

flock(fd, operation)
int fd, operation;
```

Description

The `flock` system call applies or removes an *advisory* lock on the file associated with the file descriptor, *fd*. A lock is applied by specifying an *operation* parameter that is the inclusive OR of LOCK_SH or LOCK_EX and, possibly, LOCK_NB. To unlock an existing lock, *operation* should be LOCK_UN.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency; that is, processes might still access files without using advisory locks, possibly resulting in inconsistencies.

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. At any time, multiple shared locks can be applied to a file. However, multiple exclusive locks, or shared and exclusive locks cannot be applied simultaneously on a file.

A shared lock can be upgraded to be an exclusive lock, and an exclusive lock can become shared, simply by specifying the appropriate lock type. This change results in the previous lock being released and the new lock applied. When upgrading, do not include LOCK_NB in *operation,* because there is a possibility that other processes have requests for locks, or have gained or released a lock.

Requesting a lock on an object that is already locked normally causes the caller to blocked until the lock can be acquired. If LOCK_NB is included in *operation*, the call is not blocked; instead, the call fails and the error EWOULDBLOCK is returned.

Locks are on files, not file descriptors. That is, file descriptors duplicated through `dup` or `fork` call do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent loses its lock.

Processes blocked awaiting a lock may be awakened by signals.

Return Value

Zero is returned if the operation was successful; on an error, a –1 is returned and an error code is stored in the global variable, *errno*.

Diagnostics

The `flock` call fails under the following conditions:

[EWOULDBLOCK] The file is locked and the LOCK_NB option was specified.

[EBADF] The argument *fd* is an invalid descriptor.

[EINVAL] The argument *fd* refers to an object other than a file.

[EOPNOTSUPP] Invalid operation is requested. The argument *fd* refers to a socket.

Restrictions

File region locking is not supported over NFS.

See Also

close(2), dup(2), execve(2), fork(2), open(2)

fork(2)

Name

fork – create a new process

Syntax

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

pid = fork()
pid_t pid;

Description

The `fork` system call 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 (that is, 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 a `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 processes resource utilizations are set to 0. For further information, see `setrlimit`(2).

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.

Diagnostics

The `fork` system call fails and no child process are created under the following conditions:

[EAGAIN]	The system-imposed limit {PROC_MAX} on the total number of processes under execution would be exceeded.
[EAGAIN]	The system-imposed limit {CHILD_MAX} on the total number of processes under execution by a single user would be exceeded.
[ENOMEM]	There is insufficient swap space for the new process.

See Also

execve(2), wait(2)

Name

fsync – synchronize a file's in-core state with that on disk

Syntax

fsync(*fd*)
int *fd*;

Description

The `fcync` system call causes all modified data and attributes of *fd* to be moved to a permanent storage device. This results in all in-core modified copies of buffers for the associated file to be written to a disk.

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

Return Value

A 0 value is returned on success. A –1 value indicates an error.

Diagnostics

The `fsync` call fails under the following conditions:

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

[EINVAL] The *fd* argument refers to a socket.

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

[EINTR] The `fsync()` function was interrupted by a signal.

If an error occurs on an asynchronous write over NFS, the error cannot always be returned from a `write` system call. The error code is returned on `close` or `fsync`. The following are NFS-only error messages:

[EACCESS] The requested address is protected, and the current user has inadequate permission to access it.

[ENOSPC] There is no free space remaining on the file system containing the file.

[EDQUOT] The user's quota of disk blocks on the file system containing the file has been exhausted.

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

[ESTALE] The *fd* argument is invalid because the file referred to by that file handle no longer exists or has been revoked.

[ETIMEDOUT] A write operation failed because the server did not properly respond after a period of time that is dependent on the `mount`(8nfs) options.

See Also

sync(1), close(2), sync(2), write(2), update(8)

Name

getdirentries – gets directory entries in a generic directory format

Syntax

#include <sys/dir.h>

cc = getdirentries(fd, buf, nbytes, basep)
int cc, fd;
char *buf;
int nbytes;
long *basep;

Description

The `getdirentries` system call puts directory entries from the directory referenced by the file descriptor *fd* into the buffer pointed to by *buf,* in a generic directory format. Up to *nbytes* of data are transferred. The *nbytes* of data must be greater than or equal to the block size associated with the file. For further information, see `stat`(2). Sizes less than *nbytes* can cause errors on certain file systems.

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

```
unsigned long   d_ino;
unsigned short  d_reclen;
unsigned short  d_namlen;
char            d_name[MAXNAMLEN + 1];
```

The *d_ino* entry is a number that is unique for each distinct file in the file system. Files that are linked by hard links have the same *d_ino* . For further information, see `link`(2). The *d_reclen* entry is the length, in bytes, of the directory record. The *d_namlen* entry specifies the length of the file name. The *d_name* entry contains a null-terminated file name. Thus, the actual size of *d_name* can vary from 2 to MAXNAMLEN + 1.

The generic directory structures are not necessarily tightly packed. The *d_reclen* entry may be used as an offset from the beginning of a *direct* structure to the next structure, if any.

Upon return, the actual number of bytes transferred is returned. The current position pointer associated with *fd* is set to point to the next block of entries. The pointer is not necessarily incremented by the number of bytes returned by `getdirentries`. If the value returned is zero, the end of the directory has been reached. The current position pointer may be set and retrieved by `lseek`. The `getdirentries` system call writes the position of the block read into the location pointed to by *basep*. It is not safe to set the current position pointer to any value other than a value previously returned by `lseek` or a value previously returned in the location pointed to by *basep* or zero.

Return Value

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

Diagnostics

The `getdirentries` system call fails under the following conditions:

EBADF	The *fd* is not a valid file descriptor open for reading.
ENOTDIR	The *fd* is not a directory.
EFAULT	Either *buf* or *basep* points outside the allocated address space.
EIO	While reading from or writing to the file system, an I/O error occurred.
EINTR	A read from a slow device was interrupted by the delivery of a signal before any data arrived.
EPERM	The user does not have read permission in the directory.

NOTE

The `getdirentries` system call is not the suggested interface for reading directories. The `opendir`, `readdir`, and `telldir` routines offer a standard interface. See the `directory`(3) reference page for information on these routines.

See Also

close(2), link(2), lseek(2), open(2), stat(2), directory(3)

Name

getdomainname, setdomainname – get or set name of current domain

Syntax

getdomainname(*name*, *namelen*)
char **name*;
int *namelen*;

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

Description

The `getdomainname` system call returns the domain name of the current processor, as set by `setdomainname`.

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

The purpose of domains is to allow merging of two distinct networks that have common host names. Each network can be distinguished by having a different domain name. At the current time, only the Yellow Pages service makes use of domains.

The *name* argument is the address where the name of the current domain is stored.

The *namelen* argument specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

Restrictions

Domain names are limited to 31 characters.

Return Value

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

Diagnostics

[EFAULT]	The *name* parameter contains an invalid address.
[EPERM]	The caller was not the superuser. This error message only applies to the `setdomainname` system call.

getdtablesize (2)

Name

getdtablesize – get descriptor table size

Syntax

nds = getdtablesize()
int nds;

Description

Each process has a fixed size descriptor table that is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call `getdtablesize` returns the size of this table.

See Also

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

Name

getgid, getegid – get group identity

Syntax

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

gid = getgid()
gid_t gid;

egid = getegid()
gid_t egid;
```

Description

The `getgid` system call returns the real group ID of the current process, and the `getegid` call returns 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. The `getgid` call is most useful with processes that are ''set-group-ID.''

Environment

Differs from the System V definition in that the return values are of type *int,* instead of type *unsigned short.*

See Also

getuid(2), setregid(2), setgid(3)

getgroups(2)

Name

getgroups – get group access list

Syntax

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

int
getgroups(gidsetsize, gidset)
int gidsetsize;
gid_t *gidset;
```

Description

The `getgroups` call gets the current group access list of the user process and stores it in the array *gidset*. The *gidsetsize* parameter indicates the number of entries that can be placed in *gidset* and is modified on return to indicate the actual number of groups returned.

Return Value

Upon success, the call returns the actual number of groups returned to array *gidset*. No more than NGROUPS, as defined in <sys/param.h>, are returned.

A value of –1 indicates that an error occurred, and the error code is stored in the global variable, *errno*.

Environment

POXIX

When your program is compiled in the POSIX environment, the *gidset* argument should be defined as follows:

gid_t gidset[ ];

Additionally, in the POSIX environment, if the *gidsetsize* argument is zero, `getgroups` returns the number of supplemental group IDs associated with the calling process, without modifying the array pointed to by the *gidset* argument.

Diagnostics

The `getgroups` call fails under the following conditions:

[EINVAL]	The *gidsetsize* argument is smaller than the number of groups in the group set.
[EFAULT]	The *gidset* argument specifies invalid addresses.

See Also

setgroups(2), initgroups(3x)

Name

gethostid, sethostid – get or set the unique identifier of the current host

Syntax

hostid = gethostid()
int hostid;

sethostid(hostid)
int hostid;

Description

The `sethostid` system call establishes a 32-bit identifier for the current processor that is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the superuser and is normally performed at boot time.

Return Value

The `gethostid` system call returns the 32-bit identifier for the current processor.

See Also

hostid(1), gethostname(2)

gethostname(2)

Name

gethostname, sethostname – get or set the name of the current host

Syntax

gethostname(*name, namelen*)
char **name*;
int *namelen*;

sethostname(*name, namelen*)
char **name*;
int *namelen*;

Description

The `gethostname` system call returns the standard host name for the current processor, as previously set by `sethostname`. The *namelen* parameter specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

The `sethostname` system call sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the superuser and is normally used only when the system is bootstrapped.

Return Value

0 If the call succeeds, it returns a value of zero.

–1 If the call fails, a value of –1 is returned and an error code is placed in the global location, *errno*.

Restrictions

Host names are limited to 31 characters and may contain only lower case ASCII characters a to z, numbers 0 to 9, dashes (–), underscores (_), and periods (.).

Diagnostics

The `gethostname` system call fails under the following condition:

[EFAULT] The *name* parameter points outside the process's allocated address space.

The `sethostname` system call fails under the following conditions:

[EPERM] The caller is not the superuser.

[EINVAL] The *name* or *namelen* parameter is an invalid address.

See Also

hostname(1), gethostid(2)

Name

getitimer, setitimer – get or set value of interval timer

Syntax

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

#define ITIMER_REAL     0       /* real time intervals */
#define ITIMER_VIRTUAL  1       /* virtual time intervals */
#define ITIMER_PROF     2       /* user and system virtual time */

getitimer(which, value)
int which;
struct itimerval *value;

setitimer(which, value, ovalue)
int which;
struct itimerval *value, *ovalue;
```

Description

The system provides each process with three interval timers, defined in <sys/time.h>. The `getitimer` call returns the current value for the timer specified in *which*, while the `setitimer` call sets the value of a timer (optionally, returning the previous value of the timer).

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
      struct  timeval it_interval; /* timer interval */
      struct  timeval it_value;    /* current value */
};
```

If *it_value* is nonzero, it indicates the time to the next timer expiration. If *it_interval* is nonzero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is nonzero).

Time values smaller than the resolution of the system clock are rounded up to this resolution (on MIPS, 3.906 milliseconds; on VAX, 10 milliseconds).

The ITIMER_REAL timer decrements in real time. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

Three macros for manipulating time values are defined in <sys/time.h>. The *timerclear* sets a time value to zero, *timerisset* tests if a time value is nonzero, and *timercmp* compares two time values (beware that >= and <= do not work with this macro).

Return Value

If the calls succeed, a value of 0 is returned. If an error occurs, the value –1 is returned, and a more precise error code is placed in the global variable, *errno*.

Diagnostics

The possible errors are:

[EFAULT] The *value* structure specified a bad address.

[EINVAL] A *value* structure specified a time that was too large to be handled.

See Also

gettimeofday(2), sigvec(2), pause(3)

Name

getmnt – get information about mounted file systems

Syntax

#include <sys/types.h>
#include <sys/param.h>
#include <sys/mount.h>

getmnt(*start, buffer, nbytes, mode, path*)
int **start*;
struct fs_data **buffer*;
int *nbytes, mode*;
char **path*;

Description

The `getmnt` system call retrieves information about mounted file systems.

The *mode* argument is one of the following: `STAT_ONE`, `NOSTAT_ONE`, `STAT_MANY`, or `NOSTAT_MANY`.

If *mode* is `STAT_ONE` or `NOSTAT_ONE`, then *path* is the name of a single file system for which information is desired, *start* and *nbytes* are ignored, and *buffer* is assumed to be large enough to hold one `fs_data` structure.

If *mode* is `STAT_MANY` or `NOSTAT_MANY`, then *path* is ignored. The *start* argument is the current logical location within the internal system mount table and must be initially set to `0`. The *start* argument is updated to reflect the current logical location within the system mount table, allowing successive executions of `getmnt` to retrieve information about all the mounted file systems. The *nbytes* argument defines the size of *buffer*, into which the file system information is returned. Buffer sizes must be a multiple of `sizeof(struct fs_data)` bytes. Larger buffer sizes allow information about multiple file systems to be returned.

If *mode* is `NOSTAT_ONE` or `NOSTAT_MANY`, then dynamic `fs_data` information (the number of free inodes and the number of free blocks) could be out of date, but these calls are guaranteed to return. The file system information in memory is not updated.

If *mode* is `STAT_ONE` or `STAT_MANY`, then the file system information in memory is updated. However, if the server of any file system for which information is being retrieved is down, then these calls will hang until the server responds.

When information about multiple file systems is returned, it is stored within consecutive *buffer* locations. The information for each file system is described by the structure `fs_data`:

```
struct fs_data {
       struct fs_data_req  fd_req;       /* required data */
       u_int  fd_spare[113];             /* spare */
};     /* 2560 bytes */

struct fs_data_req {    /* required part for all file systems */
       u_int  flags;    /* how mounted */
       u_int  mtsize;   /* max transfer size in bytes */
       u_int  otsize;   /* optimal transfer size in bytes */
```

```
        u_int  bsize;    /* fs block size in bytes for vm code */
        u_int  fstype;   /* see ../h/fs_types.h  */
        u_int  gtot;     /* total number of gnodes */
        u_int  gfree;    /* # of free gnodes */
        u_int  btot;     /* total number of 1K blocks */
        u_int  bfree;    /* # of free 1K blocks */
        u_int  bfreen;   /* user consumable 1K blocks */
        u_int  pgthresh; /* min size in bytes before paging*/
        int    uid;      /* uid that mounted me */
        dev_t  dev;      /* major/minor of fs */
        dev_t  pad;      /* alignment: dev_t is a short*/
        char   devname[MAXPATHLEN + 4];  /* name of dev */
        char   path[MAXPATHLEN + 4];     /* name of mount point */
}
```

Return Value

Upon successful completion, a value indicating the number of `fs_data` structures stored in *buffer* is returned. If the file system is not mounted (*mode* is `STAT_ONE` or `NOSTAT_ONE`) or there are no more file systems in the mount table (*mode* is `STAT_MANY` or `NOSTAT_MANY`), `0` is returned. Otherwise, `-1` is returned and the global variable *errno* is set to indicate the error.

Diagnostics

[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[EINVAL]	Invalid argument.
[ENAMETOOLONG]	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
[ENOENT]	The file referred to by *path* does not exist.
[EACCESS]	Search permission is denied for a component of the path prefix of *path*.
[ELOOP]	Too many symbolic links were encountered in translating *path*.
[EFAULT]	Either *buffer* or *start* causes an illegal address to be referenced.
[EIO]	An I/O error occurred while reading from the file system.

See Also

gfsi(5)

Name

getpagesize – get system page size

Syntax

pagesize = getpagesize()
int pagesize;

Description

The `getpagesize` system call returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a *system* page size and may not be the same as the underlying hardware page size.

See Also

pagesize(1), sbrk(2)

getpeername (2)

Name

getpeername – get name of connected peer

Syntax

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

getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;

Description

The `getpeername` returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return, it contains the actual size, in bytes, of the name returned.

Return Value

A zero is returned if the call succeeds, and –1 is returned if it fails.

Restrictions

Names bound to sockets in the UNIX domain are inaccessible; `getpeername` returns a zero length name.

Diagnostics

The call succeeds unless:

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

See Also

bind(2), getsockname(2), socket(2)

Name

getpgrp – get process group

Syntax

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

pgrp = getpgrp(*pid***)**
pid_t pgrp;
pid_t ***pid*****;**

Description

The system call `getprgrp` returns the process group of the specified process. If *pid* is zero, the call applies to the current process.

Process groups are used for distribution of signals and by terminals to arbitrate requests for their input. Processes that have the same process group as the terminal are the foreground and may read, while others block with a signal if they attempt to read.

This call is used by programs such as `csh`(1) to create process groups in implementing job control. The TIOCGPGRP and TIOCSPGRP calls described in `tty`(4) are used to get and set the process group of the control terminal.

Environment

When your program is compiled in the System V or POSIX environment, `getpgrp` is called without arguments and the process group of the current process is returned.

Additionally, in POSIX mode, `getpgrp` returns a value type of *pid_t*.

Diagnostics

The `getpgrp` call fails under the following condition:

[ESRCH] No such process, PID.

See Also

getuid(2), setpgrp(2), tty(4)

getpid(2)

Name

getpid, getppid – get process identification

Syntax

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

pid = getpid()
pid_t pid;

ppid = getppid()
pid_t ppid;
```

Description

The `getpid` system call returns the process ID of the current process. Most often it is used, with the host identifier `gethostid`, to generate uniquely named temporary files.

Return Value

The `getppid` system call returns the process ID of the parent of the current process.

Environment

POSIX

When your program is compiled in POSIX mode, the `getpid` and `getppid` functions return a value of type *pid_t*.

See Also

gethostid(2)

Name

getpriority, setpriority – get or set program scheduling priority

Syntax

```
#include <sys/time.h>
#include <sys/resource.h>

#define PRIO_PROCESS     0      /* process */
#define PRIO_PGRP        1      /* process group */
#define PRIO_USER        2      /* user id */

prio = getpriority(which, who)
int prio, which, who;

setpriority(which, who, prio)
int which, who, prio;
```

Description

The scheduling priority of the process, process group, or user, as indicated by *which* and *who,* is obtained with the `getpriority` call and set with the `setpriority` call. The *which* is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, and *who* is interpreted relative to *which* (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user ID for PRIO_USER). The *prio* is a value in the range –20 to 20. The default priority is 0; lower priorities cause more favorable scheduling.

The `getpriority` call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The `setpriority` call sets the priorities of all of the specified processes to the specified value. Only the superuser may lower priorities.

Return Value

Since `getpriority` can legitimately return the value –1, it is necessary to clear the external variable *errno* prior to the call, then check it afterward to determine if a –1 is an error or a legitimate value. The `setpriority` call returns 0 if there is no error or –1 if there is.

Diagnostics

The `getpriority` and `setpriority` system calls fail under the following conditions:

[ESRCH] No processes were located using the *which* and *who* values specified.

[EINVAL] The *which* was not one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.

In addition to the errors indicated above, *setpriority* can fail under the following conditions:

[EPERM] A process was located, but neither its effective nor real user ID matched the effective user ID of the caller.

[EACCES] A user other than the superuser attempted to change a process priority to a negative value.

See Also

nice(1), fork(2), renice(8)

Name

getrlimit, setrlimit – control maximum system resource consumption

Syntax

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

getrlimit(resource, rlp)
int resource;
struct rlimit *rlp;

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

Description

Limits on the consumption of system resources by the current process and each process it creates can be obtained with the `getrlimit` call and set with the `setrlimit` call.

