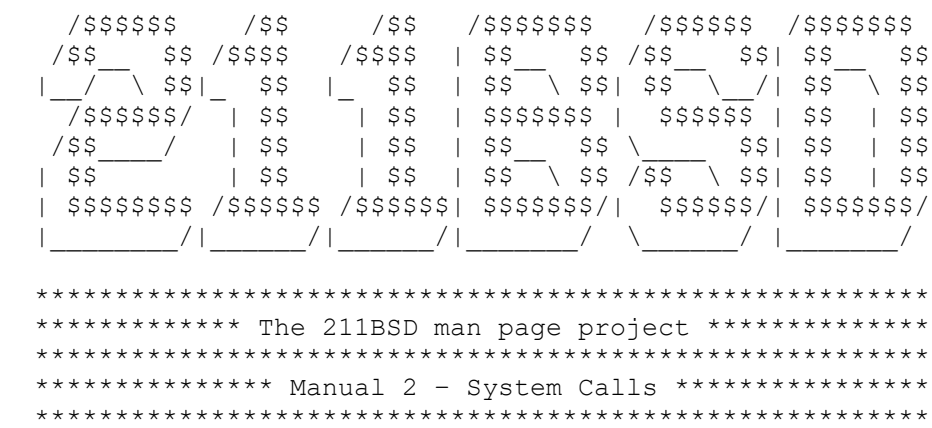

Inspired by:
==========

SimH	http://simh.trailing-edge.com/
PiDP11	https://obsolescence.wixsite.com/obsolescence/pidp-11
BSD 2.11	https://wfjm.github.io/home/211bsd/

Presented by the ShadowTron Blog
==========

https://www.youtube.com/c/shadowtronblog
www.shadowtron.com
shadowtronblog <at> gmail <dot> com

Other manuals in the series
==========

    Manual 1 - Commands and Application Programs
==> Manual 2 - System Calls
    Manual 3 - C Library Subroutines
    Manual 3F - Fortran Library
    Manual 4 - Special Files
    Manual 5 - File Formats
    Manual 6 - Games
    Manual 7 - Miscellaneous
    Manual 8 - System Maintenance

****
**** Manual 2 - System Calls ****
****

Page	Command	Description
4	intro	introduction to system calls and error numbers
15	accept	a connection on a socket
17	access	determine accessibility of file
20	acct	turn accounting on or off
21	adjtime	synchronization of the system clock
23	bind	bind a name to a socket
25	brk	change data segment size
27	chdir	change current working directory
29	chflags	chflags, fchflags - set file flags
31	chmod	change mode of file
33	chown	change owner and group of a file
35	chroot	change root directory
37	close	delete a descriptor
38	connect	initiate a connection on a socket
40	dup	duplicate a descriptor
41	execve	execute a file
45	exit	terminate a process
46	fcntl	file control
49	fetchi	fetch from user instruction space (2BSD
50	flock	apply or remove an advisory lock on an open file
52	fork	create a new process
54	fperr	get floating-point error registers (2BSD
55	fsync	synchronize a file's in-core state with the disk
56	getdtablesize	get descriptor table size
57	getfsstat	get list of all mounted filesystems
59	getgid get	group identity
60	getgroups	get group access list
61	gethostid	get/set unique identifier of current host
62	gethostname	get/set name of current host
63	getitimer	get/set value of interval timer
65	getlogin	get/set login name
67	getpagesize	get system page size
68	getpeername	get name of connected peer
69	getpgrp	get process group
70	getpid get	process identification
71	getpriority	get/set program scheduling priority
73	getrlimit	control maximum system resource consumption
76	getrusage	get information about resource utilization
79	getsockname	get socket name
80	getsockopt	get and set options on sockets
83	gettimeofday	get/set date and time
85	getuid	get user identity
87	ioctl	control device
88	kill	send signal to a process
90	killpg	send signal to a process group
91	link	make a hard link to a file
93	listen	listen for connections on a socket
94	lock	lock a process in primary memory (2BSD
95	lseek	move read/write pointer
97	mkdir	make a directory file
99	mknod	make a special file
101	mount	mount or remove file system

```
104 nostk         allow process to manage its own stack
105 open          open a file for reading,  writing,  or create file
109 phys          allow a process to access physical addresses
110 pipe          create an interprocess communication channel
112 profil        execution time profile
113 ptrace        process trace
116 quota         manipulate disk quotas
119 read          read input
121 readlink      read value of a symbolic link
122 reboot        reboot system or halt processor
124 recv          receive a message from a socket
126 rename        change the name of a file
129 rmdir         remove a directory file
131 select        synchronous I/O multiplexing
134 send          send a message from a socket
136 setgroups     set group access list
137 setpgrp       set process group
138 setquota      enable/disable quotas on a file system
140 setregid      set real and effective group ID
141 setreuid      set real and effective user ID's
142 setuid        allow a process to access physical addresses
144 shutdown      shut down part of a full-duplex connection
145 sigaction     software signal facilities
150 sigaltstack   set and/or get signal stack context
152 sigblock      block signals
153 sigpause      atomically release blocked signals and wait forinterrupt
154 sigpending    get pending signals
155 sigprocmask   manipulate current signal mask
157 sigreturn     return from signal
159 sigsetmask    set current signal mask
160 sigstack      set and/or get signal stack context
162 sigsuspend    atomically release blocked signals and wait for interrupt
163 sigvec        software signal facilities
168 sigwait       wait for a signal
169 socket        create an endpoint for communication
172 socketpair    create a pair of connected sockets
173 stat          get file status
176 statfs        get file system statistics
178 swapon        add a swap device for interleaved paging/swapping
180 symlink       make symbolic link to a file
182 sync          update super-block
183 syscall       indirect system call
184 truncate      truncate a file to a specified length
186 ucall         call a kernel subroutine from user mode
187 umask         set file creation mode mask
188 unlink        remove directory entry
190 utimes        set file times
192 vfork         spawn new process in a virtual memory efficient way
193 vhangup       virtually "hangup" the current control terminal
194 wait          wait for process to terminate
198 write         write output
```

NAME

intro - introduction to system calls and error numbers

SYNOPSIS

#include <sys/errno.h>

DESCRIPTION

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

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

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

0 Error 0
Unused.

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

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

3 ESRCH No such process
The process or process group whose number was given does not exist, or any such process is already dead.

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

5 EIO I/O error
Some physical I/O error occurred during a read or write. This error may in some cases 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 beyond the limits of the device. It may also occur when, for example, an illegal tape drive unit number is selected or a disk pack is not loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 20480 bytes (or the current limit, NCARGS in <sys/param.h>) is presented to execve.

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

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

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

11 EAGAIN 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 memory
During an execve or break, a program asks for more core or swap space than the system is able to supply, or a process size limit would be exceeded. 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. Soft limits may be increased to their corresponding hard limits.

13 EACCES Permission denied
An attempt was made to access a file in a 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
A plain file was mentioned where a block device was required, e.g., in mount.

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

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

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

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device, e.g., to read a write-only device, or the device is not configured by the system.

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

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

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

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

24 EMFILE Too many open files
As released, the limit on the number of open files per process is 64. Getdtablesize(2) will obtain the current limit. Customary configuration limit on most other UNIX systems is 20 per process.

25 ENOTTY Inappropriate ioctl for device
The file mentioned in an ioctl is not a terminal or one of the devices to which this call applies.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program that is currently open for writing. Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum (about 2.1E9 bytes).

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

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

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

31 EMLINK Too many links
An attempt to make more than 32767 hard links to a file.

32 EPIPE Broken pipe
A 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 caught or ignored.

33 EDOM Argument too large
The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

35 EWOULDBLOCK Operation would block
An operation that would cause a process to block was attempted on an object in non-blocking mode (see fcntl(2)).

36 EINPROGRESS Operation now in progress
An operation that takes a long time to complete (such as a connect(2)) was attempted on a non-blocking object

(see fcntl(2)).

37 EALREADY Operation already in progress
An operation was attempted on a non-blocking object that already had an operation in progress.

38 ENOTSOCK Socket operation on non-socket
Self-explanatory.

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

40 EMSGSIZE Message too long
A message sent on a socket was larger than the internal message buffer or some other network limit.

41 EPROTOTYPE Protocol wrong type for socket
A protocol was 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 Option not supported by protocol
A bad option or level was specified in a getsockopt(2) or setsockopt(2) call.

43 EPROTONOSUPPORT Protocol not supported
The protocol has not been configured into the system or no implementation for it exists.

44 ESOCKTNOSUPPORT Socket type not supported
The support for the socket type has not been configured into the system or no implementation for it exists.

45 EOPNOTSUPP Operation not supported on socket
For example, 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 no implementation for it exists.

47 EAFNOSUPPORT Address family not supported by protocol family
An address incompatible with the requested protocol was used. For example, you shouldn't necessarily expect to be able to use NS addresses with ARPA Internet protocols.

48 EADDRINUSE Address already in use
Only one usage of each address is normally permitted.

49 EADDRNOTAVAIL Can't assign requested address
Normally results from an attempt to create a socket with an address not on this machine.

50 ENETDOWN Network is down
A socket operation encountered a dead network.

51 ENETUNREACH Network is unreachable
A socket operation was attempted to an unreachable network.

52 ENETRESET Network dropped connection on reset
The host you were connected to crashed and rebooted.

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

54 ECONNRESET Connection reset by peer
A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.

55 ENOBUFS No buffer space available
An operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.

56 EISCONN Socket is already connected
A connect request was made on an already connected socket; or, a sendto or sendmsg request on a connected socket specified a destination when already connected.

57 ENOTCONN Socket is not connected
An request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket) no address was supplied.

58 ESHUTDOWN Can't send after socket shutdown
A request to send data was disallowed because the socket had already been shut down with a previous shutdown(2) call.

59 unused

60 ETIMEDOUT Connection timed out
A connect or send request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

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

62 ELOOP Too many levels of symbolic links
A path name lookup involved more than 8 symbolic links.

63 ENAMETOOLONG File name too long
A component of a path name exceeded 255 (MAXNAMELEN) characters, or an entire path name exceeded 1023 (MAXPATHLEN-1) characters.

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

65 EHOSTUNREACH Host is unreachable
A socket operation was attempted to an unreachable host.

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

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

DEFINITIONS

Process ID
Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 0 to 30000.

Parent process ID
A new process is created by a currently active process; (see fork(2)). The parent process ID of a process is the process ID of its creator.

Process Group ID
Each active process is a member of a process group that is identified by a positive integer called the process group ID. This is the process ID of the group leader. This grouping permits the signaling of related processes (see killpg(2)) and the job control mechanisms of csh(1).

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; (see csh(1) and tty(4)).

Real User ID and Real Group ID
Each user on the system is identified by a positive integer termed the real user ID.

Each user is also a member of one or more groups. One of these groups is distinguished from others and used in implementing accounting facilities. The positive integer corresponding to this distinguished group is termed the real group ID.

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

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

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

The group access list is an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described below in ``File Access Permissions''.

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

Special Processes
The processes with a process ID's of 0, 1, and 2 are special. Process 0 is the scheduler. Process 1 is the initialization process init, and is the ancestor of every other process in the system. It is used to control the process structure. Process 2 is the paging daemon.

Descriptor
An integer assigned by the system when a file is

referenced by open(2) or dup(2), or when a socket is created by pipe(2), socket(2) or socketpair(2), which uniquely identifies an access path to that file or socket from a given process or any of its children.

File Name

Names consisting of up to 255 (MAXNAMELEN) characters may be used to name an ordinary file, special file, or directory.

These characters may be selected from the set of all ASCII character excluding 0 (null) and the ASCII code for / (slash). (The parity bit, bit 8, must be 0.)

Note that it is generally unwise to use *, ?, [or] as part of file names because of the special meaning attached to these characters by the shell.

Path Name

A path name is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name. The total length of a path name must be less than 1024 (MAXPATHLEN) characters.

If a path name begins with a slash, the path search begins at the root directory. Otherwise, the search begins from the current working directory. A slash by itself names the root directory. A null pathname refers to the current directory.

Directory

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

Root Directory and Current Working Directory

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

File Access Permissions

Every file in the file system has a set of access permissions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access

permissions are established at the time a file is created. They may be changed at some later time through the chmod(2) call.

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

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

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

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

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

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

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

Otherwise, permission is denied.

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 include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format

used in naming message recipients, etc.

Each instance of the system supports some collection of socket types; consult 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.

SEE ALSO

intro(3), perror(3)

NAME

accept - accept a connection on a socket

SYNOPSIS

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

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

DESCRIPTION

The argument s is a socket that has been created with socket(2), bound to an address with bind(2), and is listening for connections after a listen(2). Accept 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 non-blocking, accept blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept returns an error as described below. The accepted socket, ns, may not 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 it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to select(2) a socket for the purposes of doing an accept by selecting it for read.

RETURN VALUE

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

ERRORS

The accept will fail 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 non-blocking and no connections are present to be accepted.

SEE ALSO

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

NAME

access - determine accessibility of file

SYNOPSIS

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

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

accessible = access(path, mode)
int accessible;
char *path;
int mode;
```

DESCRIPTION

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

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

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

RETURN VALUE

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

ERRORS

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

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

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

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

[ENOENT] The named file does not exist.

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

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

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

[ETXTBSY] Write access is requested for a pure procedure (shared text) file that is being executed.

[EACCES] Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix. The owner of a file has permission checked with respect to the ``owner'' read, write, and execute mode bits, members of the file's group other than the owner have permission checked with respect to the ``group'' mode bits, and all others have permissions checked with respect to the ``other'' mode bits.

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

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

SEE ALSO

chmod(2), stat(2)

NAME

acct - turn accounting on or off

SYNOPSIS

```
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 super-user.

NOTES

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 super-user. It is erroneous to try to turn on accounting when it is already on.

ERRORS

Acct will fail if one of the following is true:

[EPERM] The caller is not the super-user.

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

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

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

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix, or the path name is not a regular file.

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

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

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

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

SEE ALSO

acct(5), sa(8)

BUGS

No accounting is produced for programs running when a crash occurs. In particular non-terminating programs are never accounted for.

