# Contents

## Contents

Chapter 1

# Interface Definition

## 1.1 OVERVIEW

### 1.1.1 Rationale

This part of the X/OPEN Guide contains the X/OPEN System V Specification (XVS). It defines the system interfaces offered to application programs and the run-time behaviour of those interfaces, without imposing any particular restrictions on the way in which the interfaces are implemented.

The interfaces are defined in terms of the source code interfaces for the C programming language. C is defined in Part III of this Guide. It is possible that some implementations may make the interfaces available to languages other than C, but this Guide does not currently define the source code interfaces for any other language.

This Specification allows an application to be built using a basic set of services that are consistent across all X/OPEN systems. Applications written in C using only these interfaces and avoiding machine dependent constructs will be portable to all X/OPEN systems.

The interfaces defined have been separated into two categories; "System Calls" and "Subroutines". This is in accordance with common practice, and should not be taken to imply that the implementation of these interfaces follows the same division.

### 1.1.2 Contents

In accordance with common practice, the definitions of the various interfaces have been separated into seven chapters. Chapters two to five inclusive, and chapter seven, define the application interfaces.

The chapters have the following contents:

- **Chapter 1** introduces Part II of the Guide and includes important notes and caveats relating to the rest of the XVS.
- **Chapter 2** (System Calls) defines interfaces which are conventionally implemented as entries to the system kernel.
- **Chapter 3** (Subroutines) defines interfaces which are conventionally implemented as subroutines.

- **Chapter 4** (File Formats) defines the formats of data files which are used by system calls and subroutines.
- **Chapter 5** (Header Files) defines the contents of header files which declare constants, macros and data structures that are needed by programs using the services provided by Chapters 2 and 3.
- **Chapter 6** is empty.
- **Chapter 7** (Special Files) describes the input/output devices always present on X/OPEN systems.

## 1.2 STATUS OF INTERFACES

### 1.2.1 Mandatory

The majority of the interfaces are mandatory; they must be present in all X/OPEN systems and they must conform to the published definition.

### 1.2.2 Optional

A small number of the interfaces are optional. The presence of these interfaces is not mandatory, although if they are present they must conform to the definition. The list below shows the optional interfaces in the form *name entry*(chapter), where *name* is the name of the interface and *entry* and *chapter* are the name and chapter number of the entry in which the interface is described.

| Optional Interfaces | |
|---|---|
| *acct* | *acct*(2)† |
| *brk* | *brk*(2) |
| *chroot* | *chroot*(2)† |
| *end* | *end*(3C) |
| *monitor* | *monitor*(3C) |
| *nice* | *nice*(2)† |
| *plock* | *plock*(2)† |
| *profil* | *profil*(2)† |
| *ptrace* | *ptrace*(2)† |
| *sbrk* | *brk*(2) |
| *termio* | *termio*(7) |
| *ulimit* | *ulimit*(2) |

The interfaces marked with a dagger (†) form part of the kernel extension set in the AT&T System V Interface Definition (see section 1.2.3).

The following interfaces conventionally form the standard mathematical library (3M). These routines are collectively optional, i.e. if one is present, the whole math library is present.

| Optional Math Interface | |
|---|---|
| *bessel*(3M) | *exp*(3M) |
| *floor*(3M) | *gamma*(3M) |
| *hypot*(3M) | *matherr*(3M) |
| *sinh*(3M) | *trig*(3M) |

Some of the mandatory interfaces are affected by the presence or absence of the services provided by the optional interfaces. The fundamental behaviour of these interfaces is not changed; the effects are identified in the relevant interface descriptions. These interfaces are given in the following table.

| Interfaces affected by options | |
|---|---|
| *exec*(2) | *exit*(2) |
| *fork*(2) | *read*(2) |

### 1.2.3 Relationship to SVID

The System V Interface Definition (SVID), published by AT&T, is intended for use as a standard by applications developers. The XVS is not a distinct standard but a definition of the System V interfaces supported by X/OPEN systems, based on the SVID (Issue 1 published in Spring 1985).

With the exception of the mathematical routines (3M), and *termio*(7) in some circumstances, (see section 1.10.2), the XVS contains as mandatory all of the SVID base interfaces.

All the interfaces within the SVID kernel extension set (K_EXT) except those relating to shared memory, semaphores and message passing, are included in the XVS as individually optional routines.

Additionally, the XVS includes a number of interfaces taken from System V Release 2.0 but not defined in the SVID.

Wherever an XVS definition differs from the corresponding one in the SVID, the differences are marked in the description. The rationale for such differences from the SVID is:

- Some of the SVID "FUTURE DIRECTIONS" have been included
- The use of symbolic constants to replace explicit constants has been increased
- Alternative wording has been used for clarification

The interfaces have been specified with two goals in mind, relative to the SVID:

- Applications written to the SVID should be portable to X/OPEN systems
- X/OPEN systems should pass the SVID verification tests

To enable these goals to be met, X/OPEN systems will ensure that where symbolic constants are used in place of explicit constants in the SVID, the symbolic constants will have the values of the SVID explicit constants.

The symbolic constants should be used wherever possible for two reasons:

- They improve readability of programs
- They protect programs from the problems which arise if the values of the explicit constants ever change

Programs written to the X/OPEN specification can easily be moved to systems that do not provide definitions for these symbolic constants. All that is required is the provision of a small number of header files, containing the necessary definitions.

### 1.2.4 Subject to Change

The SVID identifies certain interfaces as possibly subject to withdrawal, by referring to them as "level 2" interfaces. They will be present until at least January 1, 1988. The only ones currently in this class are the following:

| Interfaces subject to change | |
|---|---|
| *perror* | *errno* |
| *sys_errlist* | *sys_nerr* |

The X/OPEN commitment to support of these interfaces matches that of the SVID.

### 1.2.5 User Interface Services

Neither the SVID nor the X/OPEN Guide define any interfaces for the support of terminal-independent I/O, but application writers are referred to *curses*(3X) for a list of such interfaces which are in widespread use. The list given is the *minicurses* package (taken from UNIX System V, Release 2.0).

These routines are likely to be supported on most X/OPEN Systems, but their presence cannot be guaranteed.

X/OPEN views the definition of appropriate user interface services as a matter of urgency.

## 1.3 FORMAT OF ENTRIES

The entries in each chapter are based on a common format, not all of whose parts always appear.

The **NAME** part gives the name(s) of the entry and briefly states its purpose.

The **SYNOPSIS** part summarises the use of the entry being described. If it is necessary to include a header file to use this interface, the names of such files will be shown,
e.g. **#include <stdio.h>**.

The **DESCRIPTION** part discusses the subject at hand.

The **EXAMPLE(S)** part gives example(s) of usage, where appropriate.

The **FILES** part gives the file names that are built into the subject at hand.

The **ERRORS** part gives the symbolic names of the values returned in the global variable *errno* if an error occurs.

The **RETURN VALUE** part indicates the return value, if any.

The **SEE ALSO** part gives pointers to related information.

The **APPLICATION USAGE** part gives information about the way that the subject at hand should be used.

The **FUTURE DIRECTIONS** part is generally copied from the SVID, unless the change indicated in the SVID has already been adopted. Comments found in this section should be used as a guide to current thinking; there is not necessarily a commitment to implement all of these future directions in their entirety.

The **RELATIONSHIP TO SVID** part gives the differences, if any, between this definition and that in the SVID.

The following typographical conventions are used throughout this part:

**Boldface** strings are literals and are to be typed just as they appear.

*Italic* strings usually represent substitutable argument prototypes and the names of entries found elsewhere.

Names in upper case surrounded by braces, e.g. {CONST} represent constants which are declared in appropriate header files by means of the C **#define** facility. For portability, only the symbolic names should be used, never the value that a particular implementation may happen to use. The values of most of these constants are defined in <**limits.h**>, <**values.h**> or <**unistd.h**>.

Square brackets **[ ]** around an argument prototype indicate that the argument is optional. When an argument prototype is given as "name" or "file", it always refers to a *file* name.

The notation <**file.h**> indicates a header file, also known as an include file, which is supplied as part of the applications development system, see Chapter 5 and "FILE INCLUSION" in Part III.

Ellipses ... are used to show that the previous argument may be repeated.

Whenever referring to a subject described in chapters 2 - 7, its chapter number is appended to its name, in parentheses. For example: *access*(2).

## 1.4 DEFINITIONS

Many special terms are used in the interface definitions. The descriptions of these terms follows.

### 1.4.1 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 {PID_MAX}. Process IDs between 0 and {SYSPID_MAX} are reserved for special system processes.

### 1.4.2 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.

### 1.4.3 Process Group ID

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

### 1.4.4 Process Group Leader

A process group leader is any process whose process group ID is the same as its process ID. Any process may become a group leader by calling *setpgrp*(2). A process inherits the process group ID of the process that created it, see *fork*(2) and *exec*(2).

### 1.4.5 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 terminate a group of related processes upon termination of one of the processes in the group, see *exit*(2) and *signal*(2).

### 1.4.6 Real User ID and Real Group ID

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

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for

the creation of the process. They can be reset with *setuid*(2) and *setgid*(2), respectively.

### 1.4.7 Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group-ID bit set, see *exec*(2). In addition, they can be reset with *setuid*(2) and *setgid*(2), respectively.

### 1.4.8 Super-user

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

### 1.4.9 Special Processes

Special processes are system processes as, for example, a system's process scheduler. Process IDs between 0 and {SYSPID_MAX} are reserved for special system processes.

### 1.4.10 File Descriptor

A file descriptor is a small integer used to identify a file for the purpose of doing I/O. The value of a file descriptor is from 0 to {OPEN_MAX}-1. A process may have no more than {OPEN_MAX} file descriptors open simultaneously.

