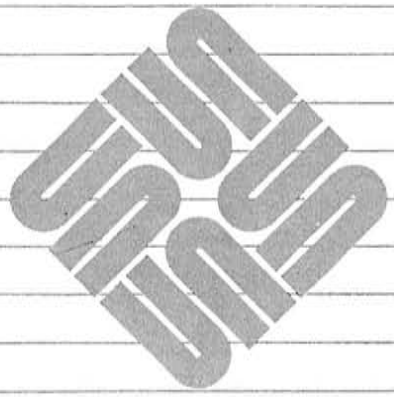
sun microsystems

UNIX Interface Reference Manual



Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part Number: 800-1303-04
Revision: A of 19 September 1986

Credits and Trademarks

Sun Workstation® is a registered trademark of Sun Microsystems, Inc.

SunStation®, Sun Microsystems®, SunCore®, SunWindows®, DVMA®, and the combination of Sun with a numeric suffix are trademarks of Sun Microsystems, Inc.

UNIX, UNIX/32V, UNIX System III, and UNIX System V are trademarks of AT&T Bell Laboratories.

Intel® and Multibus® are registered trademarks of Intel Corporation.

DEC®, PDP®, VT®, and VAX® are registered trademarks of Digital Equipment Corporation.

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. A "(2V)" heading indicates that the system call performs differently when called from programs that use the System V libraries (programs compiled using **/usr/5bin/cc**). On these pages, both the regular behavior and the System V behavior is described.

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.

Each system call description attempts to list all possible error numbers. The following is a complete list of the errors and their names as given in *<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 filename is specified and the file should exist but doesn't, or when one of the directories in a pathname 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), which 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. 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 which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not on-line or a disk pack is not loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 10240 bytes is presented to *execve*.

8 ENOEXEC Exec format error
A request is made to execute a file which, 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 (respectively, write) request is made to a file which is open only for writing (respectively, reading).

10 ECHILD No children
A *wait* was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN No more processes
A *fork* failed because 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*, *brk*, or *sbrk*, a program asks for more address space 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 address space 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 file which is not a block device was mentioned where a block device was required, for example, in *mount*.

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

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, for example, *link*.

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

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device (for example, an attempt to read a write-only device) or an attempt was made to use a device not configured by the system.

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

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

22 EINVAL Invalid argument
A system call was made with an invalid argument; for example, dismounting a non-mounted file system, mentioning an unknown signal in *sigvec* or *kill*, reading or writing a file for which *lseek* has generated a negative pointer, or some other argument inappropriate for the call. Also set by math functions, see *intro*(3).

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

24 EMFILE Too many open files
A process tried to have more open files than the system allows a process to have. The customary configuration limit is 30 per process.

25 ENOTTY Inappropriate ioctl for device
The code used in an *ioctl* call is not supported by the object that the file descriptor in the call refers to.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which 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 file size (1,082,201,088 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 file system mounted read-only.

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

32 EPIPE Broken pipe
An attempt was made to write on a pipe or socket for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is caught or ignored.

33 EDOM Math argument
The argument of a function in the math library (as described in section 3M) is out of the domain of the function.

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

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

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

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

41 EPROTOTYPE Protocol wrong type for socket
A protocol was specified which 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 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 PUP Internet 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 the peer executing a *shutdown*(2) call.

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

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 other than the connected party.

57 ENOTCONN Socket is not connected
An request to send or receive data was disallowed because the socket is not connected.

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* 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 which is inactive on the foreign host.

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

63 ENAMETOOLONG File name too long
A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 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
An attempt was made to remove a directory with entries other than . and .. by performing a *rmdir* system call or a *rename* system call with that directory specified as the target directory.

67 *unused*

68 *unused*

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.

70 ESTALE Stale NFS file handle
A client referenced a an open file, when the file has been deleted.

71 EREMOTE Too many levels of remote in path
An attempt was made to remotely mount a file system into a path which already has a remotely mounted component.

72 *unused*

73 *unused*

74 *unused*

75 ENOMSG No message of desired type
An attempt was made to receive a message of a type that does not exist on the specified message queue; see *msgop*(2).

76 *unused*

77 EIDRM Identifier removed
This error is returned to processes that resume execution due to the removal of an identifier from the IPC system's name space (see *msgctl*(2), *semctl*(2), and *shmctl*(2)).

DEFINITIONS

Descriptor
An integer assigned by the system when a file is referenced by *open*(2V), *dup*(2), or *pipe*(2) or a socket is referenced by *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.

Directory
A directory is a special type of file which contains entries which 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.

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 of 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".

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.

File Name

Names consisting of up to 255 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 filenames because of the special meaning attached to these characters by the shell. See *sh*(1). Although permitted, it is advisable to avoid the use of unprintable characters in filenames.

Message Queue Identifier

A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

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

msg_perm is an ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

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

msg_qnum is the number of messages currently on the queue. **msg_qbytes** is the maximum number of bytes allowed on the queue. **msg_lspid** is the process id of the last process that performed a *msgsnd* operation. **msg_lrpid** is the process id of the last process that performed a *msgrcv* operation. **msg_stime** is the time of the last *msgsnd* operation, **msg_rtime** is the time of the last *msgrcv* operation, and **msg_ctime** is the time of the last *msgctl*(2) operation that changed a member of the above structure.

Message Operation Permissions

In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

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

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

The effective user ID of the process is super-user.

The effective user ID of the process matches **msg_perm.[c]uid** in the data structure associated with *msqid* and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.

The effective user ID of the process does not match **msg_perm.[c]uid** and the effective group ID of the process matches **msg_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The effective user ID of the process does not match **msg_perm.[c]uid** and the effective group ID of the process does not match **msg_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

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.

Path Name and Path Prefix

A pathname 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 filename. The total length of a pathname must be less than {MAXPATHLEN} (1024) characters.

More precisely, a pathname is a null-terminated character string constructed as follows:

```
<path-name>::=<file-name>|<path-prefix><file-name>|/
<path-prefix>::=<rtprefix>|/<rtprefix>
<rtprefix>::=<dirname>/|<rtprefix><dirname>/
```

where <file-name> is a string of 1 to 255 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 255 characters (other than the ASCII slash and null) that names a directory.

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

A slash, by itself, names the root directory. A dot (.) names the current working directory.

A null pathname also refers to the current directory. However, this is not true of all UNIX systems. (On such systems, accidental use of a null pathname in routines that don't check for it may corrupt the current working directory.) For portable code, specify the current directory explicitly using ".", rather than "".

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).

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.

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 which created it.

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.

Semaphore Identifier

A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

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

sem_perm is an ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

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

The value of **sem_nsems** is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. *sem_num* values run sequentially from 0 to the value of **sem_nsems** minus 1. **sem_otime** is the time of the last *semop*(2) operation, and **sem_ctime** is the time of the last *semctl*(2) operation that changed a member of the above structure.

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

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

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

Semaphore Operation Permissions

In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

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

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

The effective user ID of the process is super-user.

The effective user ID of the process matches **sem_perm.[c]uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.

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

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

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier

A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as *shmid_ds* and contains the following members:

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

shm_perm is an *ipc_perm* structure that specifies the shared memory operation permission (see below). This structure includes the following members:

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

shm_segsz specifies the size of the shared memory segment. **shm_cpid** is the process id of the process that created the shared memory identifier. **shm_lpid** is the process id of the last process that performed a *shmop*(2) operation. **shm_nattch** is the number of processes that currently have this segment attached. **shm_atime** is the time of the last *shmat* operation, **shm_dtime** is the time of the last *shmdt* operation, and **shm_ctime** is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

Shared Memory Operation Permissions

In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

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

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

The effective user ID of the process is super-user.

The effective user ID of the process matches **shm_perm.[c]uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.

The effective user ID of the process does not match **shm_perm.[c]uid** and the effective group ID of the process matches **shm_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The effective user ID of the process does not match **shm_perm.[c]uid** and the effective group ID of the process does not match **shm_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Sockets and Address Families

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

Sockets are typed according to their communications properties. These properties 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.

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.

Super-user

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

Tty Group ID

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

SEE ALSO

intro(3), perror(3)

LIST OF SYSTEM CALLS

Name	*Appears on Page*	*Description*
_exit	exit(2)	terminate a process
accept	accept(2)	accept a connection on a socket
access	access(2)	determine accessibility of file
acct	acct(2)	turn accounting on or off
adjtime	adjtime(2)	correct the time to allow synchronization of the system cl
async_daemon	nfssvc(2)	NFS daemons

bind	bind(2)	bind a name to a socket
brk	brk(2)	change data segment size
chdir	chdir(2)	change current working directory
chmod	chmod(2)	change mode of file
chown	chown(2)	change owner and group of a file
chroot	chroot(2)	change root directory
close	close(2)	delete a descriptor
connect	connetc(2)	initiate a connection on a socket
creat	creat(2)	create a new file
dup	dup(2)	duplicate a descriptor
dup2	dup(2)	duplicate a descriptor
execve	execve(2)	execute a file
fchmod	chmod(2)	change mode of file
fchown	chown(2)	change owner and group of a file
fcntl	fcntl(2)	file control
flock	flock(2)	apply or remove an advisory lock on an open file
fork	fork(2)	create a new process
fstat	stat(2)	get file status
fsync	fsync(2)	synchronize a file's in-core state with that on disk
ftruncate	truncate(2)	truncate a file to a specified length
getdirentries	getdirentries(2)	gets directory entries in a filesystem independent format
getdomainname	getdomainname(2)	get name of current domain
getdtablesize	getdtablesize(2)	get descriptor table size
getegid	getgid(2)	get group identity
geteuid	getuid(2)	get effective user identity
getgid	getgid(2)	get group identity
getgroups	getgroups(2)	get group access list
gethostid	gethostid(2)	get unique identifier of current host
gethostname	gethostname(2)	get name of current host
getitimer	getitimer(2)	get value of interval timer
getpagesize	getpagesizename(2)	get system page size
getpeername	getpeername(2)	get name of connected peer
getpgrp	setpgrp(2V)	set and/or return the process group of a process
getpid	getpid(2)	get parent process identification
getppid	getpid(2)	get process identification
getpriority	getpriority(2)	get program scheduling priority
getrlimit	getrlimit(2)	control maximum system resource consumption
getrusage	getrusage(2)	get information about resource utilization
getsockname	getsockname(2)	get socket name
getsockopt	getsockopt(2)	get options on sockets
gettimeofday	gettimeofday(2)	get date and time
getuid	getuid(2)	get user identity
ioctl	ioctl(2)	control device
kill	kill(2)	send signal to a process
killpg	killpg(2)	send signal to a process group
link	link(2)	make a hard link to a file
listen	listen(2)	listen for connections on a socket
lseek	lseek(2)	move read/write pointer
lstat	stat(2)	get file status
mkdir	mkdir(2)	make a directory file
mknod	mknod(2)	make a special file
mmap	mmap(2)	map or unmap pages of memory
mount	mount(2)	mount file system

msgctl	msgctl(2)	message control operations
msgget	msgget(2)	get message queue
msgop	msgop(2)	message operations
msgrcv	msgop(2)	message operations
msgsnd	msgop(2)	message operations
munmap	munmap(2)	map or unmap pages of memory
nfssvc	nfssvc(2)	NFS daemons
open	open(2V)	open or create a file for reading or writing
pipe	pipe(2)	create an interprocess communication channel
profil	profil(2)	execution time profile
ptrace	ptrace(2)	process trace
quotactl	quotactl(2)	manipulate disk quotas
read	read(2V)	read input
readlink	readlink(2)	read value of a symbolic link
readv	read(2V)	read input
reboot	reboot(2)	reboot system or halt processor
recv	recv(2)	receive a message from a socket
recvfrom	recv(2)	receive a message from a socket
recvmsg	recv(2)	receive a message from a socket
rename	rename(2)	change the name of a file
rmdir	rmdir(2)	remove a directory file
sbrk	brk(2)	change data segment size
select	select(2)	synchronous I/O multiplexing
semctl	semctl(2)	semaphore control operations
semget	semget(2)	get set of semaphores
semop	semop(2)	semaphore operations
send	send(2)	send a message from a socket
sendmsg	send(2)	send a message from a socket
sendto	send(2)	send a message from a socket
setdomainname	getdomainname(2)	set name of current domain
setgroups	getgroups(2)	set group access list
sethostname	gethostname(2)	set name of current host
setitimer	getitimer(2)	set value of interval timer
setpgrp	setpgrp(2V)	set and/or return the process group of a process
setpriority	getpriority(2)	set program scheduling priority
setregid	setregid(2)	set real and effective group IDs
setreuid	setreuid(2)	set real and effective user IDs
setrlimit	getrlimit(2)	control maximum system resource consumption
setsockopt	getsockopt(2)	set options on sockets
settimeofday	gettimeofday(2)	set date and time
shmat	shmop(2)	shared memory operations
shmctl	shmctl(2)	shared memory control operations
shmdt	shmop(2)	shared memory operations
shmget	shmget(2)	get shared memory segment
shmop	shmop(2)	shared memory operations
shutdown	shutdown(2)	shut down part of a full-duplex connection
sigblock	sigblock(2)	block signals
sigpause	sigpause(2)	atomically release blocked signals and wait for interrupt
sigsetmask	sigsetmask(2)	set current signal mask
sigstack	sigstack(2)	set and/or get signal stack context
sigvec	sigvec(2)	software signal facilities
socket	socket(2)	create an endpoint for communication
socketpair	socketpair(2)	create a pair of connected sockets

stat	stat(2)	get file status
statfs	statfs(2)	get file system statistics
swapon	swapon(2)	add a swap device for interleaved paging/swapping
symlink	symlink(2)	make symbolic link to a file
sync	sync(2)	update super-block
syscall	syscall(2)	indirect system call
tell	lseek(2)	locate read/write pointer
truncate	truncate(2)	truncate a file to a specified length
umask	umask(2)	set file creation mode mask
uname	uname(2V)	get name of current UNIX system
unlink	unlink(2)	remove directory entry
unmount	umount(2)	remove a file system
utimes	utimes(2)	set file times
vadvise	vadvise(2)	give advice to paging system
vfork	vfork(2)	spawn new process in a virtual memory efficient way
vhangup	vhangup(2)	virtually "hangup" the current control terminal
wait	wait(2)	wait for process to terminate or stop
wait3	wait(2)	wait for process to terminate or stop
write	write(2V)	write output
writev	write(2V)	write output

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*, is used to read and write data to and from the socket which connected to this one; it is not used to accept more connections. The original socket *s* remains open for accepting further connections.

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

path points to a path name naming a file. *access* checks the named file 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 (that is, 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.

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.

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 of *path* is not a directory.
EINVAL	*path* contains a byte with the high-order bit set.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	The file named 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*.
EROFS	The file named by *path* is on a read-only file system and write access was requested.
ETXTBSY	The file named by *path* is a pure procedure (shared text) file that is being executed and write access was requested.
EACCES	Permission bits of the file mode do not permit the requested access to the file named by *path*.
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

acct is used to enable or disable the process accounting. If process accounting is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal; see *exit*(2) and *sigvec*(2). The effective user ID of the calling process must be super-user to use this call.

name points to a path name naming the accounting file. The accounting file format is given in *acct*(5).

The accounting routine is enabled if *name* is non-zero and no errors occur during the system call. It is disabled if *name* is zero and no errors occur during the system call.

If accounting is already turned on, and a successful *acct* call is made with a non-zero *name*, all subsequent accounting records will be written to the new accounting file.

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

The value −1 is returned if an error occurs, and external variable *errno* is set to indicate the cause of the error. Otherwise the value 0 is returned.

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 of *file* is not a directory.
EINVAL	*file* contains a character with the high-order bit set.
EINVAL	Support for accounting was not configured into the system.
ENAMETOOLONG	The length of a component of *file* exceeds 255 characters, or the length of *file* exceeds 1023 characters.
ENOENT	The named file does not exist.
EACCES	Search permission is denied for a component of the path prefix of *file*.
EACCES	The file referred to by *file* is not a regular file.
ELOOP	Too many symbolic links were encountered in translating the path name.
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 adjusts the system's notion of the current time, as returned by *gettimeofday*(2), advancing or retarding it by the amount of time specified in the **struct timeval** **delta*.

The adjustment is effected by speeding up (if **delta* is positive) or slowing down (if **delta* is negative) the system's clock by a fixed percentage, currently 10%. 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.

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

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

If *olddelta* is a NULL pointer, the corresponding information will not be returned.

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

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

The adjustment value will be silently rounded to the resolution of the system clock.

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	*delta* or *olddelta* points outside the process's allocated address space, or *olddelta* points to a region of the process' allocated address space which is not writable.
EPERM	The process's effective user ID is not that of the super-user.

SEE ALSO

settimeofday(2), date(1)

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 the name pointed to by *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 of the path name in *name* is not a directory.
EINVAL	The path name in *name* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of the path name in *name* exceeds 255 characters, or the length of the path name in *name* exceeds 1023 characters.
ENOENT	A component of the path prefix of the path name in *name* does not exist.
EACCES	Search permission is denied for a component of the path prefix of the path name in *name*.
ELOOP	Too many symbolic links were encountered in translating the path name in *name*.
EIO	An I/O error occurred while making the directory entry or allocating the inode.
EROFS	The inode would reside on a read-only file system.
EISDIR	A null path name was specified.

SEE ALSO

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

NAME

brk, sbrk – change data segment size

SYNOPSIS

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

caddr_t brk(addr)
caddr_t addr;

caddr_t sbrk(incr)
int incr;
```

DESCRIPTION

Brk

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.

Sbrk

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* normally returns the current value of the break, but –1 if it 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 – change current working directory

SYNOPSIS

chdir(path)
char *path;

DESCRIPTION

path points to the path name of a directory. *chdir* causes this directory to become the current working directory, the starting point for path names not beginning with /.

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

RETURN VALUE

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

ERRORS

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

ENOTDIR	A component of the path prefix of *path* is not a directory.
ENOTDIR	The file named by *path* is not a directory.
EINVAL	*path* contains a byte with the high-order bit set.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	The directory referred to by *path* does not exist.
ELOOP	Too many symbolic links were encountered in translating *path*.
EACCES	Search permission is denied for a component of the path prefix of *path*.
EACCES	Search permission is denied for the directory referred to by *path*.
EFAULT	*path* points outside the process's allocated address space.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

chroot(2)

NAME

chmod, fchmod – change mode of file

SYNOPSIS

#include /usr/include/sys/stat.h

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:

S_ISUID	04000 set user ID on execution
S_ISGID	02000 set group ID on execution
S_ISVTX	01000 save text image after execution (sticky bit)
S_IREAD	00400 read by owner
S_IWRITE	00200 write by owner
S_IEXEC	00100 execute (search on directory) by owner
	00070 read, write, execute (search) by group
	00007 read, write, execute (search) by others

These bit patterns are defined in **/usr/include/sys/stat.h.**

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user and the process attempts to set the set group ID bit on a file owned by a group which is not in its group access list, mode bit 02000 (set group ID on execution) is cleared.

If an executable file is set up for sharing (this is the default) then mode 01000 (save text image after execution) prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. If the effective user ID of the process is not super-user, this bit is cleared.

If a user other than the super-user writes to a file, the set user ID and set group ID bits are turned off. 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 of *path* is not a directory.
EINVAL	*path* contains a byte with the high-order bit set.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	The file referred to by *path* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *path*.
ELOOP	Too many symbolic links were encountered in translating *path*.
EPERM	The effective user ID does not match the owner of the file and the effective user ID is

not the super-user.

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

EROFS The file referred to by *path* 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.

EROFS The file referred to by *fd* resides on a read-only file system.

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

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

FILES

/usr/include/sys/stat.h

SEE ALSO

open(2V), chown(2), stat(2), sticky(8)

NAME

chown, fchown – change owner and group of a file

SYNOPSIS

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

fchown(fd, owner, group)
int fd, owner, group;

DESCRIPTION

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

fchown is particularly useful when used in conjunction with the file locking primitives (see *flock*(2)).

If *owner* or *group* is specified as –1, the corresponding ID of the file is not changed.

If a process whose effective user ID is not super-user successfully changes the group ID of a file, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

If the final component of *path* is a symbolic link, the ownership and group of the symbolic link is changed, not the ownership and group of the file or directory to which it points.

RETURN VALUE

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

ERRORS

chown will fail and the file will be unchanged if:

ENOTDIR	A component of the path prefix of *path* is not a directory.
EINVAL	*path* contains a byte with the high-order bit set.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	The file referred to by *path* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *path*.
ELOOP	Too many symbolic links were encountered in translating *path*.
EPERM	The user ID specified by *owner* is not the current owner ID of the file, or the group ID specified by *group* is not the current group ID fo the file and is not in the process' group access list, and the effective user ID is not the super-user.
EROFS	The file referred to by *path* 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 user ID specified by *owner* is not the current owner ID of the file, or the group ID specified by *group is not the current group* group access list, and the effective user ID is not the super-user.

EROFS	The file referred to by *fd* 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(2), flock(2)

NAME

chroot – change root directory

SYNOPSIS

chroot(dirname)
char *dirname;

DESCRIPTION

dirname points to a path name naming a directory. *chroot* causes this directory to become the root directory, the starting point for path names beginning with /. The current working directory is unaffected by this call. This root directory setting is inherited across *execve*(2) and by all children of this process created with *fork*(2) calls.

The effective user ID of the process must be super-user to change the root directory.

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

In order for a directory to become the root 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 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 prefix of *dirname* is not a directory.
ENOTDIR	The file referred to by *dirname* is not a directory.
EINVAL	*dirname* contains a byte with the high-order bit set.
ENAMETOOLONG	The length of a component of *dirname* exceeds 255 characters, or the length of *dirname* exceeds 1023 characters.
ENOENT	The directory referred to by *dirname* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *dirname*.
EACCES	Search permission is denied for the directory referred to by *dirname*.
ELOOP	Too many symbolic links were encountered in translating *dirname*.
EPERM	The effective user ID is not super-user.
EFAULT	*dirname* 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 *flock*(2) for further information).

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.

Close unmaps pages mapped through this file 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.
EINTR	A read from a slow device was interrupted before any data arrived by the delivery of a signal.

SEE ALSO

accept(2), flock(2), open(2V), pipe(2), socket(2), socketpair(2), execve(2), fcntl(2), mmap(2), munmap(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 permanently specifies the peer to which datagrams are to be sent; if it 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.

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.
EWOULDBLOCK	The socket is non-blocking and the and the connection cannot be completed immediately. It is possible to *select*(2) the socket while it is connecting by selecting it for writing.
EINTR	A read from a slow device was interrupted before any data arrived by the delivery of a signal.

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 of the path name in *name* is not a directory.
EINVAL	The path name in *name* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of the path name in *name* exceeds 255 characters, or the length of the entire path name in *name* exceeds 1023 characters.
ENOENT	A component of the path prefix of the path name in *name* does not exist.
ENOENT	The socket referred to by the path name in *name* does not exist.
EACCES	Search permission is denied for a component of the path prefix of the path name in

name.

ELOOP	Too many symbolic links were encountered in translating the path name in *name*.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

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

NAME

creat – create a new file

SYNOPSIS

creat(name, mode)
char *name;
int mode;

DESCRIPTION

This interface is made obsolete by

creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *name*. If the file did not exist, it is given mode *mode*, as modified by the process's mode mask (see *umask*(2)). Also see *chmod*(2) for the construction of the *mode* argument.

If the file exists, its mode and owner remain unchanged, but it is truncated to 0 length. Otherwise, the file's owner ID is set to the effective user ID of the process, the file's group ID is set to the group ID of the directory in which the file is created, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

The "save text image after execution" bit of the mode is cleared. See *chmod*(2).

Upon successful completion, the file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *execve* system calls. See *fcntl*(2).

NOTES

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

RETURN VALUE

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

ERRORS

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

ENOTDIR	A component of the path prefix of *name* is not a directory.
EINVAL	*name* contains a byte with the high-order bit set.
ENAMETOOLONG	The length of a component of *name* exceeds 255 characters, or the length of *name* exceeds 1023 characters.
ENOENT	A component of the path prefix of *name* does not exist.
ELOOP	Too many symbolic links were encountered in translating *name*.
EACCES	Search permission is denied for a component of the path prefix of *name*.
EACCES	The file referred to by *name* does not exist and the directory in which it is to be created is not writable.
EACCES	The file referred to by *name* exists, but it is unwritable.
EISDIR	The file referred to by *name* is a directory.
EMFILE	There are already too many files open.
ENFILE	The system file table is full.
ENOSPC	The directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.

ENOSPC	There are no free inodes on the file system on which the file is being created.
EDQUOT	The directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
EDQUOT	The user's quota of inodes on the file system on which the file is being created has been exhausted.
EROFS	The file referred to by *name* resides, or would reside, on a read-only file system.
ENXIO	The file is a character special or block special file, and the associated device does not exist.
ETXTBSY	The file is a pure procedure (shared text) file that is being executed.
EIO	An I/O error occurred while making the directory entry or allocating the inode.
EFAULT	*name* points outside the process's allocated address space.
EOPNOTSUPP	The file was a socket (not currently implemented).

SEE ALSO

open(2), write(2V), close(2), chmod(2), fcntl(2), umask(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.

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.

The new descriptor has the following in common with the original:

It refers to the same object that the old descriptor referred to.

It uses the same file pointer as the old descriptor. (i.e., both file descriptors share one file pointer).

It has the same access mode (read, write or read/write) as the old descriptor.

Thus if *newd* and *oldd* are duplicate references to an open file, *read*(2V), *write*(2V) 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*(2V) call. The close-on-exec flag on the new file descriptor is unset.

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

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, whose name is pointed to by *path*, 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* [arg]". When an interpreter file is *execve*'d, the system *execve*'s the specified *interpreter*. If the optional *arg* is specified, it becomes the first argument to the *interpreter*, and the name of the originally *execve*'d file becomes the second argument; otherwise, the name of the originally *execve*'d file becomes the first argument. The original argument are shifted over to become the subsequent arguments. The zeroth argument, normally the name of the *execve*'d file, is left unchanged.

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 which are not directly arguments to the command (see *environ*(5V)).

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) and *fcntl*(2)). Descriptors which remain open are unaffected by *execve*.

Ignored signals remain ignored across an *execve*, but signals that are caught are reset to their default values. 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 a *real* user ID and group ID and an *effective* user ID and group ID. The *real* ID identifies the person using the system; the *effective* ID determines their access privileges. *Execve* changes the effective user or group ID to the owner or group of the executed file if the file has the "set-user-ID" or "set-group-ID" modes. The *real* user ID and group ID are not affected.

The shared memory segments attached to the calling process will not be attached to the new process (see *shmop*(2)).

Profiling is disabled for the new process; see *profil*(2).

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)
semadj values	see*semop*(2)
working directory	see *chdir*(2)
root directory	see *chroot*(2)
control terminal	see *tty*(4)

trace flag	see *ptrace* (2) request 0)
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*(5V) 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 of the new process file is not a directory.
EINVAL	*name* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of *name* exceeds 255 characters, or the length of *name* exceeds 1023 characters.
ENOENT	One or more components of the path prefix of the new process file does not exist.
ENOENT	The new process file does not exist.
ELOOP	Too many symbolic links were encountered in translating *name*.
EACCES	Search permission is denied for a component of the new process file's path prefix.
EACCES	The new process file is not an ordinary file.
EACCES	Execute permission is denied for the new process file.
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 file requires more virtual memory than is allowed by the imposed maximum (*getrlimit*(2)).
[E2BIG]	The number of bytes in the new process file's argument list is larger than the system-imposed limit. The limit in the system as released is 10240 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	*Name*, *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 user ID is super-user, then the program has some of the powers of a super-user as well.

SEE ALSO

exit(2), fork(2), execl(3), environ(5V)

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 will 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), 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 performs a variety of functions on open descriptors. The argument *fd* is an open 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*.

References the same object 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 descriptors share the same file status flags).

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

F_GETFD — Get the close-on-exec flag associated with the 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, see *fcntl*(5) for their definitions.

F_SETFL — Set descriptor status flags, see *fcntl*(5) for their definitions.

F_GETLK — Get a description of the first lock which would block the lock specified in the *flock* structure pointed to by *arg*. The information retrieved overwrites the information in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK — Set or clear an advisory record lock according to the *flock* structure pointed to by *arg*. F_SETLK is used to establish shared (F_RDLCK) and exclusive (F_WRLCK) locks, or to remove either type of lock (F_UNLCK). If the specified lock cannot be applied, *fcntl* will return with an error value of -1.

F_SETLKW — This *cmd* is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the requesting process will sleep until the lock may be applied.

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 SIGIO facilities are enabled by setting the FASYNC flag with F_SETFL.

NOTES

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access (i.e., processes may still access files without using advisory locks, possibly resulting in inconsistencies).

The record locking mechanism allows two types of locks: shared locks (F_RDLCK) and exclusive locks (F_WRLCK). More than one process may hold a shared lock for a particular segment of a file at any given time, but multiple exclusive, or both shared and exclusive, locks may not exist simultaneously on any segment.

In order to claim a shared lock, the descriptor must have been opened with read access. The descriptor on which an exclusive lock is being placed must have been opened with write access.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type with a *cmd* of F_SETLK or F_SETLKW; the previous lock will be released and the new lock applied (possibly after other processes have gained and released the lock).

If the *cmd* is F_SETLKW and the requested lock cannot be claimed immediately (e.g., another process holds an exclusive lock that partially or completely overlaps the current request) then the calling process will block until the lock may be acquired. Processes blocked awaiting a lock may be awakened by signals.

Care should be taken to avoid deadlock situations in applications in which multiple processes perform blocking locks on a set of common records.

The record that is to be locked or unlocked is described by the *flock* structure, which is defined in *<fcntl.h>* as follows:

```
struct flock {
        short   l_type;         /* F_RDLCK, F_WRLCK, or F_UNLCK */
        short   l_whence;       /* flag to choose starting offset */
        long    l_start;        /* relative offset, in bytes */
        long    l_len;          /* length, in bytes; 0 means lock to EOF */
        short   l_pid;          /* returned with F_GETLK */
};
```

The *flock* structure describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), and size (*l_len*) of the segment of the file to be affected. *L_whence* must be set to 0, 1, or 2 to indicate that the relative offset will be measured from the start of the file, current position, or end-of-file, respectively. The process id field (*l_pid*) is only used with the F_GETLK *cmd* to return the description of a lock held by another process.

Locks may start and extend beyond the current end-of-file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end-of-file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *l_start* set to zero (0), the entire file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take affect. All locks associated with a file for a given process are removed when the file is closed or the process terminates. Locks are not inherited by the child process in a *fork*(2) system call.

In order to maintain consistency in the network case, data must not be cached on client machines. For this reason, file buffering for an NFS file is turned off when the first lock is attempted on the file. Buffering will remain off as long as the file is open. Programs that do I/O buffering in the user address space, however, may have inconsistent results (the standard I/O package, for instance, is a common source of unexpected buffering).

The advisory record locking capabilities of *fcntl* are implemented throughout the network by the **network lock daemon**; see *lockd*(8C). If the file server crashes and is rebooted, the lock daemon will attempt to recover all locks that were associated with that server. If a lock cannot be reclaimed, the process that held the lock will be issued a SIGLOST signal.

RETURN VALUE

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

F_DUPFD	A new descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_GETFL	Value of flags.

F_GETOWN	Value of 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	*Fd* is not a valid open descriptor.
EMFILE	*Cmd* is F_DUPFD and the maximum allowed number of descriptors are currently open.
EINVAL	*Cmd* is F_DUPFD and *arg* is negative or greater than the maximum allowable number (see *getdtablesize*(2)).
EFAULT	*Cmd* is F_GETLK, F_SETLK, or F_SETLKW and *arg* points to an invalid address.
EINVAL	*Cmd* is F_GETLK, F_SETLK, or F_SETLKW and the data *arg* points to is not valid.
EBADF	*Cmd* is F_SETLK or F_SETLKW and the process does not have the appropriate read or write permissions on the file.
EAGAIN	*Cmd* is F_SETLK, the lock type (*l_type*) is F_RDLCK (shared lock), and the segment of the file to be locked already has an exclusive lock held by another process. This error will also be returned if the lock type is F_WRLCK (exclusive lock) and another process already has the segment locked with either a shared or exclusive lock.
EINTR	*Cmd* is F_SETLKW and a signal interrupted the process while it was waiting for the lock to be granted.
ENOLCK	*Cmd* is F_SETLK or F_SETLKW and there are no more file lock entries available.

SEE ALSO

close(2), execve(2), getdtablesize(2), open(2V), sigvec(2), lockf(3), lockd(8C)

BUGS

File locks obtained through the *fcntl* mechanism do not interact in any way with those acquired via *flock*(2). They do, however, work correctly with the exclusive locks claimed by *lockf*(3).

F_GETLK returns F_UNLCK if the requesting process holds the specified lock. Thus, there is no way for a process to determine if it is still holding a specific lock after catching a SIGLOST signal.

In a network environment, the value of *l_pid* returned by F_GETLK is next to useless.

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, the *operation* should be LOCK_UN.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access (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. More than one process may hold a shared lock for a file at any given time, but multiple exclusive, or both shared and exclusive, locks may not exist simultaneously on a file.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; the previous lock will be 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 block 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 on success, −1 on error, with an error code stored in *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.
EOPNOTSUPP	The argument *fd* refers to an object other than a file.

SEE ALSO

open(2V), close(2), dup(2), execve(2), fcntl(2), fork(2), lockf(3)

BUGS

Locks obtained through the *flock* mechanism are known only within the system on which they were placed. Thus, multiple clients may successfully acquire exclusive locks on the same remote file. If this behavior is not explicitly desired, the *fcntl*(2) or *lockf*(3) system calls should be used instead; these make use of the services of the **network lock manager** (see lockd(8C)).

NAME

fork – create a new process

SYNOPSIS

pid = fork()
int pid;

DESCRIPTION

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

The child process has a unique process ID.

The child process has a different parent process ID (that is, the process ID of the parent process).

The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that 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 determined when the system is generated.
EAGAIN	The system-imposed limit on the total number of processes under execution by a single user would be exceeded. This limit is determined when the system is generated.
ENOMEM	There is insufficient swap space for the new process.

SEE ALSO

execve(2), wait(2)

NAME

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

SYNOPSIS

fsync(fd)
int fd;

DESCRIPTION

fsync moves all modified data and attributes of *fd* to a permanent storage device: all in-core modified copies of buffers for the associated file have been written to a disk when the call returns. Note that this is different than *sync*(2) which schedules disk I/O for all files (as though an *fsync* had been done on all files) but returns before the I/O completes.

fsync should be used by programs which require a file to be in a known state; for example, a program which contains a simple transaction facility might use it to ensure that all modifications to a file or files caused by a transaction were recorded on disk.

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

SEE ALSO

sync(2), sync(8), cron(8)

BUGS

The current implementation of this call is expensive for large files.

NAME

getdirentries – gets directory entries in a filesystem independent format

SYNOPSIS

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

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

DESCRIPTION

getdirentries attempts to put directory entries from the directory referenced by the file descriptor *fd* into the buffer pointed to by *buf*, in a filesystem independent format. Up to *nbytes* of data will be transferred. *nbytes* must be greater than or equal to the block size associated with the file, see *stat(2)*. Sizes less than this may cause errors on certain filesystems.

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

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

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

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

Upon return, the actual number of bytes transferred is returned. The current position pointer associated with *fd* is set to point to the next block of entries. The pointer is not necessarily incremented by the number of bytes returned by *getdirentries*. If the value returned is zero, the end of the directory has been reached. The current position pointer may be set and retrieved by *lseek(2)*. *getdirentries* writes the position of the block read into the location pointed to by *basep*. It is not safe to set the current position pointer to any value other than a value previously returned by *lseek(2)* or a value previously returned in the location pointed to by *basep* or zero.

RETURN VALUE

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

ERRORS

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

EBADF	*fd* is not a valid file descriptor open for reading.
EFAULT	Either *buf* or *basep* point outside the allocated address space.
EIO	An I/O error occurred while reading from or writing to the file system.
EINTR	A read from a slow device was interrupted before any data arrived by the delivery of a signal.

SEE ALSO

open(2V), lseek(2)

NAME

getdomainname, setdomainname – get/set name of current domain

SYNOPSIS

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

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

DESCRIPTION

Getdomainname returns the name of the domain for the current processor, as previously set by *setdomainname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

Setdomainname sets the domain 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.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only the yellow pages service makes use of domains.

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* parameter gave an invalid address.
EPERM	The caller was not the super-user. This error only applies to **setdomainname**.

BUGS

Domain names are limited to 255 characters.

NAME

getdtablesize – get descriptor table size

SYNOPSIS

nds = getdtablesize()
int nds;

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)

NAME

getgid, getegid – get group identity

SYNOPSIS

```
gid = getgid()
int gid;

egid = getegid()
int egid;
```

DESCRIPTION

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

The real group ID is specified at login time.

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

SEE ALSO

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

NAME

getgroups, setgroups – get or set group access list

SYNOPSIS

#include <sys/param.h>

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

setgroups(ngroups, gidset)
int ngroups, *gidset;

DESCRIPTION

Getgroups

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 entries placed in the *gidset* array. No more than NGROUPS, as defined in *<sys/param.h>*, will ever be returned.

Setgroups

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

Getgroups

A return value of greater than zero indicates the number of entries placed in the *gidset* array. A return value of –1 indicates that an error occurred, and the error code is stored in the global variable *errno*.

Setgroups

A 0 value is returned on success, –1 on error, with a error code stored in *errno*.

ERRORS

Either call fails if:

EFAULT	The address specified for *gidset* is outside the process address space.

getgroup fails if:

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

setgroups fails if:

EPERM	The caller is not the super-user.

SEE ALSO

initgroups(3)

NAME

gethostid – get unique identifier of current host

SYNOPSIS

hostid = gethostid()
long hostid;

DESCRIPTION

Gethostid returns the 32-bit identifier for the current host, which should be unique across all hosts. On the Sun, this number is taken from the CPU board's ID PROM.

SEE ALSO

hostid(1)

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 was not the super-user. Note that this error only applies to *sethostname*.

SEE ALSO

gethostid(2)

BUGS

Host names are limited to 31 characters.

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*, while the *setitimer* call sets the value of a timer (optionally returning the previous value of the timer).

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

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

If *it_value* is 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.

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).

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* or *ovalue* parameter specified a bad address.

EINVAL A *value* parameter specified a time which was too large to be handled.

SEE ALSO

sigvec(2), gettimeofday(2)

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).

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

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

BUGS

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

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. If the specified value is less than –20, a value of –20 is used; if it is greater than 20, a value of 20 is used. 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, and neither the effective nor the real user ID of the process executing the *setpriority* was super-user.
EACCES	The call to *setpriority* would have changed a process' priority to a value lower than its current value, and the effective user ID of the process executing the call was not that of the super-user.

SEE ALSO

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

BUGS

It is not possible for the process executing *setpriority*() to lower any other process down to its current priority, without requiring super-user privileges.

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 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 *brk* or *sbrk* call will fail if the data space limit is reached, or the process will be killed when the stack limit is reached (since the stack cannot be extended, there is no way to send a signal!).

A file I/O operation which 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)

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>

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

DESCRIPTION

getrusage returns information about 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 */
        int     ru_maxrss;
        int     ru_ixrss;       /* integral shared text memory size */
        int     ru_idrss;       /* integral unshared data size */
        int     ru_isrss;               /* integral unshared stack size */
        int     ru_minflt;              /* page reclaims */
        int     ru_majflt;              /* page faults */
        int     ru_nswap;               /* swaps */
        int     ru_inblock;             /* block input operations */
        int     ru_oublock;             /* block output operations */
        int     ru_msgsnd;              /* messages sent */
        int     ru_msgrcv;              /* messages received */
        int     ru_nsignals;            /* signals received */
        int     ru_nvcsw;               /* voluntary context switches */
        int     ru_nivcsw;              /* involuntary context switches */
};
```

The fields are interpreted as follows:

ru_utime — the total amount of time spent executing in user mode. Time is given in seconds:microseconds.

ru_stime — the total amount of time spent in the system executing on behalf of the process(es). Time is given in seconds:microseconds.

ru_maxrss — the maximum resident set size utilized. Size is given in pages (the size of a page, in bytes, is given by the *getpagesize*(2) system call).

ru_ixrss — an "integral" value indicating the amount of memory used by the text segment which was also shared among other processes. This value is expressed in units of pages * clock ticks (1 tick = 1/50 second). The value 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. The value is given in pages * clock ticks.

ru_isrss — an integral value of the amount of unshared memory residing in the stack segment of a process. The value is given in pages * clock ticks.

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 which required I/O activity.
ru_nswap	the number of times a process was "swapped" out of main memory.
ru_inblock	the number of times the file system had to perform input.
ru_outblock	the number of times the file system had to perform output.
ru_msgsnd	the number of messages sent over sockets.
ru_msgrcv	the number of messages received from sockets.
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

getrusage will fail if:

EINVAL	The *who* parameter is not a valid value.
EFAULT	The address specified by the *rusage* argument is not in a valid portion of the process's address space.

SEE ALSO

gettimeofday(2), wait(2)

BUGS

There is no way to obtain information about a child process which 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), getpeername(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 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; see *socket*(2). Options at other protocol levels vary in format and name, consult the appropriate entries in (4P).

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.
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

socket(2), getprotoent(3N)

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."

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.

If *tzp* is a zero pointer, the timezone information is not returned or set.

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)

BUGS

Time is never correct enough to believe the microsecond values. There should a mechanism by which, at least, local clusters of systems might synchronize their clocks to millisecond granularity.

NAME

getuid, geteuid – get user identity

SYNOPSIS

```
uid = getuid()
int uid;

euid = geteuid()
int euid;
```

DESCRIPTION

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

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

SEE ALSO

getgid(2), setreuid(2)

NAME

ioctl – control device

SYNOPSIS

#include <sys/ioctl.h>

ioctl(d, request, argp)
int d, 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.

If no error has occurred (using a STANDARD device driver), a value of 0 is returned.

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), mtio(4), tty(4)

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.

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*.
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.

SEE ALSO

getpid(2), getpgrp(2V), 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 have an effective user ID of 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 were found in the specified 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(2V), sigvec(2)

NAME

link – make a hard link to a file

SYNOPSIS

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

DESCRIPTION

name1 points to a path name naming an existing file. *name2* points to a path name naming a new directory entry to be created. A hard link to the first file is created; the link has the name pointed to by *name2*. The file named by *name1* must exist.

With hard links, both files must be on the same file system. Unless the caller is the super-user, the file named by *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 the path prefix of *name1* or *name2* is not a directory.
EINVAL	*name1* or *name2* contains a byte with the high-order bit set.
ENAMETOOLONG	The length of a component of *name1* or *name2* exceeds 255 characters, or the length of *name1* or *name2* exceeds 1023 characters.
ENOENT	A component of the path prefix of *name1* or *name2* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *name1* or *name2*.
EACCES	The requested link requires writing in a directory for which write permission is denied.
ELOOP	Too many symbolic links were encountered in translating *name1* or *name2*.
ENOENT	The file referred to by *name1* does not exist.
EEXIST	The link referred to 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 path names 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 backlog for incoming connections is 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 will receive an error with an indication of ECONNREFUSED.

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

lseek, tell – move read/write pointer

SYNOPSIS

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

pos = lseek(d, offset, whence)
long pos;
int d;
long 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.

The obsolete function *tell(fildes)* is identical to *lseek(fildes, 0L, L_INCR)*.

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, a non-negative (long) integer, 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(2V)

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 of *path* is not a directory.
EINVAL	*path* contains a byte with the high-order bit set.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	A component of the path prefix of *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*.
EROFS	The file referred to by *path* resides on a read-only file system.
EEXIST	The file referred to by *path* exists.
ENOSPC	The directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.
ENOSPC	The new directory cannot be created because there is no space left on the file system which will contain the directory.
ENOSPC	There are no free inodes on the file system on which the file is being created.
EDQUOT	The directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
EDQUOT	The new directory cannot be created because the user's quota of disk blocks on the file system which will contain the directory has been exhausted.
EDQUOT	The user's quota of inodes on the file system on which the file is being created has been exhausted.
EIO	An I/O error occurred while reading from or writing to the file system.
EFAULT	*Path* points outside the process's allocated address space.

SEE ALSO

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

NAME

mknod – make a special file

SYNOPSIS

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

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

DESCRIPTION

mknod creates a new file named by the path name pointed to by *path*. The mode of the new file (including file type bits) is initialized from *mode*. The values of the file type bits which are permitted are:

```
#define  S_IFCHR    0020000    /* character special */
#define  S_IFBLK    0060000    /* block special */
#define  S_IFREG    0100000    /* regular */
#define  S_IFIFO    0010000    /* FIFO special */
```

Values of *mode* other than those above are undefined and should not be used.

The protection part of the mode is modified by the process's mode mask (see *umask*(2)).

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the group ID of the parent directory.

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 for file types other than FIFO special.

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 fails and the file mode remains unchanged if:

ENOTDIR — A component of the path prefix of *path* is not a directory.

EINVAL — *path* contains a character with the high-order bit set.

ENAMETOOLONG — The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

ENOENT — A component of the path prefix of *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*.

EPERM — An attempt was made to create a file of type other than FIFO special and the process's effective user ID is not super-user.

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

EISDIR — The specified *mode* would have created a directory.

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

ENOSPC — There are no free inodes on the file system on which the file is being created.

EDQUOT — The directory in which the entry for the new file is being placed cannot be extended

because the user's quota of disk blocks on the file system containing the directory has been exhausted.

EDQUOT The user's quota of inodes on the file system on which the node is being created has been exhausted.

EROFS The file referred to by *path* resides on a read-only file system.

EEXIST The file referred to by *path* exists.

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

SEE ALSO

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

NAME

mmap, munmap – map or unmap pages of memory

SYNOPSIS

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

mmap(addr, len, prot, share, fd, off)
caddr_t addr; int len, prot, share, fd; off_t off;

munmap (addr, len)
caddr_t addr; int len;

DESCRIPTION

Mmap

mmap maps pages of memory from the memory device associated with the file *fd* into the address space of the calling process, one page at a time. Pages are mapped from the memory device, beginning at *off*, and into the caller's address space, beginning at *addr*, and continuing for *len* bytes. *fd* is a file descriptor obtained by opening the device from which to map pages. Only character-special devices are currently supported.

share specifies whether modifications made to mapped-in copies of pages are to be kept "private" or are to be "shared" with other references. Currently, it must be set to MAP_SHARED.

The parameter *prot* specifies the read/write accessibility of the mapped pages. The *addr* and *len* parameters, and the sum of the current position in *fd* and *off* parameters, must be multiples of the page size (found using the *getpagesize*(2) call). For this reason, local memory space beginning at *addr* should be allocated using *valloc*(2), which supplies a buffer with proper page alignment.

When mapping an area of 128K or more, the kernel releases the swap area associated with it. Consequently, when the pages are unmapped, they are marked invalid; the next call to *valloc*(2) returns the invalid pages, and any attempt to refer to those pages results in a segmentation violation. To avoid this, do not *free*(2) such large areas; instead, call *valloc*(2) again without calling *free*(2).

All pages are automatically unmapped when *fd* is closed. Specific pages can be unmapped explicitly using *munmap*.

mmap can sometimes be used to install memory-mapped devices without writing a device driver. However, this does not always work. In particular, devices that are *mmap*'ed into user space and then accessed by user programs will see those accesses in user mode. If the device contains registers that must be accessed in supervisor mode, *mmap* cannot be used to drive it. (See *Writing Device Drivers for the Sun Workstation* for more information.)

Munmap

munmap unmaps previously mapped pages starting at *addr* and continuing for *len* bytes. Unmapped pages refer, once again, to private pages within the caller's address space. Pages are initialized to zero, unless *len* is greater than or equal to 128K, in which case the pages are marked invalid.

RETURN VALUE

Each call returns 0 on success, –1 on failure.

ERRORS

Both calls fail when:

EINVAL The argument address or length is not a multiple of the page size as returned by *getpagesize*(2),*or* the length is negative.

EINVAL The entire range of pages specified in the call is not part of data space.

In addition *mmap* fails when:

EINVAL The specified *fd* does not refer to a character special device which supports mapping (e.g. a frame buffer).

EINVAL The specified *fd* is not open for reading and read access is requested, or not open for writing when write access is requested.

EINVAL The sharing mode was not specified as MAP_SHARED.

SEE ALSO

getpagesize(2), munmap(2), close(2)

BUGS

The kernel may panic when more than 128k of memory has been unmapped with *munmap*(1) and *mmap* is subsequently called with an incorrect *length* value.

If 128K of memory, or more, is unmapped as a result of closing *fd*, the resulting invalid pages cannot be reclaimed within the life of the calling process.

NAME

mount – mount file system

SYNOPSIS

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

mount(type, dir, flags, data)
int type;
char *dir;
int flags;
caddr_t data;
```

DESCRIPTION

mount attaches a file system to a directory. After a successful return, references to directory *dir* will refer to the root directory on the newly mounted file system. *dir* is a pointer to a null-terminated string containing a path name. *dir* must exist already, and must be a directory. Its old contents are inaccessible while the file system is mounted.

mount may be invoked only by the super-user.

The *flags* argument determines whether the file system can be written on, and if set-uid execution is allowed. 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.

type indicates the type of the filesystem. It must be one of the types defined in *mount.h*. *data* is a pointer to a structure which contains the type specific arguments to mount. Below is a list of the filesystem types supported and the type specific arguments to each:

MOUNT_UFS

```
struct ufs_args {
        char    *fspec;         /* Block special file to mount */
};
```

MOUNT_NFS

```
#include            <nfs/nfs.h>
#include            <netinet/in.h>
struct nfs_args {
        struct sockaddr_in  *addr; /* file server address */
        fhandle_t   *fh;        /* File handle to be mounted */
        int         flags;      /* flags */
        int         wsize;      /* write size in bytes */
        int         rsize;      /* read size in bytes */
        int         timeo;      /* initial timeout in .1 secs */
        int         retrans;    /* times to retry send */
};
```

RETURN VALUE

mount returns 0 if the action occurred, and –1 if *fspec* is inaccessible or not an appropriate file, if *name* does not exist, if *fspec* is already mounted, if *dir* is in use, or if there are already too many file systems mounted.

ERRORS

mount fails when one of the following occurs:

EPERM	The caller is not the super-user.
ENOTBLK	*fspec* is not a block device.
ENXIO	The major device number of *fspec* is out of range (this indicates no device driver exists for the associated hardware).
EBUSY	*dir* is not a directory, or another process currently holds a reference to it.

EBUSY	No space remains in the mount table.
EBUSY	The super block for the file system had a bad magic number or an out of range block size.
EBUSY	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.
ENOTDIR	A component of the path prefix in *fspec* or *dir* is not a directory.
EINVAL	The path name of *fspec* or *dir* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of the path name of *fspec* or *dir* exceeds 255 characters, or the length of the entire path name of *fspec* or *dir* exceeds 1023 characters.
ENOENT	*fspec* or *dir* does not exist.
ENOTDIR	The file named by *dir* is not a directory.
EACCES	Search permission is denied for a component of the path prefix of *fspec* or *dir*.
EFAULT	*fspec* or *dir* points outside the process's allocated address space.
ELOOP	Too many symbolic links were encountered in translating the path name of *fspec* or *dir*.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

unmount(2), mount(8)

BUGS

The error codes are in a state of disarray; too many errors appear to the caller as one value.

NAME

msgctl – message control operations

SYNOPSIS

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

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

DESCRIPTION

msgctl provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **msg_perm.uid** in the data structure associated with *msqid*. Only super user can raise the value of **msg_qbytes**.

IPC_RMID Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **msg_perm.uid** in the data structure associated with *msqid*.

ERRORS

msgctl will fail if:

EINVAL *msqid* is not a valid message queue identifier.

EINVAL *cmd* is not a valid command.

EACCES *cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)).

EPERM *cmd* is equal to **IPC_RMID** or **IPC_SET**. The effective user ID of the calling process is not equal to that of super user and it is not equal to the value of **msg_perm.uid** in the data structure associated with *msqid*.

EPERM *cmd* is equal to **IPC_SET**, an attempt is being made to increase to the value of **msg_qbytes**, and the effective user ID of the calling process is not equal to that of super user.

EFAULT *Buf* points to an illegal address.

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.

SEE ALSO

intro(2), msgget(2), msgop(2)

NAME

msgget – get message queue

SYNOPSIS

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

int msgget (key, msgflg)
key_t key;
int msgflg;
```

DESCRIPTION

msgget returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure (see *intro*(2)) are created for *key* if one of the following are true:

- *key* is equal to **IPC_PRIVATE**.
- *key* does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

msg_perm.cuid, msg_perm.uid, msg_perm.cgid, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.

msg_qnum, msg_lspid, msg_lrpid, msg_stime, and **msg_rtime** are set equal to 0.

msg_ctime is set equal to the current time.

msg_qbytes is set equal to the system limit.

ERRORS

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

EACCES	A message queue identifier exists for *key*, but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *msgflg* would not be granted.
ENOENT	A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is "false".
ENOSPC	A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.
EEXIST	A message queue identifier exists for *key* but ((*msgflg* & IPC_CREAT) & (*msgflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

Upon successful completion, a non-negative integer, namely a message queue identifier, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), msgctl(2), msgop(2)

NAME

msgop, msgsnd, msgrcv – message operations

SYNOPSIS

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

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

DESCRIPTION

msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*. {WRITE} *msgp* points to a structure containing the message. This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

mtype is a positive integer that can be used by the receiving process for message selection (see *msgrcv* below). *mtext* is any text of length *msgsz* bytes. *msgsz* can range from 0 to a system-imposed maximum.

msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to **msg_qbytes** (see *intro*(2)).

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the message will not be sent and the calling process will return immediately.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

msqid is removed from the system (see *msgctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(2)).

msgsnd will fail and no message will be sent if one or more of the following are true:

EINVAL *msqid* is not a valid message queue identifier.

EACCES Operation permission is denied to the calling process (see *intro*(2)).

EINVAL *mtype* is less than 1.

EAGAIN The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is "true".

EINVAL *msgsz* is less than zero or greater than the system-imposed limit.

EFAULT *msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see intro (2)).

msg_qnum is incremented by 1.

msg_lspid is set equal to the process ID of the calling process.

msg_stime is set equal to the current time.

msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. {READ} This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

mtype is the received message's type as specified by the sending process. *mtext* is the text of the message. *msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

msgtyp specifies the type of message requested as follows:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process will return immediately with a return value of −1 and *errno* set to ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

A message of the desired type is placed on the queue.

msqid is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2)).

msgrcv will fail and no message will be received if one or more of the following are true:

EINVAL	*msqid* is not a valid message queue identifier.
EACCES	Operation permission is denied to the calling process.
EINVAL	*msgsz* is less than 0.
[E2BIG]	*mtext* is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "false".
ENOMSG	The queue does not contain a message of the desired type and (*msgtyp* & **IPC_NOWAIT**) is "true".
EFAULT	*msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see intro (2)).

msg_qnum is decremented by 1.

msg_lrpid is set equal to the process ID of the calling process.

msg_rtime is set equal to the current time.

RETURN VALUES

If *msgsnd* or *msgrcv* return due to the receipt of a signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. If they return due to removal of *msqid* from the system, a value of −1 is returned and *errno* is set to EIDRM.

Upon successful completion, the return value is as follows:

msgsnd returns a value of 0.

msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

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

SEE ALSO

intro(2), msgctl(2), msgget(2), signal(2).

NAME

nfssvc, async_daemon – NFS daemons

SYNOPSIS

nfssvc(sock)
int sock;

async_daemon()

DESCRIPTION

Nfssvc starts an NFS daemon listening on socket *sock*. The socket must be AF_INET, and SOCK_DGRAM (protocol UDP/IP). The system call will return only if the process is killed.

Async_daemon implements the NFS daemon that handles asynchronous I/O for an NFS client. The system call never returns.

BUGS

These two system calls allow kernel processes to have user context.

SEE ALSO

mountd(8)

NAME

open – open or create a file for reading or writing

SYNOPSIS

#include <sys/file.h>

int open(path, flags [, mode])
char *path;
int flags, mode;

DESCRIPTION

path points to the pathname of a file. *open* opens the named file 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)). If the path is a null string, the kernel maps this null pathname to ., the current directory. *flags* values are constructed by ORing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier. The first time the process attempts to perform I/O on the open file it will block (not currently implemented).

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the group ID of the directory in which the file is created, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

All bits set in the file mode creation mask of the process are cleared. See *umask*(2).

The "save text image after execution" bit of the mode is cleared. See *chmod*(2).

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists. 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.

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) and *fcntl*(2).

There is a system enforced limit on the number of open file descriptors per process, whose value is returned by the *getdtablesize*(2) call.

SYSTEM V DESCRIPTION

If the O_NDELAY flag is set on an *open*, that flag is set for that file descriptor (see *fcntl*) and may affect subsequent reads and writes. See *read*(2V) and *write*(2V).

RETURN VALUE

The value −1 is returned if an error occurs, and external variable *errno* is set to indicate the cause of the error. Otherwise a non-negative numbered file descriptor for the new open file is returned.

ERRORS

Open fails if:

ENOTDIR	A component of the path prefix of *path* is not a directory.
EINVAL	*path* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	O_CREAT is not set and the named file does not exist.
ENOENT	A component of the path prefix of *path* does not exist.
ELOOP	Too many symbolic links were encountered in translating *path*.
EACCES	Search permission is denied for a component of the path prefix of *path*.
EACCES	The required permissions (for reading and/or writing) are denied for the file named by *path*.
EACCES	The file referred to by *path* does not exist, O_CREAT is specified, and the directory in which it is to be created does not permit writing.
EISDIR	The named file is a directory, and the arguments specify it is to be opened for writing.
ENXIO	O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
EMFILE	The system limit for open file descriptors per process has already been reached.
ENFILE	The system file table is full.
ENOSPC	The file does not exist, O_CREAT is specified, 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	The file does not exist, O_CREAT is specified, and there are no free inodes on the file system on which the file is being created.
EDQUOT	The file does not exist, O_CREAT is specified, and the directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
EDQUOT	The file does not exist, O_CREAT is specified, and the user's quota of inodes on the file system on which the file is being created has been exhausted.
EROFS	The named file does not exist, O_CREAT is specified, and the file system on which it is to be created is a read-only file system.
EROFS	The named file resides on a read-only file system, and the file is to be opened for writing.
ENXIO	The file is a character special or block special file, and the associated device does not

exist.

EINTR	A signal was caught during the *open* system call.
ETXTBSY	The file is a pure procedure (shared text) file that is being executed and the *open* call requests write access.
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.
EEXIST	O_EXCL and O_CREAT were both 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), fcntl(2), lseek(2), read(2V), write(2V), umask(2)

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 and returns two file descriptors, *fildes*[0] and *fildes*[1]. *fildes*[0] is opened for reading and *fildes*[1] is opened for writing. When the pipe is written using the descriptor *fildes*[1] up to 4096 bytes of data are buffered before the writing process is blocked. A read only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

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(2V), write(2V), 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 (20 milliseconds); *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 binary point at the left: 0x10000 gives a 1-1 mapping of pc's to words in *buff*; 0x8000 maps each pair of instruction words together. 0x2 maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *execve* is executed, but remains on in 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), setitimer(2), monitor(3)

NAME

ptrace – process trace

SYNOPSIS

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

ptrace(request, pid, addr, data [, addr2])
enum ptracereq request;
int pid;
char *addr;
int data;
char *addr2;

DESCRIPTION

ptrace provides a means by which a process may control the execution of another process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are five arguments whose interpretation depends on the *request* argument. Generally, *pid* is the process ID of the traced process. A process being traced behaves normally until it encounters some signal whether internally generated like 'illegal instruction' or externally generated like 'interrupt'. See *sigvec*(2) for the list. Then the traced process enters a stopped state and the tracing process is notified via *wait*(2). When the traced process is in the stopped state, its core image can be examined and modified using *ptrace*. If desired, another *ptrace* request can then cause the traced process either to terminate or to continue, possibly ignoring the signal.

Note that several different values of the *request* argument can make *ptrace* return data values — since –1 is a possibly legitimate value, to differentiate between –1 as a legitimate value and –1 as an error code, you should clear the *errno* global error code before doing a *ptrace* call, and then check the value of *errno* afterwards.

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

PTRACE_TRACEME
: This request is the only one used by the traced 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.

PTRACE_PEEKTEXT, PTRACE_PEEKDATA
: The word in the traced process's address space at *addr* is returned. If the instruction and data spaces are separate (for example, historically on a PDP-11), request PTRACE_PEEKTEXT indicates instruction space while PTRACE_PEEKDATA indicates data space. Otherwise, either request may be used, with equal results. *addr* must be even, the child must be stopped and the input *data* and *addr2* are ignored.

PTRACE_PEEKUSER
: The word of the system's per-process data area corresponding to *addr* is returned. *addr* must be a valid offset within the kernel's per-process data pages. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system (see **<sys/user.h>**).

PTRACE_POKETEXT, PTRACE_POKEDATA
: The given *data* is written at the word in the process's address space corresponding to *addr*, which must be even. No useful value is returned. If the instruction and data spaces are separate, request PTRACE_PEEKTEXT indicates instruction space while PTRACE_PEEKDATA indicates data space. The PTRACE_POKETEXT request must be used to write into a process's text space even if the instruction and data spaces are not separate. Attempts to write in a pure text space fail if another process is executing the same file.

PTRACE_POKEUSER

The process's system data is written, as it is read with request PTRACE_PEEKUSER. 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.

PTRACE_CONT
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.

PTRACE_KILL
The traced process terminates, with the same consequences as *exit*(2).

PTRACE_SINGLESTEP
Execution continues as in request PTRACE_CONT; 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 Sun, the T-bit is used and just one instruction is executed. This is part of the mechanism for implementing breakpoints.

PTRACE_ATTACH
Attach to the process identified by the *pid* argument and begin tracing it. Process *pid* does not have to be a child of the requestor, but the requestor must have permission to send process *pid* a signal and the effective user IDs of the requesting process and process *pid* must match.

PTRACE_DETACH
Detach the process being traced. Process *pid* is no longer being traced and continues its execution. The *data* argument is taken as a signal number and the process continues at location *addr* as if it had incurred that signal.

PTRACE_GETREGS
The traced process's registers are returned in a structure pointed to by the *addr* argument. The registers include the general purpose registers, the program counter and the program status word. The 'regs' structure defined in *<machine/reg.h>* describes the data that is returned.

PTRACE_SETREGS
The traced process's registers are written from a structure pointed to by the *addr* argument. The registers include the general purpose registers, the program counter and the program status word. The 'regs' structure defined in *<machine/reg.h>* describes the data that is set.

PTRACE_GETFPREGS
(Sun-3 only) The traced process's FPP status is returned in a structure pointed to by the *addr* argument. The status includes the 68881 floating point registers and the control, status, and instruction address registers. The 'fp_status' structure defined in *<machine/reg.h>* describes the data that is returned.

PTRACE_SETFPREGS
(Sun-3 only) The traced process's FPP status is written from a structure pointed to by the *addr* argument. The status includes the 68881 floating point registers and the control, status, and instruction address registers. The 'fp_status' structure defined in **<machine/reg.h>** describes the data that is set.

PTRACE_GETFPAREGS
(Sun-3 with FPA only) The traced process's FPA registers are returned in a structure pointed to by the *addr* argument. The 'fpa_regs' structure defined in **<machine/reg.h>** describes the data that is returned.

PTRACE_SETFPAREGS
(Sun-3 with FPA only) The traced process's FPA registers are written from a structure pointed to by the *addr* argument. The 'fpa_regs' structure defined in **<machine/reg.h>** describes the data that is set.

PTRACE_READTEXT, PTRACE_READDATA

Read data from the address space of the traced process. If the instruction and data spaces are separate, request PTRACE_READTEXT indicates instruction space while PTRACE_READDATA indicates data space. The *addr* argument is the address within the traced process from where the data is read, the *data* argument is the number of bytes to read, and the *addr2* argument is the address within the requesting process where the data is written.

PTRACE_WRITETEXT, PTRACE_WRITEDATA
Write data into the address space of the traced process. If the instruction and data spaces are separate, request PTRACE_READTEXT indicates instruction space while PTRACE_READDATA indicates data space. The *addr* argument is the address within the traced process where the data is written, the *data* argument is the number of bytes to write, and the *addr2* argument is the address within the requesting process from where the data is read.

As indicated, these calls (except for requests PTRACE_TRACEME and PTRACE_ATTACH) 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 WSTOPPED to indicate a stop 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 the Sun, 'word' also means a 32-bit integer.

RETURN VALUE

In general, a 0 value is returned if the call succeeds. Note that this is not always true because requests such as PTRACE_PEEKTEXT and PTRACE_PEEKDATA return legitimate values. 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.
ESRCH	The request requires the process to be one which is traced by the current process and stopped, but it is not stopped or it is not being traced by the current process.
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)

BUGS

ptrace is unique and arcane; it should be replaced with a special file which 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 requests PTRACE_TRACEME thru PTRACE_SINGLESTEP are standard UNIX *ptrace* requests. The requests PTRACE_ATTACH thru PTRACE_WRITEDATA and the fifth argument, *addr2*, are unique to Sun UNIX.

The request PTRACE_TRACEME should be able to specify signals which 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 clarify.

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

quotactl – manipulate disk quotas

SYNOPSIS

#include <ufs/quota.h>

quotactl(cmd, special, uid, addr)
int cmd;
char *special;
int uid;
caddr_t addr;

DESCRIPTION

The *quotactl* call manipulates disk quotas. The *cmd* parameter indicates a command to be applied to the user ID *uid*. *Special* is a pointer to a null-terminated string containing the path name of the block special device for the file system being manipulated. The block special device must be mounted. *Addr* is the address of an optional, command specific, data structure which is copied in or out of the system. The interpretation of *addr* is given with each command below.