NAME

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

SYNOPSIS

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

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

DESCRIPTION

Adjtime makes small adjustments to the system time, as returned by gettimeofday(2), advancing or retarding it by the time specified by the timeval delta. If delta is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If delta is positive, a larger increment than normal is used. The skew used to perform the correction is generally a fraction of one percent. 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 non-zero, then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

This call may be used by 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 call adjtime(2) is restricted to the super-user.

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.

ERRORS

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), timed(8), timedc(8), TSP: The Time Synchronization Protocol for UNIX 4.3BSD, R.

Gusella and S. Zatti

NOTES (PDP-11)

Adjtime(2) calls are executed immediately, not over a period of time, therefore, the olddelta return values for an adjtime(2) call will always be zero.

NAME

bind - bind a name to a socket

SYNOPSIS

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

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

DESCRIPTION

Bind assigns a name to an unnamed socket. When a socket is created with socket(2) it exists in a name space (address family) but has no name assigned. Bind requests that name be assigned to the socket.

NOTES

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 unlink(2)).

The rules used in name binding vary between communication domains. Consult the manual entries in section 4 for detailed information.

RETURN VALUE

If the bind is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the global errno.

ERRORS

The bind call will fail if:

[EBADF] S is not a valid 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.

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

[EFAULT] The name parameter is not in a valid

part of the 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.

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

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

[ENOENT] 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), listen(2), socket(2), getsockname(2)

NAME

brk, sbrk - change data segment size

SYNOPSIS

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

char *brk(addr)
char *addr;

char *sbrk(incr)
int incr;
```

DESCRIPTION

Brk sets the system's idea of the lowest data segment location not used by the program (called the break) to addr (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 via execve the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use sbrk.

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, e.g. "etext + rlp->rlim_max." (see end(3) for the definition of etext).

RETURN VALUE

Zero is returned if the brk could be set; -1 if the program requests more memory than the system limit. Sbrk returns -1 if the break could not be set.

ERRORS

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

[ENOMEM] The limit, as set by setrlimit(2), was exceeded.

[ENOMEM] The maximum possible size of 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), malloc(3), end(3)

BUGS

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.

NAME

chdir, fchdir - change current working directory

SYNOPSIS

```
chdir(path)
char *path;

fchdir(fd)
int fd;
```

DESCRIPTION

The path argument points to the pathname of a directory. The fd argument is a file descriptor which references a directory. The chdir function causes this directory to become the current working directory, the starting point for path names not beginning with ``/''.

The fchdir function causes the directory referenced by fd to become the current working directory, the starting point for path searches of pathnames not beginning with a slahs, '/'.

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

RETURN VALUE

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

ERRORS

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

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

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

[ENAMETOOLONG] A component of a pathname exceeded 63 characters, or an entire path name exceeded 255 characters.

[ENOENT] The named directory does not exist.

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

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

[EFAULT] Path points outside the process's allocated

address space.

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

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

[EACCES] Search permission is denied for the directory referenced by the file descriptor.

[ENOTDIR] The file descriptor fd does not reference a directory.

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

SEE ALSO

chroot(2)

NAME

chflags, fchflags - set file flags

SYNOPSIS

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

int
chflags(path, flags)
char *path;
u_short flags;

int
fchflags(fd, flags)
int fd;
u_short flags;
```

DESCRIPTION

The file whose name is given by path or referenced by the descriptor fd has its flags changed to flags .

The flags specified are formed by or'ing the following values

UF_NODUMP Do not dump the file.

UF_IMMUTABLE The file may not be changed.

UF_APPEND The file may only be appended to.

ARCHIVED File is archived.

SF_IMMUTABLE The file may not be changed.

SF_APPEND The file may only be appended to.

The UF_IMMUTABLE and UF_APPEND flags may be set or unset by either the owner of a file or the super-user.

The SF_IMMUTABLE and SF_APPEND flags may only be set or unset by the super-user. They may be set at any time, but normally may only be unset when the system is in single-user mode. (See init(8) for details.)

RETURN VALUES

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

ERRORS

Chflags will fail if:

ENOTDIR A component of the path prefix is not a

directory.

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

ENAMETOOLONG A component of a pathname exceeded 63 characters, or an entire path name exceeded 255 characters.

ENOENT The named file does not exist.

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

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

EPERM The effective user ID does not match the owner of the file and the effective user ID is not the super-user.

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

EFAULT path points outside the process's allocated address space.

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

fchflags will fail if:

EBADF The descriptor is not valid.

EINVAL fd refers to a socket, not to a file.

EPERM The effective user ID does not match the owner of the file and the effective user ID is not the super-user.

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

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

SEE ALSO

chflags(1), init(8)

HISTORY

The chflags and fchflags functions first appeared in 4.4BSD.

NAME

chmod - change mode of file

SYNOPSIS

```
chmod(path, mode)
char *path;
int mode;

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

DESCRIPTION

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

```
   ISUID   04000   set user ID on execution
   ISGID   02000   set group ID on execution
   ISVTX   01000   `sticky bit' (see below)
   IREAD   00400   read by owner
   IWRITE  00200   write by owner
   IEXEC   00100   execute (search on directory) by owner
         00070   read, write, execute (search) by group
         00007   read, write, execute (search) by others
```

If an executable file is set up for sharing (this is the default) then mode ISVTX (the `sticky bit') prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Ability to set this bit on executable files is restricted to the super-user.

If mode ISVTX (the `sticky bit') is set on a directory, an unprivileged user may not delete or rename files of other users in that directory. For more details of the properties of the sticky bit, see sticky(8).

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

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

RETURN VALUE

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

ERRORS

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

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

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

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

[ENOENT] The named file does not exist.

[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 effective user ID does not match the owner of the file and the effective user ID is not the super-user.

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

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

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

Fchmod will fail if:

[EBADF] The descriptor is not valid.

[EINVAL] Fd refers to a socket, not to a file.

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

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

SEE ALSO

chmod(1), open(2), chown(2), stat(2), sticky(8)

NAME

chown - change owner

SYNOPSIS

/usr/sbin/chown [-f -R] owner[.group] file ...

DESCRIPTION

Chown changes the owner of the files to owner. The owner may be either a decimal UID or a login name found in the password file. An optional group may also be specified. The group may be either a decimal GID or a group name found in the group-ID file.

Only the super-user can change owner, in order to simplify accounting procedures. No errors are reported when the -f (force) option is given.

When the -R option is given, chown recursively descends its directory arguments setting the specified owner. When symbolic links are encountered, their ownership is changed, but they are not traversed.

FILES

/etc/passwd

SEE ALSO

chgrp(1), chown(2), passwd(5), group(5)

[ENOENT] The named file does not exist.

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

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

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

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

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

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

Fchown will fail if:

[EBADF] Fd does not refer to a valid descriptor.

[EINVAL] Fd refers to a socket, not a file.

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

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

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

SEE ALSO

chown(8), chgrp(1), chmod(2), flock(2)

NAME

chroot - change root directory

SYNOPSIS

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

DESCRIPTION

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

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

This call is restricted to the super-user.

RETURN VALUE

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

ERRORS

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

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

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

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

[ENOENT] The named directory does not exist.

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

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

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

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

SEE ALSO
chdir(2)

NAME

close - delete a descriptor

SYNOPSIS

```
close(d)
int d;
```

DESCRIPTION

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

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

When a process forks (see fork(2)), all descriptors for the new child process reference the same objects as they did in the parent before the fork. If a new process is then to be run using execve(2), the process would normally inherit these descriptors. Most of the descriptors can be rearranged with dup2(2) or deleted with close before the execve is attempted, but if some of these descriptors will still be needed if the execve fails, it is necessary to arrange for them to be closed if the execve succeeds. For this reason, the call ``fcntl(d, F_SETFD, 1)'' is provided, which arranges that a descriptor will be closed after a successful execve; the call ``fcntl(d, F_SETFD, 0)'' restores the default, which is to not close the descriptor.

RETURN VALUE

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

ERRORS

Close will fail if:

[EBADF] D is not an active descriptor.

SEE ALSO

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

NAME

connect - initiate a connection on a socket

SYNOPSIS

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

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

DESCRIPTION

The parameter s is a socket. If it is of type SOCK_DGRAM, then this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, then 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. Generally, stream sockets may successfully connect only once; datagram sockets may use connect multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUE

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.

ERRORS

The call fails if:

[EBADF] S is not a valid descriptor.

[ENOTSOCK] 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.

[EISCONN] The socket is already connected.

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

[ECONNREFUSED] The attempt to connect was forcefully rejected.

[ENETUNREACH] The network isn't reachable from this host.

[EADDRINUSE] The address is already in use.

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

[EINPROGRESS] The socket is non-blocking and the connection cannot be completed immediately. It is possible to select(2) for completion by selecting the socket for writing.

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

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.

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

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

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

[ENOENT] The named socket does not exist.

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

[EACCES] Write access to the named socket is denied.

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

SEE ALSO

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

NAME

dup, dup2 - duplicate a descriptor

SYNOPSIS

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

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

DESCRIPTION

Dup 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(2). The new descriptor returned by the call, newd, is the lowest numbered descriptor that is not currently in use by the process.

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

In the second form of the call, the value of newd desired is specified. If this descriptor is already in use, the descriptor is first deallocated as if a close(2) call had been done first.

RETURN VALUE

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

ERRORS

Dup and dup2 fail if:

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

[EMFILE] Too many descriptors are active.

SEE ALSO

accept(2), open(2), close(2), fcntl(2), pipe(2), socket(2), socketpair(2), getdtablesize(2)

NAME

execve - execute a file

SYNOPSIS

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

DESCRIPTION

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

An interpreter file begins with a line of the form ``#! interpreter''. When an interpreter file is execve'd, the system execve's the specified interpreter, giving it the name of the originally exec'd file as an argument and shifting over the rest of the original arguments.

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

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

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

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

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

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

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

process ID	see getpid(2)
parent process ID	see getppid(2)
process group ID	see getpgrp(2)
access groups	see getgroups(2)
working directory	see chdir(2)
root directory	see chroot(2)
control terminal	see tty(4)
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), sigmask(2)

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

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

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

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

RETURN VALUE

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

ERRORS

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

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

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

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

[ENOENT] The new process file does not exist.

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

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

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

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

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

[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 (getrlimit(2)).

[E2BIG] The number of bytes in the new process's argument list is larger than the system-imposed limit. The limit in the system as released is 20480 bytes (NCARGS in <sys/param.h>.

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

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

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

CAVEATS

If a program is setuid to a non-super-user, but is executed

when the real uid is ``root'', then the program has some of the powers of a super-user as well.

SEE ALSO

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

NAME

_exit - terminate a process

SYNOPSIS

```
_exit(status)
int status;
```

DESCRIPTION

_exit terminates a process with the following consequences:

All of the descriptors open in the calling process are closed. This may entail delays, for example, waiting for output to drain; a process in this state may not be killed, as it is already dying.

If the parent process of the calling process is executing a wait or is interested in the SIGCHLD signal, then it is notified of the calling process's termination and the low-order eight bits of status are made available to it; see wait(2).

The parent process ID of all of the calling process's existing child processes are also set to 1. This means that the initialization process (see intro(2)) inherits each of these processes as well. Any stopped children are restarted with a hangup signal (SIGHUP).

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

RETURN VALUE

This call never returns.

SEE ALSO

fork(2), sigvec(2), wait(2), exit(3)

NAME

fcntl - file control

SYNOPSIS

```
#include <fcntl.h>

res = fcntl(fd, cmd, arg)
int res;
int fd, cmd, arg;
```

DESCRIPTION

Fcntl provides for control over descriptors. The argument fd is a descriptor to be operated on by cmd as follows:

F_DUPFD Return a new descriptor as follows:

Lowest numbered available descriptor greater than or equal to arg.

Same object references as the original descriptor.

New descriptor shares the same file pointer if the object was a file.

Same access mode (read, write or read/write).

Same file status flags (i.e., both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across execv(2) system calls.

F_GETFD Get the close-on-exec flag associated with the file descriptor fd. If the low-order bit is 0, the file will remain open across exec, otherwise the file will be closed upon execution of exec.

F_SETFD Set the close-on-exec flag associated with fd to the low order bit of arg (0 or 1 as above).

F_GETFL Get descriptor status flags, as described below.

F_SETFL Set descriptor status flags.

F_GETOWN Get the process ID or process group currently receiving SIGIO and SIGURG signals; process

groups are returned as negative values.

F_SETOWN Set 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.

The flags for the F_GETFL and F_SETFL flags are as follows:

O_NONBLOCK Non-blocking 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.

O_APPEND Force each write to append at the end of file; corresponds to the O_APPEND flag of open(2).

O_ASYNC Enable the SIGIO signal to be sent to the process group when I/O is possible, e.g., upon availability of data to be read.

RETURN VALUE

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

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_GETFL	Value of flags.
F_GETOWN	Value of file descriptor owner.
other	Value other than -1.

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

ERRORS

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

[EBADF] Fildes is not a valid open file descriptor.

[EMFILE] Cmd is F_DUPFD and the maximum allowed number of file descriptors are currently open.

[EINVAL] Cmd is F_DUPFD and arg is negative or greater than the maximum allowable number (see getdtablesize(2)).

[ESRCH] Cmd is F_SETOWN and the process ID given as argument is not in use.

SEE ALSO

close(2), execve(2), getdtablesize(2), open(2), sigvec(2)

BUGS

The asynchronous I/O facilities of O_NONBLOCK and O_ASYNC are currently available only for tty and socket operations.

NAME

fetchi - fetch from user instruction space (2BSD)

SYNOPSIS

```
fetchi(addr)
int *addr;
```

DESCRIPTION

Fetchi fetches the word at addr from the caller's instruction space. This system call is required on PDP-11's with separate instruction and data spaces because the mfpi instruction reads from D-space if the current and previous modes in the program status word are both user.

RETURN VALUE

Upon successful completion the contents of the caller's instruction space at addr are returned. Otherwise, a value of -1 is returned.

ERRORS

[EINVAL] The kernel has not been compiled for a processor with separate I/D.

[EFAULT] Addr points to an address not in the process's allocated instruction space.

BUGS

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

Fetchi is a kludge and exists only to circumvent an alleged security feature on some DEC PDP-11 processors.

Fetchi is unique to the PDP-11 and 2BSD; its use is discouraged.

NAME

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

SYNOPSIS

```
#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

Flock applies or removes an advisory lock on the file associated with the file descriptor fd. A lock is applied by specifying an operation parameter that is 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 (i.e., processes may 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 may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock may be upgraded to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; this results in the previous lock being released and the new lock applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object that is already locked normally causes the caller to be blocked until the lock may be acquired. If LOCK_NB is included in operation, then this will not happen; instead the call will fail and the error EWOULDBLOCK will be returned.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through dup(2) or fork(2) 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 will lose 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 left in the global location errno.