A file descriptor has associated with it information used in performing I/O on the file: a file pointer that marks the current position within the file where I/O will begin; file status and access modes (e.g. read, write, read/write), see *open*(2); and close-on-exec flag, see *fcntl*(2). Multiple file descriptors may identify the same file. A file descriptor is returned by system routines such as *creat*(2), *dup*(2), *fcntl*(2), *open*(2), or *pipe*(2). The file descriptor is used as an argument by routines such as *read*(2), *write*(2), *ioctl*(2) and *close*(2).

### 1.4.11 File Name

Names consisting of 1 to {NAME_MAX} characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding the characters "null" and "slash".

Note that it is generally unwise to use *, ?, !, [, or ] as part of file names because of the special meaning attached to these characters for filename expansion by some command interpreters, see *system*(3S). Other characters to avoid are the hyphen, blank, tab, <, >, backslash, single and double quotes, accent grave, vertical bar, carat, curly braces and parentheses. It is also advisable to avoid the use of non-printing characters in file names.

### 1.4.12 Character and Block Special Files

Character and block special files are used to refer to physical devices. Certain restrictions may apply to use of character and block special files which are implementation dependent.

### 1.4.13 FIFO Special Files

A FIFO special file is a named "pipe", see *pipe* (2) and *mknod* (2). Normally, a FIFO special file is opened in conjunction by two or more separate processes. One or more processes write data to the FIFO special file and another process reads this same data from the file on a "first-in-first-out" basis. Seeks on a FIFO special file have no meaning and cause the [ESPIPE] error.

### 1.4.14 Path Name and Path Prefix

In a C program a path name is a null-terminated character-string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name. The null string is undefined and may be considered an error.

More precisely, a path name 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 {NAME_MAX} characters other than slash and null, and <dirname> is a string of 1 to {NAME_MAX} characters (other than slash and null) that names a directory.

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

A slash by itself names the root directory. The meanings of . and .. are defined below under *Directory*.

The result of names not produced by the grammar is undefined.

### 1.4.15 Directory

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. Also, the parent directory of the root directory / is /.

### 1.4.16 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. The root directory of a process need not be the root directory of the root file system.

### 1.4.17 File Access Permissions

Read, write, and execute/search permissions on a file 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 the user ID of the owner of the file and the appropriate access bit of the "owner" portion (S_IRWXU) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion (S_IRWXG) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion (S_IRWXO) of the file mode is set.

Otherwise, the corresponding permissions are denied.

## 1.5 SIGNALS

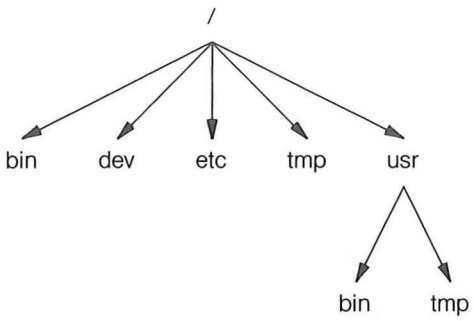
To be portable, applications should only catch or ignore the following signals:

| Signal | Description |
|---|---|
| **SIGHUP** | hangup |
| **SIGINT** | interrupt (rubout) |
| **SIGQUIT** | quit |
| **SIGILL** | illegal instruction (not reset when caught) |
| **SIGTRAP** | trace trap (not reset when caught) |
| **SIGABRT**† | process abort signal |
| **SIGFPE** | floating point exception |
| **SIGKILL** | kill (cannot be caught or ignored) |
| **SIGSYS** | bad argument to system call |
| **SIGPIPE** | write on a pipe with no one to read it |
| **SIGALRM** | alarm clock |
| **SIGTERM** | software termination signal from kill |
| **SIGUSR1** | user defined signal 1 |
| **SIGUSR2** | user defined signal 2 |

The signal marked above **SIGABRT**† has been included from a FUTURE DIRECTION indicated in the SVID.

## 1.6 DIRECTORY STRUCTURE

Below is a diagram of the minimal directory tree structure present on any system.




The following guidelines apply to the contents of these directories:

**/bin**,**/dev**,**/etc** and **/tmp** are primarily for the use of the system. Most applications should never create files in any of these directories, though they may read and execute them.

**/bin** contains executable system commands (utilities), although it may be empty.

**/dev** contains special files (I/O devices). Those which are always present in X/OPEN systems are defined in chapter 7.

**/etc** contains system data files such as **/etc/passwd**. It may also contain some executable files which are used by the system; these are not intended to be accessible to the ordinary user.

**/tmp** contains temporary files created by any utilities in **/bin**, and other system processes. Applications should use **/usr/tmp**.

**/usr/bin** and **/usr/tmp** can be used by applications as well as the system.

**/usr/bin** contains (user-level) executable application commands and system commands.

**/usr/tmp** contains temporary files created by application programs and by the system.

If the system is re-started after being halted for any reason, applications cannot rely on the contents of **/tmp** or **/usr/tmp** remaining undisturbed. It is common for both directories to be emptied by the restart procedures.

## 1.7 ENVIRONMENTAL VARIABLES

An array of strings is made available by *exec*(2) when a process begins execution, see also *system*(2). These strings are described more fully in *environ*(5), but the minimum strings that can be expected to exist and be set in any X/OPEN environment are:

| *Variable* | *Use* |
|---|---|
| HOME | Full pathname of the user's home directory, set when the user signs on. |
| PATH | A colon separated ordered list of pathnames that determine the search sequence used in locating files. |
| TERM | The kind of terminal for which output is prepared. |
| TZ | Time zone information. See also *ctime*(3C). |

## 1.8 SYSTEM RESIDENT DATA FILES

The only system-resident data files implied by the X/OPEN system definition are given in the following table, together with a reference to the appropriate chapter and entry to find their definitions.

| data file | reference |
|---|---|
| /etc/group | *group*(4) |
| /etc/passwd | *passwd*(4) |
| /etc/profile | *environ*(5) |
| /etc/utmp | *utmp*(4) |
| /etc/wtmp | *utmp*(4) |

## 1.9 SPECIAL FILES

The names of specific I/O devices are known as "special files". The only ones present in every X/OPEN system are given below, together with reference to their descriptions. The ones marked (†) are only present in systems supporting the source code transfer standard and need *not* be present in every system.

| device | reference |
|---|---|
| /dev/console | *console*(7) |
| /dev/null | *null*(7) |
| /dev/tty | *tty*(7) |
| /dev/sctfdm | *sct*(7)† |
| /dev/sctmtm | *sct*(7)† |

## 1.10 CAVEATS

### 1.10.1 Null Pointers

The descriptions of some functions refer to the NULL pointer. This is the value that is obtained by casting 0 into a pointer i.e., *(char *) 0*.

The C language guarantees that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

For consistency with the **C** language definition, this interpretation of the NULL pointer has been retained. However, reference should be made to the notes on **C** program portability in Part III of the Guide, where it is indicated that some systems do not support this definition of the NULL pointer.

(A NULL pointer should not be confused with the NULL character. The NULL character is a character with the value 0, represented in the C language as '\0'. A string, or NULL—terminated character array, is a sequence of characters, the last of which is the NULL character. A NULL string is an array of characters which contains only the NULL character.)

### 1.10.2 Termio(7)

The SVID interface for locally connected asynchronous lines, *termio*(7), is a mandatory part of the SVID base. Some X/OPEN systems may not support any asynchronous lines, or may only support them over networks. In both cases, it is impossible to support the full definition of *termio*(7). For these reasons X/OPEN cannot guarantee full support for *termio*(7) on all systems. This also slightly affects the functionality of *read*(2) and *open*(2), with respect to the O_NDELAY flag.

The *open*(2) description discusses what happens while waiting for a carrier to be detected on a communication line. It should be noted that even if full support is otherwise provided for *termio*(7), the hardware driving the line may not respond to the modem control signals. In this case, it will appear as if carrier is permanently present and there will be no delay when opening such a line.

### 1.10.3 Process IDs

The values of a Process ID are specified to range from zero to {PID_MAX}. On many systems, the value of {PID_MAX} is small enough to imply that variables of type **short** provide adequate precision to store such a value. This is not always a justifiable assumption, and application developers are warned that **int** variables should be used for this purpose.

The *ut_pid* field of the *utmp* structure (see *utmp*(5)) is explicitly declared to be of type **short**, in line with the SVID. On some systems its type may be different; programming practices which rely on the type of *ut_pid* should be avoided. In particular, if it is necessary to take the address of this structure member, the type of the resulting pointer will depend on the type of *ut_pid*.

### 1.10.4 The files <values.h> and <limits.h>

A number of limits and values which are of importance to system developers are system dependent. Their values are available in the include files <**limits.h**> and <**values.h**>, as symbolic constants. These constants are referred to in many places in the interface definitions; for example {SYSPID_MAX}. The file <**values.h**> is part of the SVID base. The file <**limits.h**>, part of the /usr/group standard, defines additional values and has also been included in the XVS.

In some cases, the same value is defined in both of these files, although the name used to refer to the value differs. In X/OPEN systems, both names are set to the same value.

Applications developers may choose to include either, neither or both of these files in a particular application, depending upon their needs. The interfaces defined in the following chapters can always be used without it being necessary to include either of these files.

## 1.11 ERRORS AND EXCEPTIONS

Most system calls and subroutines can result in exceptions, known as "error returns". An error condition is indicated by an otherwise impossible returned value. This is almost always -1; the individual descriptions specify the details.

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

A full list of error names is defined in *errno*(5). Only these symbolic names for error numbers should be used in programs, since the actual value of the error number may vary with the implementation. Certain implementations may not return all of the error types listed. Other implementations may return errors which are not included on the list.

The [EFAULT] error is caused by a program referencing data outside its legitimate address space. The reliable detection of this error cannot be guaranteed.