The *resource* parameter is one of the following:

RLIMIT_CPU	the maximum amount of cpu time (in milliseconds) to be used by each process.
RLIMIT_FSIZE	the largest size, in bytes, of any single file that may be created.
RLIMIT_DATA	the maximum size, in bytes, of the data segment for a process. This limit defines how far a program can extend its break with the `sbrk` system call.
RLIMIT_STACK	the maximum size, in bytes, of the stack segment for a process. This limit defines how far a program's stack segment can be extended, either automatically by the system or explicitly by a user, with the `sbrk` system call.
RLIMIT_CORE	the largest size, in bytes, of a *core* file that may be created.
RLIMIT_RSS	the maximum size, in bytes, to which a process's resident set size may grow when there is a shortage of free physical memory. Exceeding this limit when free physical memory is in short supply results in an unfavorable scheduling priority being assigned to the process.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded, a process may receive a signal (for example, if the cpu time is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The system uses just the soft limit field of the resources RLIMIT_CORE and RLIMIT_RSS. The *rlimit* structure is used to specify the hard and soft limits on a resource, as shown:

```
struct rlimit {
        int     rlim_cur;       /* current (soft) limit */
        int     rlim_max;       /* hard limit */
};
```

Only the superuser may raise the maximum limits. Other users may alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

An ''infinite'' value for a limit is defined as RLIM_INFINITY (0x7fffffff).

Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to `csh`.

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached. Because the stack cannot be extended, there is no way to send a signal.

A file I/O operation that creates too large a file causes the SIGXFSZ signal to be generated. This condition normally terminates the process, but may be caught. When the soft cpu time limit is exceeded, a signal SIGXCPU is sent to the process.

Return Value

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

Environment

System Five

When your program is compiled in the System V environment, the SIGXFSZ signal is not generated.

Diagnostics

The `getrlimit` call fails under the following conditions:

[EFAULT]	The address specified for *rlp* is invalid.
[EPERM]	The limit specified to `setrlimit` would have raised the maximum limit value, and the caller is not the superuser.
[EINVAL]	Resource is greater than or equal to RLIM_NLIMITS.

See Also

csh(1), quota(2)

Name

getrusage – get information about resource utilization

Syntax

```
#include <sys/time.h>
#include <sys/resource.h>

#define RUSAGE_SELF      0      /* calling process */
#define RUSAGE_CHILDREN  -1     /* terminated child processes */

getrusage(who, rusage)
int who;
struct rusage *rusage;
```

Description

The `getrusage` system call returns information describing the resources utilized by the current process or all its terminated child processes. The *who* parameter is one of RUSAGE_SELF and RUSAGE_CHILDREN. If *rusage* is nonzero, the buffer it points to will be filled in with the following structure:

```
struct  rusage {
        struct timeval ru_utime;        /* user time used */
        struct timeval ru_stime;        /* system time used */
        int     ru_maxrss;
        int     ru_ixrss;       /* integral shared text size */
        int     ru_ismrss       /* integral shared memory size */
        int     ru_idrss;       /* integral unshared data size */
        int     ru_isrss;       /* integral unshared stack size */
        int     ru_minflt;      /* page reclaims */
        int     ru_majflt;      /* page faults */
        int     ru_nswap;       /* swaps */
        int     ru_inblock;     /* block input operations */
        int     ru_oublock;     /* block output operations */
        int     ru_msgsnd;      /* messages sent */
        int     ru_msgrcv;      /* messages received */
        int     ru_nsignals;    /* signals received */
        int     ru_nvcsw;       /* voluntary context switches */
        int     ru_nivcsw;      /* involuntary context switches */
};
```

The fields are interpreted as follows:

ru_utime The total amount of time spent executing in user mode.

ru_stime The total amount of time spent in the system executing on behalf of the processes.

ru_maxrss
The maximum resident set size utilized (in bytes).

ru_ixrss An ''integral'' value indicating the amount of text memory used that was also shared among other processes. This value is expressed in units of kilobytes * seconds-of-execution and is calculated by summing the number of shared memory pages in use each time the internal system clock ticks and then averaging over 1-second intervals.

ru_ismrss An integral value of the amount of shared memory residing in the data

segment of a process (expressed in units of kilobytes * seconds-of execution).

ru_idrss An integral value of the amount of unshared memory residing in the data segment of a process (expressed in units of kilobytes * seconds-of-execution).

ru_isrss An integral value of the amount of unshared memory residing in the stack segment of a process (expressed in units of kilobytes * seconds-of-execution).

ru_minflt The number of page faults serviced without any I/O activity; here, I/O activity is avoided by ''reclaiming'' a page frame from the list of pages awaiting reallocation.

ru_majflt The number of page faults serviced that required I/O activity.

ru_nswap The number of times a process was ''swapped'' out of main memory.

ru_inblock
The number of times the file system had to perform input.

ru_oublock
The number of times the file system had to perform output.

ru_msgsnd
The number of ipc messages sent.

ru_msgrcv
The number of ipc messages received.

ru_nsignals
The number of signals delivered.

ru_nvcsw The number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed, usually to await availability of a resource.

ru_nivcsw
The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

The numbers *ru_inblock* and *ru_oublock* account only for real I/O. Data supplied by the cacheing mechanism is charged only to the first process to read or write the data.

Restrictions

There is no way to obtain information about a child process that has not yet terminated.

Diagnostics

The `getrusage` call fails under the following conditions:

[EINVAL] The *who* parameter is not a valid value on RUSAGE_SELF or RUSAGE_CHILDREN.

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

See Also

gettimeofday(2), wait(2)

getsockname (2)

Name

getsockname – get socket name

Syntax

#include <sys/typos.h>
#include <sys/socket.h>

getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;

Description

The `getsockname` system call returns the current *name* for the specified socket descriptor **s.** The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size, in bytes, of the name returned.

Return Value

A zero is returned if the call succeeds, –1 if it fails.

Restrictions

Names bound to sockets in the UNIX domain are inaccessible; `getsockname` returns a zero-length name.

Diagnostics

The call succeeds unless:

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

See Also

bind(2), socket(2)

Name

getsockopt, setsockopt – get or set options on sockets

Syntax

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

getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int optlen;
```

Description

The `getsockopt` and `setsockopt` system calls manipulate options associated with a socket. Options can exist at multiple protocol levels; they are always present at the uppermost socket level.

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

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

The *optname* parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <sys/socket.h> contains definitions for socket level options. For further information, see `socket`(2). Options at other protocol levels vary in format and name. Consult the `arp`(4p), `ip`(4p), `tcp`(4p) or `udp`(4p) reference pages for details.

Return Value

A zero is returned if the call succeeds, and –1 is returned if it fails.

Diagnostics

The `getsockopt` call fails under the following conditions:

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

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

[ENOPROTOOPT]
The option is unknown.

[EFAULT] The address pointed to by *optval* is not in a valid part of the process address space. For `getsockopt`, this error can also be returned if *optlen* is not in a valid part of the process address space.

See Also

socket(2), getprotoent(3n), *Guide to the Data Link Interface*

Name

getsysinfo – get system information

Syntax

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

getsysinfo(op, buffer, nbytes, start, arg)
unsigned        op;
char            *buffer;
unsigned        nbytes;
int             *start;
char            *arg;
```

Description

The `getsysinfo` system call retrieves information from the system.

The *op* argument specifies the operation to be performed. Values for *op* are defined in the <sys/sysinfo.h> header file.

Possible *op* values are as follows:

GSI_BOOTDEV
Return the BOOTDEV string, which is used for the installation.

GSI_NETBLK
Return the entire NETBLK structure, which is used for the network installation.

GSI_PROG_ENV
Return the compatibility mode of the process. Possible values are A_BSD, A_POSIX, A_SYSTEM_FIVE as defined in <sys/exec.h>.

GSI_MAX_UPROCS
Return the maximum number of processes allowed per user id.

GSI_TTYP
Return the major and minor numbers of the controlling terminal.

GSI_UACSYS (RISC only)
Return current value of flag that determines whether or not to print "unaligned access fixup" message on a system-wide basis.

GSI_UACPARNT (RISC only)
Return current value of flag in parent process's structure for printing unaligned access messages.

GSI_UACPROC (RISC only)
Return current value of flag in process's structure for printing of unaligned access messages.

The *nbytes* argument defines the size of *buffer* into which the system information is returned.

The *start* argument is the current logical location within the internal system table referenced by the op, and it must be initially set to `0`. The *start* argument is updated to reflect the current logical location within the system table, allowing successive

executions of `getsysinfo` to retrieve information about all the system structures specified by op.

The *start* argument is set to 0 when all system information requested by *op* has been retrieved.

The optional *arg* argument may be used by certain *op's* for additional information. When *arg* is not required, it should be set to NULL.

When information about multiple system structures is returned, it is stored within consecutive *buffer* locations. The information for each system structure is dependent upon *op*.

Return Value

Upon successful completion, a value indicating the number of requested items stored in *buffer* is returned. If the information requested by *op* is not available, *getsysinfo* returns a zero. Otherwise, –1 is returned, and the global variable, `errno`, is set to indicate the error.

Diagnostics

[EFAULT]	Either *buffer, start,* or *arg* causes an illegal address to be referenced.
[EINVAL]	The *op* argument is invalid.
[EPERM]	Permission is denied for the operation requested

See Also

setsysinfo(2)

Name

gettimeofday, settimeofday – get or set date and time

Syntax

#include <sys/time.h>

gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;

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

Description

The `gettimeofday` system call returns the system's notion of the current Greenwich time and the current time zone. Time returned is expressed relative in seconds and microseconds since midnight January 1, 1970.

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

```
struct timeval {
        long    tv_sec;             /* seconds since Jan. 1, 1970 */
        long    tv_usec;      /* and microseconds */
};

struct timezone {
        int     tz_minuteswest;      /* of Greenwich */
        int     tz_dsttime;   /* type of dst correction to apply */
};
```

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

Only the superuser can set the time of day.

Return Value

A 0 return value indicates that the call succeeded. A –1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

Diagnostics

The `gettimeofday` call fails under the following conditions:

[EFAULT]	An argument address referenced invalid memory.
[EPERM]	A user other than the superuser attempted to set the time.

See Also

date(1), stime(2), ctime(3)

getuid(2)

Name

getuid, geteuid – get user identity

Syntax

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

uid = getuid()
uid_t uid;

euid = geteuid()
uid_t euid;
```

Description

The `getuid` system call 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 which invoked them.

Environment

System Five

Differs from the System V definition in that the return values are of type *int,* instead of *unsigned short.*

POSIX

When your program is compiled in POSIX mode, the `getuid` and `geteuid` functions return a value of type *uid_t.* The `getgid` and `getegid` functions return a value of type *gid_t.*

See Also

getgid(2), setreuid(2)

Name

ioctl – control device

Syntax

#include <sys/ioctl.h>

ioctl(d, request, argp)
int d, request;
char *argp;

Description

The `ioctl` call performs a variety of functions on open descriptors. In particular, many operating characteristics of character special files (for example, terminals) can be controlled with `ioctl` requests. Certain `ioctl` requests operate on a number of device types. These include informational `ioctl` requests, such as `devio` and `nbuf`. The descriptions of various devices in the *Reference Pages, Section 4: Special Files* discuss how `ioctl` applies to them. Also consult <sys/ioctl.h> for more information.

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.

Diagnostics

The `ioctl` call fails under the following conditions:

[EBADF]	The *d* is not a valid descriptor.
[ENOTTY]	The *d* is not associated with a character special device.
[ENOTTY]	The specified request does not apply to the kind of object which the descriptor *d* references.
[EINVAL]	The *request* or *argp* is not valid.
[EFAULT]	The *argp* points to memory that is not part of the process' address space.

See Also

execve(2), fcntl(2), devio(4), intro(4n), mu(4), nbuf(4), tty(4)

kill(2)

Name

kill – send signal to a process

Syntax

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

kill(pid, sig)
pid_t pid;
int sig;
```

Description

The system call `kill` sends the signal *sig* to a process specified by the process number *pid*. The *sig* can be a signal specified in a `sigvec` call or it can be 0. If the *sig* is 0, error checking is performed, but a signal is not sent. This call 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 superuser with the exception of the signal SIGCONT. The signal SIGCONT can always be sent to a child or grandchild of the current process.

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

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.

The above two options are variants of `killpg`.

If the process number is –1, and the user is the superuser, the signal is broadcast for all processes except to system processes and the process sending the signal.

Processes may send signals to themselves.

Environment

System Five
POSIX

When your program is compiled in the System V or POSIX environment, a signal is sent if either the real or effective uid of the sending process matches the real or saved-set-uid (as described in `execve`(2)) of the receiving process. In addition, any process can use a *pid* of –1, and the signal is sent to all processes subject to these permission checks.

In POSIX mode, the *pid* argument is of type *pid_t*.

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.

Diagnostics

The `kill` system call fails under the following conditions:

[EINVAL]	The *sig* is not a valid signal number.
[EPERM]	The sending process is not the superuser, and its effective user ID does not match the effective user ID of the receiving process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

See Also

execve(2), getpgrp(2), getpid(2), killpg(2), sigvec(2), pause(3)

killpg(2)

Name

killpg – send signal to process or process group

Syntax

killpg(*pgrp, sig*)
int *pgrp, sig*;

Description

The `killpg` system call sends the signal *sig* to the process group *pgrp*. See `sigvec`(2) for a list of signals.

The sending process and members of the process group must have the same effective user ID, otherwise this call is restricted to the superuser with the exception of the signal SIGCONT. The signal SIGCONT can be sent to any process which is a descendant of the current process.

Return Value

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

Diagnostics

The `killpg` system call fails and a signal is not sent under the following conditions:

[EINVAL]	The *sig* is not a valid signal number.
[EPERM]	The sending process is not the superuser and all of the target processes have an effective user ID that differs from that of the sending process.
[ESRCH]	No process can be found corresponding to that specified by *pgrp*.

See Also

getpgrp(2), kill(2), sigvec(2)

Name

link – link to a file

Syntax

link(*name1, name2*)
char **name1, *name2*;

Description

A hard link to *name1* is created; the link has the name *name2*. The *name1* must exist.

With hard links, both *name1* and *name2* must be in the same file system. Unless the caller is the superuser, *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.

Diagnostics

The `link` system call fails and no link is created under the following conditions:

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

[ENAMETOOLONG] A component of either pathname exceeded 255 characters, or the entire length of either pathname exceeded 1023 characters.

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

[ENOENT] The file named by *name1* does not exist.

[ENOENT] When *name1* or *name2* point to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

[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.

[EEXIST] The link named by *name2* does exist.

[EPERM] The file named by *name1* is a directory, and the effective user ID is not that of superuser or the environment defined is POSIX.

[EXDEV] The link named by *name2* and the file named by *name1* are on different file systems.

[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.

[ELOOP] Too many symbolic links were encountered in translating one of the pathnames.

[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.
[EDQUOT]	The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EIO]	An I/O error occurred while reading from or writing to the file system to make the directory entry.
[ESTALE]	The file handle given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.
[ETIMEDOUT]	A connect request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communications protocol.
[EMLINK]	The number of links to the file named by *path1* would exceed {LINK_MAX}.

Environment

In the POSIX environment, linking to directories is not allowed.

See Also

symlink(2), unlink(2)

Name

listen – listen for connections on a socket

Syntax

listen(s, backlog)
int s, backlog;

Description

To accept connections, a socket is first created with a `socket` call, a backlog for incoming connections is specified with `listen`, and then the connections are accepted with the `accept` call. The `listen` call is needed only for sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full, the client receives an error with an indication of ECONNREFUSED.

Restrictions

The *backlog* is currently limited to 8.

Return Value

A 0 return value indicates success. A –1 indicates an error.

Diagnostics

The call fails under the following conditions:

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

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

[EOPNOTSUPP] The socket is not of a type that supports the operation `listen`.

See Also

accept(2), connect(2), socket(2)

lseek (2)

Name

lseek, tell – move read or write pointer

Syntax

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

pos = lseek(d, offset, whence)
off_t pos;
int d, whence;
off_t offset;

pos = tell(d)
off_t pos;
int d;
```

Description

The system call `lseek` moves the file pointer associated with a file or device open for reading or writing.

The descriptor *d* refers to a file or device open for reading or writing. The `lseek` system call 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*.

Seeking beyond the end of a file and then writing to the file creates a gap or hole that does not occupy physical space and reads as zeros.

The `tell` system call returns the offset of the current byte relative to the beginning of the file associated with the file descriptor.

Environment

System Five

If you compile a program in the System Five environment, an invalid *whence* argument causes SIGSYS to be sent. This complies with the behavior described in the System V Interface Definition (SVID), Issue 1.

Return Value

Upon successful completion, a long integer (the current file pointer value) is returned. This pointer is measured in bytes from the beginning of the file, where the first byte is byte 0. (Note that some devices are incapable of seeking. The value of the pointer associated with such a device is undefined.) If a value of –1 is returned, *errno* is set to indicate the error.

Diagnostics

The `lseek` system call fails and the file pointer remains unchanged under the following conditions:

[EBADF]	The *fildes* is not an open file descriptor.
[EINVAL]	The *whence* is not a proper value.
[ESPIPE]	The *fildes* is associated with a pipe or a socket.

See Also

dup(2), open(2)

mkdir (2)

Name

mkdir – make a directory file

Syntax

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

mkdir(*path, mode*)
char **path*;
mode_t *mode*;

Description

The `mkdir` system call 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. For further information, 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. For further information, 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*.

Diagnostics

The `mkdir` system call fails and a directory is not created if the following occurs:

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

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

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

[ENOENT] A component of the path prefix does not exist or the path argument points to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

[EACCES] Search permission is denied for a component of the path prefix, or write permission is denied on the parent directory to be created.

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

[EEXIST] The named file exists.

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

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

[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 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 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.

[EDQUOT] The directory in which the entry for the new directory is being placed cannot be extended, because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The new directory cannot be created, because the user's quota of disk blocks on the file system that will contain the directory has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the directory is being created has been exhausted.

[ESTALE] The file handle given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.

[ETIMEDOUT]
A "connect" request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

[EMLINK] The link count of the parent directory would exceed {LINK_MAX}.

See Also

chmod(2), stat(2), umask(2)

mknod(2)

Name

mknod – make a directory or a special file

Syntax

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

int mknod(path, mode, dev)
char *path;
mode_t mode;
int dev;
```

Description

The `mknod` system call creates a new file whose name is *path*. The mode of the new file (including special file bits) is initialized from *mode*, where the value of *mode* is interpreted as follows:

S_IFMT-0170000 File type; one of the following:
- S_IFIFO-0010000 FIFO special
- S_IFCHR-0020000 Character special
- S_IFDIR-0040000 Directory
- S_IFBLK-0060000 Block special
- S_IFREG-0100000
  - or 0000000 Ordinary file

S_IRWXU-0007000 Execution mode; made from the following:
- S_ISUID-0004000 Set user ID on execution
- S_ISGID-0002000 Set group ID on execution
- S_ISVTX-0001000 Save text image after execution

00777 Access permissions; made from the following:
- S_IREAD-0000400 Read by owner
- S_IWRITE-0000200 Write by owner
- S_IEXEC-0000100 Execute (search on directory) by owner
- s_IRWXG-0000070 Read, write, execute (search) by group
- S_IRWXD-0000007 Read, write, execute (search) by others

The file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's effective group ID.

Values of *mode* other than those in the preceeding list are undefined and should not be used. The low-order nine 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. For further information, see `umask`(2). If *mode* indicates a block or character special file, *dev* is a configuration dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

For file types other than FIFO special, only the superuser can invoke the `mknod` system call.

Return Value

The `mknod` system call returns a value of 0 upon successful completion. Otherwise, `mknod` returns a value of –1, and sets *errno* to indicate the error.

Diagnostics

The `mknod` system call fails and the file mode is unchanged under the following conditions:

[EPERM] The process's effective user ID is not superuser.

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

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

[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.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

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

[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.

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

[EDQUOT] The directory in which the entry for the new node is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the node is being created has been exhausted.

[ESTALE] The file handle given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.

[ETIMEDOUT] A connect request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

mkdir(1), chmod(2), execve(2), stat(2), umask(2), fs(5)

mount(2)

Name

mount, umount – mount or unmount a file system

Syntax

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

mount(*special, name, rwflag, type, options*)
char **special*, **name*;
int *rwflag*, *type*;
char **options*;

umount(*dev*)
dev_t *dev*;

Description

The `mount` system call announces to the system that a file system has been mounted on special file, *special*. References to file *name* refer to the root file on the newly mounted file system.

The *special* argument is a pointer to a null-terminated string containing the pathname of the file system being mounted.