ERRORS

The flock call fails if:

[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.

SEE ALSO

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

NAME

fork - create a new process

SYNOPSIS

```
pid = fork()
int pid;
```

DESCRIPTION

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

The child process has a unique process ID.

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

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

The child processes resource utilizations are set to 0; 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.

ERRORS

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

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent.

[EAGAIN] The system-imposed limit MAXUPRC (<sys/param.h>) 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

fperr - get floating-point error registers (2BSD)

SYNOPSIS

```
#include <pdp/fperr.h>

struct fperr
{
   short     f_fec;
   caddr_t   f_fea;
};

fperr(fpe)
struct fperr *fpe;
```

DESCRIPTION

Fperr returns the contents of the floating-point processor's error registers as they were following the last floating exception generated by the calling process. The registers are stored in the structure pointed to by fpe.

This call is required because the error registers in the PDP-11 floating-point processor are read-only. Thus, they may be changed by some other process between the time that the current process generates an exception and the time that it reads the registers. Therefore, the system saves their state at the time of an exception.

The values returned are valid only after a floating-point exception.

ERRORS

[EINVAL] The kernel has not been compiled for a processor with floating point.

SEE ALSO

Ed Gould, Jim Reeds, Vance Vaughan, UNIX Problems with Floating Point Processors

BUGS

Fperr is unique to the PDP-11 and 2BSD; its use is discouraged.

NAME

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

SYNOPSIS

```
fsync(fd)
int fd;
```

DESCRIPTION

Fsync causes all modified data and attributes of fd to be moved to a permanent storage device. This normally results in all in-core modified copies of buffers for the associated file to be written to a disk.

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

RETURN VALUE

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

ERRORS

The fsync fails if:

[EBADF] Fd is not a valid descriptor.

[EINVAL] Fd refers to a socket, not to a file.

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

SEE ALSO

sync(2), sync(8), update(8)

NAME

getdtablesize - get descriptor table size

SYNOPSIS

```
nfds = getdtablesize()
int nfds;
```

DESCRIPTION

Each process has a fixed size descriptor table, which 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), select(2)

NAME

getfsstat - get list of all mounted filesystems

SYNOPSIS

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

int
getfsstat(buf,bufsize,flags)
struct statfs *buf;
int bufsize;
int flags;
```

DESCRIPTION

Getfsstat() returns information about all mounted filesystems. Buf is a pointer to statfs structures defined as follows:

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

struct statfs {
short   f_type;          /* type of filesystem (see below) */
short   f_flags;         /* copy of mount flags */
short   f_bsize;         /* fundamental file system block size */
short   f_iosize;        /* optimal transfer block size */
long    f_blocks;        /* total data blocks in file system */
long    f_bfree;         /* free blocks in fs */
long    f_bavail;        /* free blocks avail to non-superuser */
ino_t   f_files;         /* total file nodes in file system */
ino_t   f_ffree;         /* free file nodes in fs */
u_long  f_fsid[2];        /* file system id */
long    f_spare[4];       /* spare for later */
char    f_mntonname[MNAMELEN];    /* mount point */
char    f_mntfromname[MNAMELEN];  /* mounted filesystem */
};
/*
* File system types. - Only UFS is supported so the other types are not
* given.
*/
#define   MOUNT_NONE     0
#define   MOUNT_UFS 1    /* Fast Filesystem */
```

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

If buf is given as NULL, getfsstat() returns just the number of mounted filesystems.

Normally flags is currently unused. In 4.4BSD systems the usage is specified as MNT_WAIT. If flags is set to MNT_NOWAIT, getfsstat() will return the information it has available without requesting an update from each filesystem. Thus, some of the information will be out of date, but getfsstat() will not block waiting for information from a filesystem that is unable to respond.

RETURN VALUES

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

ERRORS

Getfsstat() fails if one or more of the following are true:

[EFAULT] Buf points to an invalid address.

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

SEE ALSO

statfs(2), fstab(5), mount(8)

HISTORY

The getfsstat function first appeared in 4.4BSD.

NAME

getgid, getegid - get group process identification

SYNOPSIS

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

gid_t
getgid()

gid_t
getegid()
```

DESCRIPTION

The getgid function returns the real group ID of the calling process, getegid returns the effective group ID of the calling process.

The real group ID is specified at login time.

The real group ID is the group of the user who invoked the program. As the effective group ID gives the process additional permissions during the execution of ``set-group-ID'' mode processes, getgid is used to determine the real-user-id of the calling process.

ERRORS

The getgid and getegid functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

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

STANDARDS

Getgid and getegid conform to IEEE Std 1003.1-1988 (``POSIX'').

NAME

getgroups - get group access list

SYNOPSIS

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

ngroups = getgroups(gidsetlen, gidset)
int ngroups, gidsetlen, *gidset;
```

DESCRIPTION

Getgroups gets the current group access list of the user process and stores it in the array gidset. The parameter gidsetlen indicates the number of entries that may be placed in gidset. Getgroups returns the actual number of groups returned in gidset. No more than NGROUPS, as defined in <sys/param.h>, will ever be returned.

RETURN VALUE

A successful call returns the number of groups in the group set. A value of -1 indicates that an error occurred, and the error code is stored in the global variable errno.

ERRORS

The possible errors for getgroup are:

[EINVAL] The argument gidsetlen is smaller than the number of groups in the group set.

[EFAULT] The argument gidset specifies an invalid address.

SEE ALSO

setgroups(2), initgroups(3X)

BUGS

The gidset array should be of type gid_t, but remains integer for compatibility with earlier systems.

NAME

gethostid, sethostid - get/set unique identifier of current host

SYNOPSIS

```
hostid = gethostid()
long hostid;

sethostid(hostid)
long hostid;
```

DESCRIPTION

Sethostid 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 super-user and is normally performed at boot time.

Gethostid returns the 32-bit identifier for the current processor.

SEE ALSO

hostid(1), gethostname(2)

BUGS

32 bits for the identifier is too small.

NAME

gethostname, sethostname - get/set name of current host

SYNOPSIS

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

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

DESCRIPTION

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

Sethostname sets the name of the host machine to be name, which has length namelen. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

RETURN VALUE

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

ERRORS

The following errors may be returned by these calls:

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

[EPERM] The caller tried to set the hostname and was not the super-user.

SEE ALSO

gethostid(2)

BUGS

Host names are limited to MAXHOSTNAMELEN (from <sys/param.h>) characters, currently 64.

NAME

getitimer, setitimer - get/set value of interval timer

SYNOPSIS

```
#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 in the structure at value. The setitimer call sets a timer to the specified value (returning the previous value of the timer if ovalue is nonzero).

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 non-zero, it indicates the time to the next timer expiration. If it_interval is non-zero, 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 non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution (on the 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.

NOTES

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

NOTES (PDP-11)

On the PDP-11, setitimer rounds timer values up to seconds resolution. (This saves some space and computation in the overburdened PDP-11 kernel.)

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.

ERRORS

The possible errors are:

[EFAULT] The value parameter specified a bad address.

[EINVAL] A value parameter specified a time was too large to be handled.

SEE ALSO

sigvec(2), gettimeofday(2)

NAME

getlogin, setlogin - get/set login name

SYNOPSIS

```
#include <unistd.h>

char *
getlogin()

int
setlogin(name)
   char *name;
```

DESCRIPTION

The getlogin routine returns the login name of the user associated with the current session, as previously set by setlogin. The name is normally associated with a login shell at the time a session is created, and is inherited by all processes descended from the login shell. (This is true even if some of those processes assume another user ID, for example when su(1) is used.)

Setlogin sets the login name of the user associated with the current session to name. This call is restricted to the super-user, and is normally used only when a new session is being created on behalf of the named user (for example, at login time, or when a remote shell is invoked).

RETURN VALUES

If a call to getlogin succeeds, it returns a pointer to a null-terminated string in a static buffer. If the name has not been set, it returns NULL. If a call to setlogin succeeds, a value of 0 is returned. If setlogin fails, a value of -1 is returned and an error code is placed in the global location errno.

ERRORS

The following errors may be returned by these calls:

EFAULT The name parameter gave an invalid address.

EINVAL The name parameter pointed to a string that was too long. Login names are limited to MAXLOGNAME (from <sys/param.h>) characters, currently 16.

EPERM The caller tried to set the login name and was not the super-user.

SEE ALSO

setsid(2)

BUGS

Login names are limited in length by setlogin. However, lower limits are placed on login names elsewhere in the system (UT_NAMESIZE in <utmp.h>).

In earlier versions of the system, getlogin failed unless the process was associated with a login terminal. The current implementation (using setlogin) allows getlogin to succeed even when the process has no controlling terminal. In earlier versions of the system, the value returned by getlogin could not be trusted without checking the user ID. Portable programs should probably still make this check.

HISTORY

The setlogin function first appeared in 4.4BSD. The getlogin function was present in V7.

NAME

getpagesize - get system page size

SYNOPSIS

```
pagesize = getpagesize()
int pagesize;
```

DESCRIPTION

Getpagesize 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

sbrk(2), pagesize(1)

NAME

getpeername - get name of connected peer

SYNOPSIS

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

DESCRIPTION

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 of the name returned (in bytes). The name is truncated if the buffer provided is too small.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

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

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

NAME

getpgrp - get process group

SYNOPSIS

```
pgrp = getpgrp(pid)
int pgrp;
int pid;
```

DESCRIPTION

The process group of the specified process is returned by getpgrp. If pid is zero, then 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 foreground and may read, while others will block with a signal if they attempt to read.

This call is thus 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/set the process group of the control terminal.

SEE ALSO

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

NAME

getpid, getppid - get process identification

SYNOPSIS

```
pid = getpid()
int pid;

ppid = getppid()
int ppid;
```

DESCRIPTION

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

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

SEE ALSO

gethostid(2)

NAME

getpriority, setpriority - get/set program scheduling priority

SYNOPSIS

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

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. 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). A zero value of who denotes the current process, process group, or user. 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 super-user 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.

ERRORS

Getpriority and setpriority may return one of the following errors:

[ESRCH] No process was located using the which and who values specified.

[EINVAL] Which was not one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.

In addition to the errors indicated above, setpriority may fail with one of the following errors returned:

[EPERM] A process was located, but neither its

effective nor real user ID matched the effective user ID of the caller.

[EACCES] A non super-user attempted to lower a process priority.

SEE ALSO

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

NAME

getrlimit, setrlimit - control maximum system resource consumption

SYNOPSIS

```
#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 may 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 seconds) 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 defines how far a program may extend its break with the sbrk(2) system call.

RLIMIT_STACK the maximum size, in bytes, of the stack segment for a process; this defines how far a program's stack segment may be extended. Stack extension is performed automatically by the system.

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. This imposes a limit on the amount of physical memory to be given to a process; if memory is tight, the system will prefer to take memory from processes that are exceeding their declared resident set size.

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 rlimit structure is used to specify the hard and soft limits on a resource,

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

Only the super-user may raise the maximum limits. Other users may only 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(1).

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. When the stack limit is reached, the process receives a segmentation fault (SIGSEGV); if this signal is not caught by a handler using the signal stack, this signal will kill the process.

A file I/O operation that would create a file that is too large will cause a signal SIGXFSZ to be generated; this normally terminates the process, but may be caught. When the soft cpu time limit is exceeded, a signal SIGXCPU is sent to the offending 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.

ERRORS

The possible errors are:

[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 super-user.

SEE ALSO

csh(1), quota(2), sigvec(2), sigstack(2)

BUGS

There should be limit and unlimit commands in sh(1) as well as in csh.

NAME

getrusage - get information about resource utilization

SYNOPSIS

```
#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

Getrusage returns information describing the resources utilized by the current process, or all its terminated child processes. The who parameter is one of RUSAGE_SELF or RUSAGE_CHILDREN. The buffer to which rusage points will be filled in with the following structure:

```
   struct  rusage {
         struct timeval ru_utime;     /* user time used */
         struct timeval ru_stime;     /* system time used */
         long       ru_maxrss;
         long       ru_ixrss;         /* integral shared text memory size
*/
         long       ru_idrss;         /* integral unshared data size */
         long       ru_isrss;         /* integral unshared stack size */
         long       ru_minflt;             /* page reclaims */
         long       ru_majflt;             /* page faults */
         long       ru_ovly;          /* overlay changes */
         long       ru_nswap;         /* swaps */
         long       ru_inblock;            /* block input operations */
         long       ru_oublock;            /* block output operations */
         long       ru_msgsnd;             /* messages sent */
         long       ru_msgrcv;             /* messages received */
         long       ru_nsignals;           /* signals received */
         long       ru_nvcsw;         /* voluntary context switches */
         long       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 process(es).

ru_maxrss — the maximum resident set size utilized (in kilobytes).

ru_ixrss an "integral" value indicating the amount of memory used by the text segment 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_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.

the number of times a process requested a text overlay switch - only available under 2_10BSD.

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_outblock 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.

NOTES

The numbers ru_inblock and ru_outblock account only for real I/O; data supplied by the caching mechanism is charged only to the first process to read or write the data.

ERRORS

The possible errors for getrusage are:

[EINVAL] The who parameter is not a valid value.