Functions in the Math Library (3M) may return the conventional values 0 or HUGE (the largest single-precision floating-point number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* is set to the value [EDOM] or [ERANGE].

## 1.12 LIST OF INTERFACES

A list of all interfaces specified follows. Some of the pages in the following chapters describe a number of interfaces, so to find a particular one knowing its name, look in the column labeled "Interface". The chapter describing the interface is given in the column on its right. For example, both *isascii* and *isspace* are to be found under the overall heading of *ctype*(3C).

Those entries marked with a dagger (†) are optional.

| Interface | Entry |
|---|---|
| abort | abort (3C) |
| abs | abs (3C) |
| access | access (2) |
| acct | acct (2) † |
| acct | acct (4) |
| acct | acct (5) |
| acos | trig (3M) † |
| alarm | alarm (2) |
| asctime | ctime (3C) |
| asin | trig (3M) † |
| assert | assert (3X) |
| assert | assert (5) |
| atan | trig (3M) † |
| atan2 | trig (3M) † |
| atof | strtod (3C) |
| atoi | strtol (3C) |
| atol | strtol (3C) |
| brk | brk (2) † |
| bsearch | bsearch (3C) |
| calloc | malloc (3X) |
| ceil | floor (3M) † |
| chdir | chdir (2) |
| chmod | chmod (2) |
| chown | chown (2) |
| chroot | chroot (2) † |
| clearerr | ferror (3S) |
| clock | clock (3C) |
| close | close (2) |
| console | console (7) |
| cos | trig (3M) † |
| cosh | sinh (3M) † |
| creat | creat (2) |
| crypt | crypt (3C) |
| ctermid | ctermid (3S) |
| ctime | ctime (3C) |

| Interface | Entry |
|---|---|
| ctype | ctype (5) |
| curses | curses (3X) † |
| cuserid | cuserid (3S) |
| daylight | ctime (3C) |
| drand48 | drand (3C) |
| dup | dup (2) |
| ecvt | ecvt (3C) |
| edata | end (3C) † |
| encrypt | crypt (3C) |
| end | end (3C) † |
| endgrent | getgrent (3C) |
| endpwent | getpwent (3C) |
| endutent | getut (3C) |
| erand48 | drand (3C) |
| erf | erf (3M) † |
| erfc | erf (3M) † |
| errno | errno (5) |
| errno | perror (3C) |
| etext | end (3C) † |
| execl | exec (2) |
| execle | exec (2) |
| execlp | exec (2) |
| execv | exec (2) |
| execve | exec (2) |
| execvp | exec (2) |
| exit | exit (2) |
| exp | exp (3M) † |
| fabs | floor (3M) † |
| fclose | fclose (3S) |
| fcntl | fcntl (2) |
| fcntl | fcntl (5) |
| fcvt | ecvt (3C) |
| fdopen | fopen (3S) |
| feof | ferror (3S) |
| ferror | ferror (3S) |

| Interface | Entry |
|---|---|
| fflush | fclose (3S) |
| fgetc | getc (3S) |
| fgets | gets (3S) |
| fileno | ferror (3S) |
| floor | floor (3M) † |
| fmod | floor (3M) † |
| fopen | fopen (3S) |
| fork | fork (2) |
| fprintf | printf (3S) |
| fputc | putc (3S) |
| fputs | puts (3S) |
| fread | fread (3S) |
| free | malloc (3X) |
| freopen | fopen (3S) |
| frexp | frexp (3C) |
| fscanf | scanf (3S) |
| fseek | fseek (3S) |
| fstat | stat (2) |
| ftell | fseek (3S) |
| ftw | ftw (3C) |
| ftw | ftw (5) |
| fwrite | fread (3S) |
| gamma | gamma (3M) † |
| gcvt | ecvt (3C) |
| getc | getc (3S) |
| getchar | getc (3S) |
| getcwd | getcwd (3C) |
| getegid | getuid (2) |
| getenv | getenv (3C) |
| geteuid | getuid (2) |
| getgid | getuid (2) |
| getgrent | getgrent (3C) |
| getgrgid | getgrent (3C) |
| getgrnam | getgrent (3C) |
| getlogin | getlogin (3C) |

| Interface | Entry |
|---|---|
| getopt | getopt (3C) |
| getpass | getpass (3C) |
| getpgrp | getpid (2) |
| getpid | getpid (2) |
| getppid | getpid (2) |
| getpw | getpw (3C) |
| getpwent | getpwent (3C) |
| getpwnam | getpwent (3C) |
| getpwuid | getpwent (3C) |
| gets | gets (3S) |
| getuid | getuid (2) |
| getutent | getut (3C) |
| getutid | getut (3C) |
| getutline | getut (3C) |
| getw | getc (3S) |
| gmtime | ctime (3C) |
| group | group (4) |
| grp | grp (5) |
| gsignal | ssignal (3C) |
| hcreate | hsearch (3C) |
| hdestroy | hsearch (3C) |
| HOME | environ (5) |
| hsearch | hsearch (3C) |
| hypot | hypot (3M) † |
| ioctl | ioctl (2) |
| isalnum | ctype (3C) |
| isalpha | ctype (3C) |
| isascii | ctype (3C) |
| isatty | ttyname (3C) |
| iscntrl | ctype (3C) |
| isdigit | ctype (3C) |
| isgraph | ctype (3C) |
| islower | ctype (3C) |
| isprint | ctype (3C) |
| ispunct | ctype (3C) |

| Interface | Entry |
|---|---|
| isspace | ctype (3C) |
| isupper | ctype (3C) |
| isxdigit | ctype (3C) |
| j0 | bessel (3M) † |
| j1 | bessel (3M) † |
| jn | bessel (3M) † |
| jrand48 | drand (3C) |
| kill | kill (2) |
| l3tol | ltol (3C) |
| lcong48 | drand (3C) |
| ldexp | frexp (3C) |
| lfind | lsearch (3C) |
| limits | limits (5) |
| link | link (2) |
| localtime | ctime (3C) |
| lock | lock (5) |
| lockf | lockf (3C) |
| log | exp (3M) † |
| log10 | exp (3M) † |
| logname | logname (3X) |
| longjmp | setjmp (3C) |
| lrand48 | drand (3C) |
| lsearch | lsearch (3C) |
| lseek | lseek (2) |
| ltol3 | ltol (3C) |
| mallinfo | malloc (3X) |
| malloc | malloc (3X) |
| malloc | malloc (5) |
| mallopt | malloc (3X) |
| math | math (5) |
| matherr | matherr (3M) † |
| memccpy | memory (3C) |
| memchr | memory (3C) |
| memcmp | memory (3C) |
| memcpy | memory (3C) |

| Interface | Entry |
|---|---|
| memory | memory (5) |
| memset | memory (3C) |
| mknod | mknod (2) |
| mktemp | mktemp (3C) |
| modf | frexp (3C) |
| mon | mon (5) |
| monitor | monitor (3C) † |
| mount | mount (2) |
| mrand48 | drand (3C) |
| nice | nice (2) † |
| nrand48 | drand (3C) |
| null | null (7) |
| open | open (2) |
| passwd | passwd (4) |
| PATH | environ (5) |
| pause | pause (2) |
| pclose | popen (3S) |
| perror | perror (3C) |
| pipe | pipe (2) |
| plock | plock (2) † |
| popen | popen (3S) |
| pow | exp (3M) † |
| printf | printf (3S) |
| profil | profil (2) † |
| ptrace | ptrace (2) † |
| putc | putc (3S) |
| putchar | putc (3S) |
| putenv | putenv (3C) |
| putpwent | putpwent (3C) |
| puts | puts (3S) |
| pututline | getut (3C) |
| putw | putc (3S) |
| pwd | pwd (5) |
| qsort | qsort (3C) |
| rand | rand (3C) |

| Interface | Entry |
|---|---|
| read | read (2) |
| realloc | malloc (3X) |
| regcmp | regcmp (3X) |
| regex | regcmp (3X) |
| rewind | fseek (3S) |
| sbrk | brk (2) † |
| scanf | scanf (3S) |
| sctfd | sct (7) † |
| sctmt | sct (7) † |
| search | search (5) |
| seed48 | drand (3C) |
| setbuf | setbuf (3C) |
| setgid | setuid (2) |
| setgrent | getgrent (3C) |
| setjmp | setjmp (3C) |
| setjmp | setjmp (5) |
| setkey | crypt (3C) |
| setpgrp | setpgrp (2) |
| setpwent | getpwent (3C) |
| setuid | setuid (2) |
| setutent | getut (3C) |
| setvbuf | setbuf (3C) |
| signal | signal (2) |
| signal | signal (5) |
| signgam | gamma (3M) † |
| sin | trig (3M) † |
| sinh | sinh (3M) † |
| sleep | sleep (3C) |
| sprintf | printf (3S) |
| sqrt | exp (3M) † |
| srand | rand (3C) |
| srand48 | drand (3C) |
| sscanf | scanf (3S) |
| ssignal | ssignal (3C) |
| stat | stat (2) |

| Interface | Entry |
|---|---|
| stat | stat (5) |
| stdio | stdio (3S) |
| stdio | stdio (5) |
| stime | stime (2) |
| strcat | string (3C) |
| strchr | string (3C) |
| strcmp | string (3C) |
| strcpy | string (3C) |
| strcspn | string (3C) |
| string | string (5) |
| strlen | string (3C) |
| strncat | string (3C) |
| strncmp | string (3C) |
| strncpy | string (3C) |
| strpbrk | string (3C) |
| strrchr | string (3C) |
| strspn | string (3C) |
| strtod | strtod (3C) |
| strtok | string (3C) |
| strtol | strtol (3C) |
| swab | swab (3C) |
| sync | sync (2) |
| system | system (3S) |
| sys_errlist | perror (3C) |
| sys_nerr | perror (3C) |
| tan | trig (3M) † |
| tanh | sinh (3M) † |
| tdelete | tsearch (3C) |
| tempnam | tmpnam (3S) |
| TERM | environ (5) |
| termio | termio (5) |
| termio | termio (7) † |
| tfind | tsearch (3C) |
| time | time (2) |
| time | time (5) |