The *name* argument is a pointer to a null-terminated string containing the pathname of the root file on the newly mounted file system. The *name* must already exist and must be a directory. Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument is used to determine whether the file system can be written on; if it is 0, the file system is writable, if nonzero, the file system is write-protected. Physically write-protected disks and magnetic tape file systems must be mounted read-only. The `mount` call also detects devices that are offline at mount time and returns the appropriate error.

The type argument identifies the file system type that is being mounted. The file system types are defined in the <fs_types.h> file.

The *options* argument specifies certain parameters that can be used to define how the file system is to be mounted.

The *dev* argument identifies the device that contains the file system that is to be unmounted.

Environment

Programs compiled in the System V environment cause `mount` and `umount` to set errno to ENOTDIR, instead of EPERM (illegal char in directory name) or EROFS (directory on read only filesystem). If the process is not the superuser, errno is set to EPERM, instead of ENODEV, and if the file does not exist, errno is set to ENOENT, instead of ENODEV.

Also in the System V environment, only the low-order bit of *rwflag* is checked to determine write permission.

Return Value

The `mount` system call returns 0 upon successful completion of a mount operation; it returns –1 if the mount operation fails.

The `umount` system call announces to the system that the device *dev* no longer contains a file system. The associated directory reverts to its ordinary interpretation.

The `umount` system call returns 0 if the dismount operation succeeds; –1 if it fails.

Diagnostics

The `mount` call fails under the following conditions:

[EPERM] The caller is not the superuser.

[ENODEV] A component of *special* does not exist or the device is offline.

[ENOTBLK] The *special* is not a block device.

[ENXIO] The major device number of *special* is out of range (indicating that no device driver exists for the associated hardware).

[EINVAL] The file system type is out of range.

[EINVAL] The super block for the file system had a bad magic number or an out-or-range block size.

[EINVAL] The file system has not been unmounted cleanly, and the force option has not been set.

[ENOTDIR] A component of *name* is not a directory, or a path prefix of *special* is already mounted.

[EBUSY] Another process currently holds a reference to *name*, or *special* is already mounted.

[ENAMETOOLONG] A component of either pathname exceeded 255 characters, or the entire length of either pathname exceeded 1023 characters.

[ELOOP] Too many symbolic links were encountered in translating either pathname.

[ENOENT] A component of *name* does not exist.

[EMFILE] No space remains in the mount table.

[ENOMEM] Not enough memory was available to read the cylinder group information for the file system.

[EIO] An I/O error occurred while reading the super block or cylinder group information.

[EFAULT] The *special* or *name* points outside the process's allocated address space space.

[EROFS] The *special* is a write-locked device and the user did not set the *rwflag*.

The `umount` command fails under the following conditions:

[EPERM] The caller is not the superuser.

[EINVAL]	The requested device is not in the mount table.
[EBUSY]	A process is holding a reference to a file located on the file system.
[EIO]	An I/O error occurred while writing the super block or other cached file system information.
[EREMOTE]	An attempt has been made to mount an NFS remote file system that is not local to the specified server. This cannot occur except in response to a failed `mount`(2).
[ETIMEDOUT]	A connect request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

mount(2nfs), mount(8), umount(8)

Name

mount, umount – mount or remove an NFS file system

Syntax

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

mount(special, name, rwflag, type, options)
char *special, *name;
int rwflag, type;
char *options;

unmount(dev)
dev_t dev;
```

Description

The `mount` system call announces to the system that a remote NFS file system has been mounted on directory *name*. References to file *name* refer to the root file on the newly mounted file system.

The *special* argument is a pointer to a null-terminated string containing the pathname of the file system being mounted. It is of the form:

```
host:pathname
```

The *name* argument is a pointer to a null-terminated string containing the pathname of the root file on the newly mounted file system. The *name* must already exist and must be a directory. Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument is used to determine whether the file system can be written to; if it is 0, the file system is writable, if nonzero, the file system is write-protected.

The *type* argument identifies the file system type that is being mounted. The DEFINE statement for the *nfs* type is:

```
#define GT_NFS    0x04
```

The *nfs* file system type is defined in the <fs_types.h> file.

The *options* argument specifies certain parameters that can be used to define how the file system is to be mounted. The `mount(8nfs)` () description lists the available NFS options.

The following structure is used by the `nfs_mount` user-level routine as the fifth argument when making a `mount` system call.

```
struct nfs_gfs_mount {
       struct sockaddr_in addr;    /* File server address /*
       fhandle_t  fh;              /* File handle to be mounted /*
       int    flags;               /* Flags handler /*
       int    wsize;               /* Write size in bytes /*
       int    rsize;               /* Read size in bytes /*
       int    timeo;               /* Initial timeout in .1 secs. /*
       int    retrans;             /* Times to retry send /*
       char *hostname;             /* Server's host name /*
       char *optstr;               /* Options string /*
```

```
    int   gfs_flags;          /* GFS flags /*
    int   pg_thresh;          /* Page threshold for exec /*
};
```

Return Value

The `mount` system call returns a value of 0 upon successful completion of a operation, –1 if the operation fails.

The `umount` system call announces to the system that the remote file system mounted on directory *name* is no longer available. The directory *name* reverts to its ordinary interpretation.

The `umount` system call returns 0 if the dismount operation succeeds, –1 if it fails.

Diagnostics

The `mount` call fails under the following conditions:

[EPERM] The caller is not the superuser and is not the owner of the mount point.

[ENODEV] A component of *special* does not exist.

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

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

[EBUSY] Another process currently holds a reference to *name*.

[ENAMETOOLONG] A component of the pathname exceeded 255 characters, or the entire length of the pathname exceeded 1023 characters.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENOENT] A component of *name* does not exist.

[EMFILE] No space remains in the mount table.

[EFAULT] The *special* or *name* points outside the process's allocated address space space.

[ESTALE] The *fhandle* given in the argument was invalid. The file referred to by that file handle no longer exists or has been revoked.

The `umount` call fails under the following conditions:

[EPERM] The caller is not the superuser and is not the owner of the mount point.

[EINVAL] The requested mounted-on directory is not in the mount table.

[EBUSY] A process is holding a reference to a file located on the file system.

[EIO] An I/O error occurred while writing cached file system information.

See Also

mount(2), gfsi(5), mount(8nfs)

mprotect(2)

Name

mprotect – memory protection control

Syntax

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

int mprotect (addr, len, prot)
caddr_t addr;
int len, prot;

Description

The `mprotect` system call changes the protection of portions of an application program's data memory. Protection is performed on page cluster boundaries. The default protection for data memory on process invocation is user READ/WRITE. The *addr* argument is the beginning address of the data block and must fall on a page cluster boundary.

The *len* argument is the length of the data block, in bytes. The length of the block is rounded up to a cluster boundary, and the size of the block to be protected is returned.

The *prot* argument is the requested protection for the block of memory. Protection values affect only the user process. Protection values are defined in <mman.h> as:

```
/* protections are chosen from these bits, ORed together */
#define PROT_READ       0x1     /* pages can be read */
#define PROT_WRITE      0x2     /* pages can be written */
#define PROT_EXEC       0x4     /* pages can be executed */
```

Setting the *prot* argument to zero (0) indicates that the process cannot reference the memory block, without causing a fault.

A protected page faults if the protection is violated, and a SIGBUS signal is issued. If the process has a handler defined for the SIGBUS signal, the *code* parameter, described in `sigvec`(2) and `signal`(3), is used to pass in the virtual address that faulted.

Restrictions

The page cluster size may change in future versions of ULTRIX. As a result, `getpagesize` should be used to determine the correct *len* argument, and `sbrk` or `malloc` should be used to determine the correct *addr* argument.

If the user handles a SIGBUS signal, the signal handler must either abort the process or correct the condition that caused the protection fault (SIGBUS). If some corrective action is not taken, an infinite loop results because the faulting instruction is restarted. If the user permits the default SIGBUS handler to be used, the process aborts if a referenced page causes a fault.

The VAX architecture makes the following implications; PROT_WRITE implies (PROT_WRITE | PROT_READ | PROT_EXEC), and PROT_READ implies (PROT_READ | PROT_EXEC).

Only the application can change the `mprotect` call's private data space. This means that attempts to change text, shared memory, or stack space causes a EACCES failure.

Return Value

Upon successful completion, the size of the protected memory block, in bytes, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Diagnostics

The `mprotect` call fails under the following conditions:

[EALIGN]	The *addr* argument is not on a cluster boundary.
[EINVAL]	The *prot* argument is not a valid protection mask.
[EACCES]	The memory block is not fully contained within private data space.

See Also

getpagesize(2), sbrk(2), sigvec(2), malloc(3), signal(3)

msgctl(2)

Name

msgctl – message control operations

Syntax

#include <*sys/types.h*>
#include <*sys/ipc.h*>
#include <*sys/msg.h*>

int msgctl (*msqid, cmd, buf*)
int *msqid, cmd*;
struct *msqid_ds, buf*;

Description

The `msgctl` system call provides message control operations as specified by *cmd*. The following *cmds* are available:

IPC_STAT — Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in `intro`(2).

IPC_SET — Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can be executed only by a process that has an effective user ID that is equal to superuser or the value of either *msg_perm.uid* or *msg_perm.cuid* in the data structure associated with *msqid*. Only the superuser can raise the value of *msg_qbytes*.

IPC_RMID — Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This command can only be executed by a process that has an effective user ID equal to either that of the superuser or to the value of *msg_perm.uid* in the data structure associated with *msquid*.

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.

Diagnostics

The `msgctl` system call fails under the following conditions:

[EINVAL] — The *msqid* is not a valid message queue identifier.

[EINVAL] — The *cmd* is not a valid command.

[EACCES] — The *cmd* is equal to IPC_STAT and read operation permission is

denied to the calling process. For further information, see `intro`(2).

[EPERM] The *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of the superuser or to the value of *msg_perm.uid* in the data structure associated with *msqid*.

[EPERM] The *cmd* is equal to IPC_SET, an attempt is being made to increase to the value of *msg_qbytes*, and the effective user ID of the calling process is not equal to that of superuser.

[EFAULT] The *buf* points to an illegal address.

See Also

msgget(2), msgop(2)

msgget(2)

Name

msgget – get message queue

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;

Description

The `msgget` system call returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure are created for *key* if one of the following is true:

- The *key* is equal to IPC_PRIVATE
- The *key* does not already have a message queue identifier associated with it, and *(msgflg* & IPC_CREAT) is true. For further information, see `intro`(2).

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- The *msg_perm.cuid, msg_perm.uid, msg_perm.cgid* and *msg_perm.gid* members are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order nine bits of *msg_perm.mode* are set equal to the low-order nine bits of *msgflg*.
- The *msg_qnum*, *msg_lspid*, *msg_lrpid*, *msg_stime*, and *mgs_rtime* members are set equal to 0.
- The *msg_ctime* is set equal to the current time.
- The *msg_qbytes* is set equal to the system limit.

Return Value

Upon successful completion, a non-negative integer, which is a message queue identifier, is returned. Otherwise, a value of –1 is returned, and *errno* is set to indicate the error.

Diagnostics

The `msgget` system call fails under the following conditions:

[EACCES] A message queue identifier exists for *key* but operations permission, as specified by the low-order nine bits of *msgflg*, would not be granted. For further information, see `intro`(2).

[ENOENT] A message queue identifier does not exist for *key* and the logicl operation *(msgflg* & IPC_CREAT) is false.

[ENOSPC]	A message queue identifier is to be created, but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.
[EEXIST]	A message queue identifier exists for *key* but the logical operation ((*msgflg* & IPC_CREAT) & (*msgflg* & IPC_EXCL)) is true.

See Also

msgctl(2), msgop(2), ftok(3)

msgop(2)

Name

msgsnd, msgrcv – message operations

Syntax

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

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
void *msgp;
size_t msgsz;
int msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
void *msgp;
size_t msgsz;
long msgtyp;
int msgflg;
```

Description

There are two message operations system calls, `msgsnd` and `msgrcv`.

msgsnd

The `msgsnd` system call is used to send a message to the queue associated with the message queue identifier specified by *msqid*. The *msgp* parameter points to a structure containing the message. This structure is composed of the following members:

```
long   mtype; /* message type */
char   mtext[];      /* message text */
```

The *mtype* parameter is a positive integer that can be used by the receiving process for message selection. For more information, see the `msgrcv` section of this reference page. The *mtext* parameter is any text of length *msgsz* bytes. The *msgsz* parameter can range from 0 to a system-imposed maximum.

The *msgflg* parameter specifies the action to be taken if the number of bytes already on the queue is equal to msg_qbytes. (For further information, see `intro`(2).) The parameter also specifies what happens when the total number of messages on all queues system-wide is equal to the system-imposed limit.

If either of these conditions is true, and if (*msgflg* & IPC_NOWAIT) is true, the message is not sent and the calling process returns immediately. However, if either of the conditions is true and (*msgflg* & IPC_NOWAIT) is false, the calling process suspends execution until one of the following occurs:

- The condition responsible for the suspension no longer exists, in which case the message is sent.
- The *msqid* parameter is removed from the system. For further information, see `msgctl`(2). When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

- The calling process receives a signal that is to be caught. In this case, the message is not sent and the calling process resumes execution in the manner prescribed in `signal`(3).

The `msgsnd` system call fails and no message is sent under the following conditions:

[EINVAL]	The *msqid* parameter is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process. For more information, see `errno`(2).
[EINVAL]	The *mtype* parameter is less than 1.
[EAGAIN]	The message cannot be sent for one of the reasons cited above and (*msgflg* & IPC_NOWAIT) is true.
[EINVAL]	The *msgsz* parameter is less than zero or greater than the system-imposed limit.
[EFAULT]	The *msgp* parameter points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (for more information, see `errno`(2)):

- The *msg_qnum* is incremented by 1.
- The *msg_lspid* is set equal to the process ID of the calling process.
- The *msg_stime* is set equal to the current time.

msgrcv

The `msgrcv` system call reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

The *mtype* parameter is the received message's type, as specified by the sending process. The *mtext* parameter is the text of the message. The *msgsz* parameter specifies the size, in bytes, of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

The *msgtyp* parameter specifies the type of message requested, as follows:

- If *msgtyp* is equal to 0, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

The *msgflg* parameter specifies the action to be taken if a message of the desired type is not on the queue. These specified actions are as follows:

- If (*msgflg* & IPC_NOWAIT) is true, the calling process returns immediately with a value of –1 and *errno* set to ENOMSG.
- If (*msgflg* & IPC_NOWAIT) is false, the calling process suspends execution until one of the following occurs:

- A message of the desired type is placed on the queue.
- The *msqid* parameter is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.
- The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes execution in the manner prescribed in `signal`(3).

The `msgrcv` system call fails and no message is received under the following conditions:

[EINVAL]	The *msqid* parameter is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process.
[EINVAL]	The *msgsz* parameter is less than 0.
[E2BIG]	The *mtext* parameter is greater than *msgsz* and (*msgflg* & MSG_NOERROR) is false.
[ENOMSG]	The queue does not contain a message of the desired type and (*msgtyp* & IPC_NOWAIT) is true.
[EFAULT]	The *msgp* parameter points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid:*

- The *msg_qnum* is decremented by 1.
- The *msg_lrpid* is set equal to the process ID of the calling process.
- The *msg_rtime* is set equal to the current time.

Return Values

If the `msgsnd` or `msgrcv` system calls return due to the receipt of a signal, a value of –1 is returned to the calling process, and errno is set to EINTR. If they return due to removal of *msqid* from the system, a value of –1 is returned, and errno is set to EIDRM.

Upon successful completion, the return value is as follows:

- The `msgsnd` system call returns a value of 0.
- The `msgrcv` system call returns a value equal to the number of bytes actually placed into *mtext*.

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

See Also

errno(2), intro(2), msgctl(2), msgget(2), signal(3)

Name

nfs_svc, nfs_biod – invoke NFS daemons

Syntax

nfs_svc(*sock*)
int *sock*;

nfs_biod()

Description

The `nfs_svc` system call starts an NFS daemon listening on the socket referenced by the file descriptor *sock*. The socket must be an AF_INET address format, and a SOCK_DGRAM socket type (protocol UDP/IP). This system call is used by `nfsd`.

If the process is killed, the system call returns the diagnostic EINTR.

The `nfs_biod` implements the NFS daemon that handles asynchronous I/O for an NFS client. This system call is used by `biod`. Unlike `nfs_svc`, this system call does not return any diagnostics if the process is killed.

Diagnostics

[EINTR] The NFS daemon, `nfs_svc`, process was killed.

See Also

socket(2), biod(8nfs), nfsd(8nfs)

open (2)

Name

open – open for reading or writing

Syntax

#include <sys/file.h>
#include <limits.h> /* definition of OPEN_MAX */

open(*path, flags, mode*)
char **path*;
int *flags, mode*;

Description

The `open` system call opens a specified file and returns a descriptor for that file. The file pointer used to mark the current position within the file is set to the beginning of the file.

The file descriptor remains open across `execve` system calls. The `close` system call closes the file descriptor.

A process cannot have more than OPEN_MAX file descriptors open simultaneously.

Arguments

path is the address of a string of ASCII characters representing a path name, terminated by a null character. The path name identifies the file to be opened.

mode is only used with the O_CREAT flag. The file is created with the specified mode, as described in `chmod`(2) and modified by the process's umask value. For further information, see `umask`(2).

flags defines how the file is to be opened. This argument is formed by ORing the following values:

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY Do not block on open when opening a port (named pipe) with O_RDONLY or O_WRONLY:

If O_NDELAY is set, an `open` for read only returns without delay. An `open` for write only returns an error if no process currently has the file open for reading.

If O_NDELAY is clear, an `open` for read only blocks until a process opens the file for writing. An `open` for write only blocks until a process opens the file for reading.

O_NONBLOCK POSIX definition of O_NDELAY. See O_NDELAY for explanation of functionality.

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.
O_BLKINUSE	Block if file is in use.
O_BLKANDSET	Block if file is in use; then, set in use.
O_FSYNC	Do file writes synchronously.
O_NOCTTY	In the POSIX environment, if this flag is set and path identifies a terminal device, the `open()` function will not cause the terminal device to become the controlling terminal for the process.

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 and the file already exists, the `open` returns an error. This can be used to implement a simple exclusive access locking mechanism.

If the O_NDELAY or O_NONBLOCK flag is specified and the open call would result in the process being blocked for some reason, the open returns immediately. For example, if the process were waiting for carrier on a dialup line, an open with the O_NDELAY or O_NONBLOCK flag would return immediately. The first time the process attempts to perform I/O on the open file, it blocks.

If the O_FSYNC flag is specified, each subsequent write (see `write`(2)) for the file is synchronous, instead of the default asynchronous writes. Use this flag to ensure that the write is complete when the system call returns. With asynchronous writes, the call returns when data is written to the buffer cache. There is no guarantee that the data was actually written out to the device. With synchronous writes, the call returns when the data is written from the buffer cache to the device.

O_BLKINUSE and O_BLKANDSET provide a test and set operation similar to a semaphore. O_BLKINUSE causes the open to block if another process has marked the file as in use. The `open` blocks in the system at a point where no references to the file are established.

There are two ways to mark a file as in use:

- Use the `ioctl`(2) system call with the *request* argument set to FIOSINUSE or TIOCSINUSE. For further information, see `tty`(4).
- Use the O_BLKANDSET flag to `open`(2)

O_BLKANDSET caused the `open` to block if another process has marked the file in use. When the `open` resumes, the file is marked in use by the current process.

If O_NDELAY is used with either O_BLKINUSE or O_BLKANDSET, the `open` failed if the file is in use. The external variable `errno` is set to EWOULDBLOCK in this case.

NOTE

The in use flag cannot be inherited by a child process, nor can it be replicated by the `dup` system call.

When the in use flag is cleared, all processes that are blocked for that reason resume. The `open` continues to block if another process marks the file as in use again.

The in use flag can be cleared in three ways:

- When the file descriptor marked as in use is closed
- When the process that set the in use flag exits
- When an `ioctl` system call is issued and FIOCINUSE or TIOCCINUSE is specified in the *request* argument.

Environment

System Five

When your program is compiled using the System V environment, and O_NDELAY is specified, subsequent reads and writes are also affected.