Q_QUOTAON
: Turn on quotas for a file system. *Addr* is a pointer to a null terminated string containing the path name of file containing the quotas for the file system. The quota file must exist; it is normally created with the *quotacheck*(8) program. This call is restricted to the super-user.

Q_QUOTAOFF
: Turn off quotas for a file system. This call is restricted to the super-user.

Q_GETQUOTA
: Get disk quota limits and current usage for user *uid*. *Addr* is a pointer to a struct dqblk structure (defined in *<ufs/quota.h>*). Only the super-user may get the quotas of a user other than himself.

Q_SETQUOTA
: Set disk quota limits and current usage for user *uid*. *Addr* is a pointer to a struct dqblk structure (defined in *<ufs/quota.h>*). This call is restricted to the super-user.

Q_SETQLIM
: Set disk quota limits for user *uid*. *Addr* is a pointer to a struct dqblk structure (defined in *<ufs/quota.h>*). This call is restricted to the super-user.

Q_SYNC
: Update the on-disk copy of quota usages. 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 the error.

ERRORS

A *quotactl* call will fail when one of the following occurs:

EINVAL	*Cmd* is invalid.
EPERM	The call is privileged and the caller was not the super-user.
EINVAL	The *special* parameter is not a mounted file system or is a mounted file system without quotas enabled.
ENOTBLK	The *special* parameter is not a block device.
EFAULT	An invalid *addr* is supplied; the associated structure could not be copied in or out of the kernel.
EINVAL	The *addr* parameter is being interpreted as the path of a quota file which exists but is either not a regular file or is not on the file system pointed to by the *special* parameter.

EUSERS The quota table is full.

SEE ALSO

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). Incompatible with Melbourne quotas.

NAME

read, readv – read input

SYNOPSIS

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

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

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

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;
        int      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.

Note: For read access to a directory, use *readdir*(3). function. (Directory access using *readdir*(3) is no longer optional.)

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 which 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.

When attempting to read from a descriptor associated with an empty pipe, socket, or FIFO:

If O_NDELAY is set, the read will return a –1 and *errno* will be set to EWOULDBLOCK.

If O_NDELAY is clear, the read will block until data is written to the pipe or the file is no longer open for writing.

When attempting to read from a descriptor associated with a tty that has no data currently available:

If O_NDELAY is set, the read will return a –1 and *errno* will be set to EWOULDBLOCK.

If O_NDELAY is clear, the read will block until data becomes available.

If O_NDELAY is set, and less data are available than are requested by the *read* or *readv*, only the data that are available are returned, and the count indicates how many bytes of data were actually read.

SYSTEM V DESCRIPTION

When an attempt is made to read a descriptor which is in no-delay mode, and there is no data currently available, *read* will return a 0 instead of returning a –1 and setting *errno* to EWOULDBLOCK. Note that this

is indistinguishable

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 descriptor open for reading.
EISDIR	*d* refers to a directory which is on a file system mounted using the NFS.
EFAULT	*buf* points outside the allocated address space.
EIO	An I/O error occurred while reading from or writing to 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	One of the *iov_len* values in the *iov* array was negative.
EINVAL	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.
EFAULT	Part of *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 buffer will be unchanged if:

EINVAL *path* contained a byte with the high-order bit set.

ENAMETOOLONG
The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

ENOENT The named file 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*.

EINVAL The named file is not a symbolic link.

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

EFAULT *path* or *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 "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.

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(8S), halt(8), init(8), reboot(8)

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 may be 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 nonblocking (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*(2V). A buffer to receive any access rights sent along with

the message is specified in *msg_accrights*, which has length *msg_accrightslen*.

RETURN VALUE

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

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(2V), 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 renames the link named *from* 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:

ENOTDIR	A component of the path prefix of either *from* or *to* is not a directory.
EINVAL	Either *from* or *to* contains a byte with the high-order bit set.
ENAMETOOLONG	The length of a component of either *from* or *to* exceeds 255 characters, or the length of either *from* or *to* exceeds 1023 characters.
ENOENT	A component of the path prefix of either *from* or *to* does not exist.
ENOENT	The file named by *from* does not exist.
EACCES	A component of the path prefix of either *from* or *to* denies search permission.
EACCES	The requested rename requires writing in a directory with a mode that denies write permission.
ELOOP	Too many symbolic links were encountered while translating either *from* or *to*.
EXDEV	The link named by *to* and the file named by *from* are on different logical devices (file systems).
ENOSPC	The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.
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 reading from or writing to the file system.
EROFS	The requested rename requires writing in a directory on a read-only file system.
EFAULT	Either or both of *from* or *to* point 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.
EBUSY	*to* is a directory and is the mount point for a mounted file system.

SEE ALSO

open(2V)

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 prefix of *path* is not a directory.
ENOTDIR	The file referred to by *path* is not a directory.
EINVAL	*path* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	The directory referred to by *path* does not exist.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENOTEMPTY	The directory referred to by *path* contains files other than "." and ".." in it.
EACCES	Search permission is denied for a component of the path prefix of *path*.
EACCES	Write permission is denied for the directory containing the link to be removed.
EBUSY	The directory to be removed is the mount point for a mounted file system. EIO An I/O error occurred while reading from or writing to the file system.
EROFS	The directory to be removed resides on a read-only file system.
EFAULT	*path* points outside the process's allocated address space.

mkdir(2), unlink(2)

NAME

select – synchronous I/O multiplexing

SYNOPSIS

#include <sys/time.h>

nfds = select(width, readfds, writefds, exceptfds, timeout)
int width, *readfds, *writefds, *exceptfds;
struct timeval timeout;

DESCRIPTION

select examines the I/O descriptors specified by the bit masks *readfds*, *writefds*, and *exceptfds* to see if they are ready for reading, writing, or have an exceptional condition pending, respectively. *width* is the number of significant bits in each bit mask that represent a file descriptor. Typically *width* has the value returned by *getdtablesize*(2) for the maximum number of file descriptors or is the constant 32 (number of bits in an int). File descriptor *f* is represented by the bit "1<<f" in the mask. *select* returns, in place, a mask of those descriptors which are ready. The total number of ready descriptors is returned in *nfds*.

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, the select blocks indefinitely. To effect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as NULL pointers if no descriptors are of interest.

RETURN VALUE

select returns the number of ready descriptors that are contained in the bit masks, or –1 if an error occurred. If the time limit expires then *select* returns 0.

ERRORS

An error return from *select* indicates:

EBADF	One of the bit masks specified an invalid descriptor.
EINTR	A signal was delivered before any of the selected events occurred or the time limit expired.
EINVAL	The specified time limit is unacceptable. One of its components is negative or too large.
EFAULT	One of the pointers given in the call referred to a non-existent portion of the process' address space.

SEE ALSO

accept(2), connect(2), gettimeofday(2), read(2V), write(2V), recv(2), send(2), getdtablesize(2)

BUGS

The descriptor masks are always modified on return, even if the call returns as the result of the timeout.

NAME

semctl – semaphore control operations

SYNOPSIS

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

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
        val;
        struct semid_ds *buf;
        ushort *array;
} arg;
```

DESCRIPTION

semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum:*

GETVAL Return the value of *semval* (see *intro*(2)). {READ}

SETVAL Set the value of *semval* to *arg.val*. {ALTER} When this cmd is successfully executed, the *semadj* value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of *sempid*. {READ}

GETNCNT Return the value of *semncnt*. {READ}

GETZCNT Return the value of *semzcnt*. {READ}

The following *cmd*s return and set, respectively, every *semval* in the set of semaphores.

GETALL Place *semvals* into array pointed to by *arg.array*. {READ}

SETALL Set *semvals* according to the array pointed to by *arg.array*. {ALTER} When this cmd is successfully executed the *semadj* values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */

This cmd can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of **sem_perm.uid** in the data structure associated with *semid*.

IPC_RMID Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This cmd can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of **sem_perm.uid** in the data structure associated with *semid*.

ERRORS

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

EINVAL	*semid* is not a valid semaphore identifier.
EINVAL	*semnum* is less than zero or greater than **sem_nsems**.
EINVAL	*cmd* is not a valid command.
EACCES	Operation permission is denied to the calling process (see *intro*(2)).
ERANGE	*cmd* is **SETVAL** or **SETALL** and the value to which *semval* is to be set is greater than the system imposed maximum.
EPERM	*cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of **sem_perm.uid** in the data structure associated with *semid*.
EFAULT	*arg.buf* points to an illegal address.

RETURN VALUE

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

GETVAL	The value of *semval*.
GETPID	The value of *sempid*.
GETNCNT	The value of *semncnt*.
GETZCNT	The value of *semzcnt*.
All others	A value of 0.

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

SEE ALSO

intro(2), semget(2), semop(2).

NAME

semget – get set of semaphores

SYNOPSIS

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

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

DESCRIPTION

semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores (see *intro*(2)) are created for *key* if one of the following are true:

key is equal to **IPC_PRIVATE.**

key does not already have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

Sem_perm.cuid, sem_perm.uid, sem_perm.cgid, and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.

sem_nsems is set equal to the value of *nsems*.

sem_otime is set equal to 0 and **sem_ctime** is set equal to the current time.

ERRORS

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

EINVAL	*nsems* is either less than or equal to zero or greater than the system-imposed limit.
EACCES	A semaphore identifier exists for *key*, but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *semflg* would not be granted.
EINVAL	A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to zero.
ENOENT	A semaphore identifier does not exist for *key* and (*semflg* & **IPC_CREAT**) is "false".
ENOSPC	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.
ENOSPC	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.
EEXIST	A semaphore identifier exists for *key* but ((*semflg* & IPC_CREAT) **and** (*semflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

Upon successful completion, a non-negative integer, namely a semaphore identifier, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), semctl(2), semop(2).

NAME

semop – semaphore operations

SYNOPSIS

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

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
int nsops;
```

DESCRIPTION

semop is used to automatically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *sops* is a pointer to the array of semaphore-operation structures. *nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
short   sem_num;   /* semaphore number */
short   sem_op;    /* semaphore operation */
short   sem_flg;   /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following will occur: {ALTER}

If *semval* (see *intro*(2)) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's *semadj* value (see *exit*(2)) for the specified semaphore.

If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the *semncnt* associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occur.

semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval* and, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's *semadj* value for the specified semaphore.

The *semid* for which the calling process is awaiting action is removed from the system (see *semctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

If *sem_op* is a positive integer, the value of *sem_op* is added to *semval* and, if (*sem_flg* & **SEM_UNDO**) is "true", the value of *sem_op* is subtracted from the calling process's *semadj* value for the specified semaphore. {ALTER}

If *sem_op* is zero, one of the following will occur: {READ}

If *semval* is zero, *semop* will return immediately.

If *semval* is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If *semval* is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the *semzcnt* associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:

semval becomes zero, at which time the value of *semzcnt* associated with the specified semaphore is decremented.

The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of *semzcnt* associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

ERRORS

semop will fail if one or more of the following are true for any of the semaphore operations specified by *sops*:

EINVAL	*semid* is not a valid semaphore identifier.
EFBIG	*sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.
[E2BIG]	*nsops* is greater than the system-imposed maximum.
EACCES	Operation permission is denied to the calling process (see *intro*(2)).
EAGAIN	The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true".
ENOSPC	The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.
EINVAL	The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.
ERANGE	An operation would cause a *semval* or *semadj* value to overflow the system-imposed limit.
EFAULT	*sops* points to an illegal address.

Upon successful completion, the value of *sempid* for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

RETURN VALUE

If *semop* returns due to the receipt of a signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of −1 is returned and *errno* is set to EIDRM.

Upon successful completion, the value of *semval* at the time of the call for the last operation in the array pointed to by *sops* is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), semctl(2), semget(2).

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

S is a socket created with *socket*(2). *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 be set to MSG_OOB to send "out-of-band" data on sockets which support this notion (e.g. SOCK_STREAM).

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.

SEE ALSO

recv(2), socket(2)

NAME

setpgrp, getpgrp – set and/or return the process group of a process

SYNOPSIS

setpgrp(pid, pgrp)

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

SYSTEM V SYNOPSIS

int setpgrp ( )

DESCRIPTION

Setpgrp

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 (calling) process.

If the effective user ID is not that of the super-user, then the process to be affected must have the same effective user ID as that of the caller or be a descendant of that process.

Getpgrp

getpgrp returns the process group of the indicated process. If *pid* is zero, then the call applies to the calling 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 run in the foreground and may read from the terminal, while others block with a signal when 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.

RETURN VALUE

setpgrp returns 0 when the operation was successful. If the request failed, –1 is returned and the global variable *errno* indicates the reason.

ERRORS

setpgrp fails, and the process group is not altered when one of the following occurs:

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.

SYSTEM V DESCRIPTION

In the System V implementation, *setpgrp* takes no parameters. It sets the process group of the calling process to match its process ID, and returns the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), getuid(2), intro(2), kill(2), signal(2), tty(4)

NAME

setregid – set real and effective group IDs

SYNOPSIS

int setregid(rgid, egid)
int rgid, egid;

DESCRIPTION

setregid is used to set the real and effective group IDs of the calling process. If *rgid* is –1, the real group ID is not changed; if *egid* is –1, the effective group ID is not changed. The real and effective group IDs may be set to different values in the same call.

If the effective user ID of the calling process is super-user, the real group ID and the effective group ID can be set to any legal value.

If the effective user ID of the calling process is not super-user, either the real group ID can be set to the saved set-group ID from *execve*(2), or the effective group ID can either be set to the saved set-group ID or the real group ID. Note that if a set-GID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group ID.

In either case, if the real group ID is changed to a particular value (i.e., if *rgid* is not –1), the saved set-group ID is set to that same value.

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

Setregid will fail and neither of the group IDs will be changed if:

EPERM The calling process' effective user ID is not the super-user and a change other than changing the real group ID to the saved set-group ID, or changing the effective group ID to the real group-id or the saved set-group ID, was specified.

SEE ALSO

getgid(2), execve(2), setreuid(2), setgid(3)

NAME

setreuid – set real and effective user IDs

SYNOPSIS

int setreuid(ruid, euid)
int ruid, euid;

DESCRIPTION

setreuid is used to set the real and effective user IDs of the calling process. If *ruid* is −1, the real user ID is not changed; if *euid* is −1, the effective user ID is not changed. The real and effective user IDs may be set to different values in the same call.

If the effective user ID of the calling process is super-user, the real user ID and the effective user ID can be set to any legal value.

If the effective user ID of the calling process is not super-user, either the real user ID can be set to the effective user ID, or the effective user ID can either be set to the saved set-user ID from *execve*(2) or the real user ID. Note that if a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-user ID.

In either case, if the real user ID is changed to a particular value (i.e., if *ruid* is not −1), the saved set-user ID is set to that same value.

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

Setreuid will fail and neither of the user IDs will be changed if:

EPERM — The calling process' effective user ID is not the super-user and a change other than changing the real user ID to the effective user ID, or changing the effective user ID to the real user-id or the saved set-user ID, was specified.

SEE ALSO

getuid(2), execve(2), setregid(2), setuid(3)

NAME

shmctl – shared memory control operations

SYNOPSIS

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

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

DESCRIPTION

shmctl provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of **shm_perm.uid** in the data structure associated with *shmid*.

IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of **shm_perm.uid** in the data structure associated with *shmid*.

ERRORS

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

EINVAL *Shmid* is not a valid shared memory identifier.

EINVAL *cmd* is not a valid command.

EACCES *cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)).

EPERM *cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of **shm_perm.uid** in the data structure associated with *shmid*.

EFAULT *buf* points to an illegal address.

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.

SEE ALSO

intro(2), shmget(2), shmop(2).

NAME

shmget – get shared memory segment

SYNOPSIS

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

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

DESCRIPTION

shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of size *size* bytes (see *intro*(2)) are created for *key* if one of the following are true:

key is equal to **IPC_PRIVATE**.

key does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

shm_perm.cuid, shm_perm.uid, shm_perm.cgid, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **shm_segsz** is set equal to the value of *size*.

shm_lpid, shm_nattch, shm_atime, and **shm_dtime** are set equal to 0.

shm_ctime is set equal to the current time.

ERRORS

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

EINVAL	*size* is less than the system-imposed minimum or greater than the system-imposed maximum.
EACCES	A shared memory identifier exists for *key* but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *shmflg* would not be granted.
EINVAL	A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.
ENOENT	A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is "false".
ENOSPC	A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.
ENOMEM	A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.
EEXIST	A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) and (*shmflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

Upon successful completion, a non-negative integer, namely a shared memory identifier is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), shmctl(2), shmop(2)

NAME

shmop, shmat, shmdt – shared memory operations

SYNOPSIS

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

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr
int shmflg;

int shmdt (shmaddr)
char *shmaddr
```

DESCRIPTION

shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus **SHMLBA**)).

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "false", the segment is attached at the address given by *shmaddr*.

The segment is attached for reading if (*shmflg* & **SHM_RDONLY**) is "true" {READ}, otherwise it is attached for reading and writing {READ/WRITE}.

shmat will fail and not attach the shared memory segment if one or more of the following are true:

EINVAL	*Shmid* is not a valid shared memory identifier.
EACCES	Operation permission is denied to the calling process (see *intro*(2)).
ENOMEM	The available data space is not large enough to accommodate the shared memory segment.
EINVAL	*shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus **SHMLBA**)) is an illegal address.
EINVAL	*shmaddr* is not equal to zero, (*shmflg* & **SHM_RND**) is "false", and the value of *shmaddr* is an illegal address.
EMFILE	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
EINVAL	*shmdt* detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.
EINVAL	*shmdt* will fail and not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.

RETURN VALUES

Upon successful completion, the return value is as follows:

shmat returns the data segment start address of the attached shared memory segment.

shmdt returns a value of 0.

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

SEE ALSO

exec(2), exit(2), fork(2), intro(2), shmctl(2), shmget(2).

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)

BUGS

The *how* values should be defined constants.

NAME

sigblock – block signals

SYNOPSIS

#include <signal.h>

oldmask = sigblock(mask);
int mask;

mask = sigmask(signum)

DESCRIPTION

Sigblock adds the signals specified in *mask* 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*. The previous mask is returned, and may be restored using *sigsetmask*(2).

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), sigvec(2), sigsetmask(2), signal(3)

NAME

sigpause – atomically release blocked signals and wait for interrupt

SYNOPSIS

sigpause(sigmask)
int sigmask;

DESCRIPTION

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 0 to indicate that no signals are now to be blocked. *Sigpause* always terminates by being interrupted, returning EINTR.

In normal usage, a signal is blocked using *sigblock*(2), to begin a critical section, variables modified on the occurence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using *sigpause* with the mask returned by *sigblock*.

SEE ALSO

sigblock(2), sigvec(2), signal(3)

NAME

sigsetmask – set current signal mask

SYNOPSIS

```
#include <signal.h>

sigsetmask(mask);
int mask;

mask = sigmask(signum)
```

DESCRIPTION

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 from being blocked.

RETURN VALUE

The previous set of masked signals is returned.

SEE ALSO

kill(2), sigvec(2), sigblock(2), sigpause(2), signal(3)

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

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), signal(3)

NAME

sigvec – software signal facilities

SYNOPSIS

```
#include <signal.h>

struct sigvec {
int     (*sv_handler)();
int     sv_mask;
int     sv_flags;
};

sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

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 *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 mask specified in *vec* is not allowed to block SIGKILL, SIGSTOP, or SIGCONT. The system enforces this restriction silently.

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 (other than A-line or F-line op code)
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT trap (not generated on Suns)
SIGEMT	7*	EMT trap (A-line or F-line op code)
SIGFPE	8*	arithmetic 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 or other socket 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 changed (see *win*(4S))
SIGLOST	29*	resource lost (see *lockd*(8C))
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, except that if the SV_RESETHAND bit is set in *sv_flags*, the value of *sv_handler* for the caught signal will be set to SIG_DFL before entering the signal-catching function, unless the signal is SIGILL or SIGTRAP. If this bit is set, the bit for that signal in the signal mask will not be set; unless the signal mask associated with that signal blocks that signal, further occurrences of that signal will not be blocked. The SV_RESETHAND flag is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

The default action for a signal may be reinstated by setting *sv_handler* to SIG_DFL; this default is termination 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 the process is terminated, a "core image" will be made in the current working directory of the receiving process if the signal is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

The effective group ID and the real group ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the file group ID of the current directory

If *sv_handler* is SIG_IGN the signal is subsequently ignored, and pending instances of the signal are discarded.

Note: the signals SIGKILL, SIGSTOP, and SIGCONT cannot be ignored.

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 SV_INTERRUPT flag is not available in 4.2BSD, hence it should not be used if backward compatibility is needed. The affected system calls are *read*(2V) or *write*(2V) on a

slow device (such as a terminal or pipe or other socket, 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 and reset-signal-handler flags.

The *execve*(2) call resets all caught signals to default action and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that interrupt system calls continue to do so.

NOTES

The handler routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number. *Code* is a parameter of certain signals that provides additional detail. *Scp* is a pointer to the *sigcontext* structure (defined in **<signal.h>**), used to restore the context from before the signal.

Programs that must be portable to UNIX systems other than 4.2 BSD should use the *signal*(3) interface instead.

CODES

The following defines the codes for signals which produce them. All of these symbols are defined in **<signal.h>**:

Hardware condition	Signal	Code
Illegal instruction	SIGILL	ILL_INSTR_FAULT
Privilege violation	SIGILL	ILL_PRIVVIO_FAULT
Coprocessor protocol error	SIGILL	ILL_INSTR_FAULT
Trap #*n* (1 <= *n* <= 14)	SIGILL	ILL_TRAP*n*_FAULT
A-line op code	SIGEMT	EMT_EMU1010
F-line op code	SIGEMT	EMT_EMU1111
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
CHK or CHK2 instruction	SIGFPE	FPE_CHKINST_TRAP
TRAPV or TRAPcc or cpTRAPcc	SIGFPE	FPE_TRAPV_TRAP
IEEE floating point compare unordered	SIGFPE	FPE_FLTBSUN_TRAP
IEEE floating point inexact	SIGFPE	FPE_FLTINEX_TRAP
IEEE floating point division by zero	SIGFPE	FPE_FLTDIV_TRAP
IEEE floating point underflow	SIGFPE	FPE_FLTUND_TRAP
IEEE floating point operand error	SIGFPE	FPE_FLTOPERR_TRAP
IEEE floating point overflow	SIGFPE	FPE_FLTOVF_FAULT
IEEE floating point signaling NaN	SIGFPE	FPE_FLTNAN_TRAP

RETURN VALUE

A 0 value indicated that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate 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), setjmp(3), signal(3), tty(4)

NAME

socket – create an endpoint for communication

SYNOPSIS

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

s = socket(af, type, protocol)
int s, af, type, protocol;

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *af* parameter specifies an address format with which addresses specified in later operations using the socket should be interpreted. These formats are defined in the include file *<sys/socket.h>*. The currently understood formats are

AF_UNIX	(UNIX path names),
AF_INET	(ARPA Internet addresses),
AF_PUP	(Xerox PUP-I Internet addresses), and
AF_IMPLINK	(IMP "host at IMP" addresses).

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 with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network interfaces. The types SOCK_RAW, which is available only to the super-user, and SOCK_SEQPACKET and SOCK_RDM, which are 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 using a given address format. However, it is possible that many protocols may exist in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see *services*(5) and *protocols*(5).

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*(2V) and *write*(2V) 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_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. It is also possible to receive datagrams at such a socket with *recv*(2).

An *fcntl*(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives.

The operation of sockets is controlled by socket level *options*. These options are defined in the file *<sys/socket.h>* and explained below. *Setsockopt* and *getsockopt*(2) are used to set and get options, respectively.

SO_DEBUG	turn on recording of debugging information
SO_REUSEADDR	allow local address reuse
SO_KEEPALIVE	keep connections alive
SO_DONTROUTE	do no apply routing on outgoing messages
SO_LINGER	linger on close if data present
SO_DONTLINGER	do not linger on close

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates 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 and SO_DONTLINGER control the actions 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_DONTLINGER is specified and a *close* is issued, the system will process the close in a manner which allows the process to continue as quickly as possible.

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:

EAFNOSUPPORT	The specified address family is not supported in this version of the system.
ESOCKTNOSUPPORT	The specified socket type is not supported in this address family.
EPROTONOSUPPORT	The specified protocol is not supported.
EMFILE	The per-process descriptor table is full.
ENOBUFS	No buffer space is available. The socket cannot be created.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), recv(2), select(2), send(2), shutdown(2), socketpair(2)

Inter-Process Communication Primer in *Networking on the Sun Workstation*

BUGS

The use of keepalives is a questionable feature for this layer.

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* system 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(2V), write(2V), 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 named by *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 searchable.

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* include the following members:

```
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 of 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, in bytes */
time_t    st_atime;   /* file last access time */
time_t    st_mtime;   /* file last modify time */
time_t    st_ctime;   /* file last status change time */
long      st_blksize; /* optimal blocksize for file system i/o ops */
long      st_blocks;  /* actual number of blocks allocated */
```

st_atime Time when file data was last read or modified. Changed by the following system calls: *mknod*(2), *utimes*(2), *read*(2V), *write*(2V), and *truncate*(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*(2V).

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*(2V), *truncate*(2).

The status information word *st_mode* has bits:

```
#define S_IFMT    0170000    /* type of file */
#define S_IFIFO   0010000    /* fifo special */
#define S_IFCHR   0020000    /* character special */
```

```
#define  S_IFDIR     0040000   /* directory */
#define  S_IFBLK     0060000   /* block special */
#define  S_IFREG     0100000   /* regular file */
#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 of *path* is not a directory.

EINVAL — *path* contains a character with the high-order bit set.

ENAMETOOLONG — The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

ENOENT — The file referred to by *path* does not exist.

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

ELOOP — Too many symbolic links were encountered in translating *path*.

EFAULT — *buf* or *path* 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 — *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.

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 which depend on the time stamps being contiguous (in calls to *utimes*(2)).

SEE ALSO

chmod(2), chown(2), readlink(2), utimes(2)

NAME

statfs – get file system statistics

SYNOPSIS

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

statfs(path, buf)
char *path;
struct statfs *buf;

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:

```
typedef struct {
        long    val[2];
} fsid_t;

struct statfs {
        long    f_type;       /* type of info, zero for now */
        long    f_bsize;      /* fundamental file system block size */
        long    f_blocks;     /* total blocks in file system */
        long    f_bfree;      /* free blocks */
        long    f_bavail;     /* free blocks available to non-superuser */
        long    f_files;      /* total file nodes in file system */
        long    f_ffree;      /* free file nodes in fs */
        fsid_t  f_fsid;       /* file system id */
        long    f_spare[7];   /* spare for later */
};
```

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 VALUE

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 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	The file referred to by *path* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *path*.
ELOOP	Too many symbolic links were encountered in translating *path*.
EFAULT	*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 both 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.

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.

SEE ALSO

swapon(8), config(8)

RETURN VALUE

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

ERRORS

ENOTDIR	A component of the path prefix of *special* is not a directory.
EINVAL	*special* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of *special* exceeds 255 characters, or the length of *special* exceeds 1023 characters.
ENOENT	The device referred to by *special* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *special*.
ELOOP	Too many symbolic links were encountered in translating *special*.
EPERM	The caller is not the super-user.
ENOTBLK	The file referred to by *special* is not a block device.
EBUSY	The device referred to by *special* has already been made available for swapping.
ENODEV	The device referred to by *special* was not configured into the system as a swap device.
ENXIO	The major device number of the device referred to by *special* is out of range (this indicates no device driver exists for the associated hardware).
EIO	An I/O error occurred while reading from or writing to the file system or opening the swap device.
EFAULT	*special* points outside the process's address space.

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 one or more of the following are true:

ENOTDIR	A component of the path prefix of *name2* is not a directory.
EINVAL	*name2* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of either *name1* or *name2* exceeds 255 characters, or the length of either *name1* or *name2* exceeds 1023 characters.
ENOENT	A component of the path prefix of *name2* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *name2*.
ELOOP	Too many symbolic links were encountered in translating *name2*.
EEXIST	The file referred to by *name2* already exists.
EIO	An I/O error occurred while reading from or writing to the file system.
EROFS	The file *name2* would reside on a read-only file system.
ENOSPC	The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.
ENOSPC	The new symbolic link cannot be created because there is no space left on the file system which will contain the link.
ENOSPC	There are no free inodes on the file system on which the file is being created.
EDQUOT	The directory in which the entry for the new 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 becaue the user's quota of disk blocks on the file system which will contain the link has been exhausted.
EDQUOT	The user's quota of inodes on the file system on which the file is being created has been exhausted.
EFAULT	*name1* or *name2* points outside the process's allocated address space.

SEE ALSO

link(2), ln(1), readlink(2), 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), cron(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, ...)

DESCRIPTION

syscall performs the system call whose assembly language interface has the specified *number*, and arguments *arg* .... Symbolic constants for system calls can be found in the header file **<syscall.h>**.

The register d0 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 d1.

NAME

truncate, ftruncate – truncate a file to a specified length

SYNOPSIS

```
truncate(path, length)
char *path;
unsigned long length;

ftruncate(fd, length)
int fd;
unsigned long 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 of *path* is not a directory.
EINVAL	*path* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	The file referred to by *path* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *path*.
EACCES	Write permission is denied for the file referred to by *path*.
ELOOP	Too many symbolic links were encountered in translating *path*.
EISDIR	The file referred to by *path* is a directory.
EROFS	The file referred to by *path* resides on a read-only file system.
ETXTBSY	The file referred to by *path* is a pure procedure (shared text) file that is being executed.
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.

ftruncate succeeds unless:

EINVAL	*fd* is not a valid descriptor of a file open for writing.
EINVAL	*fd* references a socket, not a file.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

open(2V)

BUGS

Partial blocks discarded as the result of truncation are not zero filled; this can result in holes in files which do not read as zero.

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

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(2V)

NAME

uname – get name of current UNIX system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

Note: This system call is only available for use with the System V compatibility libraries. These are located in the directory */usr/5lib*, and are compiled using the System V version of the C compiler, */usr/5lib/cc*.

uname stores information identifying the current UNIX system in the structure pointed to by *name*.

uname uses the structure defined in **<sys/utsname.h>** whose members are:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
char    machine[9];
```

uname returns a null-terminated character string naming the current UNIX system in *sysname* and *nodename*. This name will be the name returned by the *gethostname*(2) system call, truncated to 8 characters. *release* and *version* further identify the operating system. *machine* contains a name that identifies the hardware that the UNIX system is running on.

SEE ALSO

uname(1V)

NAME

unlink – remove directory entry

SYNOPSIS

unlink(path)
char *path;

DESCRIPTION

unlink removes the directory entry named by the path name pointed to by *path*. 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 of *path* is not a directory.
EINVAL	*path* contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
ENOENT	The file referred to by *path* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *path*.
EACCES	Write permission is denied for the directory containing the link to be removed.
ELOOP	Too many symbolic links were encountered in translating *path*.
EPERM	The file referred to by *path* is a directory and the effective user ID of the process is not the super-user.
EBUSY	The entry to be unlinked is the mount point for a mounted file system.
EIO	An I/O error occurred while reading from or writing to the file system.
EROFS	The file referred to by *path* 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

unmount – remove a file system

SYNOPSIS

unmount(name)
char *name;

DESCRIPTION

unmount announces to the system that the directory *name* is no longer to refer to the root of a mounted file system. The directory *name* reverts to its ordinary interpretation.

RETURN VALUE

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

ERRORS

unmount may fail with one of the following errors:

EPERM	The caller is not the super-user.
ENOTDIR	A component of the path prefix of *name* is not a directory.
EINVAL	*name* is not the root of a mounted file system.
EBUSY	A process is holding a reference to a file located on the file system.
EINVAL	The path name contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of the path name exceeds 255 characters, or the length of the entire path name exceeds 1023 characters.
ENOENT	*name* does not exist.
EACCES	Search permission is denied for a component of the path prefix.
EFAULT	*name* points outside the process's allocated address space.
ELOOP	Too many symbolic links were encountered in translating the path name.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

mount(2), mount(8), umount(8)

BUGS

The error codes are in a state of disarray; too many errors appear to the caller as one value.

NAME

utimes – set file times

SYNOPSIS

```
#include <sys/types.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 of *file* is not a directory.
EINVAL	*file* contained a character with the high-order bit set.
ENAMETOOLONG	The length of a component of *file* exceeds 255 characters, or the length of *file* exceeds 1023 characters.
ENOENT	The file referred to by *file* does not exist.
EACCES	Search permission is denied for a component of the path prefix of *file*.
ELOOP	Too many symbolic links were encountered in translating *file*.
EPERM	The process is not super-user and not the owner of the file.
EIO	An I/O error occurred while reading from or writing to the file system.
EROFS	The file system containing the file is mounted read-only.
EFAULT	*file* or *tvp* points outside the process's allocated address space.