[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)

BUGS

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

NAME

getsockname - get socket name

SYNOPSIS

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

DESCRIPTION

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

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

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)

BUGS

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

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

```
#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

Getsockopt and setsockopt manipulate options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost ``socket'' level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the ``socket'' level, level is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, level should be set to the protocol number of TCP; 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 value for the requested option(s) are to be returned. For getsockopt, optlen is a value-result parameter, initially containing the size of the buffer pointed to by optval, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, optval may be supplied as 0.

Optname 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, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (4P).

Most socket-level options take an int parameter for optval. For setsockopt, the parameter should non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a struct linger parameter, defined in

<sys/socket.h>, which specifies the desired state of the option and the linger interval (see below).

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

```
SO_DEBUG    toggle recording of debugging information
SO_REUSEADDR     toggle local address reuse
SO_KEEPALIVE     toggle keep connections alive
SO_DONTROUTE     toggle routing bypass for outgoing messages
SO_LINGER  linger on close if data present
SO_BROADCAST     toggle permission to transmit broadcast messages
SO_OOBINLINE     toggle reception of out-of-band data in band
SO_SNDBUF  set buffer size for output
SO_RCVBUF  set buffer size for input
SO_TYPE    get the type of the socket (get only)
SO_ERROR   get and clear error on the socket (get only)
```

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a bind(2) 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 via 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 action taken when unsent messags are queued on socket and a close(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block 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 the setsockopt call when SO_LINGER is requested). If SO_LINGER is disabled and a close is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv or read calls without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes

allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values. Finally, SO_TYPE and SO_ERROR are options used only with setsockopt. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUE

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

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

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

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

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

SEE ALSO

ioctl(2), socket(2), getprotoent(3N)

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

gettimeofday, settimeofday - get/set date and time

SYNOPSIS

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

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

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

DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the gettimeofday call, and set with the settimeofday call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent, and the time may be updated continuously or in ``ticks.'' If tzp is zero, the time zone information will not be returned or set.

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

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

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

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

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

RETURN

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.

ERRORS

The following error codes may be set in errno:

[EFAULT] An argument address referenced invalid memory.

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

SEE ALSO

date(1), adjtime(2), ctime(3), timed(8)

NAME

getuid, geteuid - get user identification

SYNOPSIS

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

uid_t
getuid()

uid_t
geteuid()
```

DESCRIPTION

The getuid function returns the real user ID of the calling process. The geteuid function returns the effective user ID of the calling process.

The real user ID is that of the user who has invoked the program. As the effective user ID gives the process additional permissions during execution of ``set-user-ID'' mode processes, getuid is used to determine the real-user-id of the calling process.

ERRORS

The getuid and geteuid functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

getgid(2), setreuid(2)

STANDARDS

Geteuid and getuid conform to IEEE Std 1003.1-1988 (``POSIX'').

NAME

getuid, getgid - get user or group ID of the caller

SYNOPSIS

```
integer function getuid()

integer function getgid()
```

DESCRIPTION

These functions return the real user or group ID of the user of the process.

FILES

/usr/lib/libU77.a

SEE ALSO

getuid(2)

NAME

ioctl - control device

SYNOPSIS

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

ioctl(d, request, argp)
int d;
unsigned long request;
char *argp;
```

DESCRIPTION

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

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

RETURN VALUE

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

ERRORS

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

[EBADF] D is not a valid descriptor.

[ENOTTY] D is not associated with a character special device.

[ENOTTY] The specified request does not apply to the kind of object that the descriptor d references.

[EINVAL] Request or argp is not valid.

SEE ALSO

execve(2), fcntl(2), mt(4), tty(4), intro(4N)

NAME

kill - send signal to a process

SYNOPSIS

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

DESCRIPTION

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

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user. A single exception is the signal SIGCONT, which may always be sent to any descendant of the current process.

If the process number is 0, the signal is sent to all processes in the sender's process group; this is a variant of killpg(2).

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

For compatibility with System V, 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. This is a variant of killpg(2).

Processes may send signals to themselves.

RETURN VALUE

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

ERRORS

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

[EINVAL] Sig is not a valid signal number.

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

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

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

SEE ALSO

getpid(2), getpgrp(2), killpg(2), sigvec(2)

NAME

killpg - send signal to a process group

SYNOPSIS

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

DESCRIPTION

Killpg 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, or the sender must be the super-user. As a single special case the continue signal SIGCONT may be sent to any process that 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.

ERRORS

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

[EINVAL] Sig is not a valid signal number.

[ESRCH] No process can be found in the process group specified by pgrp.

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

[EPERM] The sending process is not the super-user and one or more of the target processes has an effective user ID different from that of the sending process.

SEE ALSO

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

NAME

link - make a hard link to a file

SYNOPSIS

```
link(name1, name2)
char *name1, *name2;
```

DESCRIPTION

A hard link to name1 is created; the link has the name name2. Name1 must exist.

With hard links, both name1 and name2 must be in the same file system. Unless the caller is the super-user, name1 must not be a directory. Both the old and the new link share equal access and rights to the underlying object.

RETURN VALUE

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

ERRORS

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

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

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

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

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

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

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

[ELOOP] Too many symbolic links were encountered in translating one of the pathnames.

[ENOENT] The file named by name1 does not exist.

[EEXIST] The link named by name2 does exist.

[EPERM] The file named by name1 is a directory and the effective user ID is not super-user.

[EXDEV] The link named by name2 and the file named by name1 are on different file systems.

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

[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.

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

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

SEE ALSO

symlink(2), unlink(2)

NAME

listen - listen for connections on a socket

SYNOPSIS

```
listen(s, backlog)
int s, backlog;
```

DESCRIPTION

To accept connections, a socket is first created with socket(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with listen(2), and then the connections are accepted with accept(2). The listen call applies only to 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 may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

RETURN VALUE

A 0 return value indicates success; -1 indicates an error.

ERRORS

The call fails if:

[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)

BUGS

The backlog is currently limited (silently) to 5.

NAME

lock - lock a process in primary memory (2BSD)

SYNOPSIS

```
lock(flag)
int flag
```

DESCRIPTION

If the flag argument is non-zero, the process executing this call will not be swapped unless it is required to grow. If the argument is zero, the process is unlocked. This call may only be executed by the super-user.

ERRORS

[EPERM] The caller is not the super-user.

BUGS

Locked processes interfere with the compaction of primary memory and can cause deadlock. This system call is not considered a permanent part of the system.

Lock is unique to the PDP-11 and 2BSD; its use is discouraged.

NAME

lseek - move read/write pointer

SYNOPSIS

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

#define L_SET   0 /* set the seek pointer */
#define L_INCR  1 /* increment the seek pointer */
#define L_XTND  2 /* extend the file size */

pos = lseek(d, offset, whence)
off_t pos;
int d;
off_t offset;
int whence;
```

DESCRIPTION

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

If whence is L_SET, the pointer is set to offset bytes.

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

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

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

NOTES

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

RETURN VALUE

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

ERRORS

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

[EBADF] Fildes is not an open file descriptor.

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

[EINVAL] Whence is not a proper value.

SEE ALSO

dup(2), open(2)

BUGS

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

NAME

mkdir - make a directory file

SYNOPSIS

```
mkdir(path, mode)
char *path;
int mode;
```

DESCRIPTION

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

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

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

RETURN VALUE

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

ERRORS

Mkdir will fail and no directory will be created if:

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

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

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

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

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

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

[EPERM] The path argument contains a byte with the high-order bit set.

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

system.

[EEXIST] The named file exists.

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

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

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

[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.

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

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

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

SEE ALSO

chmod(2), stat(2), umask(2)

NAME

mknod - make a special file

SYNOPSIS

```
mknod(path, mode, dev)
char *path;
int mode, dev;
```

DESCRIPTION

Mknod creates a new file whose name is path. The mode of the new file (including special file bits) is initialized from mode. (The protection part of the mode is modified by the process's mode mask (see umask(2))). The first block pointer of the i-node is initialized from dev and is used to specify which device the special file refers to.

If mode indicates a block or character special file, dev is 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.

Mknod may be invoked only by the super-user.

RETURN VALUE

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

ERRORS

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

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

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

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

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

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

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

[EPERM] The process's effective user ID is not super-user.

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

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

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

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

[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.

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

[EEXIST] The named file exists.

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

SEE ALSO

chmod(2), stat(2), umask(2)

NAME

mount, umount - mount or remove file system

SYNOPSIS

```
mount(special, name, flags)
char *special, *name;
int flags;

umount(special)
char *special;
```

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file special; from now on, references to file name will refer to the root file on the newly mounted file system. Special and name are pointers to null-terminated strings containing the appropriate path names.

Name must exist already. Name must be a directory. Its old contents are inaccessible while the file system is mounted.

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

MNT_RDONLY The filesystem should be treated as read-only; Even the super-user may not write on it.

MNT_NOEXEC Do not allow files to be executed from the filesystem.

MNT_NOSUID Do not honor setuid or setgid bits on files when executing them.

MNT_NODEV Do not interpret special files on the filesystem.

MNT_SYNCHRONOUS All I/O to the filesystem should be done synchronously.

Umount announces to the system that the special file is no longer to contain a removable file system. The associated file reverts to its ordinary interpretation.

RETURN VALUE

Mount returns 0 if the action occurred, -1 if special is inaccessible or not an appropriate file, if name does not exist, if special is already mounted, if name is in use, or if there are already too many file systems mounted.

Umount returns 0 if the action occurred; -1 if if the special file is inaccessible or does not have a mounted file system, or if there are active files in the mounted file system.

ERRORS

Mount will fail when one of the following occurs:

[ENAMETOOLONG] A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

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

[EPERM] The caller is not the super-user.

[ENOENT] A component of name does not exist.

[ENODEV] A component of special does not exist.

[ENOTBLK] 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 name is not a directory, or a path prefix of special is not a directory.

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

[EINVAL] The super block for the file system had a bad magic number or an out of range block size.

[EBUSY] Another process currently holds a reference to name, or special is already mounted.

[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] Special or name points outside the process's allocated address space.

Umount may fail with one of the following errors:

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

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

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

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

[EPERM] The caller is not the super-user.

[ENODEV] Special does not exist.

[ENOTBLK] 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).

[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.

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

SEE ALSO

mount(8), umount(8)

BUGS

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

Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

MNT_SYNCHRONOUS is not currently implemented in the kernel but may be specified because the kernel ignores it.

NAME

nostk - allow process to manage its own stack (2BSD)

SYNOPSYS

```
nostk();
```

DESCRIPTION

Nostk informs the system that the process wishes to manage its own stack. The system releases the stack segment(s) it has reserved, making them available for allocation (via brk(2)) by the user.

C programs should use nostk only with great caution and understanding of the C language calling and stack conventions. It is most useful for assembler programs that want to use the entire available address space.

SEE ALSO

stack(5)

BUGS

Nostk is unique to the PDP-11 and 2BSD; its use is discouraged.

NAME

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

SYNOPSIS

```
#include <fcntl.h>

open(path, flags, mode)
char *path;
int flags, mode;
```

DESCRIPTION

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

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

```
O_RDONLY open for reading only
O_WRONLY open for writing only
O_RDWR   open for reading and writing
O_NONBLOCK     do not block on open
O_APPEND append on each write
O_CREAT  create file if it does not exist
O_TRUNC  truncate size to 0
O_EXCL   error if create and file exists
O_NOCTTY do not acquire as controlling terminal
O_SHLOCK atomically obtain a shared lock
O_EXLOCK atomically obtain an exclusive lock
```

Opening a file with O_APPEND set causes each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, the file is truncated to zero length. If O_EXCL is set with O_CREAT, then if the file already exists, the open returns an error. This can be used to implement a simple exclusive access locking mechanism. If O_EXCL is set and the last component of the pathname is a symbolic link, the open will fail even if the symbolic link points to a non-existent name. If the O_NONBLOCK flag is specified and the open call would result in the process being blocked for some reason (e.g. waiting for carrier on a dialup line), the open returns immediately. The first time the process attempts to perform i/o on the open file it will block.

The flag O_NOCTTY indicates that even if the file is a terminal device, the call should not result in acquiring the terminal device as the controlling terminal of the caller. This flag is not the default and is currently unimplemented (it will be Real Soon Now).

When opening a file, a lock with flock(2) semantics can be obtained by setting O_SHLOCK for a shared lock, or O_EXLOCK for an exclusive lock. If creating a file with O_CREAT, the request for the lock will never fail.

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

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

The system imposes a limit on the number of file descriptors open simultaneously by one process. Getdtablesize(2) returns the current system limit.

ERRORS

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

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

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

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

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

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

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

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

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

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

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

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

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

[ENFILE] The system file table is full.

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

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

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

[EDQUOT] O_CREAT is specified, the file does not exist, and the directory in which the entry for the new fie 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.

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

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and the open call requests write access.

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

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

[EOPNOTSUPP] An attempt was made to open a socket (not currently implemented).

SEE ALSO

chmod(2), close(2), dup(2), getdtablesize(2), lseek(2), read(2), write(2), umask(2)

NAME

phys - allow a process to access physical addresses (2BSD)

SYNOPSIS

```
phys(segreg, size, physaddr)
unsigned int segreg, size, physaddr;
```

DESCRIPTION

The argument segreg specifies a process virtual (data-space) address range of 8K bytes starting at virtual address segregx8K bytes. This address range is mapped into physical address physaddrx64 bytes. Only the first sizex64 bytes of this mapping is addressable. If size is zero, any previous mapping of this virtual address range is nullified. For example, the call

```
phys(7, 1, 0177775);
```

will map virtual addresses 0160000-0160077 into physical addresses 017777500-017777577. In particular, virtual address 0160060 is the PDP-11 console located at physical address 017777560.

This call may only be executed by the super-user.

ERRORS

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

[EINVAL] Segreg is less than 0 or greater than 7.

[EINVAL] Size is less than 0 or greater than 128.

SEE ALSO

PDP-11 segmentation hardware

BUGS

On systems with ENABLE/34(tm) memory mapping boards, phys cannot be used to map in the I/O page.

This system call is very dangerous. It is not considered a permanent part of the system.

Phys is unique to the PDP-11 and 2BSD; its use is discouraged.

NAME

pipe - create an interprocess communication channel

SYNOPSIS

```
pipe(fildes)
int fildes[2];
```

DESCRIPTION

The pipe system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor fildes[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor fildes[0] will pick up the data.

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

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

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

Pipes are really a special case of the socketpair(2) call and, in fact, are implemented as such in the system.

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

RETURN VALUE

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

ERRORS

The pipe call will fail if:

[EMFILE] Too many descriptors are active.

[ENFILE] The system file table is full.

[EFAULT] The fildes buffer is in an invalid area of the process's address space.

SEE ALSO

sh(1), read(2), write(2), fork(2), socketpair(2)

BUGS

Should more than 4096 bytes be necessary in any pipe among a

loop of processes, deadlock will occur.