Return Values

Upon successful completion, an integer value greater than -1 is returned.

Diagnostics

The `open` call fails under the following conditions:

[EACCES]	The required permissions for reading, writing, or both are denied for the named flag.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	O_CREAT is specified, the file does not exist, and the directory in which it is to be created does not permit writing.
[EDQUOT]	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 the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	O_CREAT is specified, the file does not exist, and the user's quota of inodes on the file system on which the file is being created has been exhausted.
[EEXIST]	O_CREAT and O_EXCL were specified and the file exists.
[EFAULT]	The *path* points outside the process's allocated address space.
[ENFILE]	The system file table is full.
[EINVAL]	An attempt was made to open a file with the O_RDONLY and O_FSYNC flags set.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode for O_CREAT.
[EISDIR]	The named file is a directory, and the arguments specify it is to be

opened for writing.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EMFILE] {OPEN_MAX} file descriptors are currently open.

[ENAMETOOLONG]
A component of a pathname exceeds 255 characters or an entire pathname exceeds 1023 characters.

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

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

[ENOENT] The *path* argument points to an empty string and the process is running in the POSIX or SYSTEM_FIVE environment.

[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.

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

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

[ENXIO] The O_NDELAY flag is given, and the file is a communications device on which there is no carrier present.

[ENXIO] O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set and no process has the file open for reading.

[EOPNOTSUPP] An attempt was made to open a socket that is not set active.

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

[ESTALE] The file handle given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.

[ETIMEDOUT] A connect request or remote file operation failed because the connected party did not respond after a period of time determined by the communications protocol.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and the `open` call requests write access.

[EWOULDBLOCK]
The open would have blocked if the O_NDELAY was not used. The probable cause for the block is that the file was marked in use.

[EINTR] A signal was caught during the `open()` function.

See Also

chmod(2), close(2), dup(2), fcntl(2), lseek(2), read(2), write(2), umask(2), tty(4)

Name

pipe – create an interprocess channel

Syntax

include <limits.h> /*Definition of PIPE_MAX*/
pipe(*fildes*)
int *fildes[2]*;

Arguments

fildes Passing an address as an array of two integers into the `pipe` system call.

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. Their integer values will be the two lowest available at the time of the `pipe` function call. The O_NONBLOCK and FD_CLOEXEC flags will be clear on both file descriptors.

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] picks up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent `fork` calls) 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.

For further information on how `read` and `write` calls behave with pipes, see the `read`(2) and `write`(2) reference pages.

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

Restrictions

Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock may occur.

The underlying implementation of pipes is no longer socket based, but rather implemented through the file system. Any application that needs socket functionality from pipes should use the `socketpair` system call.

Return Values

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

Diagnostics

The `pipe` call fails if:

[EMFILE] Too many descriptors are active.

[ENFILE] The system file table is full.

[EFAULT] The *fildes* buffer is in an invalid area of the process's address space.

Environment

Differs from the System V definition in that ENFILE is not a possible error condition.

See Also

sh(1), fork(2), read(2), socketpair(2), write(2)

Name

plock – lock or unlock process, text, or data in memory

Syntax

#include <sys/lock.h>

int plock (op)
int op;

Description

The `plock` call allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to page outs, and the process is immune to swap outs. The `plock` call also unlocks these segments.

The *op* argument specifies the following:

PROCLOCK Lock text and data segments into memory (process lock)

TXTLOCK Lock text segment into memory (text lock)

DATLOCK Lock data segment into memory (data lock)

UNLOCK Remove locks

Return Value

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

Diagnostics

The `plock` call fails under the following conditions:

[EPERM] The effective user ID of the calling process is not superuser.

[EINVAL] The *op* argument is equal to PROCLOCK, and a process lock, a text lock, or a data lock already exists on the calling process.

[EINVAL] The *op* argument is equal to TXTLOCK, and a text lock or a process lock already exists on the calling process.

[EINVAL] The *op* argument is equal to DATLOCK, and a data lock or a process lock already exists on the calling process.

[EINVAL] The *op* argument is equal to UNLOCK, and no type of lock exists on the calling process.

Restrictions

The effective user ID of the calling process must be superuser to use this call.

Both PROCLOCK and TXTLOCK lock the text segment of a process, and a locked text segment is locked for all sharing processes.

Because the effective user ID of the calling process is superuser, take care not to lock more virtual pages than can be contained in physical memory. A deadlock can result.

See Also

execve(2), exit(2), fork(2), shmctl(2)

Name

profil – execution time profile

Syntax

profil(buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

Description

The *buff* points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (on RISC machines, 4 milliseconds; on VAX machines, 10 milliseconds); *offset* is subtracted from the pc, and the result is multiplied by *scale*. If the resulting number corresponds to a word inside *buff,* that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0x10000 gives a 1-1 mapping of program counter's to words in *buff;* 0x8000 maps each pair of instruction words together. 0x2 maps all instructions onto the beginning of *buff,* producing a non-interrupting core clock.

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *execve* is executed, but remains on in both child and parent after a *fork*. Profiling is turned off if an update in *buff* would cause a memory fault.

Return Value

A 0, indicating success, is always returned.

See Also

gprof(1), setitimer(2), monitor(3)

ptrace (2)

Name

ptrace – process trace

Syntax

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

ptrace(request, pid, addr, data)
int request, pid, *addr, data;

Description

The system call `ptrace` provides a means by which a process can control the execution of another 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. A process being traced behaves normally until it encounters some signal, whether internally generated, like "illegal instruction," or externally generated, like "interrupt." For more information, see `sigvec`(2).

Upon encountering a signal, the traced process enters a stopped state and its tracing process is notified by means of `wait`. If the traced process stops with a SIGTRAP, the process might have been stopped for a number of reasons. Two status words addressable as registers in the traced process's uarea qualify SIGTRAPs: TRAPCAUSE, which contains the cause of the trap, and TRAPINFO, which contains extra information concerning the trap.

When the traced process is in the stopped state, its core image can be examined and modified using `ptrace`. If desired, another `ptrace` request can then cause the traced process either to terminate or to continue, possibly ignoring the signal.

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

0 This request is the only one that can be used by a child process. The child process can declare that it is to be traced by its parent. All other arguments are ignored. Unexpected results occur if the parent process does not expect to trace the child process.

1,2 The word in the traced process's address space at *addr* is returned. If I and D space are separated (for example, historically on a PDP-11), request 1 indicates I space, request 2 indicates D space. The *addr* must be 4-byte aligned. The traced process must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area corresponding to *addr* is returned. The *addr* is a constant defined in ptrace.h. This space contains the registers and other information about the process; the constants correspond to fields in the *user* structure in the system.

4,5 The given *data* is written at the word in the process's address space corresponding to *addr*, which must be 4-byte aligned. The old value at the address is returned. If I and D space are separated, request 4 indicates I space, request 5 indicates D space. Attempts to write in pure procedure fail if another process is executing the same file.

6 The process's system data is written, as it is read with request 3. 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. The old value at the address is returned.

7 The *data* argument is taken as a signal number and the traced process's execution continues at location *addr* as if it had incurred that signal. Normally, the signal number is 0 to indicate that the signal causing the stop should be ignored. The signal number might be the value fetched out of the process's image, which identifies the signal that caused the stop. If *addr* is (int *)1, execution continues from where it stopped.

8 The traced process terminates.

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. TRAPCAUSE contains CAUSESINGLE. This is part of the mechanism for implementing breakpoints.

20 This is the same as zero, except it is executed by the tracing process and the pid field is nonzero. The process with that pid stops and becomes a traced process. On a signal, the traced process returns control to the tracing process, rather than the parent. The tracing process must have the same uid as the traced process.

21,22
Returns MAXREG general or MAXFREG floating registers, respectively. Their values are copied to the locations starting at the address in the tracing process specified by the *addr* argument.

24,25
Same as 20 and 21, but writes the registers instead of reading them.

26 Specifies a watchpoint in the data or stack segment of the traced process. If any byte address starting at the *addr* argument and continuing for the number of bytes specified by the *data* argument is accessed in an instruction, the traced process stops execution with a SIGTRAP. TRAPCAUSE contains CAUSEWATCH, and TRAPINFO contains the address causing the trap. This ptrace returns a watchpoint identifier (wid). MAXWIDS specifies the maximum number of watchpoints for each process.

27 The data argument specifies a wid to delete.

28 Turns off the tracing for the traced process with the specified pid.

29 Returns an open file descriptor for the file attached to *pid*. This request is useful in accessing the symbol table of a process created with the `execve` call.

As indicated, these calls (except for request 0 and 20) 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. This value indicates that the process has stopped, rather than terminated. If multiple processes are being traced, `wait` can be called multiple times, and it returns the status for the next stopped or terminated child or traced process.

To forestall possible fraud, `ptrace` inhibits the set-user-id and set-group-id facilities on subsequent `exec`(2) calls. If a traced process calls `execve`, it stops before executing the first instruction of the new image showing signal SIGTRAP. In this case, TRAPCAUSE contains CAUSEEXEC and TRAPINFO does not contain anything interesting. If a traced process calls `execve` again, the same thing occurs.

If a traced process forks, both parent and child are traced. Breakpoints from the parent are not copied into the child. At the time of the fork, the child is stopped with a SIGTRAP. The tracing process can then terminate the trace if desired. TRAPCAUSE contains CAUSEFORK and TRAPINFO contains the pid of its parent.

Restrictions

On an ULTRIX system, the `ptrace` system call succeeds only if the user owns the binary being traced or if the user is root.

The request 0 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, can be efficiently debugged.

The error indication, –1, is a legitimate function value; when an error occurs, the errno variable is set to explain the condition that caused the error.

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

Return Value

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

Diagnostics

The `ptrace` call fails under the following conditions:

[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

dbx(1), wait(2), sigvec(2)

Name

ptrace – process trace

Syntax

#include <signal.h>

ptrace(request, pid, addr, data)
int request, pid, *addr, data;

Description

The `ptrace` system call 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 `sigvec`(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:

0 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.

1,2 The word in the child process's address space at *addr* is returned. If I and D space are separated (for example, historically on a pdp-11), request 1 indicates I space, 2 D space. The *addr* must be even. The child must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area corresponding to *addr* is returned. The *addr* must be even 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.

4,5 The given *data* is written at the word in the process's address space corresponding to *addr,* which must be even. No useful value is returned. If I and D space are separated, request 4 indicates I space, 5 D space. Attempts to write in pure procedure fail if another process is executing the same file.

6 The process's system data is written, as it is read with request 3. 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.

7 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.

8 The traced process terminates.

9 Execution continues as in request 7; 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 0) 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, "word" also means a 32-bit integer, but the "even" restriction does not apply.

Environment

When your program is compiled using the System V environment, requests 7 and 9 return the value of the data argument on success, errno is set to ESRCH if the pid does not exist, **EIO** if the address is out of bounds.

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.

Restrictions

In ULTRIX, the `ptrace` system call will only succeed if the user owns the binary being traced or if the user is root.

Diagnostics

[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

adb(1), sigvec(2), wait(2)

Name

quota – manipulate disk quotas

Syntax

#include <sys/types.h>
#include <sys/param.h>
#include <sys/quota.h>

quota(cmd, uid, arg, addr)
int cmd, uid, arg;
caddr_t addr;

Description

The `quota` call manipulates disk quotas for file systems that have had quotas enabled with `setquota`. The *cmd* parameter indicates a command in the following list that is applied to the user ID *uid*. The *arg* parameter is a command specific argument and *addr* is the address of an optional, command specific data structure, which is copied in or out of the system. The interpretation of *arg* and *addr* is given with each command in the list that follows:

Q_SETDLIM
: Set disk quota limits and current usage for the user with ID *uid*. The *arg* parameter is a major-minor device indicating a particular file system. The *addr* parameter is a pointer to a struct dqblk structure, defined in <sys/quota.h>. Only the superuser can issue this call.

Q_GETDLIM
: Get disk quota limits and current use for the user with ID *uid*. The remaining parameters are identical to the Q_SETDLIM command parameters.

Q_SETDUSE
: Set disk use limits for the user with ID *uid*. The *arg* parameter is a major-minor device indicating a particular file system. The *addr* is a pointer to a struct dqusage structure, defined in <sys/quota.h>. Only the superuser can issue this call.

Q_SYNC Update the on-disk copy of quota uses. The *uid*, *arg*, and *addr* parameters are ignored.

Q_SETUID
: Change the calling process's quota limits to those of the user with ID *uid*. The *arg* and *addr* parameters are ignored. Only the superuser can issue this call.

Q_SETWARN
: Alter the disk usage warning limits for the user with ID *uid*. The *arg* is a major-minor device indicating a particular file system. The *addr* parameter is a pointer to a struct dqwarn structure, which is defined in <sys/quota.h>. Only the superuse can issue this call.

Q_DOWARN
: Warn the user with user ID *uid* about excessive disk use. This call causes the system to check its current disk use information and print a message

on the terminal of the caller for each file system on which the user is over quota. If the *arg* parameter is specified as NODEV, all file systems that have disk quotas are checked. Otherwise, *arg* indicates a specific major-minor device to be checked. Only the superuser can issue this call.

Return Value

A successful call returns 0 and, possibly, more information specific to the command specified in the *cmd* parameter; when an error occurs, the value –1 is returned and the global variable *errno* is set to indicate the reason.

Diagnostics

A *quota* call fails when one of the following occurs:

[EINVAL]	The kernel has not been compiled with the QUOTA option.
[EINVAL]	The *cmd* parameter is invalid.
[ESRCH]	No disk quota is found for the indicated user.
[EPERM]	Only the superuser can issue the call and the caller is not the superuser.
[ENODEV]	The *arg* parameter is being interpreted as a major-minor device, and it indicates an unmounted file system.
[EFAULT]	An invalid *addr* parameter is supplied; the associated structure could not be copied in or out of the kernel.
[EUSERS]	The quota table is full.

See Also

`setquota`(2,) `quotacheck`(8,) `quotaon`(8)
"Disk Quotas in a UNIX Environment", *ULTRIX Supplementary Documents, Volume 3:System Manager*

Name

read, readv – read from a file

Syntax

cc = read(*d, buf, nbytes*)
int cc, *d*;
char **buf*;
int *nbytes*;

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

cc = readv(*d, iov, iovcnt*)
int cc, *d*;
struct iovec **iov*;
int *iovcnt*;

Arguments

d	File descriptor.
buf	Character pointer where information is stored.
nbytes	Integer that tells you how many bytes to read.
iov	Pointer to an *iovec* structure.
iovcnt	The number of *iovec* structures to be processed.

Description

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

For `readv`, the *iovec* structure is defined as follows:

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

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The `readv` system call fills an area completely before proceeding to the next area.

On objects that are capable of seeking, the `read` starts at a position given by the pointer associated with *d*. See `lseek`(2) for more information. 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.

When attempting to read from an empty pipe (or FIFO):

- If no process has the pipe open for writing, `read` returns zero to indicate end-of-file.

- If some process has the pipe open for writing and O_NDELAY or O_NONBLOCK is set, `read` returns a –1, *errno* is to [EWOULDBLOCK]. If some process has the pipe open for writing and O_NDELAY and O_NONBLOCK are clear, `read` blocks until data is written or the pipe is closed by all processes that opened the pipe for writing.

Upon successful completion, `read` and `readv` return the number of bytes actually read and placed in the buffer. The system reads the number of bytes requested if the descriptor references a file which has that many bytes left before the end-of-file; this is not true in any other instance.

Unless the SV_INTERRUPT bit has been set for a signal, the `read` system calls are automatically restarted when a process receives a signal while waiting for input. See also `sigvec`(2).

Return Value

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

If the read is 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.

Diagnostics

The `read` and `readv` system calls fail if one or more of the following are true:

[EBADF] The *d* argument is not a valid file or socket descriptor open for reading.

[EFAULT] The *buf* points outside the allocated address space.

[EINTR] A read from a slow device was interrupted before any data arrived by the delivery of a signal.

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

[ESTALE] The file handle given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.

[EWOULDBLOCK] The O_DELAY or O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the read operation.

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

[EINVAL] The *iovcnt* was less than or equal to 0, or greater than 16.

[EINVAL] One of the *iov_len* values in the *iov* array was negative.

[EINVAL] The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

[EFAULT] Part of the *iov* points outside the process's allocated address space.

[ETIMEDOUT] A connect request or remote file operation failed because the connected party did not respond after a period of time determined by the communications protocol.

Environment

SYSTEM_FIVE

When you use the System V environment, note the following:

- If your program is compiled in this environment, a `read` and `readv` system call returns 0 if the file has been set up for non-blocking I/O and the read would block.
- In this environment, the parameter *nbytes* is of type *int* instead of type *unsigned.*

POSIX

In the POSIX environment, [EAGAIN] is returned in *errno* instead of [EWOULDBLOCK].

See Also

dup(2), open(2), pipe(2), sigvec(2), socket(2), socketpair(2)

readlink(2)

Name

readlink – read value of a symbolic link

Syntax

cc = readlink(path, buf, bufsiz)
int cc;
char *path, *buf;
int bufsiz;

Description

The `readlink` system call places the contents of the symbolic link *path* in the buffer *buf*, which has size *bufsiz*. The contents of the link are not null terminated when returned.

Return Value

The call returns the count of characters placed in the buffer if it succeeds, or a –1 if an error occurs, placing the error code in the global variable *errno*.

Diagnostics

The `readlink` system call fails under the following conditions:

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

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

[ENOENT] The named file does not exist.

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

[EINVAL] The named file is not a symbolic link.

[EFAULT] The *buf* extends outside the process's allocated address space.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

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

[ETIMEDOUT] A connect request or remote file operation failed, because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

lstat(2), symlink(2), stat(2)

Name

reboot – reboot system or halt processor

Syntax

#include <sys/reboot.h>

reboot(*howto*)
int *howto*;

Arguments

howto The *howto* argument is a mask of options passed to the bootstrap program.

The bits of *howto* are:

RB_HALT
: the processor is simply halted; no reboot takes place. RB_HALT should be used with caution.

RB_ASKNAME
: Interpreted by the bootstrap program itself, causing it to inquire as to what file should be booted. Normally, the system is booted from the file "xx(0,0)vmunix" without asking.

RB_SINGLE
: Normally, the reboot procedure involves an automatic disk consistency check and then multi-user operations. RB_SINGLE prevents the consistency check, rather simply booting the system with a single-user shell on the console. RB_SINGLE is interpreted by the init(8) program in the newly booted system. This switch is not available from the system call interface.

Only the superuser may `reboot` a machine.

Description

The `reboot` system call reboots the system, and is invoked automatically in the event of unrecoverable system failures. The system call interface permits only RB_HALT or RB_AUTOBOOT to be passed to the reboot program; the other flags are used in scripts stored on the console storage media or used in manual bootstrap procedures. When none of these options (for example, RB_AUTOBOOT) is given, the system is rebooted from file `vmunix` in the root file system of unit 0 of a disk chosen in a processor-specific way. Normally, an automatic consistency check of the disks is then performed.

Return Value

If successful, this call never returns. Otherwise, a –1 is returned, and an error is stored in the global variable *errno*.

Diagnostics

The `reboot` call fails under the following condition:

[EPERM] The caller is not the superuser.

See Also

crash(8v), halt(8), init(8), reboot(8)

Name

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

Syntax

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

cc = recv(s, buf, len, flags)
int cc, s;
char *buf;
int len, flags;

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

cc = recvmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

Description

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

The `recv` call can be used only on a connected socket. The `recvfrom` and `recvmsg` calls can be used to receive data on a socket, whether or not it is in a connected state. For further information, see `connect`(2).

If *from* is nonzero, the source address of the message is filled in. The *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in `cc`. If a message is too long to fit in the supplied buffer, excess bytes can be discarded, depending on the type of socket the message is received from. For further information, see `socket`(2).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking. If the socket is nonblocking, a `cc` of –1 is returned, and the external variable errno is set to EWOULDBLOCK. For further information, see `ioctl`(2).