SEE ALSO

stat(2)

NAME

vadvise – give advice to paging system

SYNOPSIS

#include <sys/vadvise.h>

vadvise(param)
int param;

DESCRIPTION

Vadvise is used to inform the system that process paging behavior merits special consideration. Parameters to *vadvise* are defined in the file **<vadvise.h>**. Currently, two calls t *vadvise* are implemented.

The call

vadvise(VA_ANOM);

advises that the paging behavior is not likely to be well handled by the system's default algorithm, since reference information is collected over macroscopic intervals (e.g. 10-20 seconds) will not serve to indicate future page references. The system in this case will choose to replace pages with little emphasis placed on recent usage, and more emphasis on referenceless circular behavior. It is *essential* that processes which have very random paging behavior (such as LISP during garbage collection of very large address spaces) call *vadvise*, as otherwise the system has great difficulty dealing with their page-consumptive demands.

The call

vadvise(VA_NORM);

restores default paging replacement behavior after a call to

vadvise(VA_ANOM);

BUGS

Will go away soon, being replaced by a per-page *madvise* facility.

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 which 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 *ioctl*s 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, wait3 – wait for process to terminate or stop

SYNOPSIS

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

pid = wait(status)
int pid;
union wait *status;

pid = wait(0)
int pid;

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

pid = wait3(status, options, rusage)
int pid;
union wait *status;
int options;
struct rusage *rusage;
```

DESCRIPTION

wait causes its caller to delay until a signal is received or one of its child processes terminates or stops due to tracing. If any child has died or stopped due to tracing and this has not been reported via *wait*, return is immediate, returning the process ID and exit status of one of those children. If that child had died, it is discarded. If there are no children, return is immediate with the value −1 returned. If there are only running or stopped but reported children, the calling processes is blocked.

On return from a successful *wait* call, *status* is nonzero, and the high byte of *status* contains the low byte of the argument to *exit* supplied by the child process; the low byte of *status* contains the termination status of the process. A more precise definition of the *status* word is given in **<sys/wait.h>**.

wait3 is an alternate interface that allows both non-blocking status collection and the collection of the status of children stopped by any means. The *status* parameter is defined as above. The *options* parameter is used to indicate the call should not block if there are no processes that have status to report (WNOHANG), and/or that children of the current process that are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal are eligible to have their status reported as well (WUNTRACED). A terminated child is discarded after it reports status, and a stopped process will not report its status more than once. 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 have status to report, *wait3* returns a *pid* of 0. The WNOHANG and WUNTRACED options may be combined by *or*'ing the two values.

NOTES

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

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

wait and *wait3* are automatically restarted when a process receives a signal while awaiting termination of a child process.

RETURN VALUE

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

wait3 returns −1 if there are no children not previously waited for; 0 is returned if WNOHANG is specified and there are no stopped or exited children.

ERRORS

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

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

EFAULT The *status* or *rusage* arguments point to an illegal address.

The call is forced to terminate prematurely due to the arrival of a signal whose.SM SV_INTERRUPT bit in **sv_flags** is set (see *sigvec*(2)). *signal*(3V), in the System V compatibility library, sets this bit for any signal it catches.

SEE ALSO

exit(2), getrusage(2)

NAME

write, writev – write output

SYNOPSIS

```
cc = write(d, buf, nbytes)
int cc, d;
char *buf;
int 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;
        int      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 O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

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 that are subject to flow control, such as sockets, pipes (or FIFOs), or terminals, *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. If such an object's buffers are full, so that it cannot accept any data, then *write* and *writev* will return –1 and set *errno* to EWOULDBLOCK. Otherwise, they will block until space becomes available.

SYSTEM V DESCRIPTION

A *write* (but not a *writev*) on an object that cannot accept any data will return a count of 0, rather than returning–1 and setting *errno* to EWOULDBLOCK.

RETURN VALUE

Upon successful completion the number of bytes actually writen 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 (or to a socket of type SOCK_STREAM that is connected to a peer socket.) Note: an attempted write of this kind will also cause you to recieve a SIGPIPE signal from the kernel. If you've not made a special provision to catch or ignore this signal, your process will die.

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.

The call is forced to terminate prematurely due to the arrival of a signal whose.SM SV_INTERRUPT bit in sv_flags is set (see *sigvec*(2)). *signal*(3V), in the System V compatibility library, sets this bit for any signal it catches.

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 One of the *iov_len* values in the *iov* array was negative.

EINVAL The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

SEE ALSO

fcntl(2), lseek(2), open(2V), pipe(2), select(2)

NAME

intro – introduction to library functions

DESCRIPTION

Section 3 describes library routines. The main C library is */lib/libc.a*, which contains all system call entry points described in section 2, as well as functions described in several subsections here. The primary functions are described in the main section 3. Functions associated with the "standard I/O library" used by many C programs are found in section 3S. The main C library also includes Internet network functions, described in section 3N, and routines providing compatibility with other UNIX systems, described in section 3C.

Other sections are:

(3F) This section, for FORTRAN library routines and functions, is contained in the *FORTRAN Programmer's Guide*.

(3M) The Math Library. C declarations for the types of functions are be obtained from the include file *<math.h>*. To use these functions with C programs compile them with the **–lm** option with *cc*(1). They are automatically loaded as needed by the FORTRAN and Pascal compilers *f77*(1) and *pc*(1).

(3V) The System V Compatibility Library. System V versions of functions that are not yet merged into the standard Sun libraries. To use these functions, compile programs with */usr/5bin/cc* , instead of */bin/cc*.

(3X) Various specialized libraries have not been given distinctive captions. Files in which such libraries are found are named on appropriate pages if they don't appear in the *libc* library.

FILES

/lib/libc.a	C Library ((2), (3), (3N) and (3C) routines)
/usr/lib/libc_p.a	Profiling C library (for *gprof*(1))
/usr/lib/libm.a	Math Library **–lm** (see section 3M)
/usr/lib/libm_p.a	Profiling version of **–lm**
/usr/lib/libcurses.a	screen management routines (see *curses*(3X)
/usr/lib/libdbm.a	data base management routines (see *dbm*(3X))
/usr/lib/libmp.a	multiple precision math library (see *mp*(3X))
/usr/lib/libtermcap.a	terminal handling routines (see *termcap*(3X))
/usr/lib/libtermcap_p.a	"
/usr/lib/libtermlib	(link to /usr/lib/libtermcap.a)
/usr/lib/libtermlib_p.a	(link to /usr/lib/libtermcap_p.a)
/usr/lib/libplot*.a	plot routines (see *plot*(3X))
/usr/lib/libresolv.a	Internet name server routines (see *resolver*(3X))

SEE ALSO

intro(3C), intro(3S), intro(3F), intro(3M), intro(3N), nm(1), ld(1), cc(1), f77(1), intro(2)

DIAGNOSTICS

Functions in the math library (section 3M) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable *errno* (see *intro*(2)) is set to the value EDOM (domain error) or ERANGE (range error). The values of EDOM and ERANGE are defined in the include file *<errno.h>*.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
a64l	a64l(3)	convert base-64 ASCII to long
abort	abort(3)	generate a fault
abs	abs(3)	integer absolute value
acos	sin(3M)	trigonometric functions
acosh	asinh(3M)	inverse hyperbolic function
addmntent	getmntent(3)	get file system descriptor file entry

alarm	alarm(3C)	schedule signal after specified time
alloca	malloc(3)	memory allocator
alphasort	scandir(3)	scan a directory
asctime	ctime(3)	convert date and time to ASCII
asin	sin(3M)	trigonometric functions
asinh	asinh(3M)	inverse hyperbolic function
assert	assert(3)	program verification
atan	sin(3M)	trigonometric functions
atanh	asinh(3M)	inverse hyperbolic function
atof	atof(3)	convert ASCII to numbers
atoi	atof(3)	convert ASCII to numbers
atol	atof(3)	convert ASCII to numbers
bcmp	bstring(3)	bit and byte string operations
bcopy	bstring(3)	bit and byte string operations
bsearch	bsearch(3)	binary search a sorted table
bzero	bstring(3)	bit and byte string operations
cabs	hypot(3M)	Euclidean distance
calloc	malloc(3)	memory allocator
cbc_crypt	des_crypt(3)	fast DES encryption
cbrt	sqrt(3M)	cube root
ceil	floor(3M)	ceiling
cfree	malloc(3)	memory allocator
clearerr	ferror(3S)	stream status inquiries
clock	clock(3C)	report CPU time used
closedir	directory(3)	directory operations
closelog	syslog(3)	control system log
copysign	ieee(3M)	copysign remainder exponent manipulations
cos	sin(3M)	trigonometric functions
cosh	sinh(3M)	hyperbolic functions
crypt	crypt(3)	DES encryption
ctermid	ctermid(3S)	generate filename for terminal
ctime	ctime(3)	convert date and time to ASCII
cuserid	cuserid(3S)	get character login name of user
des_crypt	des_crypt(3)	fast DES encryption
des_setparity	des_crypt(3)	fast DES encryption
dn_comp	resolver(3X)	Internet name server routines
dn_expand	resolver(3X)	Internet name server routines
drand48	drand48(3)	generate uniformly distributed pseudo-random numbers
drem	ieee(3M)	copysign remainder exponent manipulations
dysize	ctime(3)	convert date and time to ASCII
ecb_crypt	des_crypt(3)	fast DES encryption
ecvt	ecvt(3)	output conversion
edata	end(3)	last locations in program.
encrypt	crypt(3)	DES encryption
end	end(3)	last locations in program
endfsent	getfsent(3)	get file system descriptor file entry
endgrent	getgrent(3)	get group file entry
endhostent	gethostent(3N)	get network host entry
endmntent	getmntent(3)	get file system descriptor file entry
endnetent	getnetent(3N)	get network entry
endnetgrent	getnetgrent(3N)	get network group entry
endprotoent	getprotoent(3N)	get protocol entry
endpwent	getpwent(3)	get password file entry

endservent	getservent(3N)	get service entry
environ	execl(3)	execute a file
erand48	drand48(3)	generate uniformly distributed pseudo-random numbers
erf	erf(3M)	error functions
errno	perror(3)	system error messages
etext	end(3)	last locations in program
ether	ether(3R)	monitor traffic on the Ethernet
ether_aton	ethers(3N)	Ethernet address mapping
ether_hostton	ethers(3N)	Ethernet address mapping
ether_line(3N)	ethers	Ethernet address mapping
ether_ntoa	ethers(3N)	Ethernet address mapping
ether_ntohost	ethers(3N)	Ethernet address mapping
execl	execl(3)	execute a file
execle	execl(3)	execute a file
execlp	execl(3)	execute a file
execv	execl(3)	execute a file
execvp	execl(3)	execute a file
exit	exit(3)	terminate a process after performing cleanup
exp	exp(3M)	exponential function
fabs	floor(3M)	absolute value
fclose	fclose(3S)	close or flush a stream
fcvt	ecvt(3)	output conversion
fdopen	fopen(3S)	open a stream
feof	ferror(3S)	stream status inquiries
ferror	ferror(3S)	stream status inquiries
fflush	fclose(3S)	close or flush a stream
ffs	bstring(3)	bit and byte string operations
fgetc	getc(3S)	get character or integer from stream
fgets	gets(3S)	get a string from a stream
fileno	ferror(3S)	stream status inquiries
finite	ieee(3M)	copysign remainder exponent manipulations
floor	floor(3M)	floor function
fopen	fopen(3S)	open a stream
fprintf	printf(3S)	formatted output conversion
fputc	putc(3S)	put character or word on a stream
fputs	puts(3S)	put a string on a stream
fread	fread(3S)	buffered binary input/output
free	malloc(3)	memory allocator
freopen	fopen(3S)	open a stream
frexp	frexp(3)	split into mantissa and exponent
fscanf	scanf(3S)	formatted input conversion
fseek	fseek(3S)	reposition a stream
ftell	fseek(3S)	reposition a stream
ftime	time(3C)	get date and time
ftok	ftok(3)	standard interprocess communication package
ftw	ftw(3)	walk a file tree
fwrite	fread(3S)	buffered binary input/output
gcvt	ecvt(3)	output conversion
getc	getc(3S)	get character or integer from stream
getchar	getc(3S)	get character or integer from stream
getcwd	getcwd(3)	get pathname of current working directory
getenv	getenv(3)	value for environment name
getfsent	getfsent(3)	get file system descriptor file entry

getfsfile	getfsent(3)	get file system descriptor file entry
getfsspec	getfsent(3)	get file system descriptor file entry
getfstype	getfsent(3)	get file system descriptor file entry
getgrent	getgrent(3)	get group file entry
getgrgid	getgrent(3)	get group file entry
getgrnam	getgrent(3)	get group file entry
gethostbyaddr	gethostent(3N)	get network host entry
gethostbyname	gethostent(3N)	get network host entry
gethostent	gethostent(3N)	get network host entry
getlogin	getlogin(3)	get login name
getmntent	getmntent(3)	get file system descriptor file entry
getnetbyaddr	getnetent(3N)	get network entry
getnetbyname	getnetent(3N)	get network entry
getnetent	getnetent(3N)	get network entry
getnetgrent	getnetgrent(3N)	get network group entry
getopt	getopt(3)	get option letter from argv
getpass	getpass(3)	read a password
getprotobyname	getprotoent(3N)	get protocol entry
getprotobynumber	getprotoent(3N)	get protocol entry
getprotoent	getprotoent(3N)	get protocol entry
getpw	getpw(3)	get name from uid
getpwent	getpwent(3)	get password file entry
getpwnam	getpwent(3)	get password file entry
getpwuid	getpwent(3)	get password file entry
getrpcbyname	getrpcent(3N)	get RPC entry
getrpcbynumber	getrpcent(3N)	get RPC entry
getrpcent	getrpcent(3N)	get RPC entry
getrpcport	getrpcport(3R)	get RPC port number
gets	gets(3S)	get a string from a stream
getservbyname	getservent(3N)	get service entry
getservbyport	getservent(3N)	get service entry
getservent	getservent(3N)	get service entry
getw	getc(3S)	get character or integer from stream
getwd	getwd(3)	get current working directory pathname
gmtime	ctime(3)	convert date and time to ASCII
gsignal	signal(3)	software signals
gtty	stty(3C)	set and get terminal state
hasmntopt	getmntent(3)	get file system descriptor file entry
havedisk	rstat(3R)	get remote host performance data
hcreate	hsearch(3)	manage hash search tables
hdestroy	hsearch(3)	manage hash search tables
hsearch	hsearch(3)	manage hash search tables
htonl	byteorder(3N)	convert values between host and network byte order
htons	byteorder(3N)	convert values between host and network byte order
hypot	hypot(3M)	Euclidean distance
ieee	ieee(3M)	copysign remainder exponent manipulations
index	string(3)	string operations
inet_addr	inet(3N)	Internet address manipulation
inet_lnaof	inet(3N)	Internet address manipulation
inet_makeaddr	inet(3N)	Internet address manipulation
inet_netof	inet(3N)	Internet address manipulation
inet_network	inet(3N)	Internet address manipulation
inet_ntoa	inet(3N)	Internet address manipulation

initgroups	initgroups(3)	initialize group access list
initstate	random(3)	better random number generator
innetgr	getnetgrent(3N)	get network group entry
insque	insque(3)	insert/remove element from a queue
isalnum	ctype(3)	character classification and conversion macros
isalpha	ctype(3)	character classification and conversion macros
isascii	ctype(3)	character classification and conversion macros
isatty	ttyname(3)	find name of a terminal
iscntrl	ctype(3)	character classification and conversion macros
isdigit	ctype(3)	character classification and conversion macros
isgraph	ctype(3)	character classification and conversion macros
isinf	isinf(3)	test for indeterminate floating point values
islower	ctype(3)	character classification and conversion macros
isnan	isinf(3)	test for indeterminate floating point values
isprint	ctype(3)	character classification and conversion macros
ispunct	ctype(3)	character classification and conversion macros
isspace	ctype(3)	character classification and conversion macros
isupper	ctype(3)	character classification and conversion macros
isxdigit	ctype(3)	character classification and conversion macros
j0	j0(3M)	Bessel functions
j1	j0(3M)	Bessel functions
jn	j0(3M)	Bessel functions
jrand48	drand48(3)	generate uniformly distributed pseudo-random numbers
l64a	a64l(3)	convert long to base-64 ASCII
lcong48	drand48(3)	generate uniformly distributed pseudo-random numbers
ldexp	frexp(3)	split into mantissa and exponent
lfind	lsearch(3)	linear search and update
lgamma	lgamma(3M)	log gamma function
localtime	ctime(3)	convert date and time to ASCII
lockf	lockf(3)	advisory record locking on files
log	exp(3M)	exponential functions
log10	exp(3M)	exponential functions
logb	ieee(3M)	copysign remainder exponent manipulations
longjmp	setjmp(3)	non-local goto
lrand48	drand48(3)	generate uniformly distributed pseudo-random numbers
lsearch	lsearch(3)	linear search and update
malloc	malloc(3)	memory allocator
malloc_debug	malloc(3)	memory allocator
malloc_verify	malloc(3)	memory allocator
matherr	matherr(3M)	math library error-handling function
memalign	malloc(3)	memory allocator
memccpy	memory(3)	memory operations
memchr	memory(3)	memory operations
memcmp	memory(3)	memory operations
memcpy	memory(3)	memory operations
memset	memory(3)	memory operations
mkstemp	mktemp(3)	make a unique file name
mktemp	mktemp(3)	make a unique file name
modf	frexp(3)	split into mantissa and exponent
moncontrol	monitor(3)	prepare execution profile
monitor	monitor(3)	prepare execution profile
monstartup	monitor(3)	prepare execution profile
mrand48	drand48(3)	generate uniformly distributed pseudo-random numbers

nice	nice(3C)	set program priority
nlist	nlist(3)	get entries from name list
nrand48	drand48(3)	generate uniformly distributed pseudo-random numbers
ntohl	byteorder(3N)	convert values between host and network byte order
ntohs	byteorder(3N)	convert values between host and network byte order
on_exit	onexit(3)	name termination handler
opendir	directory(3)	directory operations
openlog	syslog(3)	control system log
optarg	getopt(3)	get option letter from argv
optind	getopt(3)	get option letter from argv
pause	pause(3C)	stop until signal
pclose	popen(3S)	initiate I/O to/from a process
perror	perror(3)	system error messages
popen	popen(3S)	initiate I/O to/from a process
pow	exp(3M)	exponential functions
printf	printf(3S)	formatted output conversion
prof	prof(3)	profile within a function
psignal	psignal(3)	system signal messages
putc	putc(3S)	put character or word on a stream
putchar	putc(3S)	put character or word on a stream
putenv	utenv(3)	change or add value to environment
putpwent	putpwent(3)	write password file entry
puts	puts(3S)	put a string on a stream
putw	putc(3S)	put character or word on a stream
qsort	qsort(3)	quicker sort
rand	rand(3C)	random number generator
random	random(3)	better random number generator
rcmd	rcmd(3N)	routines for returning a stream to a remote command
re_comp	regex(3)	regular expression handler
re_exec	regex(3)	regular expression handler
readdir	directory(3)	directory operations
realloc	malloc(3)	memory allocator
regexp	regexp(3)	regular expression compile and match routines
remque	insque(3)	insert/remove element from a queue
res_init	resolver(3X)	Internet name server routines
res_mkquery	resolver(3)	Internet name server routines
res_send	resolver(3)	Internet name server routines
rewind	fseek(3S)	reposition a stream
rewinddir	directory(3)	directory operations
rex	rex(3R)	remote execution protocol
rexec	rexec(3N)	return stream to a remote command
rindex	string(3)	string operations
rint	floor(3M)	round to nearest integer
rnusers	rnusers(3R)	return info about users on remote hosts
rquota	rquota(3R)	implement quotas on remote hosts
rresvport	rcmd(3N)	routines for returning a stream to a remote command
rstat	rstat(3R)	get remote host performance data
ruserok	rcmd(3N)	routines for returning a stream to a remote command
rusers	rnusers(3R)	return info about users on remote hosts
rwall	rwall(3R)	write to remote host
scalb	ieee(3M)	copysign remainder exponent manipulations
scandir	scandir(3)	scan a directory
scanf	scanf(3S)	formatted input conversion

seed48	drand48(3)	generate uniformly distributed pseudo-random numbers
seekdir	directory(3)	directory operations
setbuf	setbuf(3S)	assign buffering to a stream
setbuffer	setbuf(3S)	assign buffering to a stream
setegid	setuid(3)	set user and group ID
seteuid	setuid(3)	set user and group ID
setfsent	getfsent(3)	get file system descriptor file entry
setgid	setuid(3)	set user and group ID
setgrent	getgrent(3)	get group file entry
sethostent	gethostent(3N)	get network host entry
setjmp	setjmp(3)	non-local goto
setkey	crypt(3)	DES encryption
setlinebuf	setbuf(3S)	assign buffering to a stream
setlinebuf	setbuf(3S)	assign buffering to a stream
setmntent	getmntent(3)	get file system descriptor file entry
setnetent	getnetent(3N)	get network entry
setnetgrent	getnetgrent(3N)	get network group entry
setprotoent	getprotoent(3N)	get protocol entry
setpwent	getpwent(3)	get password file entry
setrgid	setuid(3)	set user and group ID
setruid	setuid(3)	set user and group ID
setservent	getservent(3N)	get service entry
setstate	random(3)	better random number generator
setuid	setuid(3)	set user and group ID
setvbuf	setbuf(3S)	assign buffering to a stream
siginterrupt	siginterrupt(3)	allow signals to interrupt system calls
signal	signal(3)	simplified software signal facilities
sin	sin(3M)	trigonometric functions
sinh	sinh(3M)	hyperbolic functions
sleep	sleep(3)	suspend execution for interval
spray	spray(3R)	scatter packets to check network
sprintf	printf(3S)	formatted output conversion
sqrt	sqrt(3M)	square root
srand	rand(3C)	random number generator
srand48	drand48(3)	generate uniformly distributed pseudo-random numbers
srandom	random(3)	better random number generator
sscanf	scanf(3S)	formatted input conversion
ssignal	ssignal(3)	software signals
stdio	intro(3S)	standard buffered input/output package
strcat	string(3)	string operations
strcmp	string(3)	string operations
strcpy	string(3)	string operations
strlen	string(3)	string operations
strncat	string(3)	string operations
strncmp	string(3)	string operations
strncpy	string(3)	string operations
strtod	strtod(3)	convert string to double-precision number
strtol	strtol(3)	convert string to integer
stty	stty(3C)	set and get terminal state
swab	swab(3)	swap bytes
sys_errlist	perror(3)	system error messages
sys_nerr	perror(3)	system error messages
sys_siglist	psignal(3)	system signal messages

syslog	syslog(3)	control system log
system	system(3)	issue a shell command
tan	sin(3M)	trigonometric functions
tanh	sinh(3M)	hyperbolic functions
tdelete	tsearch(3)	manage binary search trees
telldir	directory(3)	directory operations
tfind	tsearch(3)	manage binary search trees
time	time(3C)	get date and time
times	times(3C)	get process times
timezone	ctime(3)	convert date and time to ASCII
tmpfile	tmpfile(3S)	create a temporary file
tmpnam	tmpnam(3S)	create a name for a temporary file
toascii	ctype(3)	character classification and conversion macros
tolower	ctype(3)	character classification and conversion macros
toupper	ctype(3)	character classification and conversion macros
tsearch	tsearch(3)	manage binary search trees
ttyname	ttyname(3)	find name of a terminal
ttyslot	ttyname(3)	find name of a terminal
twalk	tsearch(3)	manage binary search trees
ualarm	ualarm(3)	schedule signal after microsecond interval
ulimit	ulimit(3C)	get and set user limits
ungetc	ungetc(3S)	push character back into input stream
usleep	usleep(3S)	suspend execution for micorsecond interval
utime	utime(3C)	set file times
valloc	valloc(3)	aligned memory allocator
values	values(3)	machine-dependent values
varargs	varargs(3)	variable argument list
vfprintf	vprintf(3S)	print formatted output of a varargs argument list
vlimit	vlimit(3C)	control maximum system resource consumption
vprintf	vprintf(3S)	print formatted output of a varargs argument list
vsprintf	vprintf(3S)	print formatted output of a varargs argument list
vtimes	vtimes(3C)	get information about resource utilization
y0	j0(3M)	Bessel functions
y1	j0(3M)	Bessel functions
yn	j0(3M)	Bessel functions
yp_all	ypclnt(3N)	YP client interface routines
yp_bind	ypclnt(3N)	YP client interface routines
yp_first	ypclnt(3N)	YP client interface routines
yp_get_default_domain	ypclnt(3N)	YP client interface routines
yp_master	ypclnt(3N)	YP client interface routines
yp_match	ypclnt(3N)	YP client interface routines
yp_next	ypclnt(3N)	YP client interface routines
yp_order	ypclnt(3N)	YP client interface routines
yp_unbind	ypclnt(3N)	YP client interface routines
ypclnt	ypclnt(3N)	YP client interface routines
yperr_string	ypclnt(3N)	YP client interface routines
yppasswd	yppasswd(3R)	update user YP password
ypprot_err	ypclnt(3N)	YP client interface routines

NAME

a64l, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

DESCRIPTION

to long integer" These functions are used to maintain numbers stored in *base-64* ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2–11, **A** through **Z** for 12–37, and **a** through **z** for 38–63.

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

l64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

BUGS

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abort – generate a fault

SYNOPSIS

abort()

DESCRIPTION

abort first closes all open files if possible, then causes an IOT signal to be sent to the process. This signal usually results in termination with a core dump, which may be used for debugging.

It is possible for *abort* to return control if SIGIOT is caught or ignored, in which case the value returned is that of the *kill*(2) system call.

SEE ALSO

adb(1), signal(3), exit(2), kill(2)

DIAGNOSTICS

If SIGIOT is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message "abort – core dumped" is written by the shell.

NAME

abs – integer absolute value

SYNOPSIS

abs(i)
int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

SEE ALSO

floor(3M) for *fabs*

BUGS

Applying the *abs* function to the most negative integer generates a result which is the most negative integer. That is, abs(0x80000000) returns 0x80000000 as a result.

NAME

assert – program verification

SYNOPSIS

#include <assert.h>

assert(expression)

DESCRIPTION

Assert is a macro that indicates *expression* is expected to be true at this point in the program. It causes an *exit*(2) with a diagnostic comment on the standard output when *expression* is false (0). Compiling with the *cc*(1) option **–DNDEBUG** effectively deletes *assert* from the program.

DIAGNOSTICS

'Assertion failed: file *f* line *n*.' *F* is the source file and *n* the source line number of the *assert* statement.

NAME

bsearch – binary search a sorted table

SYNOPSIS

#include <search.h>

char *bsearch ((char *) key, (char *) base, nel, sizeof (*key), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *key* points to a datum instance to be sought in the table. *base* points to the element at the base of the table. *nel* is the number of elements in the table. *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero as accordingly the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node, in which case it prints out the string and its length, or it prints an error message.

```
#include <stdio.h>
#include <search.h>

#define  TABSIZE          1000

struct node {                        /* these are stored in the table */
        char *string;
        int length;
};
struct node table[TABSIZE];          /* table to be searched */
        .
        .
        .
{
        struct node *node_ptr, node;
        int node_compare( );  /* routine to compare 2 nodes */
        char str_space[20];   /* space to read string into */
        .
        .
        .
        node.string = str_space;
        while (scanf("%s", node.string) != EOF) {
                node_ptr = (struct node *)bsearch((char *)(&node),
                        (char *)table, TABSIZE,
                        sizeof(struct node), node_compare);
                if (node_ptr != NULL) {
                        (void)printf("string = %20s, length = %d\n",
                                node_ptr->string, node_ptr->length);
                } else {
                        (void)printf("not found: %s\n", node.string);
                }
```

```
        }
}
/*
        This routine compares two nodes based on an
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
struct node *node1, *node2;
{
        return strcmp(node1->string, node2->string);
}
```

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

hsearch(3), lsearch(3), qsort(3), tsearch(3)

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

NAME

bstring, bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

```
bcopy(b1, b2, length)
char *b1, *b2;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b, length)
char *b;
int length;

ffs(i)
int i;
```

DESCRIPTION

The functions *bcopy*, *bcmp*, and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *string*(3) do.

Bcopy copies *length* bytes from string *b1* to the string *b2*. Overlapping strings are handled correctly.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* 0 bytes in the string *b*.

Ffs finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1 from the right. A return value of –1 indicates the value passed is zero.

CAVEAT

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy*.

NAME

crypt, setkey, encrypt – password and data encryption

SYNOPSIS

char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;

DESCRIPTION

crypt is the password encryption routine. It is *based* on the NBS Data Encryption Standard, with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The first argument to *crypt* is normally a user's typed password. The second is a 2-character string chosen from the set [**a-zA-Z0-9./**]. The *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The *setkey* and *encrypt* entries provide (rather primitive) access to the DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the above mentioned algorithm to encrypt or decrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

passwd(1), passwd(5), login(1), getpass(3)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ctime, localtime, gmtime, asctime, timezone, dysize – convert date and time to ASCII

SYNOPSIS

```
char *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)

int dysize(y)
int y;
```

DESCRIPTION

ctime converts to ASCII a long integer, pointed to by *clock*, that represents the time in seconds since Jan. 1, 1970, 00:00, Greenwich Mean Time. It returns a pointer to a 26-character string of the form:

```
Sun Sep 16 01:03:52 1973\n\0
```

Each field has a constant width. *localtime* and *gmtime* return pointers to structures containing the broken-down time. *localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time UNIX uses. *asctime* converts a broken-down time to ASCII and returns a pointer to a 26-character string.

Declarations of all the functions and externals, and the "tm" structure, are in the **<time.h>** header file. The structure declaration is:

```
struct tm {
    int tm_sec;         /* seconds (0 - 59) */
    int tm_min;         /* minutes (0 - 59) */
    int tm_hour;        /* hours (0 - 23) */
    int tm_mday;        /* day of month (1 - 31) */
    int tm_mon;         /* month of year (0 - 11) */
    int tm_year;        /* year – 1900 */
    int tm_wday;        /* day of week (Sunday = 0) */
    int tm_yday;        /* day of year (0 - 365) */
    int tm_isdst;
};
```

tm_isdst is non-zero if Daylight Savings Time is in effect.

When local time is called for, the program consults the system to determine the time zone and whether the U.S.A., Canadian, Australian, Eastern European, Middle European, or Western European daylight saving time adjustment is appropriate. The program knows about various peculiarities in time conversion over the past 10-20 years.

timezone returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Savings Time version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; e.g., in Afghanistan *timezone(–(60*4+30), 0)* is appropriate because it is 4:30 ahead of GMT and the string **GMT+4:30** is produced.

dysize returns the number of days in the argument year, either 365 or 366.

SEE ALSO

gettimeofday(2), time(3C), getenv(3), environ(5V), ctime(3V)

BUGS

The return values point to static data, whose contents are overwritten by each call.

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii, isgraph, toupper, tolower, toascii – character classification and conversion macros and functions

SYNOPSIS

#include <ctype.h>

isalpha(c)

. . .

CHARACTER CLASSIFICATION MACROS

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *isascii* is defined on all integer values; the rest are defined only where isascii(*c*) is true and on the single non-ASCII value EOF (see *stdio*(3S)).

isalpha(c) *c* is a letter

isupper(c) *c* is an upper case letter

islower(c) *c* is a lower case letter

isdigit(c) *c* is a digit [0-9].

isxdigit(c) *c* is a hexadecimal digit [0-9], [A-F], or [a-f].

isalnum(c) *c* is an alphanumeric character, that is, *c* is a letter or a digit

isspace(c) *c* is a space, tab, carriage return, newline, vertical tab, or formfeed

ispunct(c) *c* is a punctuation character (neither control nor alphanumeric)

isprint(c) *c* is a printing character, code 040(8) (space) through 0176 (tilde)

iscntrl(c) *c* is a delete character (0177) or ordinary control character (less than 040).

isascii(c) *c* is an ASCII character, code less than 0200

isgraph(c) *c* is a visible graphic character, code 041 (exclamation mark) through 0176 (tilde).

CHARACTER CONVERSION MACROS

These macros perform simple conversions on single characters.

toupper(c) converts *c* to its upper-case equivalent. Note that this *only* works where *c* is known to be a lower-case character to start with (presumably checked via *islower*).

tolower(c) converts *c* to its lower-case equivalent. Note that this *only* works where *c* is known to be a upper-case character to start with (presumably checked via *isupper*).

toascii(c) masks *c* with the correct value so that *c* is guaranteed to be an ASCII character in the range 0 thru 0x7f.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

stdio(3S), ascii(7), ctype(3V)

NAME

des_crypt, ecb_crypt, cbc_crypt, des_setparity – fast DES encryption

SYNOPSIS

```
#include <des_crypt.h>

int ecb_crypt(key, data, datalen, mode)
char *key;
char *data;
unsigned datalen;
unsigned mode;

int cbc_crypt(key, data, datalen, mode, ivec)
char *key;
char *data;
unsigned datalen;
unsigned mode;
char *ivec;

void des_setparity(key)
char *key;
```

DESCRIPTION

ecb_crypt and *cbc_crypt* implement the NBS Data Encryption Standard (DES). These routines are faster and more general purpose than *crypt*(3). They also are able to utilize DES hardware if it is available. *ecb_crypt* encrypts in Electronic Code Book (ECB) mode, which encrypts blocks of data independently. *cbc_crypt* encrypts in Cipher Block Chaining (CBC) mode, which chains together successive blocks. CBC mode protects against insertions, deletions and substitutions of blocks. Also, regularities in the clear text will not appear in the cipher text.