| Interface | Entry |
|---|---|
| times | times (2) |
| times | times (5) |
| timezone | ctime (3C) |
| tmpfile | tmpfile (3S) |
| tmpnam | tmpnam (3S) |
| toascii | conv (3C) |
| tolower | conv (3C) |
| toupper | conv (3C) |
| tsearch | tsearch (3C) |
| tty | tty (7) |
| ttyname | ttyname (3C) |
| ttyslot | ttyslot (3C) |
| twalk | tsearch (3C) |
| types | types (5) |
| TZ | environ (5) |
| tzname | ctime (3C) |
| tzset | ctime (3C) |
| ulimit | ulimit (2) |
| umask | umask (2) |
| umount | umount (2) |
| uname | uname (2) |
| ungetc | ungetc (3S) |
| unistd | unistd (5) |
| unlink | unlink (2) |
| ustat | ustat (2) |
| ustat | ustat (5) |
| utime | utime (2) |
| utmp | utmp (4) |
| utmp | utmp (5) |
| utmpname | getut (3C) |
| utsname | utsname (5) |
| values | values (5) |
| varargs | varargs (5) |
| vfprintf | vprintf (3S) |
| vprintf | vprintf (3S) |

| Interface | Entry |
|---|---|
| vsprintf | vprintf (3S) |
| wait | wait (2) |
| write | write (2) |
| wtmp | utmp (4) |
| y0 | bessel (3M) † |
| y1 | bessel (3M) † |
| yn | bessel (3M) † |
| _tolower | conv (3C) |
| _toupper | conv (3C) |
| _exit | exit (2) |

*Chapter 2*

# System Calls

This chapter describes system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always -1; the individual descriptions specify the details.

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

The error names are described in *errno* (5).

## NAME

access — determine accessibility of a file

## SYNOPSIS

```
#include <unistd.h>

int access (path, amode)
char *path;
int amode;
```

## DESCRIPTION

*Path* points to a path name naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID or equivalent in place of the effective group ID. The value of *amode* is the sum of the access modes to be checked as defined in <**unistd.h**>:

| | | |
|---|---|---|
| R_OK | 04 | read |
| W_OK | 02 | write |
| X_OK | 01 | execute (search) |
| F_OK | 00 | check existence of file |

Thus, the value of *amode* should be the sum of the values of the access modes to be checked.

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 permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

## ERRORS

*Access* fails if one or more of the following are true:

| | |
|---|---|
| [ENOTDIR] | A component of the *path* prefix is not a directory. |
| [ENOENT] | The named file does not exist. |
| [EACCES] | Search permission is denied on a component of the *path* prefix. |
| [EROFS] | Write access is requested for a file on a read-only file system. |
| [ETXTBSY] | Write access is requested for a pure procedure (shared text) file that is being executed. |
| [EACCES] | Permission bits of the file mode do not permit the |

requested access.

[EFAULT] *Path* points outside the allocated address space for the process. The reliable detection of this condition will be implementation dependent.

[EINVAL] The value of the *amode* argument is invalid.

## RETURN VALUE

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of —1 is returned and *errno* is set to indicate the error.

## SEE ALSO

chmod(2), stat(2), unistd(5), types(5).

## APPLICATIONS USAGE

The [EINVAL] error is taken from a SVID future direction. It may not be included in all implementations at present.

## RELATIONSHIP TO SVID

SVID does not use symbolic names for *amode*. It does not, therefore, call for the inclusion of the header file <**unistd.h**>, and the description does not refer to this header file.

This change is forecast as a future direction in the SVID.

## NAME

acct — enable or disable process accounting (OPTIONAL)

## SYNOPSIS

```
int acct (path)
char *path;
```

## DESCRIPTION

*Acct* is used to enable or disable the system process accounting routine. If the routine 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 *signal*(2). The effective user ID of the calling process must be superuser to use this call.

*Path* points to a path name naming the accounting file. The format of an accounting file produced as a result of calling *acct*(2) has records in the format defined by the structure *acct* in <**sys/acct.h**>.

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

## ERRORS

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

| | |
|---|---|
| [EPERM] | The effective user of the calling process is not superuser. |
| [EBUSY] | An attempt is being made to enable accounting when it is already enabled. |
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENOENT] | One or more components of the accounting file path name do not exist. |
| [EACCES] | The file named by *path* is not an ordinary file. |
| [EROFS] | The named file resides on a read-only file system. |
| [EFAULT] | *Path* 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

exit(2), signal(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

alarm — set a process alarm clock

## SYNOPSIS

```
unsigned alarm (sec)
unsigned sec;
```

## DESCRIPTION

*Alarm* instructs the alarm clock of the calling process to send the signal SIGALRM to the calling process after the number of real time seconds specified by *sec* have elapsed, see *signal*(2).

*Alarm* requests are not stacked; successive calls reset the alarm clock of the calling process.

If *sec* is 0, any previously made alarm request is canceled.

*Fork*(2) sets the alarm clock of a new process to 0. A process created by *exec*(2) inherits the time left on the old process's alarm clock.

## RETURN VALUE

*Alarm* returns the amount of time previously remaining in the alarm clock of the calling process.

## SEE ALSO

pause(2), signal(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

brk, sbrk — change data segment space allocation (OPTIONAL)

## SYNOPSIS

```
int brk (endds)
char *endds;

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

## DESCRIPTION

*Brk* and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment, see *exec*(2). The change is made by resetting the process's break value and allocating the appropriate amount of space.

*Brk* sets the system's idea of the lowest data segment location not used by the program (called the "break") to *endds* (rounded to the convenient hardware addressing size). The amount of allocated space increases as the break value increases.

*Sbrk* adds *incr* bytes to the break value and changes the allocated space accordingly.

When a program begins execution via *exec*(2) 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*.

*Brk* sets the break value to *endds* and changes the allocated space accordingly.

*Sbrk* adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased. If *sbrk* is initially called with an *incr* of 0, then the value returned is the base of the existing data segment allocation.

When obtained, the data contents of the allocated region are undefined.

## ERRORS

*Brk* and *sbrk* will fail without making any change in the allocated space if the following is true:

[ENOMEM] Such a change would result in more space being allocated than is allowed by a system-imposed maximum, see *ulimit*(2).

## RETURN VALUE

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, *brk* returns a value of -1 and *sbrk* returns a value of *(char *) -1*, and *errno* is set to indicate the error.

## SEE ALSO

exec(2), ulimit(2), malloc(3X).

## APPLICATION USAGE

*Brk* may be called with any value in the range of memory addresses which have been returned by *sbrk*. The only real use for *brk* is to free a large amount of memory allocated by *sbrk*. If *brk* is used to allocate or free memory outside of this range, such usage may have undesirable effects. If an area is freed and subsequently reallocated, the contents of the area are not necessarily preserved.

*Malloc*(3X) is the recommended way to obtain additional working space. However, programs which use *malloc*(3X) or *stdio*(3S) should not make use of either *brk* or *sbrk*.

## RELATIONSHIP TO SVID

This *optional* function is not included in the SVID. It is taken from UNIX System V Release 2.0.

## NAME

chdir — change working directory

## SYNOPSIS

```
int chdir (path)
char *path;
```

## DESCRIPTION

*Path* points to the path name of a directory. *Chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

## 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 name is not a directory. |
| [ENOENT] | The named directory does not exist. |
| [EACCES] | Search permission is denied for any component of the path name. |
| [EFAULT] | *Path* points outside the allocated address space of the process. The reliable detection of this condition will be implementation dependent. |

## 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.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

chmod — change mode of file

## SYNOPSIS

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

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

## DESCRIPTION

*Path* points to a path name naming a file. *Chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are described in <sys/stat.h>, and are interpreted as follows:

| | | |
|---|---|---|
| S_ISUID† | 04000 | Set user ID on execution |
| S_ISGID | 02000 | Set group ID on execution |
| | 01000 | Reserved |
| S_IRUSR | 00400 | Read by owner |
| S_IWUSR | 00200 | Write by owner |
| S_IXUSR | 00100 | Execute (search if a directory) by owner |
| S_IRGRP | 00040 | Read by group |
| S_IWGRP | 00020 | Write by group |
| S_IXGRP | 00010 | Execute (search) by group |
| S_IROTH | 00004 | Read by others (ie., anyone else) |
| S_IWOTH | 00002 | Write by others |
| S_IXOTH | 00001 | Execute (search) by others |

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.

For security reasons, if *chmod* is invoked by other than the superuser, the set-user-ID and set-group-ID bits of the file mode, S_ISUID and S_ISGID respectively, will be cleared.

## ERRORS

*Chmod* will fail and the file mode will be unchanged if one or more of the following are true:

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

[ENOENT] The named file does not exist.

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

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

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

[EFAULT] *Path* points outside the allocated address space of the process. The reliable detection of this condition will be implementation dependent.

## 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

chown(2), mknod(2), open(2), stat(5), types(5).

## FUTURE DIRECTIONS

Mandatory or enforcement-mode file and record locking will be added.

## RELATIONSHIP TO SVID

The use of symbolic names for *mode* has been introduced, which will define the values of access modes. As SVID does not use symbolic names, it does not call for the inclusion of the header files <**sys/stat.h**> and <**sys/types.h**>, and the description does not refer to the <**sys/stat.h**> header file. (NB. This change is forecast in the SVID Future Directions Section).

The *mode* value 01000 is defined in the SVID as "Save text image after execution" rather than being "reserved".

## NAME

chown — change owner and group of a file

## SYNOPSIS

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

## DESCRIPTION

*Path* points to a path name naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or superuser may change the ownership of a file.

For security reasons, if *chown* is invoked by other than the superuser, the set-user-ID and set-group-ID bits of the file mode, S_ISUID and S_ISGID respectively, will be cleared.

## ERRORS

*Chown* will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

| | |
|---|---|
| [ENOTDIR] | A component of the *path* prefix is not a directory. |
| [ENOENT] | The named file does not exist. |
| [EACCES] | Search permission is denied on a component of the *path* prefix. |
| [EPERM] | The effective user ID does not match the owner of the file and the effective user ID is not super-user. |
| [EROFS] | The named file resides on a read-only file system. |
| [EFAULT] | *Path* points outside the allocated address space of the process. The reliable detection of this condition will be implementation dependent. |

## 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

chmod(2), stat(5).

## FUTURE DIRECTIONS

Mandatory or enforcement-mode file and record locking will be added.

## RELATIONSHIP TO SVID

In the third paragraph of the DESCRIPTION section, SVID does not use the symbolic values S_ISUID and S_ISGID , but instead refers to the

specific bit values. (NB. This change is forecast in the SVID Future Directions section.)

The final paragaph of the Description section has been reworded for clarity.

## NAME

chroot — change root directory (OPTIONAL)

## SYNOPSIS