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

The *flags* argument to a send call is formed by ORing one or more of the values following values:

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

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

```
struct msghdr {
        caddr_t  msg_name;          /* optional address */
        int      msg_namelen;       /* size of address */
        struct   iov *msg_iov;      /* scatter/gather array */
        int      msg_iovlen;        /* # elements in msg_iov */
        caddr_t  msg_accrights;     /* access rights sent/received */
        int      msg_accrightslen;
};
```

Here, *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* can be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter gather locations, as described in `read`(2). Access rights to be sent along with the message are specified in *msg_accrights* , which has length *msg_accrightslen* .

Return Value

These calls return the number of bytes received, or –1 if an error occurred.

Diagnostics

The `recv` call fails under the following conditions:

[EBADF] The argument *s* is an invalid descriptor.

[EINVAL] The argument length of the message is less than 0.

[EMSGSIZE] The message sent on the socket was larger than the internal message buffer.

[ENOTCONN] A call was made to `recv` from an unconnected stream socket.

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

[EWOULDBLOCK] The socket is marked nonblocking and the receive operation would block.

[EINTR] The receive was interrupted by delivery of a signal before any data was available for the receive.

[EFAULT] The data was specified to be received into a nonexistent or protected part of the process address space. The argument *fromlen* points outside the process address space.

See Also

read(2), send(2), socket(2)

Name

rename – change the name of a file

Syntax

rename(*from, to*)
char **from, *to*;

Description

The `rename` system call causes the link named *from* to be renamed *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 nondirectories) and must reside on the same file system.

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

Return Value

A zero (0) value is returned if the operation succeeds. Otherwise `rename` returns –1, and the global variable *errno* indicates the reason for the failure.

Restrictions

The system can deadlock if a loop in the file system graph is present.and two processes issue the `rename` call at the same time. For example, suppose a directory, `dirname`, contains a file, `dirname/filename`. Suppose that file is hard-linked to a directory, `secondir`, and the `secondir` directory contains a file, `secondir/secondfile`. If `secondir/secondfile` is hard-linked to `dirname`, a loop exists. Now suppose one process issues the following `rename` call:

```
rename (dirname/filename secondir/secondfile)
```

At the same time, another process issues the following `rename` call:

```
rename (secondir/secondfile dirname/filename)
```

In this case, the system can deadlock. The system administrator should replace hard links to directories with symbolic links.

Diagnostics

The `rename` system call fails and neither of the argument files are affected under the following conditions:

[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of the *from* path does not exist, or a path prefix of *to* does not exist.
[ENOENT]	Either *from* or *to* points to an empty string and the environment defined is POSIX or SYSTEM_FIVE.
[EACCES]	A component of either path prefix denies search permission.
[EPERM]	The *to* file exists, the directory containing *from* is marked sticky, and neither the containing directory nor the *to* directory is owned by the effective user ID.

[EPERM] The directory containing *from* is marked sticky, and neither the containing directory nor the *from* directory is owned by the effective user ID.

[EXDEV] The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code is not returned if the implementation permits cross-device links.

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

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

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

[EINVAL] The *from* is a parent directory of *to*, or an attempt is made to rename dot (.) or dot-dot (..).

[ENAMETOOLONG] A component of either pathname exceeded 255 characters, or the entire length of either pathname exceeded 1023 characters.

[ELOOP] Too many symbolic links were encountered in translating either pathname.

[ENOTDIR] The *from* is a directory, but *to* is not a directory.

[EISDIR] The *to* is a directory, but *from* is not a directory.

[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.

[EDQUOT] The directory in which the entry for the new name is being placed cannot be extended, because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EIO] An I/O error occurred while making or updating a directory entry.

[ENOTEMPTY] The *to* is a directory and is not empty.

[EBUSY] The directory named by *from* or *to* is a mount point.

See Also

open(2)

Name

rmdir – remove a directory file

Syntax

rmdir(path)
char *path;

Description

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

If one or more processes have the directory open when the last link is removed, the dot and dot-dot entries, if present, are removed before `rmdir()` returns and no new entries may be created in the directory. The directory, however, is not removed until all references to the directory have been closed.

Return Value

A zero (0) is returned if the remove succeeds; otherwise, a –1 is returned, and an error code is stored in the global location *errno*.

Diagnostics

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

[ENOTEMPTY] The named directory contains files other than dot and dot-dot.

[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.

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

[ENOENT] The named directory does not exist or *path* points to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

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

[EACCES] Write permission is denied on the directory containing the link to be removed.

[EBUSY] The directory to be removed is the mount point for a mounted file system.

[EROFS] The directory entry to be removed resides on a read-only file system.

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

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

[ETIMEDOUT] A connect request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

mkdir(2), unlink(2)

Name

select – synchronous I/O multiplexing

Syntax

#include <sys/time.h>

nfound = select(nfds, readfds, writefds, execptfds, timeout)
int nfound, nfds, *readfds, *writefds, *execptfds;
struct timeval *timeout;

Description

The `select` system call examines the I/O descriptors specified by the bit masks *readfds*, *writefds*, and *execptfds* to see if they are ready for reading, writing, or have an exceptional condition pending. The I/O descriptors can be pointers to arrays of integers, if multiple fd's are required to be selected. File descriptor *f* is represented by the bit 1<<f in the mask. The *nfds* descriptors are checked, that is, the bits from 0 through *nfds*-1 in the masks are examined. The `select` system call returns, in place, a mask of those descriptors that are ready. The total number of ready descriptors is returned in *nfound*.

If *timeout* is a nonzero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the `select` blocks indefinitely. To affect a poll, the *timeout* argument should be nonzero, pointing to a 0-valued `timeval` structure.

Any of *readfds*, *writefds*, and *execptfds* can be given as zero (0) if no descriptors are of interest.

Return Value

The `select` system call returns the number of descriptors that are contained in the bit masks, or –1 if an error occurred. If the time limit expires, `select` returns 0.

Restrictions

If a process is blocked on a `select` waiting for input from a socket and the sending process closes the socket, the `select` notes this as an exception rather than as data. Hence, if the `select` is not currently looking for exceptions, it waits indefinitely.

The descriptor masks are always modified on return, even if the call returns as the result of the timeout.

Diagnostics

An error return from `select` indicates:

[EBADF]	One of the bit masks specified an invalid descriptor.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is unacceptable. One of its components is negative or too large.

See Also

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

Name

semctl – semaphore control operations

Syntax

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

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
        int val;
        struct semid_ds *buf;
        ushort array[];
} arg;
```

Description

The `semctl` system call provides a variety of semaphore control operations as specified by *cmd*. The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum*:

GETVAL — Return the value of *semval*. For further information, see `intro`(2).

SETVAL — Set the value of *semval* to arg.val. When this command is successfully executed, the *semadj* value corresponding to the specified semaphore in all processes is cleared.

GETPID — Return the value of *sempid*.

GETNCNT — Return the value of *semncnt*.

GETZCNT — Return the value of *semzcnt*.

The following *cmd*s return and set every *semval* in the set of semaphores:

GETALL — Place *semval*s into array pointed to by *arg.array*.

SETALL — Set *semval*s according to the array pointed to by *arg.array* When this command is successfully executed, the *semadj* values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

IPC_STAT — Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in `intro`(2).

IPC_SET — Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */
```

This command can only be executed by a process that has an effective user ID equal to superuser or to the values of **sem_perm.uid** or **sem_perm.cuid** in the data structure associated with *semid*.

IPC_RMID — Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **sem_perm.uid** in the data structure associated with *semid*.

Return Value

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

GETVAL — The value of *semval*.

GETPID — The value of *sempid*.

GETNCNT — The value of *semncnt*.

GETZCNT — The value of *semzcnt*.

All other — A value of 0.

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

Diagnostics

The `semctl` system call fails if any of the following is true:

[EINVAL] — The *semid* is not a valid semaphore identifier.

[EINVAL] — The *semnum* is less than zero or greater than **sem_nsems.**

[EINVAL] — The *cmd* is not a valid command.

[EACCES] — Operation permission is denied to the calling process. For further information, see `errno`(2).

[ERANGE] — The *cmd* is SETVAL or SETALL, and the value to which semval is to be set is greater than the system imposed maximum.

[EPERM] — The *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of superuser nor to the value of **sem_perm.uid** in the data structure associated with *semid*.

[EFAULT] — The *arg.buf* points to an illegal address.

See Also

errno(2), intro(2), semget(2), semop(2)

Name

semget – get set of semaphores

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (*key, nsems, semflg*)
key_t *key*;
int *nsems, semflg*;

Description

The `semget` system call returns the semaphore identifier associated with *key*. A semaphore identifier and associated data structure and set containing *nsems* semaphores are created for *key*, if one of the following is true:

- The *key* is equal to IPC_PRIVATE
- The *key* does not already have a semaphore identifier associated with it, and (*semflg* & IPC_CREAT) is true.

For further information, see `intro`(2).

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

- The **sem_perm.cuid**, **sem_perm.uid**, **sem_perm.cgid** and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order nine bits of **sem_perm.mode** are set equal to the low-order nine bits of *semflg*.
- The **sem_nsems** is set equal to the value of *nsems*.
- The **sem_otime** is set equal to zero (0) and **sem_ctime** is set equal to the current time.

Return Values

Upon successful completion, a nonnegative integer, namely a semaphore identifier, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Diagnostics

The *semget* system call fails if any of the following is true:

[EINVAL] The *nsems* is either less than or equal to zero or greater than the system-imposed limit

[EACCES] A semaphore identifier exists for *key*, but operation permission, as specified by the low-order nine bits of *semflg* would not be granted. For further information, see `errno`(2).

[EINVAL] A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to zero.

[ENOENT]	A semaphore identifier does not exist for *key* and (*semflg* & IPC_CREAT) is false.
[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers in the system would be exceeded.
[EEXIST]	A semaphore identifier exists for *key* but ((*semflg* & IPC_CREAT) & (*semflg* & IPC_EXCL)) is true.

See Also

semctl(2), semop(2), ftok(3)

Name

semop – semaphore operations

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (*semid, sops, nsops*)
int *semid*;
struct sembuf **sops[]*;
int *nsops*;

Description

The `semop` system call is used to atomically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by `semid`. The *sops* is a pointer to the array of semaphore-operation structures. The *nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
short  sem_num;      /* semaphore number */
short  sem_op;       /* semaphore operation */
short  sem_flg;      /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by `semid` and *sem_num*.

The *sem_op* specifies one of three semaphore operations as follows:

1. If *sem_op* is a negative integer, one of the following occurs:

- If *semval* is greater than or equal to the absolute value of *sem_op,* the absolute value of *sem_op* is subtracted from *semval.* For further information, see `intro`(2). Also, if (*sem_flg* & SEM_UNDO) is true, the absolute value of *sem_op* is added to the calling process's *semadj* value for the specified semaphore. For further information, see `exit`(2).
- If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is true, `semop` returns immediately.
- If *semval* is less than the absolute value of *sem_op* and (sem_flg & IPC_NOWAIT) is false, `semop` increments the *semncnt* associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:
- If the *semval* becomes greater than or equal to the absolute value of *sem_op.* When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval,* and if (*sem_flg* & SEM_UNDO) is true, the absolute value of *sem_op* is added to the calling process's *semadj* value for the specified semaphore.

  The `semid` for which the calling process is awaiting action is removed from the system. For further information, see `semctl`(2). When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in `signal`(3).

2. If *sem_op* is a positive integer, the value of *sem_op* is added to *semval* and, if (*sem_flg* & SEM_UNDO) is true, the value of *sem_op* is subtracted from the calling process's *semadj* value for the specified semaphore.

3. If *sem_op* is zero, one of the following occurs:

- If *semval* is zero, `semop` returns immediately.
- If *semval* is not equal to zero and (*sem_flg* & IPC_NOWAIT) is true, `semop` returns immediately.
- If *semval* is not equal to zero and (*sem_flg* & IPC_NOWAIT) is false, `semop` increments the *semzcnt* associated with the specified semaphore and suspend execution of the calling process, until one of the following occurs:

  The *semval* became zero, at which time the value of *semzcnt* associated with the specified semaphore is decremented.

  The `semid` for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of *semzcnt* associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in `signal`(3).

Upon successful completion, the value of *sempid* for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

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.

Diagnostics

The `semop` fails if any of the following is true for any of the semaphore operations specified by *sops:*

[EINVAL]	The *sempid* is not a valid semaphore identifier.
[EFBIG]	The *sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with `semid`.
[E2BIG]	The *nsops* is greater than the system-imposed maximum.
[EACCESS]	Operation permission is denied to the calling process. For further information, see `errno`(2).
[EAGAIN]	The operation would result in suspension of the calling process, but (*sem_flg* & IPC_NOWAIT) is true.
[ENOSPC]	The limit on the number of individual processes requesting an SEM_UNDO would be exceeded.
[EINVAL]	The number of individual semaphores for which the calling process request a SEM_UNDO would exceed the limit.

[ERANGE]	An operation would cause a *semval* to overflow the system-imposed limit.
[ERANGE]	An operation would cause a *semadj* value to overflow the system-imposed limit.
[EFAULT]	The *sops* points to an illegal address.
[EINTR]	The `semop` returns due to the receipt of a signal.
[EIDRM]	The `semid` has been removed from the system.

See Also

execve(2), exit(2), fork(2), semctl(2), semget(2), signal(3)

send(2)

Name

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

Syntax

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

cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;

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

cc = sendmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

Description

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

The address of the target is given by *to*, with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted. If the address specified in the argument is a broadcast address, the SO_BROADCAST option must be set for broadcasting to succeed.

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

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

The *flags* parameter can be set to MSG_OOB to send out-of-band data on sockets that support this features (for example, SOCK_STREAM).

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

The call returns the number of characters sent, or –1 if an error occurred.

Diagnostics

[EBADF]	An invalid descriptor was specified.
[EDESTADDRREQ]	A required address was omitted from an operation on a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EINVAL]	An invalid argument length for the message was specified.
[EINTR]	The send was interrupted by delivery of a signal.
[ENOTCONN]	The socket is not connected.
[ENOTSOCK]	The argument *s* is not a socket.
[EMSGSIZE]	The socket requires that messages be sent atomically, and the size of the message to be sent made this impossible.
[EPIPE]	A write on a pipe or socket for which there is no process to read the data.
[EWOULDBLOCK]	The socket is marked nonblocking, and the requested operation would block.

See Also

recv(2), getsockopt(2), socket(2)

setgroups (2)

Name

setgroups – set group access list

Syntax

#include <sys/param.h>

setgroups(ngroups, gidset)
int ngroups, *gidset;

Description

The `setgroups` system call sets the group access list of the current user process according to the array, *gidset*. The *ngroups* parameter indicates the number of entries in the array and must be no more than NGROUPS, as defined in `<sys/param.h>`.

Only the superuser can set new groups.

Return Value

A 0 value is returned on success, –1 on an error, with the error code stored in *errno*.

Diagnostics

The `setgroups` call fails if:

[EPERM]	The caller is not the superuser.
[EFAULT]	The address specified for *gidset* is outside the process address space.

See Also

getgroups(2), initgroups(3x)

Name

setpgrp – set process group

Syntax

setpgrp(pid, pgrp)
int pid, pgrp;

Description

The `setpgrp` system call sets the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, the call applies to the current process.

If the invoker is not the superuser, the affected process must either have the same effective user-id as the invoker or be a descendant of the invoking process.

Return Value

The `setpgrp` system call returns zero (0) when the operation is successful. If the request fails, –1 is returned, and the global variable *errno* indicates the reason.

Environment

SYSTEM_FIVE

When your program is compiled using the System V environment, `setpgrp` is called without arguments and the new process group id is returned if successful.

Diagnostics

The `setpgrp` system call fails and the process group is not altered, if one of the following occur:

[EPERM]	The effective user ID of the requested process is different from that of the caller and the process is not a descendent of the calling process.
[ESRCH]	The requested process does not exist.

See Also

getpgrp(2)

setquota(2)

Name

setquota – enable/disable quotas on a file system

Syntax

setquota(special, file)
char *special, *file;

Description

Disk quotas are enabled or disabled with the `setquota` call. The *special* indicates a block special device on which a mounted file system exists. If *file* is nonzero, it specifies a file in that file system from which to take the quotas. If *file* is zero, then quotas are disabled on the file system. The quota file must exist; it is normally created with the `quotacheck` program.

Only the superuser can turn quotas on or off.

Return Value

A zero (0) return value indicates a successful call. A value of –1 is returned when an error occurs, and *errno* is set to indicate the reason for failure.

Diagnostics

The `setquota` system call fails when one of the following occurs:

[ENODEV] The *special* does not exist.

[ENOTBLK] The *special* is not a block device.

[ENXIO] The major device number of *special* is out of range. (This indicates no device driver exists for the associated hardware.)

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

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

[EACCES] The *file* resides on a file system different from *special*.

[EACCES] The *file* is not a plain file.

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

[EINVAL] The kernel has not been compiled with the QUOTA option.

[ENAMETOOLONG] A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

[ENOENT] The *file* does not exist.

[ELOOP] Too many symbolic links were encountered in translating either pathname.

[EPERM] The caller is not the superuser.

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

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

[EFAULT] The *special* or *path* points outside the process's allocated address space.

See Also

quota(2), edquota(8), quotacheck(8), quotaon(8),
"Disk Quotas in a UNIX Environment", *Supplementary Documents,* Vol. III:System Manager

setregid(2)

Name

setregid – set real and effective group ID

Syntax

setregid(*rgid, egid*)
int *rgid, egid*;

Description

The real and effective group ID's of the current process are set to the arguments.

Supplying a value of –1 for either the real or effective group ID forces the system to substitute the current ID for the –1 parameter.

Environment

BSD
If the process is superuser, or `rgid` and `egid` matches with the real group ID, the effective group ID, or the saved set-group-id (as described in `execve`(2)), then the real, effective, and saved set-group-id are set to `rgid`, `egid`, and `egid`, respectively.

POSIX
SYSTEM-FIVE
When your program is compiled in POSIX or SYSTEM-FIVE mode, the following semantics apply when using the `setregid` function.

If the process is the superuser, the real, effective, and saved set-group-id (as described in `execve`(2)) are set to `rgid`, `egid`, and `egid`, respectively.

If the process is not the superuser, but the `rgid` and `egid` matches the real group ID, the effective group ID (only in SYSTEM-FIVE and BSD environment), or the saved set-group-id, then the effective ID is set to `egid`. The real group ID and the saved set-group-id are left unchanged.

Return Value

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

Diagnostics

[EPERM]	The current process is not the superuser and the `egid` and `rgid` specified does not match with the real group ID or the effective group ID (only in SYSTEM-FIVE and BSD environment) or the saved set-group-id.

See Also

getgid(2), setreuid(2), setgid(3)

Name

setreuid – set real and effective user ID's

Syntax

setreuid(*ruid, euid*)
int*ruid, euid*;

Description

The real and effective user ID's of the current process are set according to the arguments. If *ruid* or *euid* is –1, the current uid is filled in by the system.

Return Value

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

Environment

BSD
If the process is superuser, or `ruid` and `euid` matches with the real user ID, the effective user ID, or the saved set-user-id (as described in `execve`(2)), then the real, effective, and the saved set-user-id are set to `ruid`, `euid`, and `euid`, respectively.

POSIX
SYSTEM-FIVE
When your program is compiled in the POSIX or SYSTEM-FIVE mode, if both arguments to `setreuid` are –1, the system call returns a value of –1 and errno is set to [EINVAL].

The following semantics apply when using the `setreuid` function:

If the process is the superuser, the real, effective, and saved set-user-id (as described in `execve`(2)) are set to `ruid`, `euid`, and `euid`, respectively.

If the process is not the superuser, but the `ruid` and `euid` matches with the real user ID, the effective user ID (only in the SYSTEM-FIVE and BSD environments), or the saved set-user-id, then the effective ID is set to `euid`. The real user ID and the saved set-user-id are left unchanged.

Diagnostics

[EPERM]	The current process is not the superuser and the `euid` and `ruid` specified does not match with the real user ID, the effective user ID (only in SYSTEM-FIVE and BSD environment), or the saved set-user-id.

See Also

getuid(2), setregid(2), setuid(3)

setsid (2)

Name

setsid – POSIX create session and set process group ID

Syntax

#include <sys/types.h>
pid_t
setsid()

Description

The `setsid` system call creates a new session, if the calling process is not a process group leader. The calling process is the session leader of the new session, the process group leader of the new process group, and does not have a controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process.