NAME

profil - execution time profile

SYNOPSIS

```
profil(buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;
```

DESCRIPTION

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 (VAX and TAHOE: 100 ticks/second = 10 milliseconds per tick; 60 ticks/second ~= 16 milliseconds per tick); offset is subtracted from it, and the result 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 16 bits of fraction: 0xffff gives a 1-1 mapping of pc's to words in buff; 0x7fff maps each pair of instruction words together.

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 child and parent both 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), prof(1), setitimer(2), monitor(3)

NAME

ptrace - process trace

SYNOPSIS

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

ptrace(request, pid, addr, data)
int request, pid, *addr, data;
```

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a request argument. Generally, pid is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt". See 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:

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

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

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

system.

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

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

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

PT_KILL
The traced process terminates.

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

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

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

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

RETURN VALUE

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

ERRORS

[EIO] The request code is invalid.

[ESRCH] The specified process does not exist.

[EIO] The given signal number is invalid.

[EIO] The specified address is out of bounds.

[EPERM] The specified process cannot be traced.

SEE ALSO

wait(2), sigvec(2), adb(1)

NOTES (PDP-11)

On the PDP-11 the PT_WRITE_U request may also write the child process's current overlay number in the system data area; the T-bit is used to single step the processor and just one instruction is executed for the PT_STEP request; a "word" means a 16-bit integer, and the "even" restriction does apply.

BUGS

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

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

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

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

NAME

quota - manipulate disk quotas

SYNOPSIS

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

quota(cmd, uid, arg, addr)
int cmd, uid, arg;
char *addr;
```

DESCRIPTION

The quota call manipulates disk quotas for file systems that have had quotas enabled with setquota(2). The cmd parameter indicates a command to be applied to the user ID uid. Arg is a command specific argument and addr is the address of an optional, command specific, data structure that is copied in or out of the system. The interpretation of arg and addr is given with each command below.

Q_SETDLIM
Set disc quota limits and current usage for the user with ID uid. Arg is a major-minor device indicating a particular file system. Addr is a pointer to a struct dqblk structure (defined in <sys/quota.h>). This call is restricted to the super-user.

Q_GETDLIM
Get disc quota limits and current usage for the user with ID uid. The remaining parameters are as for Q_SETDLIM.

Q_SETDUSE
Set disc usage limits for the user with ID uid. Arg is a major-minor device indicating a particular file system. Addr is a pointer to a struct dqusage structure (defined in <sys/quota.h>). This call is restricted to the super-user.

Q_SYNC
Update the on-disc copy of quota usages. Arg is a major-minor device indicating the file system to be sync'ed. If the arg parameter is specified as NODEV, all file systems that have disc quotas will be sync'ed. The uid 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. This call is restricted to the super-user.

Q_SETWARN
Alter the disc usage warning limits for the user with

ID uid. Arg is a major-minor device indicating a particular file system. Addr is a pointer to a struct dqwarn structure (defined in <sys/quota.h>). This call is restricted to the super-user.

Q_DOWARN
Warn the user with user ID uid about excessive disc usage. This call causes the system to check its current disc usage information and print a message on the terminal of the caller for each file system on which the user is over quota. If the user is under quota, his warning count is reset to MAX_*_WARN (defined in <sys/quota.h>). If the arg parameter is specified as NODEV, all file systems that have disc quotas will be checked. Otherwise, arg indicates a specific major-minor device to be checked. This call is restricted to the super-user.

RETURN VALUE

A successful call returns 0, otherwise the value -1 is returned and the global variable errno indicates the reason for the failure.

ERRORS

A quota call will fail when one of the following occurs:

[EINVAL] The kernel has not been compiled with the QUOTA option.

[EINVAL] Cmd is invalid.

[ESRCH] No disc quota is found for the indicated user.

[EPERM] The call is priviledged and the caller was not the super-user.

[ENODEV] The arg parameter is being interpreted as a major-minor device and it indicates an unmounted file system.

[EFAULT] An invalid addr 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), quotaon(8), quotacheck(8)

BUGS

There should be some way to integrate this call with the

resource limit interface provided by setrlimit(2) and getrlimit(2).

The Australian spelling of disk is used throughout the quota facilities in honor of the implementors.

NAME

read, readv - read input

SYNOPSIS

```
cc = read(d, buf, nbytes)
int cc, d;
char *buf;
unsigned short nbytes;

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

cc = readv(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

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

For readv, the iovec structure is defined as

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

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

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

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

Upon successful completion, read and readv return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

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

RETURN VALUE

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

ERRORS

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

[EBADF] D is not a valid file or socket descriptor open for reading.

[EFAULT] Buf points outside the allocated address space.

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

[EINTR] A read from a slow device was interrupted before any data arrived by the delivery of a signal.

[EINVAL] The pointer associated with d was negative.

[EWOULDBLOCK] The file was marked for non-blocking I/O, and no data were ready to be read.

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

[EINVAL] Iovcnt was less than or equal to 0, or greater than 16.

[EINVAL] The sum of the iov_len values in the iov array overflowed a short.

[EFAULT] Part of the iov points outside the process's allocated address space.

SEE ALSO

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

NAME

readlink - read value of a symbolic link

SYNOPSIS

```
cc = readlink(path, buf, bufsiz)
int cc;
char *path, *buf;
int bufsiz;
```

DESCRIPTION

Readlink places the contents of the symbolic link name 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.

ERRORS

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

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

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

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

[ENOENT] The named file does not exist.

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

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

[EINVAL] The named file is not a symbolic link.

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

[EFAULT] Buf extends outside the process's allocated address space.

SEE ALSO

stat(2), lstat(2), symlink(2)

NAME

reboot - reboot system or halt processor

SYNOPSIS

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

reboot(howto)
int howto;
```

DESCRIPTION

Reboot reboots the system, and is invoked automatically in the event of unrecoverable system failures. Howto is a mask of options passed to the bootstrap program. 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 (e.g. 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. An automatic consistency check of the disks is then normally performed.

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 super-user may reboot a machine.

RETURN VALUES

If successful, this call never returns. Otherwise, a -1 is returned and an error is returned in the global variable errno.

ERRORS

[EPERM] The caller is not the super-user.

SEE ALSO

crash(8), halt(8), init(8), reboot(8)

BUGS

The notion of ``console medium'', among other things, is specific to the VAX.

NAME

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

SYNOPSIS

```
#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

Recv, recvfrom, and recvmsg are used to receive messages from a socket.

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

If from is non-zero, the source address of the message is filled in. 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 may be discarded depending on the type of socket the message is received from (see socket(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see ioctl(2)) in which case a cc of -1 is returned with the external variable errno set to EWOULDBLOCK.

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

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

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

The recvmsg call uses a msghdr structure to minimize the number of 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     iovec *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 may 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). A buffer to receive any access rights sent along with the message is specified in msg_accrights, which has length msg_accrightslen. Access rights are currently limited to file descriptors, which each occupy the size of an int.

RETURN VALUE

These calls return the number of bytes received, or -1 if an error occurred.

ERRORS

The calls fail if:

[EBADF] The argument s is an invalid descriptor.

[ENOTSOCK] The argument s is not a socket.

[EWOULDBLOCK] The socket is marked non-blocking 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 non-existent or protected part of the process address space.

SEE ALSO

fcntl(2), read(2), send(2), select(2), getsockopt(2), socket(2)

NAME

rename - change the name of a file

SYNOPSIS

```
rename(from, to)
char *from, *to;
```

DESCRIPTION

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

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

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

CAVEAT

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

RETURN VALUE

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

ERRORS

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

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

[ENAMETOOLONG] A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

[ENOENT] A component of the from path does not exist, or a path prefix of FIto does not exist.

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

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

[EPERM] The directory containing from is marked sticky, and neither the containing directory nor from are owned by the effective user ID.

[EPERM] The to file exists, the directory containing to is marked sticky, and neither the containing directory nor to are owned by the effective user ID.

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

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

[ENOTDIR] From is a directory, but to is not a directory.

[EISDIR] To is a directory, but from is not a directory.

[EXDEV] The link named by to and the file named by from are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.

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

[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.

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

[EFAULT] Path points outside the process's allocated

address space.

[EINVAL] From is a parent directory of to, or an attempt is made to rename ``.'' or ``..''.

[ENOTEMPTY] To is a directory and is not empty.

SEE ALSO
open(2)

NAME

rmdir - remove a directory file

SYNOPSIS

```
rmdir(path)
char *path;
```

DESCRIPTION

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

RETURN VALUE

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

ERRORS

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

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

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

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

[ENOENT] The named directory does not exist.

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

[ENOTEMPTY] The named directory contains files other than ``.'' and ``..'' in it.

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

[EACCES] Write permission is denied on the directory containing the link to be removed.

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

[EBUSY] The directory to be removed is the mount point for a mounted file system.

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

[EROFS] The directory entry to be removed resides on a read-only file system.

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

SEE ALSO

mkdir(2), unlink(2)

NAME

pselect, select - synchronous I/O multiplexing

SYNOPSIS

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

nfound = pselect(nfds, readfds, writefds, exceptfds, timeout, sigmask);
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timespec *timeout;
sigset_t *sigmask;

nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

Pselect and select examine the I/O descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The two functions are identical except for the type and format of the timeout value, and the additonal sigmask parameter supplied to the pselect() call.

The first nfds descriptors are checked in each set; i.e. the descriptors from 0 through nfds-1 in the descriptor sets are examined. On return, select replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in nfound.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: FD_ZERO(&fdset) initializes a descriptor set fdset to the null set. FD_SET(fd, &fdset) includes a particular descriptor fd in fdset. FD_CLR(fd, &fdset) removes fd from fdset. FD_ISSET(fd, &fdset) is nonzero if fd is a member of fdset, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to FD_SETSIZE,

which is normally at least equal to the maximum number of descriptors supported by the system.

If timeout is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a zero pointer, select blocks indefinitely. To affect a poll, the timeout argument should be non-zero, pointing to a zero-valued timeval structure.

If the sigmask parameter to pselect() is not NULL, it points to a signal mask that replaces the previous signal mask for the process for the duration of the call, and the previous mask is restored upon return; see sigprocmask(3). This is normally used so that signals can be blocked while preparing for a call to pselect() and then atomically unblocking the signals while selecting.

Any of readfds, writefds, and exceptfds may be given as zero pointers if no descriptors are of interest.

RETURN VALUE

Select returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires then select returns 0. If select returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from select indicates:

[EBADF] One of the descriptor sets 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 invalid. One of its components is negative or too large.

SEE ALSO

accept(2), connect(2), read(2), write(2), recv(2), send(2), getdtablesize(2)

BUGS

Although the provision of getdtablesize(2) was intended to allow user programs to be written independent of the kernel limit on the number of open files, the dimension of a sufficiently large bit field for select remains a problem. The default size FD_SETSIZE (currently 256) is somewhat larger than the current kernel limit to the number of open files. However, in order to accommodate programs which might

potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of <sys/types.h>.

Select should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the select call.

In 2BSD the timeout is implemented in the kernel using the callout table. Since a callout structure only has a signed short to store the number of ticks till expiration the maximum value of a kernel timeout is 32767 ticks. In the US (60hz power) this gives a maximum timeout of approximately 9 minutes. In countries using 50hz power the maximum timeout is about 13 minutes.

struct timespec on a PDP-11 is silly since the hardware has nowhere near microsecond much less nanosecond clock resolution.

NAME

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

SYNOPSIS

```
#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

Send, sendto, and sendmsg are used to transmit a message to another socket. Send may be used only when the socket is in a connected state, while sendto and sendmsg may be used at any time.

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

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

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

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

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

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g. SOCK_STREAM); the

underlying protocol must also support "out-of-band" data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

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

RETURN VALUE

The call returns the number of characters sent, or -1 if an error occurred.

ERRORS

[EBADF] An invalid descriptor was specified.

[ENOTSOCK] The argument s is not a socket.

[EFAULT] An invalid user space address was specified for a parameter.

[EMSGSIZE] The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

[EWOULDBLOCK] The socket is marked non-blocking and the requested operation would block.

[ENOBUFS] The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS] The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

SEE ALSO

fcntl(2), recv(2), select(2), getsockopt(2), socket(2), write(2)

NAME

setgroups - set group access list

SYNOPSIS

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

setgroups(ngroups, gidset)
int ngroups, *gidset;
```

DESCRIPTION

Setgroups sets the group access list of the current user process according to the array gidset. The parameter ngroups indicates the number of entries in the array and must be no more than NGROUPS, as defined in <sys/param.h>.

Only the super-user may set new groups.

RETURN VALUE

A 0 value is returned on success, -1 on error, with a error code stored in errno.

ERRORS

The setgroups call will fail if:

[EPERM] The caller is not the super-user.

[EFAULT] The address specified for gidset is outside the process address space.

SEE ALSO

getgroups(2), initgroups(3X)

BUGS

The gidset array should be of type gid_t, but remains integer for compatibility with earlier systems.

NAME

setpgrp - set process group

SYNOPSIS

```
setpgrp(pid, pgrp)
int pid, pgrp;
```

DESCRIPTION

Setpgrp sets the process group of the specified process pid to the specified pgrp. If pid is zero, then the call applies to the current process.

If the invoker is not the super-user, then the affected process must have the same effective user-id as the invoker or be a descendant of the invoking process.

RETURN VALUE

Setpgrp returns when the operation was successful. If the request failed, -1 is returned and the global variable errno indicates the reason.

ERRORS

Setpgrp will fail and the process group will not be altered if one of the following occur:

[ESRCH] The requested process does not exist.

[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.

SEE ALSO

getpgrp(2)

NAME

setquota - enable/disable quotas on a file system

SYNOPSIS

```
setquota(special, file)
char *special, *file;
```

DESCRIPTION

Disc quotas are enabled or disabled with the setquota call. 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 0, then quotas are disabled on the file system. The quota file must exist; it is normally created with the quotacheck(8) program.

Only the super-user may turn quotas on or off.

SEE ALSO

quota(2), quotacheck(8), quotaon(8)

RETURN VALUE

A 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.

ERRORS

Setquota will fail when one of the following occurs:

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

[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.

[ENODEV] Special does not exist.

[ENOENT] File does not exist.

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

[EPERM] The caller is not the super-user.

[ENOTBLK] 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).

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

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

[EACCES] File resides on a file system different from special.

[EACCES] File is not a plain file.

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

[EFAULT] Special or path points outside the process's allocated address space.

BUGS

The error codes are in a state of disarray; too many errors appear to the caller as one value.

NAME

setregid - set real and effective group ID

SYNOPSIS