```
int chroot (path)
char *path;
```

## DESCRIPTION

*Path* points to a path name naming a directory. *Chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The user's working directory is unaffected by the *chroot* system call.

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.

## ERRORS

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

| | |
|---|---|
| [ENOTDIR] | Any component of the *path* name is not a directory. |
| [ENOENT] | The named directory does not exist. |
| [EPERM] | The effective user ID is not super-user. |
| [EFAULT] | *Path* points outside the allocated address space of the process. |

## RETURN VALUE

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

## SEE ALSO

chdir(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

This *optional* function is included in the *kernel extension* set (K_EXT) in the SVID.

## NAME

close — close a file descriptor

## SYNOPSIS

```
int close (fildes)
int fildes;
```

## DESCRIPTION

*Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Close* closes the file descriptor indicated by *fildes*. All outstanding record locks on the file indicated by *filedes* that are owned by the calling process are removed.

## ERRORS

[EBADF] *Close* will return this error if *fildes* is not a valid open file descriptor.

## 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

creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2).

## APPLICATIONS USAGE

Normally, applications should only use the *stdio* routines to open, close, read, and write files. Thus, an application that had used the *stdio* routine *fopen*(3S) to open a file would use the corresponding *fclose*(3S) routine rather than *close*.

## RELATIONSHIP TO SVID

Identical to the SVID, except that the SVID reads: "Close will fail if..." in the description of [EBADF].

## NAME

creat — create a new file or rewrite an existing one

## SYNOPSIS

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

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

## DESCRIPTION

*Creat* creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID of the process; the group ID of the file is set to the effective group ID of the process; and the access permission bits (see *chmod*(2)) of the file mode are set to the value of *mode* modified as follows:

> The corresponding bits are ANDed with the process's file mode creation mask, see *umask*(2). Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared.

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 *exec* system calls, see *fcntl*(2). No process may have more than {OPEN_MAX} files open simultaneously. A new file may be created with a mode that forbids writing.

## ERRORS

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

| | |
|---|---|
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENOENT] | A component of the path name which must exist does not exist. |
| [EACCES] | Search permission is denied on a component of the path prefix. |
| [EACCES] | The file does not exist and the directory in which the file is to be created does not permit writing. |
| [EROFS] | The named file resides or would reside on a read-only file system. |

| | |
|---|---|
| [ETXTBSY] | The file is a pure procedure (shared text) file that is being executed. |
| [EACCES] | The file exists and write permission is denied. |
| [EISDIR] | The named file is an existing directory. |
| [EMFILE] | {OPEN_MAX} file descriptors are currently open in the calling process. |
| [ENOSPC] | The directory to contain the file cannot be extended. |
| [ENFILE] | The system file table is full. |
| [EFAULT] | *Path* points outside the allocated address space of the process. |

## RETURN VALUE

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

## SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2), umask(2), write(2), stat(5), types(5).

## APPLICATIONS USAGE

Normally, applications should use the *stdio* routines to open, close, read and write files. In this case, *fopen*(3S) should be used rather than *creat*(2).

*Creat* is now obsoleted by *open*(2) with O_CREAT set.

## FUTURE DIRECTIONS

Mandatory and enforcement mode file and record locking features will be added.

## RELATIONSHIP TO SVID

SVID does not use symbolic names for the access permissions specified in the *mode* parameter. It therefore also does not call for the inclusion of <**sys/stat.h**> and <**sys/types.h**> and the description does not refer to the <**sys/stat.h**> header file. See *chmod*(2).

## NAME

dup — duplicate an open file descriptor

## SYNOPSIS

```
int dup (fildes)
int fildes;
```

## DESCRIPTION

*Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer).

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

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

The file descriptor returned is the lowest one available.

## ERRORS

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

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

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

## RETURN VALUE

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

## SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

execl, execv, execle, execve, execlp, execvp — execute a file

## SYNOPSIS

```
int execl (path, arg0, arg1, ..., argn, (char *)0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, (char *)0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp)
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];
```

## DESCRIPTION

*Exec* in all its forms transforms the current process into a new process. The new process is constructed from an ordinary, executable file called the *new process file.* This file consists of a header, a text segment, and a data segment. There can be no return from a successful *exec* because the calling process image is overlaid by the new process image.

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, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to null-terminated strings that constitute the environment for the new process. *Argc* is conventionally at least one (1) and the initial member of the array points to a string containing the name of the file.

*Path* points to a path name that identifies the new process file. *File* points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH=", see *environ*(5) and *system*(3S).

*Arg0*, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* is present and points to a string that is the same as *path* (or its last component).

*Argv* is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv[0]* points to a string that is the same as *path* (or its last component). *Argv* is terminated by a NULL pointer.

*Envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer. For *execl* and *excev,* the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

```
extern char **environ;
```

and it is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to the default action (SIG_DFL) in the calling process will be set to the default action in the new process. Signals set to be ignored (SIG_IGN) by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to the default action in the new process, see *signal*(2).

If the set-user-ID-on-execution mode bit of the new process file is set, see *chmod*(2), *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process. The effective user ID and group ID of the new process are saved for use by *setuid*(2).

If the set-user-ID (set-group-ID) mode bit of the new process file is not set, then the new process inherits the calling process's real and effective user (group) ID.

Profiling is disabled for the new process, see *profil*(2). Profiling is an *optional* service.

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

nice value (see *nice*(2)) *nice* is an *optional* service.
process ID
parent process ID
process group ID

tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0; tracing is an *optional* service)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
*utime*, *stime*, *cutime*, and *cstime* (see *times*(2))

## ERRORS

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

| | |
|---|---|
| [ENOENT] | One or more components of the path name of the new process file do not exist. |
| [ENOTDIR] | A component of the new process file's path prefix is not a directory.† |
| [EACCES] | Search permission is denied for a directory listed in the new process file's path prefix. |
| [EACCES] | The new process file is not an ordinary file, see *mknod*(2) |
| [EACCES] | The new process file mode denies execution permission. |
| [ENOEXEC] | The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but is not a valid executable object. |
| [ETXTBSY] | The new process file is a pure procedure (shared text) file that is currently open for writing by some process. |
| [ENOMEM] | The new process requires more memory than is allowed by the hardware or a system-imposed maximum. |
| [E2BIG] | The number of bytes in the new process argument list is greater than the system-imposed limit of {ARG_MAX} bytes. |
| [EFAULT] | The new process file image is corrupted. |
| [EFAULT] | *Path* points to an illegal address or *argv* or *envp* point to an illegal address, directly or indirectly. |

## RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value will be -1 and *errno* will be set to indicate the error.

## SEE ALSO

alarm(2), exit(2), fork(2), nice(2), profil(2), ptrace(2), signal(2), times(2), ulimit(2), umask(2).

## APPLICATIONS USAGE

If possible, applications should use *system*(3S), which is easier to use and supplies more functions, rather than *fork*(2) and *exec*(2).

## RELATIONSHIP TO SVID

The definition is identical to the SVID entry.

For convenience of the user, the relationships between *exec*(2) and the optional facilities — *profil*(2), *nice*(2), *ptrace*(2) — have been given on these sheets. In the SVID these options are all included in the *kernel extension* set (K_EXT), and their effect on *exec* is given in a separate section, "Effect on BASE Operating System Services".

†The wording of the [ENOTDIR] entry in the ERRORS section in the SVID reads "A component of the new process path of the file prefix is not a directory".

## NAME

exit, _exit — terminate process

## SYNOPSIS

```
void exit (status)
int status;

void _exit (status)
int status;
```

## DESCRIPTION

*Exit* terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*(2), it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it; see *wait*(2). If the parent is not waiting, the child's status will be made available to it when the parent subsequently executes *wait*(2).

If the parent process of the calling process is not executing a *wait*(2), the calling process is transformed into a *zombie process*. A *zombie process* is an inactive process and it will be deleted at some later time when its parent process executes *wait*(2).

The parent process ID of all of the calling process's existing child processes and zombie processes is set to the process ID of a special system process. That is, these processes are inherited by a special system process.

If the process has a process, text or data lock, an *unlock* is performed; see *plock*(2). Process locking is an *optional* service.

An accounting record is written on the accounting file if the system's accounting routine is enabled; see *acct*(2). Accounting is an *optional* service.

If the process is a process group leader, the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

The function *exit* may cause cleanup actions, see *fclose*(3S) before the process exits. The function *_exit* circumvents all cleanup.

## RETURN VALUE

These routines do not return a value.

SEE ALSO

acct(2), plock(2), signal(2), wait(2).

APPLICATION USAGE

Normally applications should use *exit* rather than *_exit*. Not only do these routines not return a value, they do not return at all.

RELATIONSHIP TO SVID

For the convenience of the user, the interactions between *exit* and the *optional* facilities, *plock*(2) and *acct*(2), have been identified in the DESCRIPTION section. In the SVID these *optional* facilities are included in the *kernel extension* set (K_EXT), and their effect on *exit* is given in a separate section, "Effect on BASE Operating System Services".

## NAME

fcntl — file control

## SYNOPSIS