Return Value

Upon successful completion, the `setsid` system call returns the value of the process group ID of the calling process. If the `setsid` system call fails, –1 is returned, and the global variable *errno* indicates the reason.

Diagnostics

The `setsid` system call fails and a new session is not created if the following occurs:

[EPERM] The calling process is already a process group leader.

The process group ID of a process other than the calling process matches the process ID of the calling process.

See Also

getpgrp(2), setpgid(3)

Name

setsysinfo – set system information

Syntax

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

setsysinfo(op, buffer, nbytes, arg, flag)
unsigned        op;
char            *buffer;
unsigned        nbytes;
unsigned        arg;
unsigned        flag;
```

Description

The `setsysinfo` system call modifies system information. The *op* argument specifies the operation to be performed. Values for *op* are defined in the `<sys/sysinfo.h>` header file. The optional *buffer* and *nbytes* arguments are used to pass data, which varies depending upon *op*. When *buffer* and *nbytes* are not required, they should be set to NULL. The optional *arg* argument can be used with certain *op* values for additional information. When *arg* is not required, it should be set to NULL. The optional *flag* argument can be used with certain *op* and *arg* values for additional information. When *flag* is not required it should be set to NULL.

Possible *op* values are:

op = SSI_NVPAIRS

Use a list of *name-value* pairs to modify predefined system variables. *Buffer* is an array of *name-value* pairs, where *name* is one of a predefined set of system variables defined in the <sys/sysinfo.h> header file.

Possible *name* values are:

SSIN_NFSPORTMON
A Boolean that determines whether incoming NFS traffic is originating at a privileged port or not.

SSIN_NFSSETLOCK
A Boolean that determines whether NFS (daemon) style file and record locking are enabled or not.

SSIN_PROG_ENV
Set the compatibility mode of the process. Possible values are A_BSD, A_POSIX, or A_SYSV.

SSIN_UACSYS (RISC only)
A Boolean that deteremines whether or not the system prints an "unaligned access fixup" message. Use of this is restricted to the superuser.

SSIN_UACPARNT (RISC only)
A Boolean that is set in the current process's parent proc structure. It turns printing of ''unaligned access fixups'' on or off. This flag is inherited across forks and execs. If parent is init, it returns EPERM.

SSIN_UACPROC (RISC only)
A Boolean value that is set in the proc structure to turn off/on printing of "unaligned access fixup" messages. This flag is inherited across forks and execs.

The *value* is a legal value for *name*. The *nbytes* argument defines the number of *name-value* pairs in *buffer*. The *arg* and *flag* arguments are not used.

op = SSI_ZERO_STRUCT
Each member of a system structure is set to zero. The *arg* defines the structure type.

Possible values for *arg* are:

SSIS_NFS_CLSTAT
NFS client statistics.

SSIS_NFS_SVSTAT
NFS server statistics.

SSIS_RPC_STAT
RPC statistics. The *flag* argument is used for a particular *arg* value, to further define the operation or a resultant action to be performed. The *buffer* and *nbytes* arguments are not used.

Permission checking is done on a structure-by-structure basis.

op = SSI_SET_STRUCT
Each member of a system structure is set to a supplied value. The *arg* defines the structure type.

Possible values for *arg* are as defined for op SSI_STRUCT_ZERO. The *flag* argument is used for a particular *arg* value, to further define the operation or a resultant action to be performed. The *buffer* argument is the address of a structure of the appropriate type that contains the desired values. The *nbytes* argument specifies the amount of data to be transfered that is stored at *buffer*.

Return Value

A zero (0) is returned if the call succeeds. If the call fails, –1 is returned, and the global variable *errno* is set to indicate the error.

Diagnostics

[EFAULT]	Either *buffer* or *arg* causes an illegal address to be referenced.
[EINVAL]	The *op, arg,* or *flag* argument is invalid.
[EPERM]	Permission is denied for the operation requested

See Also

getsysinfo(2)

Name

shmctl – shared memory control operations

Syntax

#include <*sys/types.h*>
#include <*sys/ipc.h*>
#include <*sys/shm.h*>

int shmctl (*shmid, cmd, buf*)
int *shmid, cmd*;
struct *shmid_ds* **buf*;

Description

The `shmctl` system call provides a variety of shared memory control operations, as specified by *cmd*. The following *cmds* are available:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in `intro`(2).

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode    /* only low 9 bits */
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of the superuser or to the value of `shm_perm.uid` in the data structure associated with *shmid*.

IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of the superuser or to the value of `shm_perm.uid` in the data structure associated with *shmid*.

SHM_LOCK Lock the shared memory segment specified by *shmid* in memory. Lock prevents the shared memory segment from being swapped or paged. This *cmd* can only be executed by a process that has an effective user ID equal to the superuser.

SHM_UNLOCK Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that has an effective user ID equal to the superuser.

Return Value

Upon successful completion, a value of zero (0) is returned. Otherwise, a value of –1 is returned, and *errno* is set to indicated the error.

Diagnostics

The `shmctl` system call fails if any of the following is true:

[EINVAL]	The *shmid* is not a valid shared memory identifier.
[EINVAL]	The *cmd* is not a valid command.
[EACCES]	The *cmd* is equal to IPC_STAT, and read permission is denied to the calling process. For further information, see `errno`(2).
[EPERM]	The *cmd* is equal to IPC_RMID or IPC_SET, and the effective user ID of the calling process is not equal to that of the superuser or to the value of `shm_perm.uid` in the data structure associated with *shmid*.
[EPERM]	The *cmd* is equal to SHM_LOCK or SHM_UNLOCK and the effective user ID of the calling process is not equal to that of the superuser.
[EINVAL]	The *cmd* is equal to SHM_LOCK, and the shared memory segment is currently locked by this process.
[EINVAL]	The *cmd* is equal to SHM_UNLOCK, and the shared memory segment specified by *shmid* is not currently locked in memory by this process.
[EFAULT]	The *buf* points to an illegal address.

See Also

shmget(2), shmop(2)

Name

shmget – get shared memory segment

Syntax

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

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

Description

The `shmget` system call returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of size *size* bytes are created for *key*, if one of the following is true:

The *key* is equal to IPC_PRIVATE. For further information, see `intro`(2).

The *key* does not already have a shared memory identifier associated with it, and (*shmflg* & IPC_CREAT) is true.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

The `shm_perm.cuid`, `shm_perm.uid`, `shm_perm.cgid`, and `shm_perm.gid` are set equal to the effective user ID and effective group ID of the calling process.

The low-order nine bits of `shm_perm.mode` are set equal to the low-order nine bits of *shmflg*. The `shm_segsz` is set equal to the value of *size*.

The `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are set equal to zero (0). The **shm_ctime** is set equal to the current time.

Return Value

Upon successful completion, a non-negative integer, namely, a shared memory identifier is returned. Otherwise, a value of –1 is returned and *errno* is set to indicated the error.

Diagnostics

The `shmget` system call fails if any of the following is true:

[EINVAL]	The *size* is less than the system-imposed minimum or greater than the system-imposed maximum.
[EACCES]	A shared memory identifier exists for *key*, but operations permission, as specified by the low-order nine bits of *shmflg*, would not be granted. For further information, see `errno`(2).
[EINVAL]	A shared memory identifier exists for *key*, but the size of the segment associated with it is less than *size* and *size* is not equal to zero.

[ENOENT]	A shared memory identifier does not exist for *key*, and (*shmflg* & IPC_CREAT) is false.
[ENOSPC]	A shared memory identifier is to be created, but the system-imposed limit on the maximum number of allowed shared memory identifiers would be exceeded.
[ENOMEM]	A shared memory identifier and the associated shared memory segment are to be created, but the amount of available physical memory is not sufficient to fill the request.
[EEXIST]	A shared memory identifier exists for *key*, but ((*shmflg* & IPC_CREAT) and (*shmflg* & IPC_EXCL)) is true.

See Also

shmctl(2), shmop(2), ftok(3)

Name

shmop, shmat, shmdt – shared memory operations

Syntax

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

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *shmaddr;
```

Description

The `shmat` system call attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is true, the segment is attached at the address given by (*shmaddr*– (*shmaddr* modulus SHMLBA)).

If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is false, the segment is attached at the address given by *shmaddr*.

The segment is attached for reading if (*shmflg* & SHM_RDONLY) is true. Otherwise, it is attached for reading and writing.

The `shmdt` system call detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

Return Value

Upon successful completion, the return values are as follows:

- The `shmat` system call returns the data segment start address of the attached shared memory segment.
- The `shmdt` system call returns a value of zero (0).

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

Diagnostics

The `shmat` system call fails and not attach the shared memory segment, if any of the following is true:

[EINVAL]	The *shmid* is not a valid shared memory identifier.
[EACCES]	Operation permission is denied to the calling process. For further information, see `errno`(2).

[ENOMEM] The available data space is not large enough to accommodate the shared memory segment.

[EINVAL] The *shmaddr* is not equal to zero, and the value of (*shmaddr*– (*shmaddr* modulus SHMLBA)) is an illegal address.

[EINVAL] The *shmaddr* is not equal to zero, (*shmflg* & SHM_RND) is false, and the value of *shmaddr* is an illegal address.

[EMFILE] The number of shared memory segments attached to the calling process would exceed the system imposed limit.

The `shmdt` fails and does not detach the shared memory segment if:

[EINVAL] The *shmaddr* is not the data segment start address of a shared memory segment.

See Also

execve(2), exit(2), fork(2), shmctl(2), shmget(2)

Name

shutdown – shut down full-duplex connection

Syntax

shutdown(s, how)
int s, how;

Description

The `shutdown` call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, further receives are disallowed. If *how* is 1, further sends are disallowed. If *how* is 2, further sends and receives are disallowed.

Return Value

A zero (0) is returned if the call succeeds, –1 if it fails.

Diagnostics

The call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor.
[ENOTSOCK]	The *s* argument is a file, not a socket.
[ENOTCONN]	The specified socket is not connected.

See Also

connect(2), socket(2)

sigblock(2)

Name

sigblock – block signals

Syntax

sigblock(*mask*)
int *mask*;

Description

The `sigblock` system call causes the signals specified in *mask* to be added to the set of signals currently being blocked from delivery. Signal *i* is blocked if the *i*th bit in *mask* is a 1.

It is not possible to block SIGKILL or SIGSTOP. This restriction is silently imposed by the system.

The previous set of masked signals is returned.

See Also

kill(2), sigsetmask(2), sigvec(2)

Name

sigpause – atomically release blocked signals and wait for interrupt

Syntax

sigpause(sigmask)
int sigmask;

Description

The `sigpause` system call assigns *sigmask* to the set of masked signals and then waits for a signal to arrive. On return, the set of masked signals is restored. The *sigmask* is usually 0 to indicate that no signals are now to be blocked. The `sigpause` always terminates by being interrupted, returning EINTR.

In normal usage, a signal is blocked using `sigblock`(2) at the beginning of a critical section of code. Variables modified on the occurrence of the signal are examined to determine if there is any work to be done. The process pauses, awaiting work, by using `sigpause` with the mask returned by `sigblock`.

See Also

sigblock(2), sigvec(2)

sigpending(2)

Name

sigpending – examine pending signals

Syntax

#include <signal.h>

sigpending(*set*)
sigset_t *set;

Description

The `sigpending` system call stores the set of signals that is blocked from delivery and pending for the calling process in the space pointed to by the argument *set*.

The *set* argument is manipulated by using the `sigsetops`(3) functions.

Return Value

A zero (0) return value indicates that the call succeeded. A –1 return value indicates an error occurred, and *errno* is set to indicate the reason.

Diagnostics

The `sigpending` system call fails if the following occurs:

[EFAULT] The *set* argument points to memory that is not a valid part of the process address space.

See Also

sigprocmask(3), sigsetops(3)

Name

sigsetmask – set current signal mask

Syntax

sigsetmask(*mask*)
int *mask*;

Description

The `sigsetmask` system call sets the current signal mask (those signals that are blocked from delivery). Signal *i* is blocked if the *i*th bit in *mask* is a 1.

The system quietly disallows SIGKILL or SIGSTOP to be blocked.

The previous set of masked signals is returned.

See Also

kill(2), sigblock(2), sigpause(2), sigvec(2)

sigstack (2)

Name

sigstack – set or get signal stack context

Syntax

```
#include <signal.h>

struct sigstack {
        caddr_t         ss_sp;
        int     ss_onstack;
};

sigstack(ss, oss)
struct sigstack *ss, *oss;
```

Description

The `sigstack` system call allows users to define an alternate stack on which signals are to be processed. If *ss* is nonzero, it specifies a *signal stack* on which to deliver signals and tells the system if the process is currently executing on that stack. When a signal's action indicates its handler should execute on the signal stack (specified with a `sigvec` call), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution. If *oss* is nonzero, the current signal stack state is returned.

Signal stacks are not grown'automatically, as is done for the normal stack. If the stack overflows, unpredictable results may occur.

Return Value

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

Diagnostics

The `sigstack` system call fails and the signal stack context remains unchanged, if one of the following occurs.

[EFAULT] Either *ss* or *oss* points to memory that is not a valid part of the process address space.

See Also

sigvec(2), setjmp(3)

Name

sigvec – software signal facilities

Syntax

```
#include <signal.h>

struct sigvec {
        void            (*sv_handler)();
        sigset_t sv_mask;
        int             sv_flags;
};

sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

Description

The system defines a set of signals that can be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process can specify a handler to which a signal is delivered, or specify that a signal is to be blocked or ignored . A process can also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process. This can be changed, on a per-handler basis, so that signals are taken on a special signal stack.

All signals have the same priority. Signal routines execute with the signal that caused their invocation blocked, but other signals can occur. A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally, 0). It can be changed with a `sigblock`(2) or `sigsetmask`(2) call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process, it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described later), and the signal handler is invoked. The call to the handler is arranged so that, if the signal handling routine returns normally, the process resumes execution in the context from before the signal's delivery. If the process wishes to resume in a different context, it must arrange to restore the previous context itself.

When a signal is delivered to a process, a new signal mask is installed for the duration of the process's signal handler (or until a `sigblock` or `sigsetmask` call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and ORing in the signal mask associated with the handler to be invoked.

The `sigvec` System call assigns a handler for a specific signal. If *vec* is nonzero, it specifies a handler routine and mask to be used when delivering the specified signal. Further, if the SV_ONSTACK bit is set in *sv_flags*, the system delivers the signal to the process on a signal stack, specified with `sigstack`(2). If *ovec* is nonzero, the previous handling information for the signal is returned to the user.

The following is a list of all signals with names as in the include file `<signal.h>`:

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	Floating point exception
SIGKILL	9	Kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	Write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGURG	16.	Urgent condition present on socket
SIGSTOP	17+	Stop (cannot be caught, blocked, or ignored)
SIGTSTP	18+	Stop signal generated from keyboard
SIGCONT	19.	Continue after stop (cannot be blocked)
SIGCHLD	20.	Child status has changed
SIGTTIN	21+	Background read attempted from control terminal
SIGTTOU	22+	Background write attempted to control terminal
SIGIO	23.	I/O is possible on a descriptor (see fcntl(2))
SIGXCPU	24	Cpu time limit exceeded (see setrlimit(2))
SIGXFSZ	25	File size limit exceeded (see setrlimit(2))
SIGVTALRM	26	Virtual time alarm (see setitimer(2))
SIGPROF	27	Profiling timer alarm (see setitimer(2))
SIGWINCH	28.	Window size change
SIGLOST	29	Lock not reclaimed after server recovery
SIGUSR1	30	User-defined signal 1
SIGUSR2	31	User-defined signal 2
SIGCLD		System V name for SIGCHLD
SIGABRT		X/OPEN name for SIGIOT

The signals marked with asterisks (*) in this list cause a core image if not caught or ignored. Explanations of the meaning of the periods (.) and plus signs (+) are included in the following paragraph.

Once a signal handler is installed, it remains installed until another `sigvec` call is made or an `execve`(2) is performed. The default action for a signal can be reinstated by setting *sv_handler* to SIG_DFL. This default is termination (with a core image for signals marked with asterisks (*)), except for signals marked with periods (.) or plus signs (+). Signals marked with periods (.) are discarded if the action is SIG_DFL. Signals marked with plus signs (+) cause the process to stop. If *sv_handler* is SIG_IGN, the signal is subsequently ignored, and pending instances of the signal are discarded.

If a caught signal occurs during certain system calls, the call is normally restarted. The call can be forced to terminate prematurely with an EINTR error return, by setting the SV_INTERRUPT bit in *sv_flags*. The affected system calls are `read`, `write`, or `ioctl` on a slow device (such as a terminal, but not a file), `flock`, and `wait`.

After a `fork` or `vfork`, the child inherits all signals, the signal mask, the signal stack, and the restart/interrupt flags.

The `execve` system call resets all caught signals to default action and resets all signals to be caught on the user stack. Ignored signals remain ignored, the signal mask remains the same; signals that interrupt system calls continue to do so.

The mask specified in *vec* is not allowed to block SIGKILL, SIGSTOP, or SIGCONT. This is done silently by the system.

The SV_INTERRUPT flag is not available in ULTRIX 2.0 or earlier versions. Therefore, it should not be used if backward compatibility is needed.

Return Value

A zero (0) value indicates that the call succeeded. A –1 return value indicates an error occurred, and *errno* is set to indicated the reason.

Diagnostics

The `sigvec` system call fails and no new signal handler is installed, if one of the following occurs:

[EFAULT]	Either *vec* or *ovec* points to memory that is not a valid part of the process address space.
[EINVAL]	*Sig* is not a valid signal number.
[EINVAL]	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.
[EINVAL]	An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

Notes

The handler routine can be declared:

```
void handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number. MIPS hardware exceptions are mapped to specific signals as defined by the following table. *Code* is a parameter that is either a constant or zero. The *scp* is a pointer to the *sigcontext* structure (defined in `<signal.h>`, that is the context at the time of the signal and is used to restore the context, if the signal handler returns.

The following defines the mapping of MIPS hardware exceptions to signals and codes. All of these symbols are defined in either *<signal.h>* or *<mips/cpu.h>*:

Hardware exception	Signal	Code
Integer overflow	SIGFPE	EXC_OV
Segmentation violation	SIGSEGV	SEXC_SEGV
Illegal instruction	SIGILL	EXC_II
Coprocessor unusable	SIGILL	SEXC_CPU
Data bus error	SIGBUS	EXC_DBE
Instruction bus error	SIGBUS	EXC_IBE
Read address error	SIGBUS	EXC_RADE

Write address error	SIGBUS	EXC_WADE
User breakpoint (used by debuggers)	SIGTRAP	BRK_USERBP
Kernel breakpoint (used by prom)	SIGTRAP	BRK_KERNELBP
Taken branch delay emulation	SIGTRAP	BRK_BD_TAKEN
Not taken branch delay emulation	SIGTRAP	BRK_BD_NOTTAKEN
User single step (used by debuggers)	SIGTRAP	BRK_SSTEPBP
Overflow check	SIGTRAP	BRK_OVERFLOW
Divide by zero check	SIGTRAP	BRK_DIVZERO
Range error check	SIGTRAP	BRK_RANGE

When a signal handler is reached, the program counter in the signal context structure (*sc_pc*) points at the instruction that caused the exception, as modified by the *branch delay* bit in the *cause* register. The *cause* register at the time of the exception is also saved in the sigcontext structure (*sc_cause*). If the instruction that caused the exception is at a valid user address, it can be retrieved with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD){
    branch_instruction = *(unsigned long *)(scp->sc_pc);
    exception_instruction = *(unsigned long *)(scp->sc_pc + 4);
}
else
    exception_instruction = *(unsigned long *)(scp->sc_pc);
```

CAUSE_BD is defined in <mips/cpu.h>.

The signal handler can fix the cause of the exception and re-execute the instruction, emulate the instruction and then step over it, or perform some nonlocal redirection, such as a `longjump()` or an `exit()`.

If corrective action is performed in the signal handler and the instruction that caused the exception would then execute without a further exception, the signal handler simply returns and re-executes the instruction (even when the *branch delay* bit is set).

If execution is to continue after stepping over the instruction that caused the exception, the program counter must be advanced. If the *branch delay* bit is set, the program counter is set to the target of the branch. Otherwise, it is incremented by four. This can be done with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD)
    emulate_branch(scp, branch_instruction);
else
    scp->sc_pc += 4;
```

Emulate_branch() modifies the program counter value in the sigcontext structure to the target of the branch instruction. See `emulate_branch`(3) for more details.