Here is how to use these routines. The first parameter, *key*, is the 8-byte encryption key with parity. To set the key's parity, which for DES is in the low bit of each byte, use *des_setparity*. The second parameter, *data*, contains the data to be encrypted or decrypted. The third parameter, *datalen*, is the length in bytes of *data*, which must be a multiple of 8. The fourth parameter, *mode*, is formed by or'ing together some things. For the encryption direction 'or' in either DES_ENCRYPT or DES_DECRYPT. For software versus hardware encryption, 'or' in either DES_HW or DES_SW. If DES_HW is specified, and there is no hardware, then the encryption is performed in software and the routine returns DESERR_NOHWDEVICE. For *cbc_crypt*, the parameter *ivec* is the the 8-byte initialization vector for the chaining. It is updated to the next initialization vector upon return.

DIAGNOSTICS

DESERR_NONE
: no error.

DESERR_NOHWDEVICE
: encryption succeeded, but done in software instead of the requested hardware.

DESERR_HWERR
: an error occurred in the hardware or driver.

DESERR_BADPARAM
: bad parameter to routine.

Given a result status *stat*, the macro DES_FAILED(*stat*) is false only for the first two statuses.

RESTRICTIONS

These routines are not available for export outside the U.S.

SEE ALSO

crypt(3), des(1)

NAME

directory, opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

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

DIR *opendir(filename)
char *filename;

struct direct *readdir(dirp)
DIR *dirp;

long telldir(dirp)
DIR *dirp;

seekdir(dirp, loc)
DIR *dirp;
long loc;

rewinddir(dirp)
DIR *dirp;

closedir(dirp)
DIR *dirp;
```

DESCRIPTION

opendir opens the directory named by *filename* and associates a *directory stream* with it. *opendir* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed or is not a directory, or if it cannot *malloc*(3) enough memory to hold the whole thing.

readdir returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or detecting an invalid *seekdir* operation.

telldir returns the current location associated with the named *directory stream.*

seekdir sets the position of the next *readdir* operation on the *directory stream.* The new position reverts to the one associated with the *directory stream* when the *telldir* operation was performed. Values returned by *telldir* are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir*.

Rewinddir resets the position of the named *directory stream* to the beginning of the directory.

closedir closes the named *directory stream* and frees the structure associated with the DIR pointer.

Sample code which searchs a directory for entry ''name'' is:

```
len = strlen(name);
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
        if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                closedir(dirp);
                return FOUND;
        }
closedir(dirp);
return NOT_FOUND;
```

SEE ALSO

open(2), close(2), read(2), lseek(2), getwd(3), dir(5)

NOTES

All UNIX programs that examine directories must be converted to use this package in Sun release 3.0 and beyond. Direct reading of directories is no longer allowed. SH BUGS The new directory format is not obvious.

delim $$

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval $[0.0,~1.0).$

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0,~2 sup 31).$

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2 sup 31 ,~2 sup 31).$

Functions *srand48*, *seed48*, and *lcong48* are initialization entry points, one of which should be invoked before either *drand48*, *lrand48*, or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48*, *lrand48*, or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48*, *nrand48*, and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, $X sub i ,$ according to the linear congruential formula

$$X sub{n+1}~=~(aX sub n~+~c) sub{roman mod~m}~~~~~~~~n>=0.$$

The parameter $m~=~2 sup 48$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by

a~mark =~roman 5DEECE66D~sub 16~=~roman 273673163155~sub 8
c~lineup =~roman B~sub 16~=~roman 13~sub 8 .

The value returned by any of the functions *drand48*, *erand48*, *lrand48*, *nrand48*, *mrand48*, or *jrand48* is computed by first generating the next 48-bit $X sub i$ in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of $X sub i$ and transformed into the returned value.

The functions *drand48*, *lrand48*, and *mrand48* store the last 48-bit $X sub i$ generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions *erand48*, *nrand48*, and *jrand48* require the calling program to provide storage for the successive $X sub i$ values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of $X sub i$ into the array and pass it as an argument. By using different arguments, functions *erand48*, *nrand48*, and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of $X sub i$ to the 32 bits contained in its argument. The low-order 16 bits of $X sub i$ are set to the arbitrary value $roman 330E sub 16 .$

The initializer function *seed48* sets the value of $X sub i$ to the 48-bit value specified in the argument array. In addition, the previous value of $X sub i$ is copied into a 48-bit internal buffer, used only by *seed48*, and a pointer to this buffer is the value returned by *seed48*. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last $X sub i$ value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial $X sub i ,$ the multiplier value $a,$ and the addend value $c.$ Argument array elements *param[0-2]* specify $X sub i ,$ *param[3-5]* specify the multiplier $a,$ and *param[6]* specifies the 16-bit addend $c.$ After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, a and $c,$ specified on the previous page.

SEE ALSO

rand(3C)

NAME

ecvt, fcvt, gcvt – output conversion

SYNOPSIS

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

Ecvt converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by *ndigits*.

Gcvt converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in Fortran F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO

isinf(3), printf(3S)

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

end, etext, edata – last locations in program

SYNOPSIS

```
extern end;
extern etext;
extern edata;
```

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with *end*, but it is reset by the routines *brk*(2), *malloc*(3), standard input/output (*stdio*(3S)), the profile (**-p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see *brk*(2)).

SEE ALSO

brk(2), malloc(3)

NAME

execl, execv, execle, execlp, execvp – execute a file

SYNOPSIS

```
execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name, *argv[ ];

execle(name, arg0, arg1, ..., argn, 0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[ ];

execlp(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execvp(name, argv)
char *name, *argv[ ];

extern char **environ;
```

DESCRIPTION

These routines provide various interfaces to the *execve* system call. Refer to *execve*(2) for a description of their properties; only brief descriptions are provided here.

Exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful exec; the calling core image is lost.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[*0*], *arg*[*1*] ... address null-terminated strings. Conventionally *arg*[*0*] is the name of the file.

Two interfaces are available. *execl* is useful when a known file with known arguments is being called; the arguments to *execl* are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The *execv* version is useful when the number of arguments is unknown in advance; the arguments to *execv* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

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

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

Argv is directly usable in another *execv* because *argv*[*argc*] is 0.

Envp is a pointer to an array of strings that constitute the *environment* of the process. 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*(5V) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by *execv* and *execl* to pass the environment to any subprograms executed by the current program.

Execlp and *execvp* are called with the same arguments as *execl* and *execv*, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

FILES

/bin/sh shell, invoked if command file found by *execlp* or *execvp*

SEE ALSO

execve(2), fork(2), environ(5V), csh(1), sh(1)

UNIX Programming in *Programming Utilities for the Sun Workstation,*

UNIX Interface Overview

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see *a.out*(5)), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is −1. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

NAME

exit – terminate a process after performing cleanup

SYNOPSIS

exit(status)
int status;

DESCRIPTION

Exit terminates a process by calling *exit*(2) after calling any termination handlers named by calls to *on_exit*. Normally, this is just the Standard I/O library function *_cleanup*. *Exit* never returns.

SEE ALSO

exit(2), intro(3S), on_exit(3)

NAME

fdate – return date and time in an ASCII string

SYNOPSIS

subroutine fdate (string)
character*24 string

character*24 function fdate()

DESCRIPTION

fdate returns the current date and time as a 24 character string in the format described under *ctime*(3). Neither 'newline' nor NULL will be included.

fdate can be called either as a function or as a subroutine. If called as a function, the calling routine must define its type and length. For example:

```
character*24   fdate
write(*,*) fdate()
```

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), time(3F), idate(3F)

NAME

frexp, ldexp, modf – floating point analysis and synthesis

SYNOPSIS

```
double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;
int exp;

double modf(value, iptr)
double value, *iptr;
```

DESCRIPTION

Frexp returns the significand of a double *value* as a double quantity, x, of magnitude less than 1 and stores an integer n, indirectly through *eptr*, such that $value = x * 2^n$.

The results are not defined when *value* is an IEEE infinity or NaN.

ldexp returns the quantity:

$$value * 2^{exp}.$$

modf returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*. Thus the argument *value* and the returned values *modf* and **iptr* would satisfy, in the absence of rounding error,

$$(*iptr + modf) == value$$

and

$$0 <= modf < abs(value).$$

The results are not defined when *value* is an IEEE infinity or NaN.

Note that the definition of *modf* varies among Unix implementations; avoid *modf* in portable code.

SEE ALSO

isinf(3)

NAME

ftok – standard interprocess communication package

SYNOPSIS

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

key_t ftok(path, id)
char *path;
char id;
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *path* must be the path name of an existing file that is accessible to the process. *id* is a character which uniquely identifies a project. Note that *ftok* will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2)

DIAGNOSTICS

ftok returns (key_t) **–1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, then *ftok* is likely to return a different key than it did the original time it was called.

NAME

ftw – walk a file tree

SYNOPSIS

#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;

DESCRIPTION

ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see *stat*(2)) containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns –1, and sets the error type in *errno*.

ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. *Ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3)

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.
It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.
ftw uses *malloc*(3) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

NAME

getcwd – get pathname of current working directory

SYNOPSIS

char *getcwd (buf, size)
char *buf;
int size;

DESCRIPTION

getcwd returns a pointer to the current directory pathname. The value of *size* must be at least two greater than the length of the pathname to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*.

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(1);
}
printf("%s\n", cwd);
```

SEE ALSO

malloc(3), popen(3S), pwd(1)

DIAGNOSTICS

Returns **NULL** with *errno* set if *size* is not large enough, or if an error ocurrs in a lower-level function.

BUGS

Since this function uses *popen* to create a pipe to the *pwd* command, it is slower than *getwd* and gives poorer error diagnostics. *getcwd* is provided only for compatibility with other UNIX systems.

NAME

getenv – return value for environment name

SYNOPSIS

```
char *getenv(name)
char *name;
```

DESCRIPTION

Getenv searches the environment list (see *environ*(5V)) for a string of the form *name*=*value*, and returns a pointer to the string *value* if such a string is present, otherwise NULL pointer.

SEE ALSO

environ(5V), execve(2), putenv(3)

NAME

getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent – get file system descriptor file entry

SYNOPSIS

```
#include <fstab.h>

struct fstab *getfsent()

struct fstab *getfsspec(spec)
char *spec;

struct fstab *getfsfile(file)
char *file;

struct fstab *getfstype(type)
char *type;

int setfsent()

int endfsent()
```

DESCRIPTION

These routines are included for compatibility with 4.2 BSD; they have been superseded by the *getmntent*(3) library routines.

getfsent, *getfsspec*, *getfstype*, and *getfsfile* each return a pointer to an object with the following structure containing the broken-out fields of a line in the file system description file, **<fstab.h>**.

```
struct fstab {
        char    *fs_spec;
        char    *fs_file;
        char    *fs_type;
        int     fs_freq;
        int     fs_passno;
};
```

The fields have meanings described in *fstab*(5).

getfsent reads the next line of the file, opening the file if necessary.

setfsent opens and rewinds the file.

endfsent closes the file.

getfsspec and *getfsfile* sequentially search from the beginning of the file until a matching special file name or file system file name is found, or until EOF is encountered. *getfstype* does likewise, matching on the file system type field.

FILES

/etc/fstab

SEE ALSO

fstab(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

The return value points to static information which is overwritten in each call.

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent()

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(name)
char *name;

setgrent()

endgrent()

struct group *fgetgrent(f)
FILE
*f;
```

DESCRIPTION

Getgrent, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the group file. Each line contains a "group" structure, defined in the **<grp.h>** header file.

```
struct   group {
             char    *gr_name;
             char    *gr_passwd;
             int     gr_gid;
             char    **gr_mem;
};
```

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group ID.
gr_mem	A null-terminated array of pointers to the individual member names.

Getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *Getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found. *Getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *Endgrent* may be called to close the group file when processing is complete.

Fgetgrent returns a pointer to the next group structure in the stream *f*, which must refer to an open file in the same format as the group file **/etc/group**.

FILES

/etc/group
/etc/yp/*domainname*/group.byname
/etc/yp/*domainname*/group.bygid

SEE ALSO

getlogin(3), getpwent(3), group(5), ypserv(8)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

Unlike the corresponding routines for passwords (see *getwpent*(3)), which always search the entire file, these routines start searching from the current file location.

NAME

getlogin – get login name

SYNOPSIS

char *getlogin()

DESCRIPTION

getlogin returns a pointer to the login name as found in **/etc/utmp**. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, or if there is no entry in **/etc/utmp** for the process's terminal, it returns a NULL pointer. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and, if it fails, to call *getpwuid*(*getuid*()).

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getpwent(3), utmp(5)

DIAGNOSTICS

Returns a NULL pointer if the name is not found.

BUGS

The return values point to static data whose content is overwritten by each call.

getlogin does not work for processes running under a *pty* (for example, emacs shell buffers, or shell tools) unless the program "fakes" the login name in the **/etc/utmp** file.

NAME

getmntent, setmntent, addmntent, endmntent, hasmntopt – get file system descriptor file entry

SYNOPSIS

```
#include <stdio.h>
#include <mntent.h>

FILE *setmntent(filep, type)
char *filep;
char *type;

struct mntent *getmntent(filep)
FILE *filep;

int addmntent(filep, mnt)
FILE *filep;
struct mntent *mnt;

char *hasmntopt(mnt, opt)
struct mntent *mnt;
char *opt;

int endmntent(filep)
FILE *filep;
```

DESCRIPTION

These routines replace the *getfsent* routines for accessing the file system description file */etc/fstab*. They are also used to access the mounted file system description file */etc/mtab*.

Setmntent opens a file system description file and returns a file pointer which can then be used with *getmntent*, *addmntent*, or *endmntent*. The *type* argument is the same as in *fopen*(3). *Getmntent* reads the next line from *filep* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the filesystem description file, *<mntent.h>*. The fields have meanings described in *fstab*(5).

```
struct mntent {
        char    *mnt_fsname;   /* file system name */
        char    *mnt_dir;      /* file system path prefix */
        char    *mnt_type;     /* 4.2, nfs, swap, or xx */
        char    *mnt_opts;     /* ro, quota, etc. */
        int     mnt_freq;      /* dump frequency, in days */
        int     mnt_passno;    /* pass number on parallel fsck */
};
```

Addmntent adds the *mntent* structure *mnt* to the end of the open file *filep*. Note that *filep* has to be opened for writing if this is to work. *Hasmntopt* scans the *mnt_opts* field of the *mntent* structure *mnt* for a substring that matches *opt*. It returns the address of the substring if a match is found, 0 otherwise. *Endmntent* closes the file.

FILES

/etc/fstab
/etc/mtab

SEE ALSO

fstab(5), getfsent(3)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

The returned *mntent* structure points to static information that is overwritten in each call.

NAME

getopt, optarg, optind – get option letter from argument vector

SYNOPSIS

```
int getopt(argc, argv, optstring)
int args;
char **argv;
char *optstring;

extern char *optarg;
extern int optind, opterr;
```

DESCRIPTION

getopt returns the next option letter in *argv* that matches a letter in *optstring*. *optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *optarg* is set to point to the start of the option argument on return from *getopt*.

getopt places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns EOF. The special option — may be used to delimit the end of the options; EOF will be returned, and — will be skipped.

DIAGNOSTICS

getopt prints an error message on *stderr* and returns a question mark (?) when it encounters an option letter not included in *optstring*. This error message may be disabled by setting *opterr* to zero.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main(argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind;
        extern char *optarg;
        .
        .
        .
        while ((c = getopt(argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc();
                        break;
                case 'f':
```

```
                        infile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        bufsiza = 512;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf(stderr, "usage: . . . ");
                exit(2);
        }
        for (; optind < argc; optind++) {
                if (access(argv[optind], 4)) {
        .
        .
        .
}
```

SEE ALSO

getopt(1)

NAME

getpass – read a password

SYNOPSIS

char *getpass(prompt)
char *prompt;

DESCRIPTION

getpass reads up to a newline or EOF from the file **/dev/tty**, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

SEE ALSO

crypt(3), getpass(3V)

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw – get name from uid

SYNOPSIS

getpw(uid, buf)
char *buf;

DESCRIPTION

Getpw is made obsolete by getpwent(3).

Getpw searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

FILES

/etc/passwd

SEE ALSO

getpwent(3), passwd(5)

DIAGNOSTICS

Non-zero return on error.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(uid)
int uid;

struct passwd *getpwnam(name)
char *name;

int setpwent()

int endpwent()

struct passwd *fgetpwent(f)
FILE *f;
```

DESCRIPTION

getpwent, *getpwuid* and *getpwnam* each return a pointer to an object with the following structure containing the broken-out fields of a line in the password file. Each line in the file contains a "passwd" structure, declared in the **<pwd.h>** header file:

```
struct   passwd { /* see getpwent(3) */
         char    *pw_name;
         char    *pw_passwd;
         int     pw_uid;
         int     pw_gid;
         int     pw_quota;
         char    *pw_comment;
         char    *pw_gecos;
         char    *pw_dir;
         char    *pw_shell;
};

struct passwd *getpwent(), *getpwuid(), *getpwnam();
```

This structure is declared in **<pwd.h>** so it is not necessary to redeclare it.

The fields *pw_quota* and *pw_comment* are unused; the others have meanings described in *passwd*(5). When first called, *getpwent* returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. *getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *endpwent* may be called to close the password file when processing is complete.

fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of the password file **/etc/passwd**.

FILES

/etc/passwd
/etc/yp/*domainname*/passwd.byname
/etc/yp/*domainname*/passwd.byuid

SEE ALSO

getlogin(3), getgrent(3), passwd(5), ypserv(8), getpwent(3V)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getwd – get current working directory pathname

SYNOPSIS

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

char *getwd(pathname)
char pathname[MAXPATHLEN];
```

DESCRIPTION

Getwd copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

DIAGNOSTICS

Getwd returns zero and places a message in *pathname* if an error occurs.

BUGS

Getwd may fail to return to the current directory if an error occurs.

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

DESCRIPTION

hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *item* is a structure of type ENTRY (defined in the **<search.h>** header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer. *hcreate* allocates sufficient space for the table, and must be called before *hsearch* is used. *nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances. *hdestroy* destroys the search table, and may be followed by another call to *hcreate*.

NOTES

hsearch uses *open addressing* with a *multiplicative* hash function.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

```
#include <stdio.h>
#include <search.h>

struct info {               /* this is the info stored in the table */
        int age, room;      /* other than the key. */
};
#define NUM_EMPL    5000    /* # of elements in search table */

main( )
{
        /* space to store strings */
        char string_space[NUM_EMPL*20];
        /* space to store employee info */
        struct info info_space[NUM_EMPL];
        /* next avail space in string_space */
        char *str_ptr = string_space;
        /* next avail space in info_space */
        struct info *info_ptr = info_space;
        ENTRY item, *found_item, *hsearch( );
        /* name to look for in table */
        char name_to_find[30];
        int i = 0;
```

```
        /* create table */
        (void) hcreate(NUM_EMPL);
        while (scanf("%s%d%d", str_ptr, &info_ptr->age,
                &info_ptr->room) != EOF && i++ < NUM_EMPL) {
                /* put info in structure, and structure in item */
                item.key = str_ptr;
                item.data = (char *)info_ptr;
                str_ptr += strlen(str_ptr) + 1;
                info_ptr++;
                /* put item into table */
                (void) hsearch(item, ENTER);
        }

        /* access table */
        item.key = name_to_find;
        while (scanf("%s", item.key) != EOF) {
            if ((found_item = hsearch(item, FIND)) != NULL) {
                /* if item is in the table */
                (void)printf("found %s, age = %d, room = %d\n",
                        found_item->key,
                        ((struct info *)found_item->data)->age,
                        ((struct info *)found_item->data)->room);
            } else {
                (void)printf("no such employee %s\n",
                        name_to_find)
            }
        }
}
```

SEE ALSO

bsearch(3), lsearch(3), malloc(3), string(3), tsearch(3)

DIAGNOSTICS

Hsearch returns a NULL pointer if either the action is FIND and the item could not be found or the action is ENTER and the table is full. *hcreate* returns zero if it cannot allocate sufficient space for the table.

WARNING

hsearch and *hcreate* use *malloc*(3) to allocate space.

BUGS

Only one hash search table may be active at any given time.

NAME

initgroups – initialize group access list

SYNOPSIS

initgroups(name, basegid)
char *name;
int basegid;

DESCRIPTION

Initgroups reads through the group file and sets up, using the *setgroups*(2) call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

FILES

/etc/group

SEE ALSO

setgroups(2)

DIAGNOSTICS

Initgroups returns –1 if it was not invoked by the super-user.

BUGS

Initgroups uses the routines based on *getgrent*(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to *initgroups*.

NAME

insque, remque – insert/remove element from a queue

SYNOPSIS

```
struct qelem {
        struct  qelem *q_forw;
        struct  qelem *q_back;
        char    q_data[];
};

insque(elem, pred)
struct qelem *elem, *pred;

remque(elem)
struct qelem *elem;
```

DESCRIPTION

insque and *remque* manipulate queues built from doubly linked lists. Each element in the queue must be in the form of "struct qelem". *insque* inserts *elem* in a queue immediately after *pred*; *remque* removes an entry *elem* from a queue.

NAME

isinf, isnan – test for indeterminate floating-point values

SYNOPSIS

int isinf(value)
double value;

int isnan(value)
double value;

DESCRIPTION

Isinf returns a value of 1 if its *value* is an IEEE format infinity (two words 0x7ff00000 0x00000000) or an IEEE negative infinity, and returns a zero otherwise.

Isnan returns a value of 1 if its *value* is an IEEE format 'not-a-number' (two words 0x7ff *nnnnn* 0x *nnnnnnnn*) where *n* is not zero) or its negative, and returns a zero otherwise.

Some library routines such as *ecvt*(3) do not handle indeterminate floating-point values gracefully. Prospective arguments to such routines should be checked with *isinf* or *isnan* before calling these routines.

The *Floating-Point Programmer's Guide for the Sun Workstation* gives details for the format of IEEE standard floating-point.

NAME

lockf – advisory record locking on files

SYNOPSIS

```
#include <unistd.h>

#define F_ULOCK     0     /* Unlock a previously locked section */
#define F_LOCK      1     /* Lock a section for exclusive use */
#define F_TLOCK     2     /* Test and lock a section (non-blocking) */
#define F_TEST      3     /* Test section for other process' locks */

lockf(fd, cmd, size)
int fd, cmd;
long size;
```

DESCRIPTION

Lockf may be used to test, apply, or remove an *advisory* record lock on the file associated with the open descriptor *fd*. (See *fcntl*(2) for more information about advisory record locking.)

A lock is obtained by specifying a *cmd* parameter of F_LOCK or F_TLOCK. To unlock an existing lock, the F_ULOCK *cmd* is used. F_TEST is used to detect if a lock by another process is present on the specified segment.

F_LOCK and F_TLOCK requests differ only by the action taken if the lock may not be immediately granted. F_TLOCK will cause the function to return a -1 and set *errno* to EAGAIN if the section is already locked by another process. F_LOCK will cause the process to sleep until the lock may be granted or a signal is caught.

Size is the number of contiguous bytes to be locked or unlocked. The lock starts at the current file offset in the file and extends forward for a positive *size* or backward for a negative *size* (preceeding but not including the current offset). A segment need not be allocated to the file in order to be locked; however, a segment may not extend to a negative offset relative to the beginning of the file. If *size* is zero, the lock will extend from the current offset through the end-of-file. If such a lock starts at offset 0, then the entire file will be locked (regardless of future file extensions).

NOTES

The descriptor *fd* must have been opened with O_WRONLY or O_RDWR permission in order to establish locks with this function call.

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

RETURN VALUE

Zero is returned on success, –1 on error, with an error code stored in *errno*.

ERRORS

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

EBADF	*Fd* is not a valid open descriptor.
EBADF	*Cmd* is F_LOCK or F_TLOCK and the process does not have write permission on the file.
EAGAIN	*Cmd* is F_TLOCK or F_TEST and the section is already locked by another process.
EINTR	*Cmd* is F_LOCK and a signal interrupted the process while it was waiting for the lock to be granted.
ENOLCK	*Cmd* is F_LOCK, F_TLOCK, or F_ULOCK and there are no more file lock entries available.

SEE ALSO

fcntl(2), lockd(8C)

BUGS

File locks obtained through the *lockf* mechanism do not interact in any way with those acquired via *flock*(2). They do, however, work correctly with the locks claimed by *fcntl*(2).

NAME

lsearch, lfind – linear search and update

SYNOPSIS

```
#include <stdio.h>
#include <search.h>

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

DESCRIPTION

lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. **key** points to the datum to be sought in the table. **base** points to the first element in the table. **nelp** points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. **compar** is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

This fragment will read in ≤ TABSIZE strings of length ≤ ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>

#define TABSIZE 50
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
        unsigned nel = 0;
        int strcmp( );
        . . .
        while (fgets(line, ELSIZE, stdin) != NULL &&
          nel < TABSIZE)
                (void) lsearch(line, (char *)tab, &nel,
                        ELSIZE, strcmp);
        . . .
```

SEE ALSO

bsearch(3), hsearch(3), tsearch(3).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc, cfree, memalign, valloc, alloca, malloc_debug, malloc_verify – memory allocator

SYNOPSIS

```
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;

cfree(ptr)
char *ptr;

char *memalign(alignment, size)
unsigned alignment;
unsigned size;

char *valloc(size)
unsigned size;

char *alloca(size)
int size;
```

DESCRIPTION

These routines provide a general-purpose memory allocation package. They maintain a table of free blocks for efficient allocation and coalescing of free storage. When there is no suitable space already free, the allocation routines call *sbrk* (see *brk*(2)) to get more memory from the system.

Each of the allocation routines returns a pointer to space suitably aligned for storage of any type of object. They return a null pointer if the request cannot be completed (see DIAGNOSTICS).

Malloc returns a pointer to a block of at least *size* bytes beginning on a word boundary. A null (0) pointer is returned if *size* bytes of memory cannot be allocated.

Free releases a previously allocated block. Its argument is a pointer to a block previously allocated by *malloc, calloc, realloc, valloc,* or *memalign.*

malloc, calloc, realloc, valloc, or *memalign.*

Realloc changes the size of the block referenced by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. For backwards compatibility, *realloc* accepts a pointer to a block freed since the most recent call to *malloc, calloc, realloc, valloc,* or *memalign.* Note that using *realloc* with a block freed *before* the most recent call to *malloc, calloc, realloc, valloc,* or *memalign* is an error.

Calloc uses *malloc* to allocate space for an array of *nelem* elements of size *elsize,* initializes the space to zeros, and returns a pointer to the initialized block. The block can be freed with *free* or *cfree.*

Memalign allocates *size* bytes on a specified alignment boundary, and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of *alignment.* Note that the value of *alignment* must be a power of two, and must be greater than or equal to the size of a word.

Valloc(size) is equivalent to *memalign(getpagesize(), size)*.

Alloca allocates *size* bytes of space in the stack frame of the caller, and returns a pointer to the allocated block. This temporary space is automatically freed when the caller returns.

SEE ALSO

"Fast Fits" by C. J. Stephenson, in Proceedings of the ACM 9th Symposium on Operating Systems, *SIGOPS Operating Systems Review*, vol. 17, no. 5, October 1983.

Core Wars, in *Scientific American*, May 1984.

DIAGNOSTICS

Malloc, calloc, realloc, valloc, and *memalign* return a null pointer (0) and set *errno* if arguments are invalid, or if there is insufficient available memory, or if the heap has been detectably corrupted, e.g. by storing outside the bounds of a block.

More detailed diagnostics can be made available to programs using *malloc, calloc, realloc, valloc, memalign, cfree,* and *free*, by including a special relocatable object file at link time (see FILES). This file also provides routines for control of error handling and diagnosis, as defined below. Note that these routines are *not* defined in the standard library.

int malloc_debug(level)
int level;

int malloc_verify()

Malloc_debug sets the level of error diagnosis and reporting during subsequent calls to *malloc, calloc, realloc, valloc, memalign, cfree,* and *free*. The value of *level* is interpreted as follows:

Level 0 — *Malloc, calloc, realloc, valloc, memalign, cfree,* and *free* behave the same as in the standard library.

Level 1 — *Malloc, calloc, realloc, valloc, memalign, cfree,* and *free* abort with a message to *stderr* if errors are detected in arguments or in the heap. If a bad block is encountered, its address and size are included in the message.

Level 2 — Same as level 1, except that the entire heap is examined on every call to *malloc, calloc, realloc, valloc, memalign, cfree,* and *free*.

Malloc_debug returns the previous error diagnostic level. The default level is 1.

Malloc_verify attempts to determine if the heap has been corrupted. It scans all blocks in the heap (both free and allocated) looking for strange addresses or absurd sizes, and also checks for inconsistencies in the free space table. *Malloc_verify* returns 1 if all checks pass without error, and otherwise returns 0. The checks can take a significant amount of time, so it should not be used indiscriminately.

ERRORS

Malloc, calloc, realloc, valloc, memalign, cfree, and *free* will set *errno* if:

EINVAL — An invalid argument was given. The value of *ptr* given to *free, cfree,* or *realloc* must be a pointer to a block previously allocated by *malloc, calloc, realloc, valloc,* or *memalign*. The EINVAL condition also occurs if the heap is found to have been corrupted. More detailed information may be obtained by enabling range checks using *malloc_debug*.

ENOMEM — *size* bytes of memory could not be allocated.

FILES

/usr/lib/debug/malloc.o — diagnostic versions of *malloc*, *free*, etc.

BUGS

Alloca is both machine- and compiler-dependent; its use is discouraged.

Since *realloc* accepts a pointer to a block freed since the last call to *malloc*, *calloc*, *realloc*, *valloc*, or *memalign*, a degradation of performance results. The semantics of *free* should be changed so that the contents of a previously freed block are undefined.

NAME

memory, memccpy, memchr, memcmp, memcpy, memset – memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

memset These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

memccpy copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* characters of *s2*.

memchr returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*, or a NULL pointer if *c* does not occur.

memcmp compares its arguments, looking at the first *n* characters only, and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*.

memcpy copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

memset sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

NOTE

For user convenience, all these functions are declared in the optional *<memory.h>* header file.

BUGS

memcmp uses native character comparison, which is signed on some machines and unsigned on other machines. Thus the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

mktemp, mkstemp – make a unique file name

SYNOPSIS

```
char *mktemp(template)
char *template;

mkstemp(template)
char *template;
```

DESCRIPTION

mktemp creates a unique file name, typically in a temporary filesystem, by replacing *template* with a unique file name, and returns the address of *template*. The string in *template* should contain a file name with six trailing Xs; *mktemp* replaces the Xs with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file. *mkstemp* makes the same replacement to the template but returns a file descriptor for the template file open for reading and writing. *mkstemp* avoids the race between testing whether the file exists and opening it for use.

Notes:

- *mktemp* and *mkstemp* actually *change* the template string which you pass; this means that you cannot use the same template string more than once — you need a fresh template for every unique file you want to open.
- When *mktemp* or *mkstemp* are creating a new unique filename they check for the prior existence of a file with that name. This means that if you are creating more than one unique filename, it is bad practice to use the same root template for multiple invocations of *mktemp* or *mkstemp*.

SEE ALSO

getpid(2), open(2V), tmpfile(3S), tmpnam(3S).

DIAGNOSTICS

mkstemp returns an open file descriptor upon success. It returns −1 if no suitable file could be created.

BUGS

It is possible to run out of letters.

NAME

monitor, monstartup, moncontrol – prepare execution profile

SYNOPSIS

monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)(), (*highpc)();
short buffer[];

monstartup(lowpc, highpc)
int (*lowpc)(), (*highpc)();

moncontrol(mode)

DESCRIPTION

There are two different forms of monitoring available: An executable program created by:

```
cc -p . . .
```

automatically includes calls for the *prof*(1) monitor and includes an initial call to its start-up routine *monstartup* with default parameters; *monitor* need not be called explicitly except to gain fine control over profil buffer allocation. An executable program created by:

```
cc -pg . . .
```

automatically includes calls for the *gprof*(1) monitor.

Monstartup is a high level interface to *profil*(2). *Lowpc* and *highpc* specify the address range that is to be sampled; the lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *Monstartup* allocates space using *sbrk*(2) and passes it to *monitor* (see below) to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. Only calls of functions compiled with the profiling option **-p** of *cc*(1) are recorded.