```
#include <unistd.h>
#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd;
```

## DESCRIPTION

*Fcntl* provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Arg* value and type are specific to the type of command.

The *cmd* values available are:

F_DUPFD Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

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

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

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

F_GETFD Get the close-on-exec flag associated with the file descriptor *fildes*. 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 *fildes* to the low-order bit of *arg* (0 or 1 as above).

F_GETFL Get file status flags: O_RDONLY, O_WRONLY, O_RDWR, O_NDELAY, O_APPEND.

F_SETFL Set file status flags to *arg*. Only certain flags can be set, see *fcntl*(5).

F_GETLK Get the first lock which blocks the lock description given by the variable of type *struct flock* pointed to by *arg* (see below). The information retrieved overwrites the information passed to *fcntl* 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 a file segment lock according to the variable of type *struct flock* pointed to by *arg* (see below). The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set, *fcntl* will return immediately with an error value of —1.

F_SETLKW This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

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

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

The structure *flock* describes the type *(l_type),* starting offset *(l_whence),* relative offset *(l_start),* size *(l_len),* process ID *(l_pid)* and system ID *(l_sysid)* of the segment of the file to be affected.

*l_whence* is †SEEK_SET, SEEK_CUR, SEEK_END, to indicate that the relative offset will be measured from the start of the file, current position or end of the file, respectively.

The process ID and system ID fields are only used with the F_GETLK *cmd* to return the value for a blocking lock. Locks may start and extend beyond the current end of a 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_start* set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment from the middle of a larger locked segment leaves two smaller segments for 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 effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

## ERRORS

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

| | |
|---|---|
| [EBADF] | *Fildes* is not a valid open file descriptor. |
| [EMFILE] | *Cmd* is F_DUPFD and {OPEN_MAX} file descriptors are currently open in the calling process. |
| [EINVAL] | *Cmd* is F_DUPFD and *arg* is negative or greater than or equal to {OPEN_MAX}. |
| [EINVAL] | *Cmd* is F_GETLK, F_SETLK, or F_SETLKW and *arg* or the data it points to is not valid. |
| [EAGAIN] | *Cmd* is F_SETLK the type of lock *(l_type)* is a read (F_RDLCK) or write (F_WRLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write lock and the segment of a file to be locked is already read or write locked by another process. |
| [ENOLCK] | *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock and there are no more file locks available (too many segments are locked). |
| [EDEADLK] | *Cmd* is F_SETLK, the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free, would cause a deadlock. |

## RETURN VALUE

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

| | |
|---|---|
| F_DUPFD | A new file descriptor |
| F_GETFD | Value of flag (only the low-order bit is defined) |
| F_SETFD | Value other than —1 |
| F_GETFL | Value of file flags |
| F_SETFL | Value other than —1 |
| F_GETLK | Value other than —1 |
| F_SETLK | Value other than —1 |
| F_SETLKW | Value other than —1 |

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

## SEE ALSO

close(2), exec(2), open(2), lockf(3C), fcntl(5).

## FUTURE DIRECTIONS

Mandatory or enforcement-mode file and record locking will be added. When mandatory file/record locking is set on a file, see *chmod*(2), future *read* and *write* system calls on the file will be affected by the record locks in effect.

## RELATIONSHIP TO SVID

†The SVID uses the absolute values 0, 1 and 2 for *l_whence* instead of the symbolic values SEEK_SET, SEEK_CUR and SEEK_END. These names come from the <**unistd.h**> file described in Appendix BASE: 1.6, Comparison to the 1984 /usr/group Standard.

In the SVID, the third sentence of DESCRIPTION reads: "*Arg* is specific to the type of command".

In the SVID, the second paragraph of DESCRIPTION reads: "The *commands* available are:".

In the SVID the structure of *flock* is given in the middle of the Description of *fcntl* as well as in <**fcntl.h**>.

## NAME

fork — create a new process

## SYNOPSIS

**int fork ()**

## DESCRIPTION

*Fork* causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

environment
close-on-exec flag (see *exec*(2))
signal handling settings (i.e., SIG_DFL, SIG_IGN, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status (see *profil*(2); profiling is an *optional* service)
nice value (see *nice*(2) — *nice* is an *optional* service)
process group ID
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0; ptrace is an *optional* service)
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

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

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

Process locks, text locks and data locks are not inherited by the child (see *plock*(2); process locking is an *optional* service).

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

## 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 system-wide {PROC_MAX} or by a single user ID {CHILD_MAX} would be exceeded.

[ENOMEM] The process requires more space than the system is able to supply.

## 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 *errno* is set to indicate the error.

## SEE ALSO

exec(2), nice(2), plock(2), ptrace(2), signal(2), times(2), ulimit(2), umask(2), wait(2).

## APPLICATION USAGE

If possible, applications should use *system*(3S), which is easier to use and supplies more functions, rather than *fork*(2) and *exec*(2).

## RELATIONSHIP TO SVID

For user convenience, the interactions between *fork* and the *optional* facilities of *profil*(2), *ptrace*(2), *plock*(2) and *nice*(2) have been included in the DESCRIPTION section. In the SVID these *optional* facilities are included in the *kernel extension* set (K_EXT), and their effect on *fork* is given in a separate section "Effect on BASE Operating System Services".

## NAME

getpid, getpgrp, getppid — get process, process group, and parent process IDs

## SYNOPSIS

```
int getpid ()
int getpgrp ()
int getppid ()
```

## DESCRIPTION

*Getpid* returns the process ID of the calling process.

*Getpgrp* returns the process group ID of the calling process.

*Getppid* returns the parent process ID of the calling process.

## SEE ALSO

exec(2), fork(2), setpgrp(2), signal(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

getuid, geteuid, getgid, getegid — get real user, effective user, real group, and effective group IDs

## SYNOPSIS

**unsigned short getuid ()**

**unsigned short geteuid ()**

**unsigned short getgid ()**

**unsigned short getegid ()**

## DESCRIPTION

*Getuid* returns the real user ID of the calling process.

*Geteuid* returns the effective user ID of the calling process.

*Getgid* returns the real group ID of the calling process.

*Getegid* returns the effective group ID of the calling process.

## SEE ALSO

setuid(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

ioctl — control device

## SYNOPSIS

**int ioctl (fildes, request, arg)**
**int fildes, request;**

## DESCRIPTION

*Ioctl* performs a variety of functions on devices, typically character special files. *Fildes* is an open file descriptor. *Request* selects the function to be performed and will depend on the device being addressed. *Arg* value and type are specific to the device and request.

## ERRORS

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

| | |
|---|---|
| [EBADF] | *Fildes* is not a valid open file descriptor. |
| [ENOTTY] | *Fildes* is not associated with a device that accepts control functions. |
| [EINVAL] | *Request* or *arg* is not valid. |
| [EINTR] | A signal was caught during the *ioctl* operation. |

## RETURN VALUE

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

## RELATIONSHIP TO SVID

In the SVID, the last sentence of the DESCRIPTION section reads: ''*Arg* also is specific to the device and request'', and the error [ENOTTY] reads: ''*fildes* is not associated with a character special device''.

## NAME

kill — send a signal to a process or a group of processes

## SYNOPSIS

```
#include <signal.h>
int kill (pid, sig)
int pid;
int sig;
```

## DESCRIPTION

*Kill* sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is super-user.

The processes with a process ID less than or equal to {SYSPID_MAX} are *special processes*.

If *pid* is greater than 0, *sig* will be sent to the process whose process ID is equal to *pid*. †If a signal is sent to a *special process*, the effect is implementation defined.

If *pid* is 0, *sig* will be sent to all processes, excluding the *special processes*, whose process group ID is equal to the process group ID of the sender.

If *pid* is —1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes, excluding the *special processes*, whose real user ID is equal to the effective user ID of the sender.

If *pid* is —1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding the *special processes*.

If *pid* is negative but not —1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

## ERRORS

*Kill* will fail and no signal will be sent if one or more of the following are true:

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

[EPERM] *Sig* is **SIGKILL** and *pid* is less than {SYSPID_MAX}.

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

[EPERM] The user ID of the sending process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process.

## RETURN VALUE

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

## SEE ALSO

getpid(2), setpgrp(2), signal(2), signal(5).

## RELATIONSHIP TO SVID

The sentence "The processes with a process ID less than {SYSPID_MAX} are *special processes*" is additional to the SVID wording. The SVID treats {SYSPID_MAX} as 1.

†The second sentence of the paragraph starting "If *pid* is greater than 0..." is additional to the SVID.

The introduction of the error [EPERM] is forecast in the Future Directions section in the SVID.

## NAME

link — link to a file

## SYNOPSIS