```
#include <unistd.h>

int
setregid(rgid, egid)
   gid_t rgid, egid
```

DESCRIPTION

The real and effective group ID's of the current process are set to the arguments. Unprivileged users may change the real group ID to the effective group ID and vice-versa; only the super-user may make other changes.

Supplying a value of -1 for either the real or effective group ID forces the system to substitute the current ID in place of the -1 parameter.

The setregid function was intended to allow swapping the real and effective group IDs in set-group-ID programs to temporarily relinquish the set-group-ID value. This function did not work correctly, and its purpose is now better served by the use of the setegid function (see setuid(2)).

When setting the real and effective group IDs to the same value, the standard setgid function is preferred.

RETURN VALUES

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

ERRORS

EPERM The current process is not the super-user and a change other than changing the effective group-id to the real group-id was specified.

SEE ALSO

getgid(2), setegid(2), setgid(2), setuid(2)

HISTORY

The setregid function call appeared in 4.2BSD and was dropped in 4.4BSD.

NAME

setreuid - set real and effective user ID's

SYNOPSIS

```
#include <unistd.h>

int
setreuid(ruid, euid)
  uid_t ruid, euid
```

DESCRIPTION

The real and effective user IDs 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. Unprivileged users may change the real user ID to the effective user ID and vice-versa; only the super-user may make other changes.

The setreuid function has been used to swap the real and effective user IDs in set-user-ID programs to temporarily relinquish the set-user-ID value. This purpose is now better served by the use of the seteuid function (see setuid(2)).

When setting the real and effective user IDs to the same value, the standard setuid function is preferred.

RETURN VALUES

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

ERRORS

EPERM The current process is not the super-user and a change other than changing the effective user-id to the real user-id was specified.

SEE ALSO

getuid(2), seteuid(2), setuid(2)

HISTORY

The setreuid function call appeared in 4.2BSD and was dropped in 4.4BSD.

NAME

setuid, seteuid,setgid,setegid - allow a process to access physical addresses

SYNOPSIS

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

int
setuid(uid)
  uid_t uid

int
seteuid(euid)
  uid_t euid

int
setgid(gid)
  gid_t gid

int
setegid(egid)
  gid_t egid
```

DESCRIPTION

The setuid function sets the real and effective user IDs and the saved set-user-ID of the current process to the specified value. The setuid function is permitted if the specified ID is equal to the real user ID of the process, or if the effective user ID is that of the super user.

The setgid function sets the real and effective group IDs and the saved set-group-ID of the current process to the specified value. The setgid function is permitted if the specified ID is equal to the real group ID of the process, or if the effective user ID is that of the super user.

The seteuid function (setegid) sets the effective user ID (group ID) of the current process. The effective user ID may be set to the value of the real user ID or the saved set-user-ID (see intro(2) and execve(2); in this way, the effective user ID of a set-user-ID executable may be toggled by switching to the real user ID, then re-enabled by reverting to the set-user-ID value. Similarly, the effective group ID may be set to the value of the real group ID or the saved set-user-ID.

RETURN VALUES

Upon success, these functions return 0; otherwise -1 is returned.

If the user is not the super user, or the uid specified is not the real, effective ID, or saved ID, these functions

return -1.

SEE ALSO
getuid(2), getgid(2)

STANDARDS
The setuid and setgid functions are compliant with the IEEE Std 1003.1-1988 (``POSIX'') specification with _POSIX_SAVED_IDS not defined. The seteuid and setegid functions are extensions based on the POSIX concept of _POSIX_SAVED_IDS , and have been proposed for a future revision of the standard.

NAME

shutdown - shut down part of a full-duplex connection

SYNOPSIS

```
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, then further receives will be disallowed. If how is 1, then further sends will be disallowed. If how is 2, then further sends and receives will be disallowed.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF] S is not a valid descriptor.

[ENOTSOCK] S is a file, not a socket.

[ENOTCONN] The specified socket is not connected.

SEE ALSO

connect(2), socket(2)

NAME

sigaction - software signal facilities

SYNOPSIS

```
#include <signal.h>

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

sigaction(sig, act, oact)
int sig;
struct sigaction *act;
struct sigaction *oact;
```

DESCRIPTION

The system defines a set of signals that may 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 may specify a handler to which a signal is delivered, or specify that a signal is to be ignored. A process may also specify that a default action is to be taken by the system when a signal occurs. A signal may also be blocked, in which case its delivery is postponed until it is unblocked. The action to be taken on delivery is determined at the time of delivery. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special signal stack.

Signal routines execute with the signal that caused their invocation blocked, but other signals may yet 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 empty). It may be changed with a sigprocmask(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 then it is delivered to the process. Signals may be delivered any time a process enters the operating system (e.g., during a system call, page fault or trap, or clock interrupt). If multiple signals are ready to be delivered at the same time, any signals that could be caused by traps are delivered first. Additional signals may be processed at the same time, with each appearing to interrupt the handlers for the previous signals before their first instructions. The set of pending

signals is returned by the sigpending(2) function. When a caught signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then 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' signal handler (or until a sigprocmask call is made). This mask is formed by taking the union of the current signal mask set, the signal to be delivered, and the signal mask associated with the handler to be invoked.

Sigaction assigns an action for a specific signal. If act is non-zero, it specifies an action (SIG_DFL, SIG_IGN, or a handler routine) and mask to be used when delivering the specified signal. If oact is non-zero, the previous handling information for the signal is returned to the user.

Once a signal handler is installed, it remains installed until another sigaction call is made, or an execve(2) is performed. A signal-specific default action may be reset by setting sa_handler to SIG_DFL. The defaults are process termination, possibly with core dump; no action; stopping the process; or continuing the process. See the signal list below for each signal's default action. If sa_handler is SIG_DFL, the default action for the signal is to discard the signal, and if a signal is pending, the pending signal is discarded even if the signal is masked. If sa_handler is set to SIG_IGN current and pending instances of the signal are ignored and discarded.

Options may be specified by setting sa_flags. If the SA_NOCLDSTOP bit is set when installing a catching function for the SIGCHLD signal, the SIGCHLD signal will be generated only when a child process exits, not when a child process stops. Further, if the SA_ONSTACK bit is set in sa_flags, the system will deliver the signal to the process on a signal stack, specified with sigstack(2).

If a signal is caught during the system calls listed below, the call may be forced to terminate with the error EINTR, the call may return with a data transfer shorter than requested, or the call may be restarted. Restart of pending calls is requested by setting the SA_RESTART bit in sa_flags. The affected system calls include open(2), read(2), write(2), sendto(2), recvfrom(2), sendmsg(2) and recvmsg(2) on a communications channel or a slow device

(such as a terminal, but not a regular file) and during a wait(2) or ioctl(2). However, calls that have already committed are not restarted, but instead return a partial success (for example, a short read count).

After a fork(2) or vfork(2) all signals, the signal mask, the signal stack, and the restart/interrupt flags are inherited by the child.

Execve(2) reinstates the default action for all signals which were caught and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that restart pending system calls continue to do so.

The following is a list of all signals with names as in the include file <signal.h>:

NAME	Action	Description
SIGHUP	terminate	terminal line hangup
SIGINT	terminate	interrupt program
SIGQUIT	core	quit program
SIGILL	core	illegal instruction
SIGTRAP	core	trace trap
SIGIOT	core	abort(2) call (same as SIGABRT)
SIGEMT	core	emulate instruction executed
SIGFPE	core	floating-point exception
SIGKILL	terminate	kill program
SIGBUS	core	bus error
SIGSEGV	core	segmentation violation
SIGSYS	core	system call given invalid argument
SIGPIPE	terminate	write on a pipe with no reader
SIGALRM	terminate	real-time timer expired
SIGTERM	terminate	software termination signal
SIGURG	discard	urgent condition present on socket
SIGSTOP	stop	stop (cannot be caught or ignored)
SIGTSTP	stop	stop generated from keyboard
SIGCONT	discard	continue after stop
SIGCHLD	discard	child status has changed
SIGTTIN	stop	background read attempted on control terminal
SIGTTOU	stop	background write attemped to control terminal
SIGIO	discard	I/O is possible on a descriptor (see fcntl(2))
SIGXCPU	terminate	cpu time limit exceeded (see setrlimit(2))
SIGXFSZ	terminate	file size limit exceeded (see setrlimit(2))

SIGVTALRM	terminate	virtual time alarm (see setitimer(2))
SIGPROF	terminate	profiling timer alarm (see setitimer(2))
SIGWINCH	discard	Window size change
SIGINFO	discard	status request from keyboard
SIGUSR1	terminate	User defined signal 1
SIGUSR2	terminate	User defined signal 2

NOTE

The mask specified in act is not allowed to block SIGKILL or SIGSTOP. This is done silently by the system.

RETURN VALUES

A 0 value indicated that the call succeeded. A -1 return value indicates an error occurred and errno is set to indicated the reason.

EXAMPLE

The handler routine can be declared:

```
int 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. Code is a parameter that is either a constant or the code provided by the hardware. Scp is a pointer to the sigcontext structure (defined in <signal.h>, used to restore the context from before the signal.

ERRORS

Sigaction will fail and no new signal handler will be installed if one of the following occurs:

EFAULT	Either act or oact 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.

STANDARDS

The sigaction function is defined by IEEE Std1003.1-1988 (``POSIX''). The SA_ONSTACK and SA_RESTART flags are Berkeley extensions, as are the signals, SIGTRAP, SIGEMT, SIGBUS, SIGSYS, SIGURG, SIGIO, SIGXCPU, SIGXFSZ, SIGVTALRM, SIGPROF, SIGWINCH, and SIGINFO. Those signals are available on most BSD-derived systems.

BUGS

The networking related syscalls are not properly restarted in 2.11BSD. The SIGINFO signal is not implemented in 2.11BSD.

SEE ALSO

kill(1), fcntl(2), ptrace(2), kill(2), setitimer(2), setrlimit(2), sigaction(2), sigprocmask(2), sigsuspend(2), sigblock(2), sigsetmask(2), sigpause(2), sigstack(2), sigvec(2), setjmp(3), siginterrupt(3), sigsetops(3), tty(4)

NAME

sigaltstack - set and/or get signal stack context

SYNOPSIS

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

struct sigaltstack {
   caddr_t ss_base;
   int  ss_size;
   int   ss_flags;
};

int
sigaltstack(ss, oss)
struct sigaltstack *ss;
struct sigaltstack *oss;
```

DESCRIPTION

Sigaltstack allows users to define an alternate stack on which signals are to be processed. If ss is non-zero, it specifies a pointer to and the size of 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 sigaction(2) 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 SA_DISABLE is set in ss_flags, ss_base and ss_size are ignored and the signal stack will be disabled. Trying to disable an active stack will cause sigaltstack to return -1 with errno set to EINVAL. A disabled stack will cause all signals to be taken on the regular user stack. If the stack is later re-enabled then all signals that were specified to be processed on an alternate stack will resume doing so.

If oss is non-zero, the current signal stack state is returned. The ss_flags field will contain the value SA_ONSTACK if the process is currently on a signal stack and SA_DISABLE if the signal stack is currently disabled.

NOTES

The value SIGSTKSZ is defined to be the number of bytes/chars that would be used to cover the usual case when allocating an alternate stack area. The following code fragment is typically used to allocate an alternate stack.

```
if ((sigstk.ss_base = malloc(SIGSTKSZ)) == NULL)
   /* error return */
```

```
sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk,0) < 0)
   perror("sigaltstack");
```

An alternative approach is provided for programs with signal handlers that require a specific amount of stack space other than the default size. The value MINSIGSTKSZ is defined to be the number of bytes/chars that is required by the operating system to implement the alternate stack feature. In computing an alternate stack size, programs should add MINSIGSTKSZ to their stack requirements to allow for the operating system overhead.

Signal stacks are automatically adjusted for the direction of stack growth and alignment requirements. Signal stacks may or may not be protected by the hardware and are not ``grown'' automatically as is done for the normal stack. If the stack overflows and this space is not protected unpredictable results may occur.

RETURN VALUES

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

ERRORS

Sigaltstack will fail and the signal stack context will remain 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.
EINVAL	An attempt was made to disable an active stack.
ENOMEM	Size of alternate stack area is less than or equal to MINSIGSTKSZ .

SEE ALSO

sigaction(2), setjmp(3)

HISTORY

The predecessor to sigaltstack, the sigstack system call, appeared in 4.2BSD.

NAME

sigblock - block signals

SYNOPSIS

```
#include <signal.h>

omask = sigblock(mask);
long omask, mask;

mask = sigmask(signum)
long mask;
int signum;
```

DESCRIPTION

This interface is made obsolete by: sigprocmask(2).

Sigblock causes the signals specified in mask to be added to the set of signals currently being blocked from delivery. Signals are blocked if the corresponding bit in mask is a 1; the macro sigmask is provided to construct the mask for a given signum.

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

kill(2), sigprocmask(2), sigaction(2), sigsetmask(2), sigsetops(2)

HISTORY

The sigblock function call appeared in 4.2BSD and has been deprecated.

NAME

sigpause - atomically release blocked signals and wait for interrupt

SYNOPSIS

```
sigpause(sigmask)
long sigmask;
```

DESCRIPTION

This interface is made obsolete by: sigsuspend(2).

Sigpause() 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. Sigmask is usually 0L to indicate that no signals are now to be blocked. Sigpause always terminates by being interrupted, returning -1 with errno set to EINTR.

SEE ALSO

sigsuspend(2), kill(2), sigaction(2), sigprocmask(2), sigblock(2), sigvec(2)

HISTORY

The sigpause function call appeared in 4.2BSD and has been deprecated.

NAME

sigpending - get pending signals

SYNOPSIS