To profile the entire program, it is sufficient to use

```
extern etext();
. . .
monstartup(0x8000, etext);
```

Etext lies just above all the program text, see *end*(3).

To stop execution monitoring and write the results on the file *mon.out*, use

```
monitor(0);
```

then *prof*(1) can be used to examine the results.

Moncontrol is used to selectively control profiling within a program. This works with either *prof*(1) or *gprof*(1) type profiling. When the program starts, profiling begins. To stop the collection of histogram ticks and call counts use *moncontrol*(0); to resume the collection of histogram ticks and call counts use *moncontrol*(1). This allows the cost of particular operations to be measured. Note that an output file will be produced upon program exit irregardless of the state of *moncontrol*.

Monitor is a low level interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. At most *nfunc* call counts can be kept. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled. *Monitor* divides the buffer into space to record the histogram of program counter samples over the range *lowpc* to *highpc*, and space to record call counts of functions compiled with the **-p** option to *cc*(1).

To profile the entire program, it is sufficient to use

```
extern etext();
. . .
monitor(0x8000, etext, buf, bufsize, nfunc);
```

FILES

mon.out

SEE ALSO

cc(1), prof(1), gprof(1), profil(2), sbrk(2)

NAME

nlist – get entries from name list

SYNOPSIS

#include <nlist.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];

DESCRIPTION

nlist examines the name list in the executable file whose name is pointed to by *filename*, list of values and puts them in the array of nlist structures pointed to by *nl*. The name list *nl* consists of an array of structures containing names, types and values. The list is terminated with a null name; that is, a null string is in the name position of the structure. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(5) for the structure declaration.

This subroutine is useful for examining the system name list kept in the file **/vmunix**. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(5)

DIAGNOSTICS

All type entries are set to 0 if the file cannot be read or if does not contain a valid name list.

nlist returns –1 upon error.

NAME

on_exit – name termination handler

SYNOPSIS

int on_exit(procp, arg)
void (*procp)();
caddr_t arg;

DESCRIPTION

On_exit names a routine to be called after a program calls *exit*(3) or returns normally, and before its process terminates. The routine named is called as

(*procp)(status, arg);

where *status* is the argument with which *exit* was called, or zero if *main* returns. Typically, *arg* is the address of an argument vector to *(*procp)*, but may be an integer value. Several calls may be made to *on_exit*, specifying several termination handlers. The order in which they are called is the reverse of that in which they were given to *on_exit*.

SEE ALSO

exit(3)

DIAGNOSTICS

On_exit returns zero normally, or nonzero if the procedure name could not be stored.

BUGS

Currently there is a limit of 20 termination handlers, including any invoked implicitly (for example, by *gprof*(1) or *tcov*(1) processing). Calls to *on_exit* beyond this number will fail.

NOTES

This call is specific to Sun Unix and should not be used if portability is a concern.

Standard I/O exit processing is always done last.

NAME

perror, sys_errlist, sys_nerr, errno – system error messages

SYNOPSIS

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];

int errno;
```

DESCRIPTION

perror produces a short error message on the standard error describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno* (see *intro*(2)), which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *sys_nerr* is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2), psignal(3)

NAME

prof – profile within a function

SYNOPSIS

#define MARK
#include <prof.h>

void MARK (name)

DESCRIPTION

MARK will introduce a mark called *name* that will be treated the same as a function entry point. Execution of the mark will add to a counter for that mark, and program-counter time spent will be accounted to the immediately preceding mark or to the function if there are no preceding marks within the active function.

name may be any combination of up to six letters, numbers or underscores. Each *name* in a single compilation must be unique, but may be the same as any ordinary program symbol.

For marks to be effective, the symbol MARK must be defined before the header file **<prof.h>** is included. This may be defined by a preprocessor directive as in the synopsis, or by a command line argument, such as:

```
cc -p -DMARK foo.c
```

If MARK is not defined, the *MARK*(name) statements may be left in the source files containing them and will be ignored.

EXAMPLE

In this example, marks can be used to determine how much time is spent in each loop. Unless this example is compiled with *MARK* defined on the command line, the marks are ignored.

```
#include <prof.h>

func( )
{
        int i, j;

        .
        .
        .
        MARK(loop1);
        for (i = 0; i < 2000; i++) {
                . . .
        }
        MARK(loop2);
        for (j = 0; j < 2000; j++) {
                . . .
        }
}
```

SEE ALSO

prof(1), profil(2), monitor(3)

NAME

psignal, sys_siglist – system signal messages

SYNOPSIS

```
psignal(sig, s)
unsigned sig;
char *s;

char *sys_siglist[];
```

DESCRIPTION

Psignal produces a short message on the standard error file describing the indicated signal. First the argument string *s* is printed, then a colon, then the name of the signal and a new-line. Most usefully, the argument string is the name of the program which incurred the signal. The signal number should be from among those found in *<signal.h>*.

To simplify variant formatting of signal names, the vector of message strings *sys_siglist* is provided; the signal number can be used as an index in this table to get the signal name without the newline. The define NSIG defined in *<signal.h>* is the number of messages provided for in the table; it should be checked because new signals may be added to the system before they are added to the table.

SEE ALSO

perror(3), signal(3)

NAME

putenv – change or add value to environment

SYNOPSIS

int putenv (string)
char *string;

DESCRIPTION

string points to a string of the form *"name=value."* *putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

DIAGNOSTICS

putenv returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero.

SEE ALSO

exec(2), getenv(3), malloc(3), environ(7).

WARNINGS

putenv manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*. However, *envp* (the third argument to *main*) is not changed.

This routine uses *malloc*(3) to enlarge the environment.

After *putenv* is called, environmental variables are not in alphabetical order.

A potential error is to call *putenv* with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

putpwent is the inverse of *getpwent*(3). Given a pointer to a passwd structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*, which matches the format of lines in the password file **/etc/passwd**.

DIAGNOSTICS

putpwent returns non-zero if an error was detected during its operation, otherwise zero.

SEE ALSO

getpwent(3).

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

This routine is of limited utility, since most password files are maintained as Yellow Pages files, and cannot be updated with this routine.

NAME

qsort – quicker sort

SYNOPSIS

qsort(base, nel, width, compar)
char *base;
int (*compar)();

DESCRIPTION

qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

base points to the element at the base of the table. *nel* is the number of elements in the table. *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. As the function must return an integer less than, equal to, or greater than zero, so must the first argument to be considered be less than, equal to, or greater than the second.

NOTES

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The order in the output of two items which compare as equal is unpredictable.

SEE ALSO

bsearch(3), lsearch(3), string(3), sort(1)

NAME

random, srandom, initstate, setstate – better random number generator; routines for changing generators

SYNOPSIS

long random()

srandom(seed)
int seed;

char *initstate(seed, state, n)
unsigned seed;
char *state;
int n;

char *setstate(state)
char *state;

DESCRIPTION

random uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16\times(2^{31}-1)$.

random/*srandom* have (almost) the same calling sequence and initialization properties as *rand*/*srand*. The difference is that *rand*(3C) produces a much less random sequence — in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by *random* are usable. For example, ''random()&01'' will produce a random binary value.

Unlike *srand*, *srandom* does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting/changing random number generators). Like *rand*(3C), however, *random* will by default produce a sequence of numbers that can be duplicated by calling *srandom* with *1* as the seed.

The *initstate* routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by *initstate* to decide how sophisticated a random number generator it should use -- the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. *initstate* returns a pointer to the previous state information array.

Once a state has been initialized, the *setstate* routine provides for rapid switching between states. *setstate* returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to *initstate* or *setstate*.

Once a state array has been initialized, it may be restarted at a different point either by calling *initstate* (with the desired seed, the state array, and its size) or by calling both *setstate* (with the state array) and *srandom* (with the desired seed). The advantage of calling both *setstate* and *srandom* is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

DIAGNOSTICS

If *initstate* is called with less than 8 bytes of state information, or if *setstate* detects that the state information has been garbled, error messages are printed on the standard error output.

SEE ALSO

rand(3C)

BUGS

About 2/3 the speed of *rand*(3C).

NAME

regex, re_comp, re_exec – regular expression handler

SYNOPSIS

char *re_comp(s)
char *s;

re_exec(s)
char *s;

DESCRIPTION

Re_comp compiles a string into an internal form suitable for pattern matching. *Re_exec* checks the argument string against the last string passed to *re_comp*.

Re_comp returns 0 if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If *re_comp* is passed 0 or a null string, it returns without changing the currently compiled regular expression.

Re_exec returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and -1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both *re_comp* and *re_exec* may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for *ed*(1), given the above difference.

SEE ALSO

ed(1), ex(1), egrep(1), fgrep(1), grep(1)

DIAGNOSTICS

Re_exec returns -1 for an internal error.

Re_comp returns one of the following strings if an error occurs:

No previous regular expression

Regular expression too long

unmatched \(

missing]

too many \(\) pairs

unmatched \)

NAME

regexp – regular expression compile and match routines

SYNOPSIS

```
#define INIT <declarations>
#define GETC( ) <getc code>
#define PEEKC( ) <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step (string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;
```

DESCRIPTION

This page describes general-purpose regular expression matching routines.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC( )	Return the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.
PEEKC( )	Return the next character in the regular expression. Successive calls to PEEKC( ) should return the same character (which should also be the next character returned by GETC( )).
UNGETC(*c*)	Cause the argument *c* to be returned by the next call to GETC( ) (and PEEKC( )). No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.
RETURN(*pointer*)	This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs that have memory allocation to manage.

ERRORS

ERROR(*val*) This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.

46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

compile(instring, expbuf, endbuf, eof)

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs that call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf–expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character that marks the end of the regular expression. For example, in an editor like *ed*(1), this character would usually a /.

Each program that includes this file must have a #define statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for GETC( ), PEEKC( ) and UNGETC( ). Otherwise it can be used to declare external variables that might be used by GETC( ), PEEKC( ) and UNGETC( ). See the example below of the declarations taken from *grep*(1).

There are other functions in this file that perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

step(string, expbuf)

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression that was obtained by a call of the function *compile*.

The function *step* returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, *loc1* will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will try to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns non-zero indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called; simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* will break out of

the loop that backs up and will return zero. This could be used by an editor like *ed*(1) or *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like s/y*//g do not loop forever.

The additional external variables *sed* and *nbra* are used for special purposes.

EXAMPLES

The following is an example of how the regular expression macros and calls could look in a command like *grep*(1):

```
#define INIT      register char *sp = instring;
#define GETC( )   (*sp++)
#define PEEKC( )  (*sp)
#define UNGETC(c)       (—sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr( )

#include <regexp.h>
...
                (void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
                if (step(linebuf, expbuf))
                        succeed( );
```

FILES

/usr/include/regexp.h

BUGS

The handling of *circf* is kludgy.

NAME

scandir, alphasort – scan a directory

SYNOPSIS

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

scandir(dirname, namelist, select, compar)
char *dirname;
struct direct *(*namelist[ ]);
int (*select)();
int (*compar)();

alphasort(d1, d2)
struct direct **d1, **d2;
```

DESCRIPTION

Scandir reads the directory *dirname* and builds an array of pointers to directory entries using *malloc*(3). The second parameter is a pointer to an array of structure pointers. The third parameter is a pointer to a routine which is called with a pointer to a directory entry and should return a non zero value if the directory entry should be included in the array. If this pointer is null, then all the directory entries will be included. The last argument is a pointer to a routine which is passed to *qsort*(3) to sort the completed array. If this pointer is null, the array is not sorted. *Alphasort* is a routine which will sort the array alphabetically.

Scandir returns the number of entries in the array and a pointer to the array through the parameter *namelist*.

SEE ALSO

directory(3), malloc(3), qsort(3)

DIAGNOSTICS

Returns –1 if the directory cannot be opened for reading or if *malloc*(3) cannot allocate enough memory to hold all the data structures.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

val = setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;

val = _setjmp(env)
jmp_buf env;

_longjmp(env, val)
jmp_buf env;
```

DESCRIPTION

Setjmp and *longjmp* are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* for later use by *longjmp*. *Setjmp* also saves the register environment. If a *longjmp* call will be made, the routine which called *setjmp* should not return until after the *longjmp* has returned control (see below).

Longjmp restores the environment saved by the last call of *setjmp*, and then returns in such a way that execution continues as if the call of *setjmp* had just returned the value *val* to the function that invoked *setjmp*. The calling function must not itself have returned in the interim, otherwise *longjmp* will be returning control to a possibly non-existent environment. All memory-bound data have values as of the time *longjmp* was called. The machine registers are restored to the values they had at the time that *setjmp* was called. But, because the **register** storage class is only a hint to the C compiler, variables declared as **register** variables may not necessarily be assigned to machine registers, so their values are unpredictable after a *longjmp*. This is especially a problem for programmers trying to write machine-independent C routines.

The following code fragment indicates the flow of control of the *setjmp* and *longjmp* combination:

```
. . . function declaration
        jmp_buf my_environment;

         . . . code . . .
        if (setjmp(my_environment)) {
                this is the code after the return from longjmp
                 . . . more code . . . .
                register variables have unpredictable values
                 . . . more code . . . .
        } else {
                this is the return from setjmp
                 . . . more code . . . .
                Do not modify register variables
                in this leg of the code
                 . . . more code . . . .
        }
```

Setjmp and *longjmp* save and restore the signal mask *sigsetmask*(2), while *_setjmp* and *_longjmp* manipulate only the C stack and registers.

SEE ALSO

sigsetmask(2), sigvec(2), signal(3)

BUGS

Setjmp does not save current notion of whether the process is executing on the signal stack. The result is that a longjmp to some place on the signal stack leaves the signal stack state incorrect.

longjmp never returns zero in the Sun implementation.

NAME

setuid, seteuid, setruid, setgid, setegid, setrgid – set user and group ID

SYNOPSIS

setuid(uid)
seteuid(euid)
setruid(ruid)

setgid(gid)
setegid(egid)
setrgid(rgid)

DESCRIPTION

Setuid (*setgid*) sets both the real and effective user ID (group ID) of the current process to as specified.

Seteuid (*setegid*) sets the effective user ID (group ID) of the current process.

Setruid (*setrgid*) sets the real user ID (group ID) of the current process.

These calls are only permitted to the super-user or if the argument is the real or effective ID.

SEE ALSO

setreuid(2), setregid(2), getuid(2), getgid(2)

DIAGNOSTICS

Zero is returned if the user (group) ID is set; −1 is returned otherwise, with the global variable *errno* set as for setreuid or setregid.

NAME

siginterrupt – allow signals to interrupt system calls

SYNOPSIS

siginterrupt(sig, flag);
int sig, flag;

DESCRIPTION

siginterrupt is used to change the system call restart behavior when a system call is interrupted by the specified signal. If the flag is false (0), then system calls will be restarted if they are interrupted by the specified signal and no data has been transferred yet. System call restart is the default behavior on 4.2 BSD, and on Sun UNIX in the 4.2 environment, when the *signal (3)* routine is used.

If the flag is true (1), then restarting of system calls is disabled. If a system call is interrupted by the specified signal and no data has been transferred, the system call will return -1 with errno set to EINTR. Interrupted system calls that have started transferring data will return the amount of data actually transferred. System call interrupt is the signal behavior found on older UNIX systems, such as 4.1 BSD and System V UNIX. It is the default behavior on Sun UNIX in the System V environment when the *signal* routine is used; therefore, this routine is useful in that environment only if a signal that a *sigvec (2)* specified should restart system calls is to be changed not to restart them.

Note that the new 4.2 BSD signal handling semantics are not altered in any other way. Most notably, signal handlers always remain installed until explicitly changed by a subsequent *sigvec* call, and the signal mask operates as documented in *sigvec*, unless the SV_RESETHAND bit has been used to specify that the pre-4.2 BSD signal behavior is to be used. Programs may switch between restartable and interruptible system call operation as often as desired in the execution of a program.

Issuing a *siginterrupt*(3) call during the execution of a signal handler will cause the new action to take place on the next signal to be caught.

NOTES

This library routine uses an extension of the *sigvec*(2) system call that is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

RETURN VALUE

A 0 value indicates that the call succeeded. A -1 value indicates that an invalid signal number has been supplied.

SEE ALSO

sigvec(2), sigblock(2), sigpause(2), sigsetmask(2).

NAME

signal – simplified software signal facilities

SYNOPSIS

#include <signal.h>

(*signal(sig, func))()
int (*func)();

DESCRIPTION

signal is a simplified interface to the more general *sigvec*(2) facility. Programs that use *signal* in preference to *sigvec* are more likely to be portable to all UNIX systems.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see *tty*(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the *signal* call allows signals either to be ignored or to cause an interrupt to a specified location. 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 (other than A-line or F-line op code)
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT trap (not generated on Suns)
SIGEMT	7*	EMT trap (A-line or F-line op code)
SIGFPE	8*	arithmetic 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 changed (see *win*(4S))
SIGLOST	29*	resource lost (see *lockd*(8C))
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.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; 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 *func* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. **Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.**

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is automatically restarted. In particular this can occur during a *read* or *write*(2V) on a slow device (such as a terminal; but not a file) and during a *wait*(2).

The value of *signal* is the previous (or initial) value of *func* for the particular signal.

After a *fork*(2) or *vfork*(2) the child inherits all signals. An *execve*(2) resets all caught signals to the default action; ignored signals remain ignored.

NOTES

The handler routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number. *Code* is a parameter of certain signals that provides additional detail. *scp* is a pointer to the *sigcontext* structure (defined in **<signal.h>**), used to restore the context from before the signal.

CODES

The following defines the codes for signals which produce them. All of these symbols are defined in **<signal.h>**:

Hardware condition	Signal	Code
Illegal instruction	SIGILL	ILL_INSTR_FAULT
Privilege violation	SIGILL	ILL_PRIVVIO_FAULT
Coprocessor protocol error	SIGILL	ILL_INSTR_FAULT
Trap #*n* (1 <= *n* <= 14)	SIGILL	ILL_TRAP_FAULT
A-line op code	SIGEMT	EMT_EMU1010
F-line op code	SIGEMT	EMT_EMU1111
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
CHK or CHK2 instruction	SIGFPE	FPE_CHKINST_TRAP
TRAPV or TRAPcc or cpTRAPcc	SIGFPE	FPE_TRAPV_TRAP
IEEE floating point compare unordered	SIGFPE	FPE_FLTBSUN_TRAP
IEEE floating point inexact	SIGFPE	FPE_FLTINEX_TRAP
IEEE floating point division by zero	SIGFPE	FPE_FLTDIV_TRAP
IEEE floating point underflow	SIGFPE	FPE_FLTUND_TRAP
IEEE floating point operand error	SIGFPE	FPE_FLTOPERR_TRAP
IEEE floating point overflow	SIGFPE	FPE_FLTOVF_FAULT
IEEE floating point signaling NaN	SIGFPE	FPE_FLTNAN_TRAP

RETURN VALUE

The previous action is returned on a successful call. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

signal will fail and no action will take place if one of the following occur:

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), sigvec(2), sigblock(2), sigsetmask(2), sigpause(2), sigstack(2), setjmp(3), tty(4)

NAME

sleep – suspend execution for interval

SYNOPSIS

sleep(seconds)
unsigned seconds;

DESCRIPTION

sleep suspends the current process from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and may be an arbitrary amount longer because of other activity in the system.

sleep is implemented by setting an interval timer and pausing until it expires. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous value of the timer, the process sleeps only until the timer would have expired, and the signal which occurs with the expiration of the timer is sent one second later.

SEE ALSO

setitimer(2), sigpause(2), usleep(3)

NAME

ssignal, gsignal – software signals

SYNOPSIS

#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;

DESCRIPTION

ssignal and *gsignal* implement a software facility similar to *signal*(3).

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to *ssignal* associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action function* or one of the manifest constants SIG_DFL (default) or SIG_IGN (ignore). *ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns SIG_DFL.

gsignal raises the signal identified by its argument, *sig*:

If an action function has been established for *sig*, then that action is reset to SIG_DFL and the action function is entered with argument *sig*. *gsignal* returns the value returned to it by the action function.

If the action for *sig* is SIG_IGN, *gsignal* returns the value 1 and takes no other action.

If the action for *sig* is SIG_DFL, *gsignal* returns the value 0 and takes no other action.

If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

SEE ALSO

signal(3)

NAME

string, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, index, rindex – string operations

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;

#include <string.h>

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, c;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

strcat appends a copy of string *s2* to the end of string *s1*. *strncat* appends at most *n* characters. Each returns a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *strncmp* makes the same comparison but compares at most *n* characters.

strcpy copies string *s2* to *s1*, stopping after the null character has been copied. *strncpy* copies exactly *n* characters, truncating or null-padding *s2*. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

strlen returns the number of characters in *s*, not including the terminating null character.

strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

index (*rindex*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. These functions are identical to *strchr* (*strchr*) and merely have different names.

strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string *s1* immediately following that token. In this way subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

NOTE

For user convenience, all these functions, except for *index* and *rindex*, are declared in the optional **<string.h>** header file. All these functions, including *index* and *rindex* but excluding *strchr*, *strrchr*, *strpbrk*, *strspn*, *strcspn*, and *strtok*, are declared in the optional **<strings.h>** include file; the reason for this is also historical.

WARNINGS

strcmp and *strncmp* use native character comparison, which is signed on the Sun, but may be unsigned on other machines. Thus the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

On the Sun processor, as well as on many other machines, you can *NOT* use a NULL pointer to indicate a null string. A NULL pointer is an error and results in an abort of the program. If you wish to indicate a null string, you must have a pointer that points to an explicit null string. On some implementations of the C language on some machines, a NULL pointer, if dereferenced, would yield a null string; this highly non-portable trick was used in some programs. Programmers using a NULL pointer to represent an empty string should be aware of this portability issue; even on machines where dereferencing a NULL pointer does not cause an abort of the program, it does not necessarily yield a null string.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtod, atof – convert string to double-precision number

SYNOPSIS

double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;

DESCRIPTION

strtod returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

strtod recognizes an optional string of spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, **ptr* is set to *str*, and zero is returned.

atof(str) is equivalent to *strtod(str, (char **)NULL)*.

SEE ALSO

ctype(3), scanf(3S), strtol(3).

DIAGNOSTICS

If the correct value would cause overflow, ±HUGE is returned (according to the sign of the value), and *errno* is set to ERANGE.
If the correct value would cause underflow, zero is returned and *errno* is set to ERANGE.

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;

DESCRIPTION

strtol returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by *isspace* in *ctype*(3)) are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment or by an explicit cast.

atol(str) is equivalent to *strtol(str, (char **)NULL, 10)*.

Atoi(str) is equivalent to *(int) strtol(str, (char **)NULL, 10)*.

SEE ALSO

ctype(3), scanf(3S), strtod(3)

BUGS

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNOPSIS

swab(from, to, nbytes)
char *from, *to;

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between high-ender machines (IBM 360's, MC68000's, etc) and low-ender machines (PDP-11's and VAX'es).

Nbytes should be even.

The *from* and *to* addresses should not overlap in portable programs.

NAME

syslog, openlog, closelog – control system log

SYNOPSIS

#include <syslog.h>

openlog(ident, logstat)
char *ident;

syslog(priority, message, parameters ...)
char *message;

closelog()

DESCRIPTION

Syslog arranges to write the *message* onto the system log maintained by *syslog*(8). The message is tagged with *priority*. The message looks like a *printf*(3S) string except that **%m** is replaced by the current error message (collected from *errno*). A trailing newline is added if needed. This message will be read by *syslog*(8) and output to the system console or files as appropriate.

If special processing is needed, *openlog* can be called to initialize the log file. Parameters are *ident* which is prepended to every message, and *logstat* which is a bit field indicating special status; current values are:

LOG_PID log the process id with each message: useful for identifying instantiations of daemons.

Openlog returns zero on success. If *syslog* cannot send datagrams to *syslog*(8), then it writes on */dev/console* instead. If */dev/console* cannot be written, standard error is used. In either case, it returns -1.

Closelog can be used to close the log file. It is automatically closed on a successful exec system call (see *execve*(2)).

EXAMPLES

```
syslog(LOG_SALERT, "who: internal error 23");

openlog("serverftp", LOG_PID);
syslog(LOG_INFO, "Connection from host %d", CallingHost);
```

SEE ALSO

syslog(8)

NAME

system – issue a shell command

SYNOPSIS

system(string)
char *string;

DESCRIPTION

System causes the *string* to be given to *sh*(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

SEE ALSO

popen(3S), execve(2), wait(2)

DIAGNOSTICS

Exit status 127 (may be displayed as "32512") indicates the shell couldn't be executed.

NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tfind ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );

DESCRIPTION

tsearch, *tfind*, *tdelete*, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

tsearch is used to build and access the tree. *key* is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to **key* (the value pointed to by *key*), a pointer to this found datum is returned. Otherwise, **key* is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. *rootp* points to a variable that points to the root of the tree. A NULL value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. *tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

twalk traverses a binary search tree. *root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type **typedef enum { preorder, postorder, endorder, leaf } VISIT;** (defined in the **<search.h>** header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <search.h>
#include <stdio.h>

struct node {              /* pointers to these are stored in the tree */
        char *string;
        int length;
};
char string_space[10000]; /* space to store strings */
struct node nodes[500];          /* nodes to store */
struct node *root = NULL;        /* this points to the root */

main( )
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        void print_node( ), twalk( );
        int i = 0, node_compare( );

        while (gets(strptr) != NULL && i++ < 500)  {
                /* set node */
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);
                /* put node into the tree */
                (void) tsearch((char *)nodeptr, &root,
                        node_compare);
                /* adjust pointers, so we don't overwrite tree */
                strptr += nodeptr->length + 1;
                nodeptr++;
        }
        twalk(root, print_node);
}
/*
        This routine compares two nodes, based on an
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
struct node *node1, *node2;
{
        return strcmp(node1->string, node2->string);
}
/*
        This routine prints out a node, the first time
        twalk encounters it.
*/

void
print_node(node, order, level)
struct node **node;
VISIT order;
int level;
{
        if (order == preorder || order == leaf)  {
```

```
                (void)printf("string = %20s,  length = %d\n",
                    (*node)->string, (*node)->length);
            }
        }
```

SEE ALSO

bsearch(3), hsearch(3), lsearch(3).

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.

A NULL pointer is returned by *tsearch*, *tfind* and *tdelete* if *rootp* is NULL on entry.

If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

WARNINGS

The *root* argument to *twalk* is one level of indirection less than the *rootp* arguments to *tsearch* and *tdelete*.

There are two nomenclatures used to refer to the order in which tree nodes are visited. *tsearch* uses preorder, postorder and endorder to respectively refer to visting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

BUGS

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname(filedes)

isatty(filedes)

DESCRIPTION

ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *filedes*.

isatty returns 1 if *filedes* is associated with a terminal device, 0 otherwise.

FILES

/dev/*

SEE ALSO

ioctl(2), ttys(5)

DIAGNOSTICS

ttyname returns a NULL pointer if *filedes* does not describe a terminal device in directory **/dev**.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ttyslot – find the slot in the utmp file of the current process

SYNOPSIS

ttyslot()

DESCRIPTION

ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/ttys** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES

/etc/ttys

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

ualarm – schedule signal after interval in microseconds

SYNOPSIS

unsigned ualarm(value, interval)
unsigned value;
unsigned interval;

DESCRIPTION

This is a simplified interface to setitimer(2).

Ualarm causes signal SIGALRM see *signal*(3), to be sent to the invoking process in a number of microseconds given by the *value* argument. Unless caught or ignored, the signal terminates the process.

If the *interval* argument is non-zero, the SIGALRM signal will be sent to the process every *interval* microseconds after the timer expires (e.g. after *value* microseconds have passed).

Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 2147483647 microseconds.

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

SEE ALSO

getitimer(2), setitimer(2), sigpause(2), sigvec(2), signal(3), sleep(3), alarm(3), usleep(3)

NAME

usleep – suspend execution for interval in microseconds

SYNOPSIS

usleep(useconds)
unsigned useconds;

DESCRIPTION

for interval in microseconds" The current process is suspended from execution for the number of microseconds specified by the argument. The actual suspension time may be an arbitrary amount longer because of other activity in the system or because of the time spent in processing the call.

The routine is implemented by setting an interval timer and pausing until it occurs. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous timer, the process sleeps only until the signal would have occurred, and the signal is sent a short time later.

This routine is implemented using *setitimer*(2); it requires eight system calls each time it is invoked. A similar but less compatible function can be obtained with a single *select*(2); it would not restart after signals, but would not interfere with other uses of *setitimer*.

SEE ALSO

setitimer(2), getitimer(2), sigpause(2), ualarm(3), sleep(3), alarm(3)

NAME

values – machine-dependent values

SYNOPSIS

#include <values.h>

DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures. The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS(*type*)	The number of bits in a specified type (e.g., int).
HIBITS	The value of a short integer with only the high-order bit set (in most implementations, 0x8000).
HIBITL	The value of a long integer with only the high-order bit set (in most implementations, 0x80000000).
HIBITI	The value of a regular integer with only the high-order bit set (usually the same as HIBITS or HIBITL).
MAXSHORT	The maximum value of a signed short integer (in most implementations, 0x7FFF ≡ 32767).
MAXLONG	The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF ≡ 2147483647).
MAXINT	The maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG).
MAXFLOAT, LN_MAXFLOAT	The maximum value of a single-precision floating-point number, and its natural logarithm.
MAXDOUBLE, LN_MAXDOUBLE	The maximum value of a double-precision floating-point number, and its natural logarithm.
MINFLOAT, LN_MINFLOAT	The minimum positive value of a single-precision floating-point number, and its natural logarithm.
MINDOUBLE, LN_MINDOUBLE	The minimum positive value of a double-precision floating-point number, and its natural logarithm.
FSIGNIF	The number of significant bits in the mantissa of a single-precision floating-point number.
DSIGNIF	The number of significant bits in the mantissa of a double-precision floating-point number.

FILES

/usr/include/values.h

SEE ALSO

intro(3), intro(3M)

NAME

varargs – handle variable argument list

SYNOPSIS

#include <varargs.h>

function(va_alist)
va_dcl

va_list *pvar*;

va_start(*pvar*);

f = va_arg(*pvar*, *type*);

va_end(*pvar*);

DESCRIPTION

This set of macros provides a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists (such as *printf*(3S)) but do not use *varargs* are inherently non-portable, since different machines use different argument passing conventions.

va_alist is used in a function header to declare a variable argument list.

va_dcl is a declaration for **va_alist**. No semicolon should follow **va_dcl**.

va_list is a type defined for the variable used to traverse the list. One such variable must always be declared.

va_start(*pvar*) is called to initialize *pvar* to the beginning of the list.

va_arg(*pvar*, *type*) will return the next argument in the list pointed to by *pvar*. *type* is the type to which the expected argument will be converted when passed as an argument. In standard C, arguments that are **char** or **short** are converted to **int** and should be accessed as **int**, arguments that are **unsigned char** or **unsigned short** are converted to **unsigned int** and should be accessed as **unsigned int**, and arguments that are **float** are converted to **double** and should be accessed as **double**. Different types can be mixed, but it is up to the routine to know what type of argument is expected, since it cannot be determined at runtime.

va_end(*pvar*) is used to finish up.

Multiple traversals, each bracketed by **va_start** ... **va_end**, are possible.

va_alist must encompass the entire arguments list. This insures that a **#define** statement can be used to redefine or expand its value.

The argument list (or its remainder) can be passed to another function using a pointer to a variable of type **va_list**— in which case a call to **va_arg** in the subroutine advances the argument-list pointer with respect to the caller as well.

EXAMPLE

This example is a possible implementation of *execl*(3).

```
#include <varargs.h>
#define MAXARGS      100

/*      execl is called by
                execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[MAXARGS];
```

```
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *)) != (char *)0)
                ;
        va_end(ap);
        return execv(file, args);
}
```

BUGS

It is up to the calling routine to specify how many arguments there are, since it is not possible to determine this from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list. *Printf* can tell how many arguments are supposed to be there by the format.

The macros *va_start* and *va_end* may be arbitrarily complex; for example, *va_start* might contain an opening brace, which is closed by a matching brace in *va_end*. Thus, they should only be used where they could be placed within a single complex statement.

NAME

intro – introduction to compatibility library functions

DESCRIPTION

These functions constitute the compatibility library portion of *libc*. They are automatically loaded as needed by the C compiler *cc*(1). The link editor searches this library under the "–lc" option. Use of these routines (instead of newer equivalent routines) is encouraged for the sake of program portability. Manual entries for the functions in this library describe the proper routine to use.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
alarm	alarm(3C)	schedule signal after specified time
clock	clock(3C)	report CPU time used
ftime	time(3C)	get date and time
gtty	stty(3C)	set and get terminal state
nice	nice(3C)	set program priority
pause	pause(3C)	stop until signal
rand	rand(3C)	random number generator
srand	rand(3C)	random number generator
stty	stty(3C)	set and get terminal state
time	time(3C)	get date and time
times	times(3C)	get process times
ulimit	ulimit(3C)	get and set user limits
utime	utime(3C)	set file times
vlimit	vlimit(3C)	control maximum system resource consumption
vtimes	vtimes(3C)	get information about resource utilization

NAME

alarm – schedule signal after specified time

SYNOPSIS

alarm(seconds)
unsigned seconds;

DESCRIPTION

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

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is canceled. Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 2147483647 seconds.