```
int link (path1, path2)
char *path1, *path2;
```

## DESCRIPTION

*Path1* points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

## ERRORS

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

| | |
|---|---|
| [ENOTDIR] | A component of either path prefix is not a directory. |
| [ENOENT] | A component of either path name which must exist does not exist. |
| [EACCES] | A component of either path prefix denies search permission. |
| [EEXIST] | The link named by *path2* exists. |
| [EPERM] | The file named by *path1* is a directory and the effective user ID is not super-user. |
| [EXDEV] | The link named by *path2* and the file named by *path1* are on different logical devices (file systems) and the implementation does not permit cross device links. |
| [EACCES] | The requested link requires writing in a directory with a mode that denies write permission. |
| [EROFS] | The requested link requires writing in a directory on a read-only file system. |
| [EMLINK] | The maximum number of links to a file would be exceeded. |
| [EFAULT] | *Path* points outside the allocated address space of the process. |
| [ENOSPC] | The directory to contain the file cannot be extended. |

## 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

unlink(2).

RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

lseek — move read/write file pointer

## SYNOPSIS

```
#include <unistd.h>

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;
```

## DESCRIPTION

*Fildes* is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *Lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is SEEK_SET (0), the pointer is set to *offset* bytes.

If *whence* is SEEK_CUR (1), the pointer is set to its current location plus *offset*.

If *whence* is SEEK_END (2), the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned.

## ERRORS

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

| | |
|---|---|
| [EBADF] | *Fildes* is not an open file descriptor. |
| [ESPIPE] | *Fildes* is associated with a pipe or fifo. |
| [EINVAL] | and SIGSYS signal. *Whence* is not one of the valid numbers. |
| [EINVAL] | The resulting file pointer would be negative. |

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

## RETURN VALUE

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

## SEE ALSO

creat(2), dup(2), fcntl(2), open(2), types(5), unistd(5).

## APPLICATION USAGE

Normally, applications should use the *stdio* library routines to open, close, read, write and manipulate files. Thus, an application that had used the *stdio* routine *fopen*(3S) to open a file would use *fseek*(3S)

rather than *lseek*(2).

## RELATIONSHIP TO SVID

The SVID uses absolute values rather than the symbolic values SEEK_SET, SEEK_CUR, SEEK_END for *whence*. It therefore also does not call for the inclusion of <**unistd.h**>. This change is forecast in the SVID Future Directions section, in the *lseek*(OS) entry. It also causes a minor wording change for the error "[EINVAL] and **SIGSYS** signal".

The error "[EINVAL]: The resulting file pointer would be negative" is additional wording to the SVID.

## NAME

mknod — make a directory, or a special or ordinary file

## SYNOPSIS

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

int 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 is initialized from *mode*. Where the value of *mode* is interpreted as follows:

file type; one of the following:

| | | |
|---|---|---|
| S_IFIFO† | 0010000 | FIFO special |
| S_IFCHR | 0020000 | character special |
| S_IFDIR | 0040000 | directory |
| S_IFBLK | 0060000 | block special |
| S_IFREG | 0100000 | ordinary file |
| | 0000000 | ordinary file |
| S_ISUID | 0004000 | set user ID on execution |
| S_ISGID | 0002000 | set group ID on execution |
| | 0001000 | reserved |

access permissions; constructed from the following:

| | | |
|---|---|---|
| S_IRWXU | 0000700 | read, write, execute (search) by owner |
| S_IRUSR | 0000400 | read by owner |
| S_IWUSR | 0000200 | write by owner |
| S_IXUSR | 0000100 | execute (search on directory) by owner |
| S_IRWXG | 0000070 | read, write, execute (search) by group |
| S_IRGRP | 0000040 | read by group |
| S_IWGRP | 0000020 | write by group |
| S_IXGRP | 0000010 | execute (search on directory) by group |
| S_IRWXO | 0000007 | read, write, execute (search) by others |
| S_IROTH | 0000004 | read by others |
| S_IWOTH | 0000002 | write by others |
| S_IXOTH | 0000001 | execute (search on directory) by others |

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 effective group ID of the process.

Values of *mode* other than those above are undefined and should not be used. The owner, group and other permision bits of *mode* are modified by the process's file mode creation mask: all bits whose corresponding bit in the process's file mode creation mask is set are cleared, see

*umask*(2). If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

*Mknod* may be invoked only with the effective user ID of the super-user for file types other than FIFO special.

## ERRORS

*Mknod* will fail and the new file will not be created if one or more of the following are true:

[EPERM] The effective user ID of the process is not super-user and the file type is not FIFO special.

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

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

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

[EROFS] The directory in which the file is to be created is located on a read-only file system.

[EEXIST] The named file exists.

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

[ENOSPC] The directory which would contain the new file cannot be extended.

## 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

chmod(2), exec(2), pipe(2), stat(2), umask(2), stat(5), types(5).

## RELATIONSHIP TO SVID

† The SVID does not use the symbolic names for *mode*; it only gives the absolute values. It therefore also does not call for the inclusion of <**sys/stat.h**> and <**sys/types.h**>. The "Appendix BASE: 1.6 Comparison to the 1984 /usr/group Standard" part of the SVID shows these names as a future direction for the <**stat.h**> header file.

In the SVID, the mode value 0001000 is identified as "save text image after execution" instead of being "reserved".

The table has been changed from that of the SVID to give all the values of access permissions for *group* and *others*.

In the SVID, the last sentence of the DESCRIPTION reads: " *Mknod* may be invoked only by the super-user for file types other than ...".

## NAME

mount — mount a file system

## SYNOPSIS

```
int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;
```

## DESCRIPTION

*Mount* requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if 1, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

*Mount* may be invoked only with an effective user ID of the super-user.

## ERRORS

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

| | |
|---|---|
| [EPERM] | The effective user ID is not super-user. |
| [ENOENT] | Any of the named files does not exist. |
| [ENOTDIR] | A component of a path prefix is not a directory. |
| [ENOTBLK] | *Spec* is not a block special device. |
| [ENXIO] | The device associated with *spec* does not exist. |
| [ENOTDIR] | *Dir* is not a directory. |
| [EFAULT] | *Spec* or *dir* points outside the allocated address space of the process. |
| [EBUSY] | *Dir* is currently mounted on, is someone's current working directory, or is otherwise busy. |
| [EBUSY] | The device associated with *spec* is currently mounted. |
| [EBUSY] | There are no more mount table entries. |

## 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

umount(2).

## RELATIONSHIP TO SVID

Identical to the SVID except for the last sentence of the description, which in the SVID reads: " *Mount* may be invoked only by the super-user."

## NAME

nice — change priority of a process (OPTIONAL)

## SYNOPSIS

```
int nice (incr)
int incr;
```

## DESCRIPTION

*Nice* adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a positive number for which a more positive value results in lower CPU priority.

A maximum *nice* value of 39 and a minimum *nice* value of 0 are imposed by the system. Requests for values above or below these limits result in the *nice* value being set to the corresponding limit.

## ERRORS

[EPERM] *Nice* will fail and not change the *nice* value if *incr* is negative or greater than 40 and the effective user ID of the calling process is not super-user.

## RETURN VALUE

Upon successful completion, *nice* returns the new *nice* value minus 20. Otherwise, a value of —1 is returned and *errno* is set to indicate the error.

## SEE ALSO

exec(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

This *optional* facility is included in the SVID *kernel extension* set (K_EXT).

## NAME

open — open for reading or writing

## SYNOPSIS

```
#include <fcntl.h>

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

## DESCRIPTION

*Path* points to a path name naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by OR-ing 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** This flag may affect subsequent reads and writes, see *read*(2) and *write*(2).

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: (see CAVEATS, Chapter 1, discussion of *termio* )

If O_NDELAY is set:

The open will return without waiting for carrier.

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 effective group ID of the process, and the access permission bits (see *chmod*(2)) of the file mode are set to the value of *mode* modified as follows, see *creat*(2):

The corresponding bits are ANDed with the process's file mode creation mask. See *umask*(2). Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared.

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.

The file pointer used to mark the current position within the file is set to the beginning of the file.

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

## ERRORS

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

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

[ENOENT] O_CREAT is not set and the named file does not exist. A component of the path name which must exist does not exist.

[EACCES] A component of the path prefix denies search permission or the file does not exist and the directory in which the file is to be created does not permit writing.

[EACCES] *Oflag* permission is denied for the named file.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[EROFS] The named file resides on a read-only file system and *oflag* is write or read/write.

[EMFILE] {OPEN_MAX} file descriptors are currently open.

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

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.

| | |
|---|---|
| [EFAULT] | *Path* points outside the allocated address space of the process. |
| [EEXIST] | O_CREAT and O_EXCL are set, and the named file exists. |
| [ENXIO] | O_NDELAY is set, the named file is a FIFO, O_WRONLY is set and no process has the file open for reading. |
| [EINTR] | A signal was caught during the *open* system call. |
| [ENFILE] | The system file table is full. {SYS_OPEN} files are open. |
| [ENOSPC] | The directory which would contain the new file cannot be expanded, the file does not exist, and O_CREAT is specified. |

## RETURN VALUE

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

## SEE ALSO

close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), write(2), fcntl(5).

## APPLICATION USAGE

Normally applications should use the *stdio* routines to open, close, read and write files. Thus applications should use the *stdio* routine *fopen*(3S) rather than using *open*(2).

## FUTURE DIRECTIONS

Mandatory or enforcement-mode file and record locking features will be added.

## RELATIONSHIP TO SVID

Identical to the SVID entry, except for the reference to "CAVEATS, Chapter 1".

## NAME

pause — suspend process until signal

## SYNOPSIS

**int pause ()**

## DESCRIPTION

*Pause* suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

## ERRORS

[EINTR] See **RETURN VALUE** below.

## RETURN VALUE

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function see *signal*(2), the calling process resumes execution from the point of suspension; *pause*(2) returns a value of —1 and *errno* is set to [EINTR].

## SEE ALSO

alarm(2), kill(2), signal(2), wait(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

pipe — create an interprocess channel

## SYNOPSIS

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

## DESCRIPTION

*Pipe* 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 and the O_NDELAY flag is clear.

Up to {PIPE_MAX} bytes of data are buffered by the pipe before the writing process is blocked. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out basis.