```
#include <signal.h>

int
sigpending(set)
sigset_t *set;
```

DESCRIPTION

The sigpending function returns a mask of the signals pending for delivery to the calling process in the location indicated by set. Signals may be pending because they are currently masked, or transiently before delivery (although the latter case is not normally detectable).

RETURN VALUES

A 0 value indicated that the call succeeded. A -1 return value indicates an error occurred and errno is set to indicated the reason.

ERRORS

If sigpending fails then errno will contain one of the following:

[EFAULT] set contains an invalid address.

SEE ALSO

sigaction(2), sigprocmask(2)

STANDARDS

The sigpending function is defined by IEEE Std1003.1-1988 (``POSIX'').

NAME

sigprocmask - manipulate current signal mask

SYNOPSIS

```
#include <signal.h>

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

sigset_t
sigmask(signum)
int signum;
```

DESCRIPTION

The sigprocmask function examines and/or changes the current signal mask (those signals that are blocked from delivery). Signals are blocked if they are members of the current signal mask set.

If set is not null, the action of sigprocmask depends on the value of the parameter how. The signal mask is changed as a function of the specified set and the current mask. The function is specified by how using one of the following values from <signal.h>:

SIG_BLOCK The new mask is the union of the current mask and the specified set.

SIG_UNBLOCK The new mask is the intersection of the current mask and the complement of the specified set.

SIG_SETMASK The current mask is replaced by the specified set.

If oset is not null, it is set to the previous value of the signal mask. When set is null, the value of how is insignificant and the mask remains unset providing a way to examine the signal mask without modification.

The system quietly disallows SIGKILL or SIGSTOP to be blocked.

RETURN VALUES

A 0 value indicated that the call succeeded. A -1 return value indicates an error occurred and errno is set to indicated the reason.

ERRORS

The sigprocmask call will fail and the signal mask will be unchanged if one of the following occurs:

EINVAL	how has a value other than those listed here.
EFAULT	set or oset contain an invalid address.

SEE ALSO

kill(2), sigaction(2), sigsetops(3), sigsuspend(2)

STANDARDS

The sigprocmask function call is expected to conform to IEEE Std1003.1-1988 (``POSIX'').

NAME

sigreturn - return from signal

SYNOPSIS

```
#include <signal.h>

struct    sigcontext {
   int  sc_onstack;
   long sc_mask;
   int  sc_sp;
   int  sc_fp;
   int  sc_ap;
   int  sc_pc;
   int  sc_ps;
};

sigreturn(scp);
struct sigcontext *scp;
```

DESCRIPTION

Sigreturn allows users to atomically unmask, switch stacks, and return from a signal context. The processes signal mask and stack status are restored from the context. The system call does not return; the users stack pointer, frame pointer, argument pointer, and processor status longword are restored from the context. Execution resumes at the specified pc. This system call is used by the trampoline code, and longjmp(3) when returning from a signal to the previously executing program.

NOTES

This system call is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

The definition of the sigcontext structure is machine dependent (the structure cited above is that for a VAX running 4.3BSD); no program should depend on its internal structure. Setjmp(3) may be used to build sigcontext structures in a machine independent manner.

RETURN VALUE

If successful, the system call does not return. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

Sigreturn will fail and the process context will remain unchanged if one of the following occurs.

[EFAULT] Scp points to memory that is not a valid part of the process address space.

[EINVAL] The process status longword is invalid or would improperly raise the privilege level of the process.

SEE ALSO

sigvec(2), setjmp(3)

NOTES (PDP-11)

On the PDP-11 the field sc_ap (argument pointer) does not exist and the field sc_fp (frame pointer) is the PDP-11 register r5. Additionally, three new fields sc_r0, sc_r1 and sc_ovno are present on the PDP-11 which hold register values r0 and r1 and the text overlay number to restore (see ld(1)).

```
struct  sigcontext {
        int   sc_onstack; /* sigstack state to restore */
        long  sc_mask;    /* signal mask to restore */
        int   sc_sp;      /* sp to restore */
        int   sc_fp;      /* fp to restore */
        int   sc_r1;      /* r1 to restore */
        int   sc_r0;      /* r0 to restore */
        int   sc_pc;      /* pc to restore */
        int   sc_ps;      /* psl to restore */
        int   sc_ovno     /* overlay to restore */
};
```

NAME

sigsetmask - set current signal mask

SYNOPSIS

```
#include <signal.h>

omask = sigsetmask(mask);
long omask, mask;

mask = sigmask(signum)
long mask;
int signum;
```

DESCRIPTION

This interface is made obsolete by:

Sigsetmask sets the current signal mask (those signals that are blocked from delivery). Signals are blocked if the corresponding bit in mask is a 1; the macro sigmask is provided to construct the mask for a given signum.

The system quietly disallows SIGKILL, SIGSTOP, or SIGCONT to be blocked.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

kill(2), sigvec(2), sigblock(2), sigpause(2)

HISTORY

The sigsetmask function call appeared in 4.2BSD and has been deprecated.

NAME

sigstack - set and/or get signal stack context

SYNOPSIS

```
#include <signal.h>

struct sigstack {
 caddr_t     ss_sp;
 int    ss_onstack;
};

sigstack(ss, oss);
struct sigstack *ss, *oss;
```

DESCRIPTION

This interface has been made obsolete sigaltstack(2).

Sigstack allows users to define an alternate stack on which signals are to be processed. If ss is non-zero, 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(2) 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 non-zero, the current signal stack state is returned.

NOTES

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

ERRORS

Sigstack will fail and the signal stack context will remain 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)

HISTORY

The sigstack function call appeared in 4.2BSD and has been

deprecated.

NAME

sigsuspend - atomically release blocked signals and wait for interrupt

SYNOPSIS

```
#include <signal.h>

int
sigsuspend(sigmask)
sigset_t *sigmask
```

DESCRIPTION

Sigsuspend() temporarily changes the blocked signal mask to the set to which sigmask points, and then waits for a signal to arrive; on return the previous set of masked signals is restored. The signal mask set is usually empty to indicate that all signals are to be unblocked for the duration of the call.

In normal usage, a signal is blocked using sigprocmask(2) to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using sigsuspend with the previous mask returned by sigprocmask.

RETURN VALUES

The sigsuspend function always terminates by being interrupted, returning -1 with errno set to EINTR. If EFAULT is set in errno then set contains an invalid address.

SEE ALSO

sigprocmask(2), sigaction(2), sigsetops(3)

STANDARDS

The sigsuspend function call conforms to IEEE Std1003.1-1988 (``POSIX'').

NAME

sigvec - software signal facilities

SYNOPSIS

```
#include <signal.h>

struct sigvec {
 int    (*sv_handler)();
 long   sv_mask;
 int    sv_flags;
};

sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

DESCRIPTION

This interface has been made obsolete sigaction(2).

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurence 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 may specify a handler to which a signal is delivered, or specify that a signal is to be blocked or ignored. A process may 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 may 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 may yet 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(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 then 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 below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then 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' 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 or'ing in the signal mask associated with the handler to be invoked.

Sigvec assigns a handler for a specific signal. If vec is non-zero, 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 will deliver the signal to the process on a signal stack, specified with sigstack(2). If ovec is non-zero, 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
SIGUSR1   30   user defined signal 1
SIGUSR2   31   user defined signal 2
```

The starred signals in the list above cause a core image if not caught or ignored.

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 may be reinstated by setting sv_handler to SIG_DFL; this default is termination (with a core image for starred signals) except for signals marked with @ or '+'. Signals marked with @ are discarded if the action is SIG_DFL; signals marked with '+' 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(2) or write(2) on a slow device (such as a terminal; but not a file) and during a wait(2).

After a fork(2) or vfork(2) the child inherits all signals, the signal mask, the signal stack, and the restart/interrupt flags.

Execve(2) 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.

NOTES

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 4.2BSD, hence it should not be used if backward compatibility is needed.

RETURN VALUE

A 0 value indicated that the call succeeded. A -1 return value indicates an error occurred and errno is set to indicated the reason.

ERRORS

Sigvec will fail and no new signal handler will be 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).

SEE ALSO

kill(1), ptrace(2), kill(2), sigblock(2), sigsetmask(2), sigpause(2), sigstack(2), sigvec(2), setjmp(3), siginterrupt(3), tty(4)

NOTES (VAX-11)

The handler routine can be declared:

```
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 below. Code is a parameter that is either a constant as given below 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). 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	
Protection violation	SIGBUS	
Reserved instruction	SIGILL	ILL_RESAD_FAULT
Customer-reserved instr.	SIGEMT	
Reserved operand	SIGILL	ILL_PRIVIN_FAULT
Reserved addressing	SIGILL	ILL_RESOP_FAULT
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	
Compatibility-mode	SIGILL	hardware supplied code
Chme	SIGSEGV	
Chms	SIGSEGV	

Chmu SIGSEGV

NOTES (PDP-11)

The handler routine can be declared:

```
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 below. Code is a parameter that is a constant as given below. 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:		
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
Illegal return code	SIGFPE	FPE_CRAZY
Bad op code	SIGFPE	FPE_OPCODE_TRAP
Bad operand	SIGFPE	FPE_OPERAND_TRAP
Maintenance trap	SIGFPE	FPE_MAINT_TRAP
Length access control	SIGSEGV	
Protection violation (odd address)	SIGBUS	
Reserved instruction	SIGILL	ILL_RESAD_FAULT
Customer-reserved instr.	SIGEMT	
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

The handler routine must save any registers it uses and restore them before returning. On the PDP-11, the kernel saves r0 and r1 before calling the handler routine, but expect the handler to save any other registers it uses. The standard entry code generated by the C compiler for handler routines written in C automatically saves the remaining general registers, but floating point registers are not saved. As a result there is currently no [standard] method for a handler routine written in C to perform floating point operations without blowing the interrupted program out of the water.

BUGS

This manual page is still confusing.

NAME

sigwait - wait for a signal

SYNOPSIS

```
#include <signal.h>

int sigwait(set, sig)
sigset_t *set;
int *sig;
```

DESCRIPTION

Sigwait checks for a pending signal in set, clears it from the set of pending signals and returns the signal number in the location referenced by sig. If more than one of the signals contained in set is pending then sigwait selects only one and acts upon it. If no signal contained in set is pending, then sigwait waits for a signal to arrive. All of the signals contained in set should be blocked or unpredictable results may occur.

RETURN VALUES

The sigwait function returns 0 if successful and the signal number is stored in the location referenced by sig.

ERRORS

The sigwait function may return one of the following errors:

EINVAL	The set argument contains an invalid or unsupported signal number.
EFAULT	Sig points to memory that is not a valid part of the process address space.

SEE ALSO

sigprocmask(2)

STANDARDS

The sigwait function call conforms to IEEE Std1003.1-1998 (``POSIX'').

NAME

socket - create an endpoint for communication

SYNOPSIS

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

s = socket(domain, type, protocol)
int s, domain, type, protocol;
```

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The domain parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file <sys/socket.h>. The currently understood formats are

```
PF_UNIX     (UNIX internal protocols),
PF_INET     (ARPA Internet protocols),
PF_NS       (Xerox Network Systems protocols), and
PF_IMPLINK  (IMP "host at IMP" link layer).
```

The socket has the indicated type, which specifies the semantics of communication. Currently defined types are:

```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM
```

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for PF_NS. SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW, which is available only to the super-user, and SOCK_RDM, which is planned, but not yet implemented, are not described here.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. 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; see protocols(3N).

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 may be sent or received on it. A connection to another socket is created with a connect(2) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) calls. When a session has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in send(2) and received as described in recv(2).

The communications protocols used to implement a SOCK_STREAM insure 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 a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send(2) calls. Datagrams are generally received with recvfrom(2), which returns the next datagram with its return address.

An fcntl(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

The operation of sockets is controlled by socket level options. These options are defined in the file <sys/socket.h>. Setsockopt(2) and getsockopt(2) are used to

set and get options, respectively.

RETURN VALUE

A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The socket call fails if:

[EPROTONOSUPPORT] The protocol type or the specified protocol is not supported within this domain.

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

[EACCESS] Permission to create a socket of the specified type and/or protocol is denied.

[ENOBUFS] Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), write(2) ``An Introductory 4.3BSD Interprocess Communication Tutorial.'' (reprinted in UNIX Programmer's Supplementary Documents Volume 1, PS1:7) ``An Advanced 4.3BSD Interprocess Communication Tutorial.'' (reprinted in UNIX Programmer's Supplementary Documents Volume 1, PS1:8)

NAME

socketpair - create a pair of connected sockets

SYNOPSIS

```
#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.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

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

read(2), write(2), pipe(2)

BUGS

This call is currently implemented only for the UNIX domain.

NAME

stat, lstat, fstat - get file status

SYNOPSIS

```
#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

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

Lstat is like stat except in the case where the named file is a symbolic link, in which case lstat returns information about the link, while stat returns information about the file the link references.

Fstat obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an open call.

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

```
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[2];
  };
```

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

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

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

The status information word st_mode has 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_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 (see chmod(2)).

RETURN VALUE

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

ERRORS

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

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

directory.

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

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

[ENOENT] The named file does not exist.

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

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

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

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

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

[EBADF] Fildes is not a valid open file descriptor.

[EFAULT] Buf points to an invalid address.

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

CAVEAT

The fields in the stat structure currently marked st_spare1, st_spare2, and st_spare3 are present in preparation for inode time stamps expanding to 64 bits. This, however, can break certain programs that depend on the time stamps being contiguous (in calls to utimes(2)).

SEE ALSO

chmod(2), chown(2), utimes(2)

BUGS

Applying fstat to a socket (and thus to a pipe) returns a zero'd buffer, except for the blocksize field, and a unique device and inode number.