For SIGFPE's generated by floating-point instructions (*code* == 0) the *floating-point control and status* register at the time of the exception is also saved in the sigcontext structure (*sc_fpc_csr*). This register has the information on which exceptions have occurred. When a signal handler is entered, the register contains the value at the time of the exception but with the *exceptions bits* cleared. On a return from the signal handler, the exception bits in the floating-point control and status register are also cleared so that another SIGFPE will not occur (all other bits are restored from *sc_fpc_csr*).

For SIGSEGV and SIGBUS errors, the faulting virtual address is saved in *sc_badvaddr* in the signal context structure.

The SIGTRAPs caused by `break` instructions noted in the previous table and all other yet to be defined `break` instructions fill the *code* parameter with the first argument to the `break` instruction (bits 25-16 of the instruction).

See Also

kill(1), kill(2), ptrace(2), sigblock(2), sigpause(2), sigsetmask(2), sigstack(2), setjmp(3), siginterrupt(3), tty(4)

'AX **sigvec (2)**

Name

sigvec – software signal facilities

Syntax

```
#include <signal.h>

struct sigvec {
void            (*sv_handler)();
sigset_t sv_mask;
int             sv_flags;
};

sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

Description

The system defines a set of signals that can be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt; the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process can specify a handler to which a signal is delivered, or specify that a signal is to be blocked or ignored. A process can also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process. This can be changed on a per-handler basis so that signals are taken on a special signal stack.

All signals have the same priority. Signal routines execute with the signal that caused their invocation to be blocked , but other signals can occur. A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a `sigblock` or `sigsetmask` call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process, it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described later), and the signal handler is invoked. The call to the handler is arranged so that, if the signal handling routine returns normally, the process resumes execution in the context from before the signal's delivery. If the process wishes to resume in a different context, it must arrange to restore the previous context itself.

When a signal is delivered to a process, a new signal mask is installed for the duration of the process's signal handler (or until a `sigblock` or `sigsetmask` call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and ORing in the signal mask associated with the handler to be invoked.

The `sigvec` system call assigns a handler for a specific signal. If *vec* is nonzero, it specifies a handler routine and mask to be used when delivering the specified signal. Further, if the SV_ONSTACK bit is set in *sv_flags,* the system delivers the signal to the process on a signal stack, specified with `sigstack`. If *ovec* is nonzero, the previous handling information for the signal is returned to the user.

The following is a list of all signals with names as in the include file <signal.h>:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16.	urgent condition present on socket
SIGSTOP	17+	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18+	stop signal generated from keyboard
SIGCONT	19.	continue after stop
SIGCHLD	20.	child status has changed
SIGTTIN	21+	background read attempted from control terminal
SIGTTOU	22+	background write attempted to control terminal
SIGIO	23.	I/O is possible on a descriptor (see fcntl(2))
SIGXCPU	24	cpu time limit exceeded (see setrlimit(2))
SIGXFSZ	25	file size limit exceeded (see setrlimit(2))
SIGVTALRM	26	virtual time alarm (see setitimer(2))
SIGPROF	27	profiling timer alarm (see setitimer(2))
SIGWINCH	28.	window size change
SIGLOST	29	lock not reclaimed after server recovery
SIGUSR1	30	user defined signal 1
SIGUSR2	31	user defined signal 2
SIGCLD		System V name for SIGCHLD
SIGABRT		X/OPEN name for SIGIOT

The signals marked with asterisks (*) in this list cause a core image if not caught or ignored. Explanations of the meaning of the periods (.) and plus signs (+) are included in the following paragraph.

Once a signal handler is installed, it remains installed until another `sigvec` call is made or an `execve` is performed. The default action for a signal can be reinstated by setting *sv_handler* to SIG_DFL. This default is termination (with a core image for signals marked with asterisks (*)), except for signals marked with periods (.) or plus signs (+). Signals marked with periods (.) are discarded if the action is SIG_DFL. Signals marked with plus signs (+) cause the process to stop. If *sv_handler* is SIG_IGN the signal is subsequently ignored, and pending instances of the signal are discarded.

If a caught signal occurs during certain system calls, the call is normally restarted. The call can be forced to terminate prematurely with an EINTR error return, by setting the SV_INTERRUPT bit in *sv_flags*. The affected system calls are `read`, `write`, or `ioctl` on a slow device (such as a terminal; but not a file), `flock`, and `wait`.

After a `fork` or `vfork`, the child inherits all signals, the signal mask, the signal stack, and the restart/interrupt flags.

The `execve` system call resets all caught signals to default action and resets all signals to be caught on the user stack. Ignored signals remain ignored, the signal mask remains the same; signals that interrupt system calls continue to do so.

The mask specified in *vec* is not allowed to block SIGKILL or SIGSTOP. This is done silently by the system.

The SV_INTERRUPT flag is not available in ULTRIX 2.0 or earlier versions. Therefore, it should not be used if backward compatibility is needed.

Notes

The handler routine can be declared:

```
void handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number into which the hardware faults and traps are mapped as defined in the following table. The *code* is a parameter that is either a constant or, for compatibility mode faults, the code provided by the hardware. Compatibility mode faults are distinguished from the other SIGILL traps by having PSL_CM set in the psl. The *scp* is a pointer to the *sigcontext* structure (defined in `<signal.h>`), used to restore the context from before the signal.

The following defines the mapping of hardware traps to signals and codes. All of these symbols are defined in `<signal.h>`:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by zero fault	SIGFPE	FPE_FLTDIV_FAULT
Floating underflow fault	SIGFPE	FPE_FLTUND_FAULT
Length access control	SIGSEGV	faulting virtual addr
Protection violation	SIGBUS	faulting virtual addr
Reserved instruction	SIGILL	ILL_PRIVIN_FAULT
Customer-reserved instr.	SIGEMT	
Reserved operand	SIGILL	ILL_RESOP_FAULT
Reserved addressing	SIGILL	ILL_RESAD_FAULT
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	
Compatibility-mode	SIGILL	hardware-supplied code
Chme	SIGSEGV	
Chms	SIGSEGV	
Chmu	SIGSEGV	

Return Values

A zero (0) value indicates that the call succeeded. A –1 return value indicates an error occurred, and *errno* is set to indicate the reason.

Diagnostics

The `sigvec` system call fails and no new signal handler is installed, if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory that is not a valid part of the process address space.

[EINVAL] The *sig* argument is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

Environment

SYSTEM_FIVE

You can not use the `sigvec` call in your program under SYSTEM_FIVE environment.

See Also

kill(1), kill(2), ptrace(2), sigblock(2), sigpause(2), sigsetmask(2), sigstack(2), setjmp(3), siginterrupt(3), tty(4)

socket(2)

Name

socket – create an endpoint for communication

Syntax

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

s = socket(af, type, protocol)
int s, af, type, protocol;

Description

The `socket` system call creates an endpoint for communication and returns a descriptor.

The operation of sockets is controlled by socket-level options, defined in the file `<sys/socket.h>` and explained in the section, Socket-level Options. The calls `setsockopt`(2) and `getsockopt`(2) are used to set and get options.

Arguments

The *af* parameter specifies an address format. Addresses specified in later operations using the socket are interpreted according to these formats. The formats are defined in the include file `<sys/socket.h>`:

`AF_UNIX`	UNIX path names
`AF_INET`	ARPA Internet addresses
`AF_IMPLINK`	IMP "host at IMP" addresses
`AF_DLI`	For access to broadcast devices (Ethernet)

The *type* argument specifies the semantics of communication. The defined types are:

```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
```

The SOCK_STREAM and SOCK_DGRAM types are available only if your system includes the TCP/IP network. For example, if you can use the `rlogin` command to log in to a remote ULTRIX node, your system supports these socket types.

A SOCK_STREAM type provides sequenced, reliable, 2-way-connection-based byte streams with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed maximum length, typically small).

SOCK_RAW sockets provide access to internal network interfaces and are available only to the super-user.

The SOCK_SEQPACKET type is the socket protocol to request when you want to communicate with other Digital systems using DECnet.

Socket types are discussed further in following sections.

The *protocol* argument specifies the protocol to be used with the socket. Normally, only a single protocol exists to support a particular socket type using a given address format. However, it is possible that many protocols may exist, in which case a

particular protocol must be specified in this manner. The protocol number to use is particular to the communication domain in which communication is to take place. For further information, see `services`(5) and `protocols`(5).

Socket Type SOCK_STREAM

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data can be sent or received on it. A connection to another socket is created with a `connect` call. Once connected, data can be transferred using `read` and `write` calls or some variant of the `send` and `recv` calls. When a session has been completed, a `close` may be performed. Out-of-band data can also be transmitted as described in `send`(2) and received as described in `recv`(2).

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with –1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets ''warm'' by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period (for example, 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes processes that do not handle the signal to exit.

Socket Types SOCK_DGRAM and SOCK_RAW

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in `send`(2) calls. It is also possible to receive datagrams at these sockets with `recv`(2).

An `fcntl`(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives.

SOCK_DGRAM sockets are the only type of socket allowed by the Data Link Interface.

Socket Type SOCK_SEQPACKET

SOCK_SEQPACKET sockets are similar to datagrams except that they are guaranteed to be received in the sequence that they are sent. They are also guaranteed to be error-free.

Socket-Level Options

The operation of sockets is controlled by socket-level options. These options are defined in the file, `<sys/socket.h>`. The calls `setsockopt` and `getsockopt` are used to set and get options.

Options other than SO_LINGER take an integer parameter that should be nonzero, if the option is to be enabled, or zero (0), if it is to be disabled. SO_LINGER uses a ''linger'' structure parameter defined in `<sys/socket.h>`. This structure specifies the desired state of the option and the linger interval (see the following).

SO_DEBUG	Turn on recording of debugging information
SO_REUSEADDR	Allow local address reuse
SO_KEEPALIVE	Keep connections alive
SO_DONTROUTE	Do not apply routing on outgoing messages
SO_LINGER	Linger on close if data present
SO_BROADCAST	Permit sending of broadcast messages
SO_ACCEPTCONN	Socket has had listen()
SO_USELOOPBACK	Bypass hardware when possible
SO_OOBINLINE	Leave received OOB data in line

SO_DEBUG enables debugging in the underlying protocol modules.

SO_REUSEADDR indicates the rules used in validating addresses supplied in a `bind` call should allow reuse of local addresses.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified through a SIGPIPE signal.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface, according to the network portion of the destination address.

SO_LINGER controls the actions taken when unsent messages are queued on the socket and a `close` is performed. When using the `setsockopt` to set the linger values, the option value for the SO_LINGER command is the address of a linger structure:

```
struct  linger {
        int     l_onoff;        /* option on/off */
        int     l_linger;       /* linger time */
};
```

If the socket promises reliable delivery of data and l_onoff is nonzero, the system blocks the process on the `close` attempt until it is able to transmit the data or until it decides it is unable to deliver the information. A timeout period, termed the linger interval, is specified in l_linger in seconds. If l_onoff is set to zero (0) and a `close` is issued, the system processes the close in a manner that allows the process to continue as quickly as possible.

SO_BROADCAST is used to enable or disable broadcasting on the socket.

Return Value

A –1 is returned if an error occurs. Otherwise, the return value is a descriptor to be used in other calls to refer to the socket.

Diagnostics

The `socket` call fails if:

[EAFNOSUPPORT] The specified address family is not supported in this version of the system.

[ESOCKTNOSUPPORT]
The specified socket type is not supported in this address family.

[EPROTONOSUPPORT]
The specified protocol is not supported.

[EPROTOTYPE]
Request for a type of socket for which there is no supporting protocol.

[EMFILE] The per-process descriptor table is full.

[ENOBUFS]
No buffer space is available. The socket cannot be created.

See Also

accept(2), bind(2), close(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read (2), recv(2), select(2), send(2), setsockopt(2), shutdown(2), socketpair(2), protocols(5), services(5), write(2),
"A 4.2 BSD Interprocess Communication Primer," *ULTRIX Supplementary Documents,* Vol. III:System Manager,
Guide to the Data Link Interface

socketpair (2)

Name

socketpair – create a pair of connected sockets

Syntax

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

socketpair(d, type, protocol, sv)
int d, type, protocol;
int sv[2];

Description

The `socketpair` call creates an unnamed pair of connected sockets in the specified domain *d*, of the specified *type*, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv*[0] and *sv*[1]. The two sockets are indistinguishable.

Return Value

A zero (0) is returned if the call succeeds, –1 if it fails.

Diagnostics

The call succeeds unless:

[EMFILE]	Too many descriptors are in use by this process.
[EAFNOSUPPORT]	The specified address family is not supported on this machine.
[EPROTONOSUPPORT]	The specified protocol is not supported on this machine.
[EOPNOSUPPORT]	The specified protocol does not support creation of socket pairs.
[EFAULT]	The address *sv* does not specify a valid part of the process address space.

See Also

pipe(2), read(2), write(2)

Name

startcpu – start a CPU

Syntax

startcpu(*cpunumber*)
int *cpunumber*;

Description

The `startcpu` system call starts the CPU specified by *cpunumber*. Any non-boot CPU can be started using this system call. Only a superuser can execute this system call.

Return Values

The `startcpu` call returns 0 if the CPU was started successfully, or else it returns -1 and sets errno appropriately.

Diagnostics

[EPERM]	The caller is not a superuser
[EBUSY]	The CPU is already running
[ENODEV]	No CPU present by the given *cpunumber*
[EINVAL]	Invalid value for *cpunumber*. A valid *cpunumber* is between 0 and 31.

See Also

stopcpu(2), startcpu(8), stopcpu(8)

stat(2)

Name

stat, lstat, fstat – get file status

Syntax

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

stat(*path, buf*)
char **path*;
struct stat **buf*;

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

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

Description

The `stat` system call obtains information about the file *path*. Read, write, or execute permission of the named file is not required, but all directories specified in the path name that leads to the file must be reachable.

The `lstat` system call is like `stat`, except when a named file is a symbolic link. In this instance, `lstat` returns information about the link; `stat` returns information about the file that is referenced by the link.

The `fstat` system call and the `open` system call obtain the same information about an open file referenced by the argument descriptor.

The *buf* is a pointer to a `stat` structure. Information about a file is placed in the `stat` structure. The contents of the structure pointed to by *buf* includes the following:

```
struct stat {
  dev_t    st_dev;     /* device inode resides on */
  ino_t    st_ino;     /* this inode's number */
  u_short  st_mode;    /* protection */
  short    st_nlink;   /* number or hard links to the file */
  short    st_uid;     /* user-id of owner */
  short    st_gid;     /* group-id of 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;   /* file last access time */
  int      st_spare1;
  time_t   st_mtime;    /* file last modify time */
  int      st_spare2;
  time_t   st_ctime;    /* file last status change time */
  int      st_spare3;
  long st_blksize; /* optimal blocksize for file system i/o ops */
  long st_blocks;  /* actual number of blocks allocated */
  long st_spare4;
  u_long st_gennum; /* file generation number */
    };
```

st_atime — The time when file data was last read or modified. This is changed by the system calls mknod(2) utimes(2) and read(2) For efficiency, st_atime is not set when a directory is searched.

st_mtime — The time when data was last modified. It is not set by changes of owner, group, link count, or mode. It is changed by the system calls mknod(2) utimes(2) and write(2)

st_ctime — The time when file status was last changed. It is set by writing and changing the i-node. It can be changed by the following system calls: chmod(2) chown(2) link(2) mknod(2) unlink(2) utimes(2 and write(2)

The status information word *st_mode* has the following bits:

```
#define S_IFMT   0170000 /* type of file */
#define S_IFDIR  0040000 /* directory */
#define S_IFCHR  0020000 /* character special */
#define S_IFBLK  0060000 /* block special */
#define S_IFREG  0100000 /* regular */
#define S_IFLNK  0120000 /* symbolic link */
#define S_IFSOCK 0140000 /* socket */
#define S_IFIFO  0010000 /* FIFO - named pipe */
#define S_ISUID  0004000 /* set user id on execution */
#define S_ISGID  0002000 /* set group id on execution */
#define S_ISVTX  0001000 /* save swapped text even after use */
#define S_IREAD  0000400 /* read permission, owner */
#define S_IWRITE 0000200 /* write permission, owner */
#define S_IEXEC  0000100 /* execute/search permission, owner */
```

The mode bits 0000070 and 0000007 encode group and others permissions. For further information, see chmod(2).

When *fd* is associated with a pipe, fstat returns a buffer with only st_blksize set.

Environment

SYSTEM_FIVE

Unlike the System V definition, ELOOP is a possible error condition.

Restrictions

Applying fstat to a socket returns a zeroed buffer and [EOPNOTSUPP].

The fields in the stat structure marked *st_spare1*, *st_spare2*, and *st_spare3* are used when inode time stamps expand to 64 bits. This, however, can break certain programs that depend on the time stamps being contiguous in calls to utimes.

Return Value

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

Diagnostics

The stat and lstat system calls fail if any of the following is true:

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

[EFAULT] — The *buf* or *name* points to an invalid address.

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG] A component of a pathname exceeds 255 characters, or an entire path name exceeds 1023 characters.

[ENOENT] The named file does not exist or *path* points to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

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

The `fstat` system call fails if one or more of the following are true:

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

[EFAULT] The *buf* points to an invalid address.

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

[EOPNOTSUPP] The file descriptor points to a socket.

[ETIMEDOUT] A connect request or remote file operation failed because the connected party did not respond after a period of time determined by the communications protocol.

See Also

chmod(2), chown(2), link(2), mknod(2), read(2), unlink(2), utimes(2), write(2)

Name

stopcpu – stop a CPU

Syntax

stopcpu(*cpunumber*)
int *cpunumber*;

Description

The `stopcpu` system call stops the CPU specified by *cpunumber*. Any nid-boot CPU can be stopped using this system call. Only a superuser can execute the `stopcpu` system call.

Return Values

The `stopcpu` call returns 0 if the CPU was stopped successfully, or else it returns -1 and sets errno appropriately.

Diagnostics

[EPERM]	The caller is not a superuser
[EACCES]	Trying to stop boot CPU
[EBUSY]	The CPU is already stopped or no such CPU present
[EINVAL]	Invalid value for *cpunumber*. A value *cpunumber* is between 0 and 31.

See Also

startcpu (2), startcpu (8), stopcpu (8)

swapon (2)

Name

swapon – add a swap device for interleaved paging/swapping

Syntax

swapon(special)
char *special;

Description

The `swapon` system call makes the block device *special* available to the system for allocation for paging and swapping. The names of potentially available devices are known to the system and defined at system configuration time. The size of the swap area on *special* is calculated at the time the device is first made available for swapping.

Restrictions

There is no way to stop swapping on a disk so that the pack may be dismounted.

Diagnostics

The `swapon` system call succeeds unless:

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

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

[ENOENT] The named device does not exist.

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The caller is not the super-user.

[ENOTBLK] The *special* is not a block device.

[EBUSY] The device specified by *special* has already been made available for swapping.

[EINVAL] The device configured by *special* was not configured into the system as a swap device.

[ENXIO] The major device number of *special* is out of range. (This indicates that no device driver exists for the associated hardware.)

[EIO] An I/O error occurred while opening the swap device.

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

See Also

config(8), swapon(8)

Name

symlink – make symbolic link to a file

Syntax

symlink(name1, name2)
char *name1, *name2;

Description

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name can be an arbitrary path name. The files need not be on the same file system.

Return Value

Upon successful completion, a zero (0) value is returned. If an error occurs, the error code is stored in *errno*, and a –1 value is returned.

Diagnostics

The symbolic link is made, unless one or more of the following are true:

[ENOTDIR] A component of the *name2* prefix is not a directory.

[EEXIST] The *name2* already exists.

[EACCES] A component of the *name2* path prefix denies search permission.