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

SEE ALSO

sigpause(2), sigvec(2), signal(3), sleep(3), ualarm(3), usleep(3)

NAME

clock – report CPU time used

SYNOPSIS

long clock ( )

DESCRIPTION

clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2) or *system*(3).

The resolution of the clock is 16.667 milliseconds.

SEE ALSO

wait(2), system(3), times(3C) times(3V)

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

NAME

nice – change priority of a process

SYNOPSIS

nice(incr)

DESCRIPTION

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without undue impact on system performance.

Negative increments are illegal, except when specified by the super-user. The priority is limited to the range –20 (most urgent) to 20 (least). Requests for values above or below these limits result in the scheduling priority being set to the corresponding limit.

The priority of a process is passed to a child process by *fork*(2). For a privileged process to return to normal priority from an unknown state, *nice* should be called successively with arguments –40 (goes to priority –20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

RETURN VALUE

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

ERRORS

The priority is not changed if:

EACCES The value of *incr* specified was negative, and the effective user ID is not super-user.

SEE ALSO

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

NAME

pause – stop until signal

SYNOPSIS

pause()

DESCRIPTION

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

RETURN VALUE

Always returns –1.

ERRORS

Pause always returns:

EINTR	The call was interrupted.

SEE ALSO

kill(2), select(2), sigpause(2)

NAME

rand, srand – simple random number generator

SYNOPSIS

```
srand(seed)
int seed;

rand()
```

DESCRIPTION

rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$.

srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of *rand* leave a great deal to be desired. *drand48*(3) and *random*(3) provide much better, though more elaborate, random-number generators.

SEE ALSO

drand48(3), random(3), rand(3V)

BUGS

The low bits of the numbers generated are not very random; use the middle bits. In particular the lowest bit alternates between 0 and 1.

NAME

stty, gtty – set and get terminal state

SYNOPSIS

```
#include <sgtty.h>

stty(fd, buf)
int fd;
struct sgttyb *buf;

gtty(fd, buf)
int fd;
struct sgttyb *buf;
```

DESCRIPTION

This interface is obsoleted by ioctl(2).

Stty sets the state of the terminal associated with *fd*. *Gtty* retrieves the state of the terminal associated with *fd*. To set the state of a terminal the call must have write permission.

The *stty* call is actually "ioctl(fd, TIOCSETP, buf)", while the *gtty* call is "ioctl(fd, TIOCGETP, buf)". See *ioctl*(2) and *tty*(4) for an explanation.

DIAGNOSTICS

If the call is successful 0 is returned, otherwise −1 is returned and the global variable *errno* contains the reason for the failure.

SEE ALSO

ioctl(2), tty(4)

NAME

time, ftime – get date and time

SYNOPSIS

```
timeofday = time(0)

timeofday = time(tloc)
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>
ftime(tp)
struct timeb *tp;
```

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If *tloc* is nonnull, the return value is also stored in the place to which *tloc* points.

The *ftime* entry fills in a structure pointed to by its argument, as defined by *<sys/timeb.h>*:

```
struct timeb
{
        time_t    time;
        unsigned short millitm;
        short     timezone;
        short     dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, 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.

SEE ALSO

date(1), gettimeofday(2), settimeofday(2), ctime(3)

NAME

times – get process times

SYNOPSIS

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

times(buffer)
struct tms *buffer;
```

DESCRIPTION

This interface is obsoleted by getrusage(2).

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ is 60.

This is the structure returned by *times*:

```
struct tms {
        time_t  tms_utime;          /* user time */
        time_t  tms_stime;          /* system time */
        time_t  tms_cutime;         /* user time, children */
        time_t  tms_cstime;         /* system time, children */
};
```

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

SEE ALSO

time(1V), getrusage(2), wait3(2), time(3C)

NAME

ulimit – get and set user limits

SYNOPSIS

long ulimit(cmd, newlimit)
int cmd;

DESCRIPTION

This function is included for System V compatibility.

This routine provides for control over process limits. The *cmd* values available are:

1 Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the process's file size limit to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than the super-user attempts to increase its file size limit.

3 Get the maximum possible break value. See *brk*(2).

RETURN VALUE

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

SEE ALSO

brk(2), setrlimit(2), write(2V)

NAME

utime – set file times

SYNOPSIS

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

utime(file, timep)
char *file;
time_t timep[2];
```

DESCRIPTION

The *utime* call uses the 'accessed' and 'updated' times in that order from the *timep* 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.

SEE ALSO

utimes(2), stat(2)

NAME

vlimit – control maximum system resource consumption

SYNOPSIS

#include <sys/vlimit.h>

vlimit(resource, value)

DESCRIPTION

This facility is superseded by getrlimit(2).

Limits the consumption by the current process and each process it creates to not individually exceed *value* on the specified *resource*. If *value* is specified as −1, then the current limit is returned and the limit is unchanged. The resources which are currently controllable are:

LIM_NORAISE A pseudo-limit; if set non-zero then the limits may not be raised. Only the super-user may remove the *noraise* restriction.

LIM_CPU the maximum number of cpu-seconds to be used by each process

LIM_FSIZE the largest single file which can be created

LIM_DATA the maximum growth of the data+stack region via *sbrk*(2) beyond the end of the program text

LIM_STACK the maximum size of the automatically-extended stack region

LIM_CORE the size of the largest core dump that will be created.

LIM_MAXRSS a soft limit for the amount of physical memory (in bytes) to be given to the program. If memory is tight, the system will prefer to take memory from processes which are exceeding their declared LIM_MAXRSS.

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, or the process is killed when the stack limit is reached (since the stack cannot be extended, there is no way to send a signal!).

A file i/o operation which would create a file which is too large will cause a signal SIGXFSZ to be generated, this normally terminates the process, but may be caught. When the cpu time limit is exceeded, a signal SIGXCPU is sent to the offending process; to allow it time to process the signal it is given 5 seconds grace by raising the cpu time limit.

SEE ALSO

csh(1)

BUGS

If LIM_NORAISE is set, then no grace should be given when the cpu time limit is exceeded.

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

NAME

vtimes – get information about resource utilization

SYNOPSIS

vtimes(par_vm, ch_vm)
struct vtimes *par_vm, *ch_vm;

DESCRIPTION

This facility is superseded by getrusage(2).

Vtimes returns accounting information for the current process and for the terminated child processes of the current process. Either *par_vm* or *ch_vm* or both may be 0, in which case only the information for the pointers which are non-zero is returned.

After the call, each buffer contains information as defined by the contents of the include file *<sys/vtimes.h>*:

```
struct vtimes {
        int     vm_utime;               /* user time (*HZ) */
        int     vm_stime;               /* system time (*HZ) */
        /* divide next two by utime+stime to get averages */
        unsigned vm_idsrss;             /* integral of d+s rss */
        unsigned vm_ixrss;              /* integral of text rss */
        int     vm_maxrss;              /* maximum rss */
        int     vm_majflt;              /* major page faults */
        int     vm_minflt;              /* minor page faults */
        int     vm_nswap;               /* number of swaps */
        int     vm_inblk;               /* block reads */
        int     vm_oublk;               /* block writes */
};
```

The *vm_utime* and *vm_stime* fields give the user and system time respectively in 60ths of a second (or 50ths if that is the frequency of wall current in your locality.) The *vm_idrss* and *vm_ixrss* measure memory usage. They are computed by integrating the number of memory pages in use each over cpu time. They are reported as though computed discretely, adding the current memory usage (in 512 byte pages) each time the clock ticks. If a process used 5 core pages over 1 cpu-second for its data and stack, then *vm_idsrss* would have the value 5*60, where *vm_utime+vm_stime* would be the 60. *Vm_idsrss* integrates data and stack segment usage, while *vm_ixrss* integrates text segment usage. *Vm_maxrss* reports the maximum instantaneous sum of the text+data+stack core-resident page count.

The *vm_majflt* field gives the number of page faults which resulted in disk activity; the *vm_minflt* field gives the number of page faults incurred in simulation of reference bits; *vm_nswap* is the number of swaps which occurred. The number of file system input/output events are reported in *vm_inblk* and *vm_oublk* These numbers 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.

SEE ALSO

getrusage(2), wait3(2)

NAME

intro – introduction to mathematical library functions and constants

SYNOPSIS

#include <math.h>

DESCRIPTION

The include file **<math.h>** contains declarations of all the functions in the Math Library *libm* (described in Section 3M), as well as various functions in the C Library (Section 3C) that return floating-point values. Functions in this library are automatically loaded as needed by the Fortran compiler *f77*(1). The link editor searches this library under the "–lm" option.

<math.h> also defines the structure and constants used by the *matherr*(3M) error-handling mechanisms, including the following constant used as an error-return value:

HUGE	The maximum value of a double-precision floating-point number.

The following mathematical constants are defined for user convenience:

M_E	The base of natural logarithms (e).
M_LOG2E	The base-2 logarithm of e.
M_LOG10E	The base-10 logarithm of e.
M_LN2	The natural logarithm of 2.
M_LN10	The natural logarithm of 10.
M_PI	The ratio of the circumference of a circle to its diameter. (There are also several fractions of its reciprocal and its square root.)
M_SQRT2	The positive square root of 2.
M_SQRT1_2	The positive square root of 1/2.

For the definitions of various machine-dependent "constants," see the description of the **<values.h>** header file.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
acos	sin(3M)	inverse trigonometric functions
acosh	asinh(3M)	inverse hyperbolic function
asin	sin(3M)	inverse trigonometric function
asinh	asinh(3M)	inverse hyperbolic function
atan	sin(3M)	inverse trigonometric function
atan2	sin(3M)	inverse trigonometric function
atanh	asinh(3M)	inverse hyperbolic function
cabs	hypot(3M)	complex magnitude
cbrt	sqrt(3M)	cube root
ceil	floor(3M)	ceiling function
copysign	ieee(3M)	copy sign bit
cos	sin(3M)	trigonometric function
cosh	sinh(3M)	hyperbolic function
drem	ieee(3M)	remainder
erf	erf(3M)	error function
erfc	erf(3M)	complementary error function
exp	exp(3M)	exponential function
expm1	exp(3M)	exp(X)-1
fabs	floor(3M)	absolute value function
finite	ieee(3M)	test for finite number
floor	floor(3M)	floor function

hypot	hypot(3M)	Euclidean distance
j0	j0(3M)	Bessel function
j1	j0(3M)	Bessel function
jn	j0(3M)	Bessel function
lgamma	lgamma(3M)	log gamma function
log	exp(3M)	natural logarithm
log10	exp(3M)	common logarithm
log1p	exp(3M)	log(1+X)
logb	ieee(3M)	exponent extraction
matherr	matherr(3M)	math library error-handling routines
pow	exp(3M)	power x**y
rint	floor(3M)	round to nearest integral value
scalb	ieee(eM)	exponent adjustment
sin	sin(3M)	trigonometric function
sinh	sinh(3M)	hyperbolic function
sqrt	sqrt(3M)	square root
tan	sin(3M)	trigonometric function
tanh	sinh(3M)	hyperbolic function
y0	j0(3M)	Bessel function
y1	j0(3M)	Bessel function
yn	j0(3M)	Bessel function

NAME

asinh, acosh, atanh – inverse hyperbolic functions

SYNOPSIS

```
#include <math.h>

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;
```

DESCRIPTION

These functions compute the designated inverse hyperbolic functions for real arguments. They inherit much of their (roundoff, etc.) error from *log1p*, as described in *exp*(3M).

SEE ALSO

intro(3M), exp(3M)

DIAGNOSTICS

Acosh returns a NaN if the argument is less than 1.

Atanh returns a NaN if the argument has absolute value greater than 1.

NAME

erf, erfc – error functions

SYNOPSIS

```
#include <math.h>

double erf(x)
double x;

double erfc(x)
double x;
```

DESCRIPTION

Erf (x) returns the error function of x; where erf (x) := $(2/\sqrt{\pi}) \int_0^x \exp(-t^2)\, dt$.

Erfc (x) returns 1.0–erf (x).

The entry for erfc is provided because of the extreme loss of relative accuracy if erf (x) is called for large x and the result subtracted from 1. (e.g. for x = 10, 12 places are lost).

SEE ALSO

intro(3M)

NAME

exp, log, log10, pow – exponential, logarithm, power

SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double expm1(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double log1p(x)
double x;

double pow(x, y)
double x, y;
```

DESCRIPTION

Exp returns the exponential function of x.

Expm1 returns exp(x)-1 accurately even for tiny x.

Log returns the natural logarithm of x.

Log10 returns the base 10 logarithm.

Log1p returns log(1+x) accurately even for tiny x;

Pow returns x^y.

SEE ALSO

hypot(3M), sinh(3M), intro(2)

DIAGNOSTICS

These functions handle exceptional arguments in the spirit of IEEE standard P754 for binary floating point arithmetic. Log(x) for x < 0, log10(x) for x < 0, pow(0.0,0.0), pow(infinity,0.0), and pow(1.0,infinity) are invalid, as is pow(x,y) if x < 0 and y is not an integer value or infinite value; in all these cases NaN function values are returned and errno is set to EDOM.

NAME

floor, ceil, fabs, rint – absolute value, floor, ceiling and round-to-nearest functions

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;

double rint(x)
double x;
```

DESCRIPTION

Fabs returns the absolute value $|x|$.

Floor returns the value of the greatest integer less than or equal to x.

Ceil returns the value of the least integer greater than or equal to x.

Rint returns the value of the integer nearest x in the direction of the prevailing rounding mode.

SEE ALSO

abs(3)

NAME

hypot, cabs – Euclidean distance

SYNOPSIS

```
#include <math.h>

double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y;} z;
```

DESCRIPTION

Hypot and *cabs* return

sqrt(x*x + y*y),

taking precautions against unwarranted overflows.

SEE ALSO

exp(3M) for *sqrt*

NAME

ieee, copysign, drem, finite, logb, scalb – copysign, remainder, exponent manipulations

SYNOPSIS

```
#include <math.h>

double copysign(x,y)
double x,y;

double drem(x,y)
double x,y;

int finite(x)
double x;

double logb(x)
double x;

double scalb(x,n)
double x;
int n;
```

DESCRIPTION

These functions are required for, or recommended by the IEEE standard 754 for floating–point arithmetic.

Copysign(x,y) returns x with its sign changed to y's.

Drem(x,y) returns the remainder r := x − n*y where n is the integer nearest the exact value of x/y; moreover if |n − x/y| = 1/2 then n is even. Consequently the remainder is computed exactly and |r| ≤ |y|/2. But drem(x,0) is exceptional; see below under DIAGNOSTICS.

Finite(x) = 1 just when −∞ < x < +∞,
= 0 otherwise (when |x| = ∞ or x is *NaN*.)

Logb(x) returns x's exponent n, a signed integer converted to double–precision floating–point and so chosen that 1 ≤ |x|/2**n < 2 unless x = 0 or (only on machines that conform to IEEE 754) |x| = ∞ or x lies between 0 and the Underflow Threshold; see below under "BUGS".

Scalb(x,n) = x*(2**n) computed, for integer n, without first computing 2**n.

SEE ALSO

floor(3M), intro(3M)

DIAGNOSTICS

IEEE 754 defines drem(x,0) and drem(∞,y) to be invalid operations that produce a *NaN*.

IEEE 754 defines logb(±∞) = +∞ and logb(0) = −∞, and requires the latter to signal Division–by–Zero.

IEEE 754 currently specifies that logb(denormalized no.) = logb(tiniest normalized no. > 0) but the consensus has changed to the specification in the new proposed IEEE standard p854, namely that logb(x) satisfy

1 ≤ scalb(|x|,−logb(x)) < Radix ... = 2 for IEEE 754

for every x except 0, ∞ and *NaN*. Almost every program that assumes 754's specification will work correctly if logb follows 854's specification instead.

IEEE 754 requires copysign(x,*NaN*) = ±x but says nothing else about the sign of a *NaN* - (*N*ot *a N*umber.)

NAME

j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x)
double x;
int n;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
double x;
int n;
```

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS

Negative arguments cause *y0*, *y1*, and *yn* to return a huge negative value and set *errno* to EDOM.

NAME

lgamma, gamma – log gamma function

SYNOPSIS

```
#include <math.h>

double lgamma(x)
double x;

double gamma(x)
double x;
```

DESCRIPTION

Lgamma

lgamma
returns $\ln|\Gamma(x)|$ where $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\,dt$ for $x > 0$ and $\Gamma(x) = \pi/(\Gamma(1-x)\sin(\pi x))$ for $x < 1$.

The external integer *signgam* returns the sign of $\Gamma(x)$.

Gamma

Gamma returns $\ln|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer *signgam*. The following C program might be used to calculate Γ:

```
y = gamma(x);
#ifdef vax
if (y > 88.0)
#endif
#ifdef sun
if (y > 706.0)
#endif
error();
y = exp(y);
if(signgam)
y = -y;
```

IDIOSYNCRASIES

Do **not** use the expression *signgam*exp(lgamma(x))* to compute $g := \Gamma(x)$. Instead use a program like this (in C):

```
lg = lgamma(x); g = signgam*exp(lg);
```

Only after *lgamma* has returned can *signgam* be correct. Note too that $\Gamma(x)$ must overflow when x is large enough, underflow when $-x$ is large enough, and spawn a division by zero when x is a nonpositive integer.

DIAGNOSTICS

For very large arguments over/underflows will occur inside the *lgamma* routine.

gamma returns a huge value for negative integer arguments.

SEE ALSO

intro(3M)

BUGS

gamma should return a positive indication of error.

Only in the UNIX math library for C was the name gamma ever attached to $ln\Gamma$. Elsewhere, for instance in IBM's FORTRAN library, the name GAMMA belongs to Γ and the name ALGAMA to $\ln\Gamma$ in single precision; in double the names are DGAMMA and DLGAMA. Why should C be different?

NAME

matherr – math library error-handling function

SYNOPSIS

#include <math.h>

int matherr (x)
struct exception *x;

DESCRIPTION

matherr is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors, by including a function named *matherr* in their programs. *matherr* must be of the form described above. When an error occurs, a pointer to the exception structure *x* will be passed to the user-supplied *matherr* function. This structure, which is defined in the **<math.h>** header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain error
SING	argument singularity
OVERFLOW	overflow range error
UNDERFLOW	underflow range error
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element *name* points to a string containing the name of the function that incurred the error. The variables *arg1* and *arg2* are the arguments with which the function was invoked. *Retval* is set to the default value that will be returned by the function unless the user's *matherr* sets it to a different value.

If the user's *matherr* function returns non-zero, no error message will be printed, and *errno* will not be set.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to EDOM or ERANGE and the program continues.

NOTE

In the Sun environment, the facilities provided by *matherr* are only available when a program is built with the software floating point library, as there would be a substantial performance penalty imposed by providing these facilities with the libraries that support various Sun floating point hardware options.

EXAMPLE

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
        switch (x->type) {
        case DOMAIN:
                /* change sqrt to return sqrt(-arg1), not 0 */
                if (!strcmp(x->name, "sqrt")) {
                        x->retval = sqrt(-x->arg1);
                        return (0); /* print message and set errno */
```

```
                }
        case SING:
                /* all other domain or sing errors, print message and abort */
                fprintf(stderr, "domain error in %s\n", x->name);
                abort( );
        case PLOSS:
                /* print detailed error message */
                fprintf(stderr, "loss of significance in %s(%g) = %g\n",
                        x->name, x->arg1, x->retval);
                return (1); /* take no other action */
        }
        return (0); /* all other errors, execute default procedure */
}
```

ERROR HANDLING

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL:	–	–	–	–	M, 0	*
y0, y1, yn (arg ≤ 0)	M, –H	–	–	–	–	–
EXP:	–	–	H	0	–	–
LOG, LOG10:						
(arg < 0)	M, –H	–	–	–	–	–
(arg = 0)	–	M, –H	–	–	–	–
POW:	–	–	±H	0	–	–
neg ** non-int	M, 0	–	–	–	–	–
0 ** non-pos						
SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	H	–	–	–
HYPOT:	–	–	H	–	–	–
SINH:	–	–	±H	–	–	–
COSH:	–	–	H	–	–	–
SIN, COS, TAN: –	–	–	–	M, 0	*	
ASIN, ACOS, ATAN2: M, 0	–	–	–	–	–	

ABBREVIATIONS

*	As much as possible of the value is returned.
M	Message is printed (EDOM error).
H	HUGE is returned.
–H	–HUGE is returned.
±H	HUGE or –HUGE is returned.
0	0 is returned.

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(y, x)
double x, y;
```

DESCRIPTION

Sin, *cos* and *tan* return trigonometric functions of radian arguments.

Asin returns the arc sin in the range $-\pi/2$ to $\pi/2$.

Acos returns the arc cosine in the range 0 to π.

Atan returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arc tangent of y/x in the range $-\pi$ to π.

DIAGNOSTICS

These functions handle exceptional arguments in the spirit of IEEE standard P754 for binary floating point arithmetic. When x is infinity in sin(x), cos(x), or tan(x), or when |x| > 1 in asin(x) or acos(x), the functions return NaN values and errno is set to EDOM.

NAME

sinh, cosh, tanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

These functions compute the designated hyperbolic functions for real arguments.

DIAGNOSTICS

These functions handle exceptional arguments in the spirit of IEEE standard P754 for binary floating point arithmetic. Thus *sinh* and *cosh* return infinity on overflow.

NAME

sqrt, cbrt – cube root, square root

SYNOPSIS

```
#include <math.h>

double cbrt(x)
double x;

double sqrt(x)
double x;
```

DESCRIPTION

Cbrt(x) returns the cube root of x.

Sqrt(x) returns the square root of x.

SEE ALSO

intro(3M)

DIAGNOSTICS

ERROR (due to Roundoff etc.)

Cbrt is accurate to within 0.7 *ulps*.

Sqrt on a machine that conforms to IEEE 754 is correctly rounded in accordance with the rounding mode in force; the error is less than half an *ulp* in the default mode (round–to–nearest). An *ulp* is one *U*nit in the *L*ast *P*lace carried.

NAME

intro – introduction to network library functions

DESCRIPTION

This section describes functions that are applicable to the DARPA Internet network, which are part of the standard C library.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
endhostent	gethostent(3N)	get network host entry
endnetent	getnetent(3N)	get network entry
endprotoent	getprotoent(3N)	get protocol entry
endservent	getservent(3N)	get service entry
gethostbyaddr	gethostent(3N)	get network host entry
gethostbyname	gethostent(3N)	get network host entry
gethostent	gethostent(3N)	get network host entry
getnetbyaddr	getnetent(3N)	get network entry
getnetbyname	getnetent(3N)	get network entry
getnetent	getnetent(3N)	get network entry
getprotobyname	getprotoent(3N)	get protocol entry
getprotobynumber	getprotoent(3N)	get protocol entry
getprotoent	getprotoent(3N)	get protocol entry
getrpcbyname	getrpcent(3N)	get rpc entry
getrpcbynumber	getrpcent(3N)	get rpc entry
getrpcent	getrpcent(3N)	get rpc entry
getservbyname	getservent(3N)	get service entry
getservbyport	getservent(3N)	get service entry
getservent	getservent(3N)	get service entry
htonl	byteorder(3N)	convert values between host and network byte order
htons	byteorder(3N)	convert values between host and network byte order
inet_addr	inet(3N)	Internet address manipulation
inet_lnaof	inet(3N)	Internet address manipulation
inet_makeaddr	inet(3N)	Internet address manipulation
inet_netof	inet(3N)	Internet address manipulation
inet_network	inet(3N)	Internet address manipulation
inet_ntoa	inet(3N)	Internet address manipulation
ntohl	byteorder(3N)	convert values between host and network byte order
ntohs	byteorder(3N)	convert values between host and network byte order
rcmd	rcmd(3N)	routines for returning a stream to a remote command
rexec	rexec(3N)	return stream to a remote command
rresvport	rcmd(3N)	routines for returning a stream to a remote command
ruserok	rcmd(3N)	routines for returning a stream to a remote command
sethostent	gethostent(3N)	get network host entry
setnetent	getnetent(3N)	get network entry
setprotoent	getprotoent(3N)	get protocol entry
setservent	getservent(3N)	get service entry
yp_all	ypclnt(3N)	YP client interface routines
yp_bind	ypclnt(3N)	YP client interface routines
yp_first	ypclnt(3N)	YP client interface routines
yp_get_default_domain		ypclnt(3N)YP client interface routines
yp_master	ypclnt(3N)	YP client interface routines
yp_match	ypclnt(3N)	YP client interface routines
yp_next	ypclnt(3N)	YP client interface routines
yp_order	ypclnt(3N)	YP client interface routines

yp_unbind	ypclnt(3N)	YP client interface routines
ypclnt	ypclnt(3N)	YP client interface routines
yperr_string	ypclnt(3N)	YP client interface routines
ypprot_err	ypclnt(3N)	YP client interface routines

NAME

byteorder, htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

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

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. On machines such as the Sun these routines are defined as null macros in the include file *<netinet/in.h>*.

These routines are most often used in conjunction with Internet addresses and ports as returned by *gethostent*(3N) and *getservent*(3N).

SEE ALSO

gethostent(3N), getservent(3N)

BUGS

The VAX handles bytes backwards from most everyone else in the world. This is not expected to be fixed in the near future.

NAME

ethers, ether_ntoa, ether_aton, ether_ntohost, ether_hostton, ether_line – Ethernet address mapping operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>

char *
ether_ntoa(e)
        struct ether_addr *e;

struct ether_addr *
ether_aton(s)
        char *s;

ether_ntohost(hostname, e)
        char *hostname;
        struct ether_addr *e;

ether_hostton(hostname, e)
        char *hostname;
        struct ether_addr *e;

ether_line(l, e, hostname)
        char *l;
        struct ether_addr *e;
        char *hostname;
```

DESCRIPTION

ether_ntoa, ether_aton, ether_ntohost, ether_hostton, ether_line

These routines are useful for mapping 48 bit Ethernet numbers to their ASCII representations or their corresponding host names, and vice versa.

The function *ether_ntoa* converts a 48 bit Ethernet number pointed to by *e* to its standard ACSII representation; it returns a pointer to the ASCII string. The representation is of the form: "x:x:x:x:x:x" where *x* is a hexadecimal number between 0 and ff. The function *ether_aton* converts an ASCII string in the standard representation back to a 48 bit Ethernet number; the function returns NULL if the string cannot be scanned successfully.

The function *ether_ntohost* maps an Ethernet number (pointed to by *e*) to its associated hostname. The string pointed to by *hostname* must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure. Inversely, the function *ether_hostton* maps a hostname string to its corresponding Ethernet number; the function modifies the Ethernet number pointed to by *e*. The function also returns zero upon success and non-zero upon failure.

The function *ether_line* scans a line (pointed to by *l*) and sets the hostname and the Ethernet number (pointed to by *e*). The string pointed to by *hostname* must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure. The format of the scanned line is described by *ethers*(5).

FILES

/etc/ethers (or the yellowpages' maps ethers.byaddr and ethers.byname)

SEE ALSO

ethers(5)

NAME

gethostent, gethostbyaddr, gethostbyname, sethostent, endhostent – get network host entry

SYNOPSIS

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

struct hostent *gethostent()

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

sethostent(stayopen)
int stayopen
endhostent()
```

DESCRIPTION

Gethostent, *gethostbyname*, and *gethostbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network host data base, */etc/hosts*.

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* address type */
        int     h_length;       /* length of address */
        char    *h_addr;/* address */
};
```

The members of this structure are:

h_name — Official name of the host.

h_aliases — A zero terminated array of alternate names for the host.

h_addrtype — The type of address being returned; currently always AF_INET.

h_length — The length, in bytes, of the address.

h_addr — A pointer to the network address for the host. Host addresses are returned in network byte order.

Gethostent reads the next line of the file, opening the file if necessary.

Sethostent opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base will not be closed after each call to *gethostent* (either directly, or indirectly through one of the other ''gethost'' calls).

Endhostent closes the file.

Gethostbyname and *gethostbyaddr* sequentially search from the beginning of the file until a matching host name or host address is found, or until EOF is encountered. Host addresses are supplied in network order.

FILES

/etc/hosts
/etc/yp/*domainname*/hosts.byname
/etc/yp/*domainname*/hosts.byaddr

SEE ALSO

hosts(5), ypserv(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net, type)
long net;
int type;

setnetent(stayopen)
int stayopen;

endnetent()
```

DESCRIPTION

getnetent, *getnetbyname*, and *getnetbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, **/etc/networks**.

```
struct    netent {
          char      *n_name;        /* official name of net */
          char      **n_aliases;    /* alias list */
          int       n_addrtype;     /* net number type */
          long      n_net;          /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

n_aliases A zero terminated list of alternate names for the network.

n_addrtype The type of the network number returned; currently only AF_INET.

n_net The network number. Network numbers are returned in machine byte order.

getnetent reads the next line of the file, opening the file if necessary.

setnetent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetent* (either directly, or indirectly through one of the other "getnet" calls).

endnetent closes the file.

Getnetbyname and *getnetbyaddr* sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks
/etc/yp/*domainname*/networks.byname
/etc/yp/*domainname*/networks.byaddr

SEE ALSO

networks(5), ypserv(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

Only Internet network numbers are currently understood.

NAME

getnetgrent, setnetgrent, endnetgrent, innetgr – get network group entry

SYNOPSIS

innetgr(netgroup, machine, user, domain)
char *netgroup, *machine, *user, *domain;

setnetgrent(netgroup)
char *netgroup

endnetgrent()

getnetgrent(machinep, userp, domainp)
char **machinep, **userp, **domainp;

DESCRIPTION

Inngetgr returns 1 or 0, depending on whether *netgroup* contains the machine, user, domain triple as a member. Any of the three strings machine, user, or domain can be NULL, in which case it signifies a wild card.

Getnetgrent returns the next member of a network group. After the call, machinep will contain a pointer to a string containing the name of the machine part of the network group member, and similarly for userp and domainp. If any of machinep, userp or domainp is returned as a NULL pointer, it signifies a wild card. Getnetgrent will malloc space for the name. This space is released when a endnetgrent call is made. Getnetgrent returns 1 if it succeeding in obtaining another member of the network group, 0 if it has reached the end of the group.

Setnetgrent establishes the network group from which getnetgrent will obtain members, and also restarts calls to getnetgrent from the beginning of the list. If the previous setnetgrent call was to a different network group, a endnetgrent call is implied. *Endnetgrent* frees the space allocated during the getnetgrent calls.

FILES

/etc/netgroup
/etc/yp/*domain*/netgroup
/etc/yp/*domain*/netgroup.byuser
/etc/yp/*domain*/netgroup.byhost

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen;

endprotoent()
```

DESCRIPTION

getprotoent, *getprotobyname*, and *getprotobynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, **/etc/protocols**.

```
struct  protoent {
        char    *p_name;        /* official name of protocol */
        char    **p_aliases;    /* alias list */
        int     p_proto; /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero terminated list of alternate names for the protocol.

p_proto The protocol number.

getprotoent reads the next line of the file, opening the file if necessary.

setprotoent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getprotoent* (either directly, or indirectly through one of the other "getproto" calls).

endprotoent closes the file.

getprotobyname and *getprotobynumber* sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

FILES

/etc/protocols
/etc/yp/*domainname*/protocols.byname
/etc/yp/*domainname*/protocols.bynumber

SEE ALSO

protocols(5), ypserv(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

NAME

getrpcent, getrpcbyname, getrpcbynumber – get RPC entry

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbyname(name)
char *name;

struct rpcent *getrpcbynumber(number)
int number;

setrpcent(stayopen)
int stayopen

endrpcent()
```

DESCRIPTION

Getrpcent, *getrpcbyname*, and *getrpcbynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the rpc program number data base, */etc/rpc*.

```
struct  rpcent {
        char    *r_name;        /* name of server for this rpc program */
        char    **r_aliases;    /* alias list */
        long    r_number;       /* rpc program number */
};
```

The members of this structure are:

r_name The name of the server for this rpc program.

r_aliases A zero terminated list of alternate names for the rpc program.

r_number The rpc program number for this service.

Getrpcent reads the next line of the file, opening the file if necessary.

Setrpcent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getrpcent* (either directly, or indirectly through one of the other "getrpc" calls).

Endrpcent closes the file.

Getrpcbyname and *getrpcbynumber* sequentially search from the beginning of the file until a matching rpc program name or program number is found, or until EOF is encountered.