NAME

statfs, fstatfs - get file system statistics

SYNOPSIS

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

int
statfs(path,buf)
char *path;
struct statfs *buf;

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

DESCRIPTION

Statfs() returns information about a mounted file system. Path is the path name of any file within the mounted filesystem. Buf is a pointer to a statfs structure defined as follows:

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

struct statfs {
short   f_type;          /* type of filesystem (see below) */
short   f_flags;         /* copy of mount flags */
short   f_bsize;         /* fundamental file system block size */
short   f_iosize;        /* optimal transfer block size */
long    f_blocks;        /* total data blocks in file system */
long    f_bfree;         /* free blocks in fs */
long    f_bavail;        /* free blocks avail to non-superuser */
ino_t   f_files;         /* total file nodes in file system */
ino_t   f_ffree;         /* free file nodes in fs */
u_long  f_fsid[2];        /* file system id */
long    f_spare[4];       /* spare for later */
char    f_mntonname[MNAMELEN];    /* mount point */
char    f_mntfromname[MNAMELEN];  /* mounted filesystem */
};
/*
* File system types. - Only UFS is supported so the other types are not
* given.
*/
#define   MOUNT_UFS 1    /* Fast Filesystem */
```

Fields that are undefined for a particular file system are set to -1. Fstatfs() returns the same information about an open file referenced by descriptor fd.

RETURN VALUES

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

ERRORS

Statfs() fails if one or more of the following are true:

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

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

[ENAMETOOLONG] The length of a component of path exceeds 63 characters, or the length of path exceeds 255 characters.

[ENOENT] The file referred to by path does not exist.

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

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

[EFAULT] Buf or path points to an invalid address.

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

Fstatfs() fails if one or more of the following are true:

[EBADF] Fd is not a valid open file descriptor.

[EFAULT] Buf points to an invalid address.

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

HISTORY

The statfs function first appeared in 4.4BSD.

NAME

swapon - add a swap device for interleaved paging/swapping

SYNOPSIS

```
swapon(special)
char *special;
```

DESCRIPTION

Swapon 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.

RETURN VALUE

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

ERRORS

Swapon succeeds unless:

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

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

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

[ENOENT] The named 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] Special is not a block device.

[EBUSY] The device specified by special has already been made available for swapping

[EINVAL] The device configured by special was not configured into the system as a swap device.

[ENXIO] The major device number of special is out of range (this indicates no device driver exists

for the associated hardware).

[EIO] An I/O error occurred while opening the swap device.

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

SEE ALSO

swapon(8), config(8)

BUGS

There is no way to stop swapping on a disk so that the pack may be dismounted.

This call will be upgraded in future versions of the system.

NAME

symlink - make symbolic link to a file

SYNOPSIS

```
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 may be an arbitrary path name; the files need not be on the same file system.

RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in errno and a -1 value is returned.

ERRORS

The symbolic link is made unless on or more of the following are true:

[ENOTDIR] A component of the name2 prefix is not a directory.

[EINVAL] Either name1 or name2 contains a character with the high-order bit set.

[ENAMETOOLONG] A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] A component of the name2 path prefix denies search permission.

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

[EEXIST] Name2 already exists.

[EIO] An I/O error occurred while making the directory entry for name2, or allocating the inode for name2, or writing out the link contents of name2.

[EROFS] The file name2 would reside on a read-only file system.

[ENOSPC] The directory in which the entry for the new symbolic link is being placed cannot be

extended because there is no space left on the file system containing the directory.

[ENOSPC] The new symbolic link cannot be created because there 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 sym- bolic link has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the symbolic link is being created has been exhausted.

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

[EFAULT] Name1 or name2 points outside the process's allocated address space.

SEE ALSO

link(2), ln(1), unlink(2)

NAME

sync - update super-block

SYNOPSIS

sync()

DESCRIPTION

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

Sync should be used by programs that examine a file system, for example fsck, df, etc. Sync is mandatory before a boot.

SEE ALSO

fsync(2), sync(8), update(8)

BUGS

The writing, although scheduled, is not necessarily complete upon return from sync.

NAME

syscall - indirect system call

SYNOPSIS

```
#include <syscall.h>

syscall(number, arg, ...)    (VAX-11)
```

DESCRIPTION

Syscall performs the system call whose assembly language interface has the specified number, register arguments r0 and r1 and further arguments arg. Symbolic constants for system calls can be found in the header file <syscall.h>.

The r0 value of the system call is returned.

DIAGNOSTICS

When the C-bit is set, syscall returns -1 and sets the external variable errno (see intro(2)).

BUGS

There is no way to simulate system calls such as pipe(2), which return values in register r1.

NAME

truncate - truncate a file to a specified length

SYNOPSIS

```
truncate(path, length)
char *path;
off_t length;

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

DESCRIPTION

Truncate 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 VALUES

A value of 0 is returned if the call succeeds. If the call fails a -1 is returned, and the global variable errno specifies the error.

ERRORS

Truncate succeeds unless:

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

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

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

[ENOENT] The named file does not exist.

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

[EACCES] The named file is not writable by the user.

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

[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.

[EIO] An I/O error occurred updating the inode.

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

Ftruncate succeeds unless:

[EBADF] The fd is not a valid descriptor.

[EINVAL] The fd references a socket, not a file.

[EINVAL] The fd is not open for writing.

SEE ALSO

open(2)

BUGS

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

NAME

ucall - call a kernel subroutine from user mode (2BSD)

SYNOPSIS

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

ucall(priority, function, arg0, arg1)
int priority, arg0, arg1;
caddr_t function;
```

DESCRIPTION

Ucall causes the processor priority to be set to priority and the specified kernel function to be called with arguments arg0 and arg1. Priority is one of PSL_BR0, ..., PSL_BR7. Processor priority is reset to PSL_BR0 when function returns.

Ucall is allowed only if the user is the superuser. It is obviously extremely dangerous if misused. It's only current use is at system boot time to configure system devices by calling device drivers ...

ERRORS

[EPERM] The caller is not the super-user.

SEE ALSO

autoconfig(8)

BUGS

No address validations are attempted.

Ucall is unique to the PDP-11 and 2BSD; its use is discouraged.

NAME

umask - set file creation mode mask

SYNOPSIS

```
oumask = umask(numask)
int oumask, numask;
```

DESCRIPTION

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

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

RETURN VALUE

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

SEE ALSO

chmod(2), mknod(2), open(2)

NAME

unlink - remove directory entry

SYNOPSIS

```
unlink(path)
char *path;
```

DESCRIPTION

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

RETURN VALUE

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

ERRORS

The unlink succeeds unless:

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

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

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

[ENOENT] The named file does not exist.

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

[EACCES] Write permission is denied on the directory containing the link to be removed.

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

[EPERM] The named file is a directory and the effective user ID of the process is not the super-user.

[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.

[EBUSY] The entry to be unlinked is the mount point for a mounted file system.

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

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

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

SEE ALSO

close(2), link(2), rmdir(2)

NAME

utimes - set file times

SYNOPSIS

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

utimes(file, tvp)
char *file;
struct timeval tvp[2];
```

DESCRIPTION

The utimes call uses the "accessed" and "updated" times in that order from the tvp vector to set the corresponding recorded times for file.

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

RETURN VALUE

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

ERRORS

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

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

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

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

[ENOENT] The named file does not exist.

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

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

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

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

[EFAULT] File or tvp points outside the process's

allocated address space.

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

SEE ALSO
stat(2)

NAME

vfork - spawn new process in a virtual memory efficient way

SYNOPSIS

```
pid = vfork()
int pid;
```

DESCRIPTION

Vfork can be used to create new processes without fully copying the address space of the old process, which is horrendously inefficient in a paged environment. It is useful when the purpose of fork(2) would have been to create a new system context for an execve. Vfork differs from fork in that the child borrows the parent's memory and thread of control until a call to execve(2) or an exit (either by a call to exit(2) or abnormally.) The parent process is suspended while the child is using its resources.

Vfork returns 0 in the child's context and (later) the pid of the child in the parent's context.

Vfork can normally be used just like fork. It does not work, however, to return while running in the childs context from the procedure that called vfork since the eventual return from vfork would then return to a no longer existent stack frame. Be careful, also, to call _exit rather than exit if you can't execve, since exit will flush and close standard I/O channels, and thereby mess up the parent processes standard I/O data structures. (Even with fork it is wrong to call exit since buffered data would then be flushed twice.)

SEE ALSO

fork(2), execve(2), sigvec(2), wait(2),

DIAGNOSTICS

Same as for fork.

BUGS

This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of vfork as it will, in that case, be made synonymous to fork.

To avoid a possible deadlock situation, processes that are children in the middle of a vfork are never sent SIGTTOU or SIGTTIN signals; rather, output or ioctls are allowed and input attempts result in an end-of-file indication.

NAME

vhangup - virtually ``hangup'' the current control terminal

SYNOPSIS

vhangup()

DESCRIPTION

Vhangup is used by the initialization process init(8) (among others) to arrange that users are given "clean"' terminals at login, by revoking access of the previous users' processes to the terminal. To effect this, vhangup searches the system tables for references to the control terminal of the invoking process, revoking access permissions on each instance of the terminal that it finds. Further attempts to access the terminal by the affected processes will yield i/o errors (EBADF). Finally, a hangup signal (SIGHUP) is sent to the process group of the control terminal.

SEE ALSO

init (8)

BUGS

Access to the control terminal via /dev/tty is still possible.

This call should be replaced by an automatic mechanism that takes place on process exit.

NAME

wait, waitpid, wait4, wait3 - wait for process terminatation

SYNOPSIS

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

pid = wait(status)
int pid;
union wait *status;

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

pid = waitpid(wpid, status, options);
int pid;
int wpid;
union wait *status;
int options;

pid = wait3(status, options, rusage);
int pid;
union wait *status;
int options;
struct rusage *rusage;

pid = wait4(wpid, status, options, rusage);
int pid;
int wpid;
union wait *status;
int options;
struct rusage *rusage;
```

DESCRIPTION

The wait function suspends execution of its calling process until status information is available for a terminated child process, or a signal is received. On return from a successful wait call, the status area contains termination information about the process that exited as defined below.

The wait4 call provides a more general interface for programs that need to wait for certain child processes, that need resource utilization statistics accummulated by child processes, or that require options. The other wait functions are implemented using wait4 .

The wpid parameter specifies the set of child processes for which to wait. If wpid is -1, the call waits for any child process. If wpid is 0, the call waits for any child process in the process group of the caller. If wpid is greater than zero, the call waits for the process with process id wpid . If wpid is less than -1, the call waits for any process

whose process group id equals the absolute value of wpid .

The status parameter is defined below. The options parameter contains the bitwise OR of any of the following options. The WNOHANG option is used to indicate that the call should not block if there are no processes that wish to report status. If the WUNTRACED option is set, children of the current process that are stopped due to a SIGTTIN , SIGTTOU , SIGTSTP , or SIGSTOP signal also have their status reported.

If rusage is non-zero, a summary of the resources used by the terminated process and all its children is returned (this information is currently not available for stopped processes).

When the WNOHANG option is specified and no processes wish to report status, wait4 returns a process id of 0.

The waitpid call is identical to wait4 with an rusage value of zero. The older wait3 call is the same as wait4 with a wpid value of -1.

The following macros may be used to test the manner of exit of the process. One of the first three macros will evaluate to a non-zero (true) value:

WIFEXITED(status) - True if the process terminated normally by a call to _exit(2) or exit(2) .

WIFSIGNALED(status) - True if the process terminated due to receipt of a signal.

WIFSTOPPED(status) - True if the process has not terminated, but has stopped and can be restarted. This macro can be true only if the wait call specified the WUNTRACED option or if the child process is being traced (see ptrace(2)).

Depending on the values of those macros, the following macros produce the remaining status information about the child process:

WEXITSTATUS(status) - If WIFEXITED(status) is true, evaluates to the low-order 8 bits of the argument passed to _exit(2) or exit(2) by the child.

WTERMSIG(status) - If WIFSIGNALED(status) is true, evaluates to the number of the signal that caused the termination of the process.

WCOREDUMP(status) If WIFSIGNALED(status) is true, evaluates as true if the termination of the process was accompanied by

the creation of a core file containing an image of the process when the signal was received.

WSTOPSIG(status) If WIFSTOPPED(status) is true, evaluates to the number of the signal that caused the process to stop.

NOTES

See sigvec(2) for a list of termination signals. A status of 0 indicates normal termination.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are assigned the parent process 1 ID (the init process ID).

If a signal is caught while any of the wait calls is pending, the call may be interrupted or restarted when the signal-catching routine returns, depending on the options in effect for the signal; see intro(2), System call restart.

RETURN VALUES

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

If wait4(), wait3() or waitpid() returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If there are no children not previously awaited, -1 is returned with errno set to [ECHILD]. Otherwise, if WNOHANG is specified and there are no stopped or exited children, 0 is returned. If an error is detected or a caught signal aborts the call, a value of -1 is returned and errno is set to indicate the error.

ERRORS

Wait() will fail and return immediately if:

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

[EFAULT] The status or rusage arguments point to an illegal address. (May not be detected before exit of a child process.)

[EINTR] The call was interrupted by a caught signal, or the signal had the SV_INTERRUPT flag set.

STANDARDS

The wait and waitpid functions are defined by POSIX; wait4 and wait3 are not specified by POSIX. The WCOREDUMP macro and the ability to restart a pending wait call are

extensions to the POSIX interface.

SEE ALSO

exit(2) , sigvec(2) A wait function call appeared in Version 6 AT&T UNIX.

NAME

write, writev - write output

SYNOPSIS

```
cc = write(d, buf, nbytes)
int cc, d;
char *buf;
unsigned short nbytes;

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

cc = writev(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Write attempts to write nbytes of data to the object referenced by the descriptor d from the buffer pointed to by buf. Writev performs the same action, but gathers the output data from the iovcnt buffers specified by the members of the iov array: iov[0], iov[1], ..., iov[iovcnt-1].

For writev, the iovec structure is defined as

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

Each iovec entry specifies the base address and length of an area in memory from which data should be written. Writev will always write a complete area before proceeding to the next.

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

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

If the real user is not the super-user, then write clears the set-user-id bit on a file. This prevents penetration of system security by a user who "captures" a writable set-user-id file owned by the super-user.

When using non-blocking I/O on objects such as sockets that are subject to flow control, write and writev may write

fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

RETURN VALUE

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

ERRORS

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

[EBADF] D is not a valid descriptor open for writing.

[EPIPE] An attempt is made to write to a pipe that is not open for reading by any process.

[EPIPE] An attempt is made to write to a 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 or the maximum file size.

[EFAULT] Part of iov or data to be written to the file points outside the process's allocated address space.

[EINVAL] The pointer associated with d was negative.

[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.

[EWOULDBLOCK] The file was marked for non-blocking I/O, and no data could be written immediately.

In addition, writev may return one of the following errors:

[EINVAL] Iovcnt was less than or equal to 0, or greater than 16.

[EINVAL] The sum of the iov_len values in the iov array overflowed a short.

SEE ALSO
fcntl(2), lseek(2), open(2), pipe(2), select(2)