[EROFS] The file *name2* would reside on a read-only file system.

[EFAULT] The *name1* or *name2* points outside the process's allocated address space.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG]
A component of either pathname exceeded MAXNAMELEN characters, or the entire length of either pathname exceeded MAXPATHNAME characters.

[ENOENT] The named file does not exist.

[EIO] An I/O error occurred while making the directory entry for *name2*, or allocating the inode for *name2*, or writing out the link contents of *name2*.

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

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

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

[EDQUOT] The directory in which the entry for the new symbolic link is being

	placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	The new symbolic link cannot be created because the user's quota of disk blocks on the file system that will contain the symbolic link has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the user's symbolic link is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[ETIMEDOUT]	A connect request or remote file operation failed, because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

ln(1), link(2), readlink(2), stat(2), unlink(2)

Name

sync – update super-block

Syntax

sync()

Description

The `sync` system call causes all information in memory that should be on disk to be written out. This includes modified superblocks, modified i-nodes, and delayed block I/O.

Programs that examine a file system, for example, `fsck` or `df`, use the `sync` system call. The writing, although scheduled, is not necessarily complete upon return from `sync`.

See Also

sync(1), fsync(2), update(8)

syscall (2)

Name

syscall – indirect system call

Syntax

syscall(number, args, ...)

Description

The `syscall` system call performs the system call whose assembly language interface has the specified *number,* and further arguments *args*. There may be no arguments.

The return value of the system call is returned.

Diagnostics

If an error occurs, `syscall` returns –1 and sets the external variable *errno*.

See Also

errno(2)

Name

syscall – indirect system call

Syntax

syscall(number, arg, ...)

Description

The `syscall` system call performs the system call whose assembly language interface has the specified *number*, register arguments *r0* and *r1*, and further arguments *arg*.

The *r0* value of the system call is returned.

Restrictions

There is no way to simulate system calls such as `pipe`, which return values in register *r1*.

Diagnostics

When the C-bit is set, `syscall` returns –1 and sets the external variable *errno*.

See Also

errno(2), pipe(2)

truncate (2)

Name

truncate, ftruncate – truncate a file to a specified length

Syntax

truncate(path, length)
char *path;
int length;

ftruncate(fd, length)
int fd, length;

Description

The `truncate` system call causes the file named by *path* or referenced by *fd* to be truncated to, at most, *length* bytes in size. If the file previously was larger than this size, the extra data is lost. With `ftruncate`, the file must be open for writing.

Return Value

A value of zero (0) is returned if the call succeeds. If the call fails, a –1 is returned, and the global variable *errno* specifies the error.

Restrictions

Partial blocks discarded as the result of truncation are not zero-filled. This can result in holes in files that do not read as zero.

Diagnostics

The `truncate` system call succeeds unless:

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

[ENOENT] The named file does not exist.

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

[EISDIR] The named file is a directory.

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

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed.

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

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EIO] An I/O error occurred updating the inode.

The `ftruncate` system call succeeds unless:

[EBADF] The *fd* is not a valid descriptor.

[EINVAL]	The *fd* references a socket, not a file.
[ETIMEDOUT]	A connect request or remote file operation failed, because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

open(2)

umask(2)

Name

umask – set file creation mask

Syntax

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

oumask = umask(*numask***)**
mode_t oumask, ***numask*****;**

Description

The `umask` system call sets the process's file mode creation mask to `numask` and returns the previous value of the mask. The low-order nine bits of `numask` are used whenever a file is created, clearing corresponding bits in the file mode. (For further information, see `chmod`(2).) This clearing allows each user to restrict the default access to his or her files.

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

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

Environment

POSIX
When your program is compiled in POSIX mode, the *numask* argument is of type mode_t and the `umask` function returns a value of type mode_t.

See Also

chmod(2), mknod(2), open(2)

Name

uname – get name of current system

Syntax

#include <limits.h>
#include <sys/utsname.h>

int uname (name)
struct utsname *name;

Description

The `uname` system call stores information identifying the current system in the structure pointed to by *name*.

The `uname` system call uses the structure defined in `<sys/utsname.h>` whose members are:

```
char    sysname[SYS_NMLN];
char    nodename[SYS_NMLN];
char    release[SYS_NMLN];
char    version[SYS_NMLN];
char    machine[SYS_NMLN];
```

The constant SYS_NMLN is defined in `<limits.h>`.

The `uname` system call returns a null-terminated character string naming the current ULTRIX system in the character array, *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. The *release* and *version* further identify the operating system. The *machine* contains a standard name that identifies the hardware that the ULTRIX system is running on.

Return Value

Upon successful completion, a nonnegative value is returned. Otherwise, –1 is returned, and *errno* is set to indicate the error.

Diagnostics

[EFAULT] The `uname` system call fails if *name* points to an invalid address.

unlink(2)

Name

unlink – remove directory entry

Syntax

unlink(*path*)
char **path*;

Description

The `unlink` system call 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.

Diagnostics

The `unlink` system call succeeds unless:

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

[ENOENT] The named file does not exist or *path* points to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

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

[EACCES] Write permission is denied on the directory containing the link to be removed.

[EBUSY] The entry to be unlinked is the mount point for a mounted file system.

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

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

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

[EPERM] The named file is a directory and the effective user ID of the process is not the superuser.

[EPERM] The named file is a directory and the environment is defined is POSIX.

[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.

[EIO]	An I/O error occurred while deleting the directory entry or deallocating the inode.
[ETIMEDOUT]	A connect request or remote file operation failed, because the connected party did not properly respond after a period of time that is dependent on the communications protocol.
[ETXTBSY]	The named file is the last link to a shared text executable and the environment defined is POSIX or SYSTEM_FIVE.

Environment

Differs from the System V definition in that ELOOP is a possible error condition.

See Also

close(2), link(2), rmdir(2)

ustat(2)

Name

ustat – get file system statistics

Syntax

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

int ustat (dev, buf)
dev_t dev;
struct ustat *buf;

Description

The `ustat` call returns information about a mounted file system. The *dev* argument is a device number identifying a device containing a mounted file system. The *buf* argument is a pointer to a `ustat` structure that includes the following elements:

```
daddr_t  f_tfree;       /* Total free blocks (Kbytes) */
ino_t    f_tinode;      /* Number of free inodes */
char     f_fname[512];  /* Filsys name */
char     f_fpack[6];    /* Filsys pack name */
```

The *f_fpack* always returns a null string.

Environment

SYSTEM V

Differs from System V definition in that the size of the f_fname structure element is 512 instead of 6, and the *dev* parameter is type dev_t instead of int.

Diagnostics

The `ustat` call fails if any of the following is true:

[EINVAL]	The *dev* argument is not the device number of a device containing a mounted file system.
[EFAULT]	The *buf* argument points outside the process's allocated address space.
[ETIMEDOUT]	A connect request or remote file operation failed, because the connected party did not properly respond after a period of time that is dependent on the communications protocol.

See Also

stat(2), fs(5)

Name

utimes – set file times

Syntax

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

utimes(*file, tvp***)**
char ****file*;**
struct timeval ****tvp[2]*;**

Description

The `utimes` call uses the accessed and updated times from the *tvp* vector to set the corresponding recorded times for *file*.

If *tvp* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file, the superuser, or have write permission to use `utimes` in this manner.

If *tvp* is not NULL, the caller must be the owner of the file or the superuser.

The inode-changed time of the file is set to the current time.

Return Value

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

Diagnostics

The `utimes` system call fails if one or more of the following are true:

[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	The *tvp* argument is NULL and the caller is not the owner of the file; write access is denied.
[EFAULT]	The *file* or *tvp* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading or writing the affected inode.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENAMETOOLONG]	A component of a pathname exceeds 255 characters, or an entire pathname exceeds 1023 characters.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The *tvp* argument is not NULL, the caller has write access, the caller is not the owner of the file, and the caller is not the superuser.

[EROFS]	The file system containing the file is mounted read-only.
[ETIMEDOUT]	A connect request or remote file operation failed, because the connected party did not respond after a period of time determined by the communications protocol.

See Also

stat(2)

Name

vfork – spawn new process in a virtual memory-efficient way

Syntax

pid = vfork()
int pid;

Description

The `vfork` can be used to create new processes without fully copying the address space of the old process, which is inefficient in a paged environment. It is useful when the purpose of `fork` would have been to create a new system context for an `execve`. The `vfork` system call differs from `fork` in that the child borrows the parent's memory and thread of control until a call to `execve` or an exit (either by a call to `exit`(2) or abnormally.) The parent process is suspended while the child is using its resources.

The `vfork` system call returns a value of zero (0) in the child's context and, later, the pid of the child in the parent's context.

The `vfork` system call can normally be used just like `fork`. It does not work, however, to return while running in the childs context from the procedure which called `vfork`, because the eventual return from `vfork` would then return to a nonexistent stack frame. Be careful, also, to call *_exit* rather than *exit* if you cannot call `execve`, because *exit* will flush and close standard I/O channels and thereby cause problems in the parent process's standard I/O data structures. Even with `fork` it is wrong to call *exit*, because buffered data would then be flushed twice.

Restrictions

To avoid a possible deadlock situation, processes which are children in the middle of a `vfork` are never sent SIGTTOU or SIGTTIN signals. Rather, output or *ioctl*s are allowed, and input attempts result in an end-of-file indication.

Diagnostics

Same as for `fork`.

See Also

execve(2), fork(2), sigvec(2), wait(2)

vhangup(2)

Name

vhangup – virtually hang up the current control terminal

Syntax

vhangup ( )

Description

The `vhangup` system call initializes a terminal line. For example, the `init` command uses `vhangup` to ensure that the previous user's processes cannot access the terminal anymore.

First, `vhangup` searches the system tables for references to the current terminal (the control terminal of the invoking process) and revokes access permissions on each instance of the terminal that it finds.

The `vhangup` system call also removes all references to the inode that corresponds to the control terminal. The `vhangup` system call then invokes the kernel's device close routine to turn the terminal off. Finally, `vhangup` sends a hangup signal (SIGHUP) to the process group of the control terminal. For further information, see `tty`(4) for a description of process groups.

When `vhangup` finishes, a terminal line is initialized; no other processes refer to this line. The only way for other processes to access the control terminal is through the special file, `/dev/tty`. All other requests will yield I/O errors (EBADF).

See Also

init(8)

Name

wait, wait3, waitpid – wait for process to terminate

Syntax

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

pid = wait(*status***)**
pid_t pid;
union wait *****status;

pid = wait((*union wait****)0)**
pid_t pid;

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

pid = wait3(*status, options, rusage***)**
pid_t pid;
union wait *****status;
int *options;*
struct *rusage* ***rusage;**

pid = waitpid(*pid, status, options***)**
pid_t *pid;*
union wait *****status;
int *options;*

Description

The `wait` system call causes its caller to delay either until a signal is received or one of its child processes terminates. If a child process has died since the last `wait`, return is immediate, returning the process id and exit status of one of the terminated child processes. If a child process does not exist, return is immediate, with the value –1 returned.

On return from a successful `wait` call, if *status* is nonzero, 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>`.

The `wait3` system call provides an alternate interface for programs that must not block when collecting the status of child processes. The *status* parameter is defined as above. The *options* parameter is used to indicate that the call should not block if there are no processes that wish to report status (WNOHANG), or that only children of the current process, which are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal, should have their status reported (WUNTRACED). If *rusage* is nonzero, a summary of the resources used by the terminated process and all its children is returned (this information is not available for stopped processes).

When the WNOHANG option is specified and no processes wish to report status, `wait3` returns a *pid* of zero (0). The WNOHANG and WUNTRACED options can be combined by ORing the two values.

See sigvec(2) for a list of termination statuses (signals). A 0 status indicates normal termination. A special status (0177) is returned for a process stopped by the process tracing mechanism, 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.

The waitpid system call provides an interface for programs that want to wait for a specific child process or child processes from specific process groups. The waitpid system call behaves as follows:

- If *pid* is equal to –1, status is requested for any child process.
- If *pid* is greater than zero, it specifies the process ID of a single child process for which status is requested.
- If *pid* is equal to zero, status is requested for any child process whose process group ID is equal to that of the calling process.
- If *pid* is less than –1, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

The *status* and *options* arguments are defined as above. The waitpid system call behaves identically to the wait system call, if the *pid* argument has a value of –1 and the *options* argument has a value of zero (0).

The wait, wait3, and waitpid system calls are automatically restarted when a process receives a signal while awaiting termination of a child process, unless the SV_INTERRUPT bit has been set for that signal. See sigvec(2).

The following macros, defined in <sys/wait.h> can be used to interpret the information contained in the *status* parameter returned by the wait functions; the *stat_val* argument is the value pointed to by the *status* argument.

WIFEXITED(*stat_val*)
Evaluates to a nonzero value, if status was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)
If the value of WIFEXITED(*stat_val*) is nonzero, this macro evaluates to the low-order eight bits of the *status* argument that the child process passes to _exit or exit, or the value the child process returned from main.

WIFSIGNALED(*stat_val*)
Evaluates to a nonzero value, if status was returned for a child process that terminated due to the receipt of a signal that was not caught.

WTERMSIG(*stat_val*)
If the value of WIFSIGNALED(*stat_val*) is nonzero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)

Evaluates to a nonzero value, if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)
If the value of WIFSTOPPED(*stat_val*) is nonzero, this macro evaluates to the number of the signal that caused the child process to stop.

Return Value

If `wait`, `wait3`, or `waitpid` 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.

The `wait3` and `waitpid` system calls return –1, if there are no children not previously waited for. A value of zero (0) is returned, if WNOHANG is specified and there are no stopped or exited children.

Environment

SYSTEM_FIVE

When your program is compiled using the System V environment, when the SIGCLD signal is being ignored, `wait` continues until all children terminate. SIGCLD is the same as SIGCHLD.

In addition, when using the System V environment, *status* is of type int *.

POSIX

When using the POSIX environment, *status* is of type int *.

In addition, the SV_INTERRUPT flag is always set in POSIX mode, causing the above system calls to always fail, if interrupted by a signal.

Diagnostics

The `wait`, `wait3`, or `waitpid` system calls fail and return is immediate, if any of the following is true:

[ECHILD]	The calling process has no existing unwaited-for child processes.
[ECHILD]	The process or process group specified by *pid* does not exist or is not a child of the calling process.
[EINTR]	The function was interrupted by a signal. The value of the location pointed to by *status* is undefined.
[EINVAL]	The value of the *options* argument is not valid.
[EFAULT]	The *status* or *rusage* arguments point to an illegal address.

See Also

exit(2), ptrace(2), sigvec(2)

write (2)

Name

write, writev – write on a file

Syntax

write (*fd, buf, nbytes*)
int *fd*;
char **buf*;
int *nbytes*;

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

writev (*fd, iov, ioveclen*)
int *fd*;
struct *iovec* **iov*;
int *ioveclen*;

Arguments

fd — Descriptor returned by a `creat`, `open`, `dup`, `fcntl`, `pipe`, or `socket` system call.

buf — Points to the buffer containing the data to be written.

nbytes — Positive integer defining the number of bytes to be written from the buffer.

iov — Points to a data structure of type `iovec`, which defines the starting location of the set of vectors forming the array and the length of each individual vector in the array to be written.

This structure is defined in `<sys/uio.h>` as follows:

```
struct iovec {

        caddr_t   iov_base ;
        int       iov_len ;
} ;
```

The `caddr_t` data type is defined in `<sys/types.h>` and is the recommended way to define an address for a character value. In any case, the address `iov_base` is the starting address of the set of vectors. The integer value `iov_len` is the length of each individual vector, in bytes.

ioveclen — Defines the number of vectors in the array of data to be written. Note that the numbering of the vectors begins with 0 and proceeds through *ioveclen* –1.

Description

The `write` system call attempts to write a buffer of data to a file. The `writev` system call attempts to write an array of buffers of data to a file.

When a file is opened to a device capable of seeking (such as a disk or tape), the write starts at the position given by the file pointer associated with the file descriptor, *fd*. This file pointer is the offset, in bytes, from the beginning of the file where the write is to begin. When the file is first opened, the file pointer is set at 0. It can be modified by the `read`(2) `lseek`(2) and `write` system calls. When the `write` call returns, the file pointer is incremented by the number of bytes actually written.

When the file is opened to a device not capable of seeking (such as sockets, pipes, or terminals), the write starts at the current position. The value of the pointer associated with such an object is undefined.

By default, `write` does asynchronous writes. That is, after the data is written to a buffer cache, control returns to the program. The actual write to a device takes place after control returns. However, if you use an `open` or `fcntl` call to open a file for synchronous writes, control does not return to the program until after the buffer cache has been written to the device.

If a program is using `write` to a remote file over NFS, and an asynchronous write error occurs, then all subsequent `write` requests will return -1 and errno will be set to the asynchronous error code. Also, a subsequent `fsync`(2) or `close`(2) will likewise fail. The return code from `close`(2) should be inspected by any program that can `write` over NFS.

Write requests to a pipe (or FIFO) are handled the same as a regular file, with the following exceptions:

- A file offset is not associated with a pipe. Therefore, each `write` request appends to the end of the pipe.
- Write requests less than or equivalent to {PIPE_BUF} bytes are not interleaved with data from other processes doing writes on the same pipe. Write requests greater than {PIPE_BUF} bytes can interleave on arbitrary boundaries with writes by other processes.
- If the O_NDELAY and O_NONBLOCK flags are clear, a write can cause the process to block, but, under normal completion, it returns nbytes.
- If the O_NDELAY or O_NONBLOCK flag is set, the `write` function does not block the process. Write requests less than or equal to {PIPE_BUF} bytes either succeed and return nbytes or –1, and errno is set to [EWOULDBLOCK]. Write requests that exceed {PIPE_BUF} bytes can return complete success, partial write, or no success, and errno is to [EWOULDBLOCK].

Environment

SYSTEM V

When your program is compiled using the System V environment, and the file was opened with the O_NDELAY flag set, a `write` to a full pipe (or FIFO) returns a zero (0), rather than an error, as for the ULTRIX non-System V environment.

Differs from the System V definition in that the value *nbytes* is *int*, rather than *unsigned*.

When your program is compiled using POSIX environment, EAGAIN is returned in errno, in place of EWOULDBLOCK.

Return Value

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

Diagnostics

The `write` system call fails and the file pointer will remain unchanged, if any of the following is true:

[EACCESS] The file does not permit writing. NFS only.

[EBADF] The *fd* argument is not a valid descriptor open for writing.

[EPIPE] An attempt was made to write to a pipe that is not open for reading by any process.

[EPIPE] An attempt was made to write to a socket of type SOCK_STREAM 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, set by `ulimit`(2) or the maximum file size (approximately 2 Gigabytes).

[EFAULT] Part of the array pointed to by *iov* or data to be written to the file points outside the process's allocated address space.

[EWOULDBLOCK] The O_NDELAY or O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.

[ENOSPC] There is no free space remaining on the file system containing the file.

[EDQUOT] The user's quota of disk blocks on the file system containing the file has been exhausted.

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

[EINTR] The write operation was interrupted, no data was transferred.

[EINVAL] The *nbytes* argument is negative.

[EROFS] The file is on a read-only file system. NFS only.

[ESTALE] The *fd* argument is invalid because the file referred to by that file handle no longer exists or has been revoked. NFS only.

[ETIMEDOUT] A write operation failed because the server did not properly respond after a period of time that is dependent on the `mount`(8nfs) options. NFS only.

See Also

close(2), creat(2), dup(2), fcntl(2), fsync(2), lseek(2), open(2), pipe(2), socket(2)

Index

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Reference Pages Section 2: System Calls
AA-LY15B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

What do you like best about this manual? _____

What do you like least about this manual? _____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

-- Do Not Tear – Fold Here and Tape ----------

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

--- Do Not Tear – Fold Here ----------

Cut
Along
Dotted
Line

Reader's Comments

ULTRIX
Reference Pages Section 2: System Calls
AA-LY15B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

What do you like best about this manual? _____

What do you like least about this manual? _____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

-- Do Not Tear – Fold Here and Tape ----------

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

--- Do Not Tear – Fold Here ----------

Cut
Along
Dotted
Line