FILES

/etc/rpc
/etc/yp/*domainname*/rpc.bynumber

SEE ALSO

rpc(5), rpcinfo(8), ypservices(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getservent, getservbyport, getservbyname, setservent, endservent – get service entry

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port; char *proto;

setservent(stayopen)
int stayopen;

endservent()
```

DESCRIPTION

getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, **/etc/services**.

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A zero terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto The name of the protocol to use when contacting the service.

getservent reads the next line of the file, opening the file if necessary.

setservent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getservent* (either directly, or indirectly through one of the other "getserv" calls).

endservent closes the file.

getservbyname and *getservbyport* sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

FILES

/etc/services
/etc/yp/*domainname*/services.byname

SEE ALSO

getprotoent(3N), services(5), ypserv(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Expecting port numbers to fit in a 32 bit quantity is probably naive.

NAME

inet inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof, inet_ntoa – Internet address manipulation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long
inet_addr(cp)
char *cp;

inet_network(cp)
char *cp;

struct in_addr
inet_makeaddr(net, lna)
int net, lna;

inet_lnaof(in)
struct in_addr in;

inet_netof(in)
struct in_addr in;

char *
inet_ntoa(in)
struct in_addr in;
```

DESCRIPTION

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard ".'' notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof* break apart Internet host addresses, returning the network number and local network address part, respectively.

The routine *inet_ntoa* returns a pointer to a string in the base 256 notation "d.d.d.d" described below.

All Internet address are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the "." notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on the VAX the bytes referred to above appear as "d.c.b.a". That is, VAX bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a "." notation may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

gethostent(3N), getnetent(3N), hosts(5), networks(5),

DIAGNOSTICS

The value −1 is returned by *inet_addr* and *inet_network* for malformed requests.

BUGS

The problem of host byte ordering versus network byte ordering is confusing. A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed.

The return value from *inet_ntoa* points to static information which is overwritten in each call.

NAME

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

SYNOPSIS

```
rem = rcmd(ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
u_short inport;
char *locuser, *remuser, *cmd;
int *fd2p;

s = rresvport(port);
int *port;

ruserok(rhost, superuser, ruser, luser);
char *rhost;
int superuser;
char *ruser, *luser;
```

DESCRIPTION

Rcmd is a routine used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. *Rresvport* is a routine which returns a descriptor to a socket with an address in the privileged port space. *Ruserok* is a routine used by servers to authenticate clients requesting service with *rcmd*. All three functions are present in the same file and are used by the *rshd*(8C) server (among others).

Rcmd looks up the host **ahost* using *gethostbyname*(3N), returning −1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in *rshd*(8C).

The *rresvport* routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by *rcmd* and several other routines. Privileged addresses consist of a port in the range 0 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

Ruserok takes a remote host's name, as returned by a *gethostent*(3N) routine, two user names and a flag indicating if the local user's name is the super-user. It then checks the files */etc/hosts.equiv* and, possibly, *.rhosts* in the current working directory (normally the local user's home directory) to see if the request for service is allowed. A 0 is returned if the machine name is listed in the "hosts.equiv" file, or the host and remote user name are found in the ".rhosts" file; otherwise *ruserok* returns -1. If the *superuser* flag is 1, the checking of the "host.equiv" file is bypassed.

SEE ALSO

rlogin(1C), rsh(1C), rexec(3N), rexecd(8C), rlogind(8C), rshd(8C)

BUGS

There is no way to specify options to the *socket* call which *rcmd* makes.

NAME

rexec – return stream to a remote command

SYNOPSIS

rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
u_short inport;
char *user, *passwd, *cmd;
int *fd2p;

DESCRIPTION

Rexec looks up the host **ahost* using *gethostbyname*(3N), returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's *.netrc* file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call ''getservbyname("exec", "tcp")'' (see *getservent*(3N)). The protocol for connection is described in detail in *rexecd*(8C).

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then a auxiliary channel to a control process will be setup, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rcmd(3N), rexecd(8C)

BUGS

There is no way to specify options to the *socket* call which *rexec* makes.

NAME

rpc – library routines for remote procedure calls

DESCRIPTION

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

FUNCTIONS

auth_destroy()	destroy authentication information handle
authnone_create()	return RPC authentication handle with no checking
authunix_create()	return RPC authentication handle with UNIX permissions
authunix_create_default()	return default UNIX authentication handle
callrpc()	call remote procedure, given [prognum,versnum,procnum]
clnt_broadcast()	broadcast remote procedure call everywhere
clnt_call()	call remote procedure associated with client handle
clnt_destroy()	destroy client's RPC handle
clnt_freeres()	free data allocated by RPC/XDR system when decoding results
clnt_geterr()	copy error information from client handle to error structure
clnt_pcreateerror()	print message to stderr about why client handle creation failed
clnt_perrno()	print message to stderr corresponing to condition given
clnt_perror()	print message to stderr about why RPC call failed
clnt_sperrno()	print message to a string corresponding to condition given
clnt_sperror()	print message to a string
clntraw_create()	create toy RPC client for simulation
clnttcp_create()	create RPC client using TCP transport
clntudp_create()	create RPC client using UDP transport
get_myaddress()	get the machine's IP address
pmap_getmaps()	return list of RPC program-to-port mappings
pmap_getport()	return port number on which waits supporting service
pmap_rmtcall()	instructs portmapper to make an RPC call
pmap_set()	establish mapping between [prognum,versnum,procnum] and port
pmap_unset()	destroy mapping between [prognum,versnum,procnum] and port
registerrpc()	register procedure with RPC service package
rpc_createerr	global variable indicating reason why client creation failed
svc_destroy()	destroy RPC service transport handle
svc_fds	global variable with RPC service file descriptor mask
svc_freeargs()	free data allocated by RPC/XDR system when decoding arguments
svc_getargs()	decodes the arguments of an RPC request
svc_getcaller()	get the network address of the caller of a procedure
svc_getreq()	returns when all associated sockets have been serviced
svc_register()	associates prognum and versnum with service dispatch procedure
svc_run()	wait for RPC requests to arrive and call appropriate service
svc_sendreply()	send back results of a remote procedure call
svc_unregister()	remove mapping of [prognum,versnum] to dispatch routines
svcerr_auth()	called when refusing service because of authentication error
svcerr_decode()	called when service cannot decode its parameters
svcerr_noproc()	called when service hasn't implemented the desired procedure
svcerr_noprog()	called when program is not registered with RPC package
svcerr_progvers()	called when version is not registered with RPC package
svcerr_systemerr()	called when service detects system error
svcerr_weakauth()	called when refusing service because of insufficient authentication
svcraw_create()	creates a toy RPC service transport for testing
svctcp_create()	creates an RPC service based on TCP transport

svcudp_create()	creates an RPC service based on UDP transport
xdr_accepted_reply()	generates RPC-style replies without using RPC package
xdr_authunix_parms()	generates UNIX credentials without using RPC package
xdr_callhdr()	generates RPC-style headers without using RPC package
xdr_callmsg()	generates RPC-style messages without using RPC package
xdr_opaque_auth()	describes RPC messages, externally
xdr_pmap()	describes parameters for portmap procedures, externally
xdr_pmaplist()	describes a list of port mappings, externally
xdr_rejected_reply()	generates RPC-style rejections without using RPC package
xdr_replymsg()	generates RPC-style replies without using RPC package
xprt_register()	registers RPC service transport with RPC package
xprt_unregister()	unregisters RPC service transport from RPC package

SEE ALSO

Remote Procedure Call Programming Guide, in *Networking on the Sun Workstation*.

NAME

xdr – library routines for external data representation

DESCRIPTION

These routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls are transmitted using these routines.

FUNCTIONS

xdr_array()	translate arrays to/from external representation
xdr_bool()	translate Booleans to/from external representation
xdr_bytes()	translate counted byte strings to/from external representation
xdr_destroy()	destroy XDR stream and free associated memory
xdr_double()	translate double precision to/from external representation
xdr_enum()	translate enumerations to/from external representation
xdr_float()	translate floating point to/from external representation
xdr_getpos()	return current position in XDR stream
xdr_inline()	invoke the in-line routines associated with XDR stream
xdr_int()	translate integers to/from external representation
xdr_long()	translate long integers to/from external representation
xdr_opaque()	translate fixed-size opaque data to/from external representation
xdr_reference()	chase pointers within structures
xdr_setpos()	change current position in XDR stream
xdr_short()	translate short integers to/from external representation
xdr_string()	translate null-terminated strings to/from external representation
xdr_u_int()	translate unsigned integers to/from external representation
xdr_u_long()	translate unsigned long integers to/from external representation
xdr_u_short()	translate unsigned short integers to/from external representation
xdr_union()	translate discriminated unions to/from external representation
xdr_void()	always return one (1)
xdr_wrapstring()	package RPC routine for XDR routine, or vice-versa
xdrmem_create()	initialize an XDR stream
xdrrec_create()	initialize an XDR stream with record boundaries
xdrrec_endofrecord()	mark XDR record stream with an end-of-record
xdrrec_eof()	mark XDR record stream with an end-of-file
xdrrec_skiprecord()	skip remaining record in XDR record stream
xdrstdio_create()	initialize an XDR stream as standard I/O FILE stream

SEE ALSO

External Data Representation Protocol Specification, in *Networking on the Sun Workstation*.

NAME

ypclnt, yp_get_default_domain, yp_bind, yp_unbind, yp_match, yp_first, yp_next, yp_all, yp_order, yp_master, yperr_string, ypprot_err – yellow pages client interface

SYNOPSIS

```
#include <rpcsvc/ypclnt.h>

yp_bind(indomain);
char *indomain;

void yp_unbind(indomain)
char *indomain;

yp_get_default_domain(outdomain);
char **outdomain;

yp_match(indomain, inmap, inkey, inkeylen, outval, outvallen)
char *indomain;
char *inmap;
char *inkey;
int inkeylen;
char **outval;
int *outvallen;

yp_first(indomain, inmap, outkey, outkeylen, outval, outvallen)
char *indomain;
char *inmap;
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;

yp_next(indomain, inmap, inkey, inkeylen, outkey, outkeylen, outval, outvallen);
char *indomain;
char *inmap;
char *inkey;
int inkeylen;
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;

yp_all(indomain, inmap, incallback);
char *indomain;
char *inmap;
struct ypall_callback incallback;

yp_order(indomain, inmap, outorder);
char *indomain;
char *inmap;
int *outorder;

yp_master(indomain, inmap, outname);
char *indomain;
char *inmap;
char **outname;

char *yperr_string(incode)
int incode;
```

ypprot_err(incode)
unsigned int incode;

DESCRIPTION

This package of functions provides an interface to the yellow pages (YP) network lookup service. The package can be loaded from the standard library, */lib/libc.a*. Refer to ypfiles(5) and ypserv(8) for an overview of the yellow pages, including the definitions of *map* and *domain* , and a description of the various servers, databases, and commands that comprise the YP.

All input parameters names begin with **in**. Output parameters begin with **out**. Output parameters of type *char* ** should be addresses of uninitialized character pointers. Memory is allocated by the YP client package using *malloc*(3), and may be freed if the user code has no continuing need for it. For each *outkey* and *outval*, two extra bytes of memory are allocated at the end that contain NEWLINE and NULL, respectively, but these two bytes are not reflected in *outkeylen* or *outvallen* . *indomain* and *inmap* strings must be non-null and null-terminated. String parameters which are accompanied by a count parameter may not be null, but may point to null strings, with the count parameter indicating this. Counted strings need not be null-terminated.

All functions in this package of type **int** return 0 if they succeed, and a failure code (YPERR_*xxxx*) otherwise. Failure codes are described under **DIAGNOSTICS** below.

The YP lookup calls require a map name and a domain name, at minimum. It is assumed that the client process knows the name of the map of interest. Client processes should fetch the node's default domain by calling **yp_get_default_domain() ,** and use the returned *outdomain* as the *indomain* parameter to successive YP calls.

To use the YP services, the client process must be "bound" to a YP server that serves the appropriate domain using *yp_bind*. Binding need not be done explicitly by user code; this is done automatically whenever a YP lookup function is called. *yp_bind* can be called directly for processes that make use of a backup strategy (e.g., a local file) in cases when YP services are not available.

Each binding allocates (uses up) one client process socket descriptor; each bound domain costs one socket descriptor. However, multiple requests to the same domain use that same descriptor. *yp_unbind()* is available at the client interface for processes that explicitly manage their socket descriptors while accessing multiple domains. The call to *yp_unbind()* make the domain *unbound*, and free all per-process and per-node resources used to bind it.

If an RPC failure results upon use of a binding, that domain will be unbound automatically. At that point, the ypclnt layer will retry forever or until the operation succeeds, provided that *ypbind* is running, and either

a) the client process can't bind a server for the proper domain, or

b) RPC requests to the server fail.

If an error is not RPC-related, or if *ypbind* is not running, or if a bound *ypserv* process returns any answer (success or failure), the ypclnt layer will return control to the user code, either with an error code, or a success code and any results.

yp_match returns the value associated with a passed key. This key must be exact; no pattern matching is available.

yp_first returns the first key-value pair from the named map in the named domain.

yp_next() returns the next key-value pair in a named map. The *inkey* parameter should be the *outkey* returned from an initial call to *yp_first()* (to get the second key-value pair) or the one returned from the nth call to *yp_next()* (to get the nth + second key-value pair).

The concept of first (and, for that matter, of next) is particular to the structure of the YP map being processing; there is no relation in retrieval order to either the lexical order within any original (non-YP) data base, or to any obvious numerical sorting order on the keys, values, or key-value pairs. The only ordering guarantee made is that if the *yp_first()* function is called on a particular map, and then the *yp_next()*

function is repeatedly called on the same map at the same server until the call fails with a reason of YPERR_NOMORE, every entry in the data base will be seen exactly once. Further, if the same sequence of operations is performed on the same map at the same server, the entries will be seen in the same order.

Under conditions of heavy server load or server failure, it is possible for the domain to become unbound, then bound once again (perhaps to a different server) while a client is running. This can cause a break in one of the enumeration rules; specific entries may be seen twice by the client, or not at all. This approach protects the client from error messages that would otherwise be returned in the midst of the enumeration. The next paragraph describes a better solution to enumerating all entries in a map.

yp_all provides a way to transfer an entire map from server to client in a single request using TCP (rather than UDP as with other functions in this package). The entire transaction take place as a single RPC request and response. You can use *yp_all* just like any other YP procedure, identify the map in the normal manner, and supply the name of a function which will be called to process each key-value pair within the map. You return from the call to *yp_all* only when the transaction is completed (successfully or unsuccessfully), or your *"foreach"* function decides that it doesn't want to see any more key-value pairs.

The third parameter to *yp_all* is

```
struct ypall_callback *incallback {
        int (*foreach)();
        char *data;
};
```

The function *foreach* is called

```
foreach(instatus, inkey, inkeylen, inval, invallen, indata);
int instatus;
char *inkey;
int inkeylen;
char *inval;
int invalllen;
char *indata;
```

The *instatus* parameter will hold one of the return status values defined in `<rpcsvc/yp_prot.h>` — either `YP_TRUE` or an error code. (See *ypprot_err* , below, for a function which converts a YP protocol error code to a ypclnt layer error code.)

The key and value parameters are somewhat different than defined in the synopsis section above. First, the memory pointed to by the *inkey* and *inval* parameters is private to the *yp_all* function, and is overwritten with the arrival of each new key-value pair. It is the responsibility of the *foreach* function to do something useful with the contents of that memory, but it does not own the memory itself. Key and value objects presented to the *foreach* function look exactly as they do in the server's map — if they were not newline-terminated or null-terminated in the map, they won't be here either.

The *indata* parameter is the contents of the *incallback->data* element passed to *yp_all* . The *data* element of the callback structure may be used to share state information between the *foreach* function and the mainline code. Its use is optional, and no part of the YP client package inspects its contents — cast it to something useful, or ignore it as you see fit.

The *foreach* function is a Boolean. It should return zero to indicate that it wants to be called again for further received key-value pairs, or non-zero to stop the flow of key-value pairs. If *foreach* returns a non-zero value, it is not called again; the functional value of *yp_all* is then 0.

yp_order returns the order number for a map.

yp_master returns the machine name of the master YP server for a map.

yperr_string returns a pointer to an error message string that is null-terminated but contains no period or newline.

ypprot_err takes a YP protocol error code as input, and returns a ypclnt layer error code, which may be used in turn as an input to *yperr_string* .

FILES

/usr/include/rpcsvc/ypclnt.h
/usr/include/rpcsvc/yp_prot.h

SEE ALSO

ypfiles(5), ypserv(8),

DIAGNOSTICS

All integer functions return 0 if the requested operation is successful, or one of the following errors if the operation fails.

```
#define YPERR_BADARGS   1  /* args to function are bad */
#define YPERR_RPC       2  /* RPC failure - domain has been unbound */
#define YPERR_DOMAIN    3  /* can't bind to server on this domain */
#define YPERR_MAP       4  /* no such map in server's domain */
#define YPERR_KEY       5  /* no such key in map */
#define YPERR_YPERR     6  /* internal yp server or client error */
#define YPERR_RESRC     7  /* resource allocation failure */
#define YPERR_NOMORE    8  /* no more records in map database */
#define YPERR_PMAP      9  /* can't communicate with portmapper */
#define YPERR_YPBIND    10 /* can't communicate with ypbind */
#define YPERR_YPSERV    11 /* can't communicate with ypserv */
#define YPERR_NODOM     12 /* local domain name not set */
```

NAME

intro – introduction to RPC service library functions

DESCRIPTION

These functions constitute the RPC service library, *librpcsvc*. In order to get the link editor to load this library, use the **–lrpcsvc** option of *cc*. Declarations for these functions may be obtained from various include files *<rpcsvc/*.h>*.

LIST OF FUNCTIONS

routine	*on page*	*description*
ether	ether(3R)	monitor traffic on the Ethernet
getrpcport	getrpcport(3R)	get RPC port number
havedisk	rstat(3R)	determine if remote machine has disk
rex	rex(3r)	remote execution protocol
rnusers	rnusers(3R)	return number of users on remote machine
rquota	rquota(3R)	implement quotas on remote machines
rstat	rstat(3R)	get performance data from remote kernel
rusers	rnusers(3R)	return information about users on remote machine
rwall	rwall(3R)	write to specified remote machines
spray	spray(3R)	scatter data in order to check the network
yppasswd	yppasswd(3R)	update user password in yellow pages

NAME

ether – monitor traffic on the Ethernet

SYNPOSIS

#include <rpcsvc/ether.h>

RPC INFO

program number:

ETHERPROG

xdr routines:

```
xdr_etherstat(xdrs, es)
        XDR *xdrs;
        struct etherstat *es;
xdr_etheraddrs(xdrs, ea)
        XDR *xdrs;
        struct etheraddrs *ea;
xdr_etherhtable(xdrs, hm)
        XDR *xdrs;
        struct etherhmem **hm;
xdr_etherhmem(xdrs, hm)
        XDR *xdrs;
        struct etherhmem **hm;
xdr_etherhbody(xdrs, hm)
        XDR *xdrs;
        struct etherhmem *hm;
xdr_addrmask(xdrs, am)
        XDR *xdrs;
        struct addrmask *am;
```

Xdr_etherhmem processes a single *etherhmem* structure. *Xdr_etherhtable* processes an array of HASHSIZE **struct etherhmem*s. The ***etherhmem* field of *etheraddrs* is actually a hashtable, that is, it is a pointer to an array of HASHSIZE *hmem* pointers.

procs:

ETHERPROC_GETDATA
: no args, returns struct etherstat

ETHERPROC_ON
: no args or results, puts server in promiscuous mode

ETHERPROC_OFF
: no args or results, puts server in promiscuous mode

ETHERPROC_GETSRCDATA
: no args, returns struct etheraddrs with information about source of packets

ETHERPROC_GETDSTDATA
: no args, returns struct etheraddrs with information about destination of packets

ETHERPROC_SELECTSRC
: takes struct mask as argument, no results sets a mask for source

ETHERPROC_SELECTDST
: takes struct mask as argument, no results sets a mask for dst

ETHERPROC_SELECTPROTO
: takes struct mask as argument, no results sets a mask for proto

```
            ETHERPROC_SELECTLNTH
                    takes struct mask as argument, no results
                    sets a mask for lnth

    versions:
            ETHERVERS_ORIG

    structures:
            /*
             * all ether stat's except src, dst addresses
             */
            struct etherstat {
                    struct timeval      e_time;
                    unsigned long       e_bytes;
                    unsigned long       e_packets;
                    unsigned long       e_bcast;
                    unsigned long       e_size[NBUCKETS];
                    unsigned long       e_proto[NPROTOS];
            };
            /*
             * member of address hash table
             */
            struct etherhmem {
                    int h_addr;
                    unsigned h_cnt;
                    struct etherhmem *h_nxt;
            };
            /*
             * src, dst address info
             */
            struct etheraddrs {
                    struct timeval      e_time;
                    unsigned long       e_bytes;
                    unsigned long       e_packets;
                    unsigned long       e_bcast;
                    struct etherhmem **e_addrs;
            };
            /*
             * for size, a_addr is lowvalue, a_mask is high value
             */
            struct addrmask {
                    int a_addr;
                    int a_mask;         /* 0 means wild card */
            };
```

SEE ALSO

traffic(1C), etherfind(8C), etherd(8C)

NAME

getrpcport – get RPC port number

SYNOPSIS

```
int getrpcport(host, prognum, versnum, proto)
        char *host;
        int prognum, versnum, proto;
```

DESCRIPTION

Getrpcport returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will return that port number.

NAME

rex – remote execution protocol

SYNOPSIS

#include <sys/ioctl.h>
#include <rpcsvc/rex.h>

DESCRIPTION

This server will execute commands remotely. the working directory and environment of the command can be specified, and the standard input and output of the command can be arbitrarily redirected. An option is provided for interactive I/O for programs that expect to be running on terminals. Note that this service is only provided with the TCP transport.

RPC INFO

program number:
REXPROG

xdr routines:

```
int xdr_rex_start(xdrs, start);
        XDR *xdrs;
        struct rex_start *start;
int  xdr_rex_result(xdrs, result);
        XDR *xdrs;
        struct rex_result *result;
int xdr_rex_ttymode(xdrs, mode);
        XDR *xdrs;
        struct rex_ttymode *mode;
int xdr_rex_ttysize(xdrs, size);
        XDR *xdrs;
        struct ttysize *size;
```

procs:

REXPROC_START
Takes rex_start structure, starts a command executing, and returns a rex_result structure.
REXPROC_WAIT
Takes no arguments, waits for a command to finish executing, and returns a rex_result structure.
REXPROC_MODES
Takes a rex_ttymode structure, and sends the tty modes.
REXPROC_WINCH
Takes a ttysize structure, and sends window size information.

versions:

REXVERS_ORIG
Original version

structures:

```
#define REX_INTERACTIVE         1       /* Interative mode */
struct rex_start {
        char **rst_cmd;                 /* list of command and args */
        char *rst_host;                 /* working directory host name */
        char *rst_fsname;               /* working directory file system name */
        char *rst_dirwithin;            /* working directory within file system */
        char **rst_env;                 /* list of environment */
        u_short rst_port0;              /* port for stdin */
        u_short rst_port1;              /* port for stdin */
        u_short rst_port2;              /* port for stdin */
```

```
        u_long rst_flags;               /* options - see #defines above */
};

struct rex_result {
        int rlt_stat;                   /* integer status code */
        char *rlt_message;              /* string message for human consumption */
};

struct rex_ttymode {
        struct sgttyb basic;            /* standard unix tty flags */
        struct tchars more;             /* interrupt, kill characters, etc. */
        struct ltchars yetmore;         /* special Berkeley characters */
        u_long andmore;                 /* and Berkeley modes */
};
```

SEE ALSO

on(1C), rexd(8C)

NAME

rnusers, rusers – return information about users on remote machines

SYNOPSIS

```
#include <rpcsvc/rusers.h>

rnusers(host)
        char *host

rusers(host, up)
        char *host
        struct utmpidlearr *up;
```

DESCRIPTION

Rnusers returns the number of users logged on to *host* (–1 if it cannot determine that number). *Rusers* fills the *utmpidlearr* structure with data about *host*, and returns 0 if successful. The relevant structures are:

```
struct utmparr {                    /* RUSERSVERS_ORIG */
        struct utmp **uta_arr;
        int uta_cnt
};

struct utmpidle {
        struct utmp ui_utmp;
        unsigned ui_idle;
};

struct utmpidlearr {                /* RUSERSVERS_IDLE */
        struct utmpidle **uia_arr;
        int uia_cnt
};
```

RPC INFO

program number:

RUSERSPROG

xdr routines:

```
int xdr_utmp(xdrs, up)
        XDR *xdrs;
        struct utmp *up;
int xdr_utmpidle(xdrs, ui);
        XDR *xdrs;
        struct utmpidle *ui;
int xdr_utmpptr(xdrs, up);
        XDR *xdrs;
        struct utmp **up;
int xdr_utmpidleptr(xdrs, up);
        XDR *xdrs;
        struct utmpidle **up;
int xdr_utmparr(xdrs, up);
        XDR *xdrs;
        struct utmparr *up;
int xdr_utmpidlearr(xdrs, up);
        XDR *xdrs;
        struct utmpidlearr *up;
```

procs:

RUSERSPROC_NUM

No arguments, returns number of users as an *unsigned long*.

RUSERSPROC_NAMES
No arguments, returns *utmparr* or *utmpidlearr*, depending on version number.
RUSERSPROC_ALLNAMES
No arguments, returns *utmparr* or *utmpidlearr*, depending on version number.
Returns listing even for *utmp* entries satisfying *nonuser()* in *utmp.h*.

versions:
RUSERSVERS_ORIG
RUSERSVERS_IDLE

structures:

SEE ALSO

rusers(1C)

NAME

rquota – implement quotas on remote machines

SYNPOSIS

#include <rpcsvc/rquota.h>

RPC INFO

program number:

RQUOTAPROG

xdr routines:

```
xdr_getquota_args(xdrs, gqa);
        XDR *xdrs;
        struct getquota_args *gqa;
xdr_getquota_rslt(xdrs, gqr);
        XDR *xdrs;
        struct getquota_rslt *gqr;
xdr_rquota(xdrs, rq);
        XDR *xdrs;
        struct rquota *rq;
```

procs:

RQUOTAPROC_GETQUOTA
RQUOTAPROC_GETACTIVEQUOTA
Arguments of *struct getquota_args*.
Returns *struct getquota_rslt*.
Uses UNIX authentication.
Returns quota only on filesystems with quota active.

versions:

RQUOTAVERS_ORIG

structures:

```
struct getquota_args {
        char *gqa_pathp;        /* path to filesystem of interest */
        int gqa_uid;            /* inquire about quota for uid */
};
/*
 * remote quota structure
 */
struct rquota {
        int rq_bsize;           /* block size for block counts */
        bool_t rq_active;       /* indicates whether quota is active */
        u_long rq_bhardlimit;   /* absolute limit on disk blks alloc */
        u_long rq_bsoftlimit;   /* preferred limit on disk blks */
        u_long rq_curblocks;    /* current block count */
        u_long rq_fhardlimit;   /* absolute limit on allocated files */
        u_long rq_fsoftlimit;   /* preferred file limit */
        u_long rq_curfiles;     /* current # allocated files */
        u_long rq_btimeleft;    /* time left for excessive disk use */
        u_long rq_ftimeleft;    /* time left for excessive files */
};
enum gqr_status {
        Q_OK = 1,               /* quota returned */
        Q_NOQUOTA = 2,          /* noquota for uid */
        Q_EPERM = 3             /* no permission to access quota */
};
```

```
struct getquota_rslt {
        enum gqr_status gqr_status;          /* discriminant */
        struct rquota gqr_rquota;   /* valid if status == Q_OK */
};
```

SEE ALSO

quota(1), quotactl(2)

NAME

rstat, havedisk – get performance data from remote kernel

SYNOPSIS

```
#include <rpcsvc/rstat.h>

havedisk(host)
        char *host;

rstat(host, statp)
        char *host;
        struct statstime *statp;
```

DESCRIPTION

Havedisk returns 1 if *host* has a disk, 0 if it does not, and –1 if this cannot be determined. *Rstat* fills in the *statstime* structure for *host*, and returns 0 if it was successful. The relevant structures are:

```
struct stats {                          /* RSTATVERS_ORIG */
        int cp_time[CPUSTATES];
        int dk_xfer[DK_NDRIVE];
        unsigned v_pgpgin;      /* these are cumulative sum */
        unsigned v_pgpgout;
        unsigned v_pswpin;
        unsigned v_pswpout;
        unsigned v_intr;
        int if_ipackets;
        int if_ierrors;
        int if_opackets;
        int if_oerrors;
        int if_collisions;
};
struct statsswtch {                     /* RSTATVERS_SWTCH */
        int cp_time[CPUSTATES];
        int dk_xfer[DK_NDRIVE];
        unsigned v_pgpgin;      /* these are cumulative sum */
        unsigned v_pgpgout;
        unsigned v_pswpin;
        unsigned v_pswpout;
        unsigned v_intr;
        int if_ipackets;
        int if_ierrors;
        int if_opackets;
        int if_oerrors;
        int if_collisions;
        unsigned v_swtch;
        long avenrun[3];
        struct timeval boottime
};
struct statstime {                      /* RSTATVERS_TIME */
        int cp_time[CPUSTATES];
        int dk_xfer[DK_NDRIVE];
        unsigned v_pgpgin;      /* these are cumulative sum */
        unsigned v_pgpgout;
        unsigned v_pswpin;
        unsigned v_pswpout;
        unsigned v_intr;
```

```
        int if_ipackets;
        int if_ierrors;
        int if_opackets;
        int if_oerrors;
        int if_collisions;
        unsigned v_swtch;
        long avenrun[3];
        struct timeval boottime;
        struct timeval curtime;
};
```

RPC INFO

program number:

RSTATPROG

xdr routines:

```
int xdr_stats(xdrs, stat)
        XDR *xdrs;
        struct stats *stat;
int xdr_statsswtch(xdrs, stat)
        XDR *xdrs;
        struct statsswtch *stat;
int xdr_statstime(xdrs, stat)
        XDR *xdrs;
        struct statstime *stat;
int xdr_timeval(xdrs, tv)
        XDR *xdrs;
        struct timeval *tv;
```

procs:

RSTATPROC_HAVEDISK
: Takes no arguments, returns *long* which is true if remote host has a disk.

RSTATPROC_STATS
: Takes no arguments, return *struct statsxxx*, depending on version.

versions:

RSTATVERS_ORIG
RSTATVERS_SWTCH
RSTATVERS_TIME

SEE ALSO

perfmeter(1), rup(1C), rstatd(8C)

NAME

rwall – write to specified remote machines

SYNOPSIS

#include <rpcsvc/rwall.h>

rwall(host, msg);
char *host, *msg;

DESCRIPTION

Rwall causes *host* to print the string *msg* to all its users. It returns 0 if successful.

RPC INFO

program number:
WALLPROG

procs:
WALLPROC_WALL
Takes string as argument (wrapstring), returns no arguments.
Executes *wall* on remote host with string.

versions:
RSTATVERS_ORIG

SEE ALSO

rwall(1), shutdown(8), rwalld(8C)

NAME

spray – scatter data in order to check the network

SYNOPSIS

#include <rpcsvc/spray.h>

RPC INFO

program number:

SPRAYPROG

xdr routines:

```
xdr_sprayarr(xdrs, arr);
        XDR *xdrs;
        struct sprayarr *arr;
xdr_spraycumul(xdrs, cumul);
        XDR *xdrs;
        sturct spraycumul *cumul;
```

procs:

SPRAYPROC_SPRAY

Takes no arguments, returns no value.
Increments a counter in server daemon.
The server does not return this call, so the caller should have a timeout of 0.

SPRAYPROC_GET

Takes no arguments, returns *struct spraycumul* with value of counter and clock.

SPRAYPROC_CLEAR

Takes no arguments and returns no value.
Zeros out counter and clock.

versions:

SPRAYVERS_ORIG

structures:

```
struct spraycumul {
        unsigned counter;
        struct timeval clock;
};
struct sprayarr {
        int *data,
        int lnth
};
```

SEE ALSO

spray(8), sprayd(8)

NAME

yppasswd – update user password in yellow pages

SYNPOSIS

```
#include <rpcsvc/yppasswd.h>

yppasswd(oldpass, newpw)
        char *oldpass
        struct passwd *newpw;
```

DESCRIPTION

If *oldpass* is indeed the old user password, this routine replaces the password entry with *newpw*. It returns 0 if successful.

RPC INFO

program number:

YPPASSWDPROG

xdr routines:

```
xdr_ppasswd(xdrs, yp)
        XDR *xdrs;
        struct yppasswd *yp;
xdr_yppasswd(xdrs, pw)
        XDR *xdrs;
        struct passwd *pw;
```

procs:

YPPASSWDPROC_UPDATE

Takes *struct yppasswd* as argument, returns integer.
Same behavior as *yppasswd()* wrapper.
Uses UNIX authentication.

versions:

YPPASSWDVERS_ORIG

structures:

```
struct yppasswd {
        char *oldpass;    /* old (unencrypted) password */
        struct passwd newpw;      /* new pw structure */
};
```

SEE ALSO

yppasswd(1), yppasswdd(8C)