## ERRORS

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

[EMFILE] If {OPEN_MAX} —1 or more file descriptors are currently open in this process.

[ENFILE] The system file table would overflow.

## 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

read(2), write(2).

## FUTURE DIRECTIONS

[EFAULT] will be returned in *errno* if the argument is not a valid address for this process.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

plock — lock process, text, or data in memory (OPTIONAL)

## SYNOPSIS

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

int plock (op)
int op;
```

## DESCRIPTION

*Plock* allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *Plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *Op* specifies the following:

| | |
|---|---|
| PROCLOCK | lock text and data segments into memory (process lock) |
| TXTLOCK | lock text segment into memory (text lock) |
| DATLOCK | lock data segment into memory (data lock) |
| UNLOCK | remove locks |

## ERRORS

*Plock* will fail and not perform the requested operation if one or more of the following are true:

| | |
|---|---|
| [EPERM] | The effective user ID of the calling process is not super-user. |
| [EINVAL] | *Op* is equal to PROCLOCK and a process lock, a text lock, or a data lock already exists on the calling process. |
| [EINVAL] | *Op* is equal to TXTLOCK and a text lock, or a process lock already exists on the calling process. |
| [EINVAL] | *Op* is equal to DATLOCK and a data lock, or a process lock already exists on the calling process. |
| [EINVAL] | *Op* is equal to UNLOCK and no type of lock exists on the calling process. |

## RETURN VALUE

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

## SEE ALSO

exec(2), exit(2), fork(2), lock(5).

APPLICATION USAGE

*Plock*(2) should not be used by most applications. Only programs that must have the type of real-time control it provides should use it.

RELATIONSHIP TO SVID

Identical to the SVID entry.

This *optional* facility is included in the SVID *kernel extension* set (K_EXT).

## NAME

profil — execution time profile (OPTIONAL)

## SYNOPSIS

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

## DESCRIPTION

*Buff* points to an area of store whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick ({CLK_TCK} times per second); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to an entry inside *buff*, that entry is incremented. An "entry" is defined as a series of bytes with length *sizeof(short)*.

The interpretation of *scale* is implementation defined.

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 *exec*(2) is executed, but remains on in child and parent both after a *fork*(2). Profiling will be turned off if an update in *buff* would cause a memory fault.

## RETURN VALUE

Not defined.

## APPLICATION USAGE

*Profil*(2) would normally be used in an application program only during its development, to analyse the program's performance.

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that the SVID gives an explicit interpretation of *scale* as follows:

"The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(octal) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock)."

This *optional* facility is included in the SVID *kernel extension* set (K_EXT).

## NAME

ptrace — process trace (OPTIONAL)

## SYNOPSIS

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

## DESCRIPTION

*Ptrace* provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging. The child process behaves normally until it encounters a signal (see *signal*(2)) at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func* see *signal*(2). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If instruction (I) and data (D) space are separated, request 1 returns a word from I space, and request 2 returns a word from D space. If I and D space are not separated either request 1 or request 2 may be used with equal results. The *data* argument is ignored. These two requests will fail if *addr* is not the start address of a word, in which case a value of —1 is returned to the parent process and the parent's *errno* is set to [EIO].

3 With this request, the word at location *addr* in the child's USER area in the system's address space is returned to the parent process. The *data* argument is ignored. This

request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of —1 is returned to the parent process and the parent's *errno* is set to [EIO].

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated, request 4 writes a word into I space, and request 5 writes a word into D space. If I and D space are not separated, either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. Upon failure a value of —1 is returned to the parent process and the parent's *errno* is set to [EIO].

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. Entries that can be written are implementation specific but might include general registers of the Processor Status Word.

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent. This request will fail if *data* is not 0 or a valid signal number, in which case a value of —1 is returned to the parent process and the parent's *errno* is set to [EIO].

8 This request causes the child to terminate with the same consequences as *exit*(2).

9 This request is implementation dependent but if operative, it is used to request single-stepping through the instruc-

tions of the child.

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

## ERRORS

*Ptrace* will in general fail if one or more of the following are true:

[EIO] *Request* is an illegal number. See the summary for each request type above.

[ESRCH] *Pid* identifies a child that does not exist or has not executed a *ptrace* with request 0.

## RETURN VALUE

Upon failure, a value of —1 is returned. Return values on successful completion are specific to the request type (see above).

## SEE ALSO

exec(2), signal(2), wait(2).

## APPLICATION USAGE

*Ptrace*(2) should not be used by applications. It is only used by software debugging programs and it is hardware dependent.

Parts of this may not be implementable on some hardware; other hardware may require that it be extended.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

This *optional* facility is included in the SVID *kernel extension* set (K_EXT).

## NAME

read — read from file

## SYNOPSIS

```
int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

## DESCRIPTION

*Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

*Read* attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking, e.g. terminals, always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl*(2) and *termio*(7)) or if the number of bytes left in the file is less than *nbyte* bytes or if the file is a pipe or a special file. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an empty pipe (or FIFO):

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

When attempting to read a file associated with a character special file that has no data currently available:

> If O_NDELAY is clear, the read will block until the data becomes available. (This functionality of O_NDELAY depends on the implementation of the *termio*(7) interface)† see CAVEATS, Chapter 1.

## ERRORS

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

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

[EFAULT] *Buf* points outside the allocated address space.

[EINTR] A signal was caught during the *read* system call.

[EIO] An I/O error occurred on a special file.

[ENXIO] A request was made of a non-existent special file, or the request was outside the capabilities of the device.

## RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a —1 is returned and *errno* is set to indicate the error.

## SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), lockf(3C), open(2), pipe(2), signal(2), termio(7).

## APPLICATION USAGE

Normally, applications should use the *stdio* library routines to open, close, read, and write files. Thus, an application that used the *stdio* routine *fopen*(3S) to open a file should use the *stdio* routine *fread*(3S) rather than *read*(2) to read it.

## FUTURE DIRECTIONS

*Read* on a pipe, FIFO, or *tty*‡ line with the O_NDELAY flag set will return -1 rather than 0 when no data was present at the time of the *read*.

[EAGAIN] will be returned in *errno* when no data is available on a pipe, FIFO or *tty*‡ line being read.

*Read* will be enhanced to provide enforcement-mode record and file locking features.

## RELATIONSHIP TO SVID

† This sentence additional to SVID.
‡ Subject to restrictions imposed by *optional* functionality of *termio*(7).

The errors [EIO] and [ENXIO] are additional to the SVID.

NAME

setpgrp — set process group ID

SYNOPSIS

```
int setpgrp ()
```

DESCRIPTION

*Setpgrp* sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

RETURN VALUE

*Setpgrp* returns the value of the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), kill(2), signal(2).

RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

setuid, setgid — set user and group IDs

## SYNOPSIS

```
int setuid (uid)
int uid;

int setgid (gid)
int gid;
```

## DESCRIPTION

*Setuid* (*setgid*) is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but the saved set-user (group) ID from *exec*(2) is equal to *uid*(*gid*), the effective user (group) ID is set to *uid*(*gid*).

## ERRORS

[EPERM] *Setuid* (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user.

[EINVAL] The *uid* is out of range.

## 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

getuid(2), exec(2).

## APPLICATION USAGE

The type of the argument taken by *setuid* differs from the type returned by *getuid*(2). This may prompt diagnostics from *lint* (see Part 3) but is otherwise harmless.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

signal — specify what to do upon receipt of a signal

## SYNOPSIS

```
#include <signal.h>

int (*signal (sig, func))()
int sig;
int (*func)();
```

## DESCRIPTION

*Signal* allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

*Sig* can be assigned any one of the following except SIGKILL:

| | |
|---|---|
| SIGHUP | hangup |
| SIGINT | interrupt |
| SIGQUIT† | quit |
| SIGILL† | illegal instruction (not reset when caught) |
| SIGTRAP† | trace trap (not reset when caught) |
| SIGABRT | process abort signal |
| SIGFPE† | floating point exception |
| SIGKILL | kill (cannot be caught or ignored) |
| SIGSEGV† | segmentation violation |
| SIGSYS† | bad argument to system call |
| SIGPIPE | write on a pipe with no one to read it |
| SIGALRM | alarm clock |
| SIGTERM | software termination signal |
| SIGUSR1 | user-defined signal 1 |
| SIGUSR2 | user-defined signal 2 |

(†) The default action for these signals is an abnormal process termination. See SIG_DFL below.

For portability, applications should use the signals listed above and no others. (For example, the System V signals SIGEMT, SIGBUS, and SIGIOT are implementation dependent and are not listed above). Specific implementations may have other implementation-specific sig-

nals.

*Func* is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL terminate process upon receipt of a signal
Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). In addition if *sig* is one of the signals shown with an * above, implementation-dependent abnormal process termination routines such as a core dump, may be invoked.

SIG_IGN ignore signal
The signal *sig* is to be ignored.

Note: the signal SIGKILL cannot be ignored.

*function address* - catch signal
Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Additional arguments may be passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of *func* for the caught signal will be set to SIG_DFL unless the signal is SIGILL, or SIGTRAP.

*Signal* will not catch an invalid function argument, *func,* and results are undefined when an attempt is made to execute the function at the bad address.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted, except for implementation defined signals where this may not be true.

When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately, the signal catching function will be executed and then the interrupted system call may return a —1 to the calling process with *errno* set to [EINTR].

Note: The signal SIGKILL cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending SIGKILL signal.

## ERRORS

[EINVAL] *Signal* will fail if *sig* is an illegal signal number, or is SIGKILL.

## RETURN VALUE

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of (int (*)())-1 is returned and *errno* is set to indicate the error.

## SEE ALSO

kill(2), pause(2), wait(2), setjmp(3C) signal(5).

## APPLICATIONS USAGE

For portability, applications should use only the symbolic names of signals rather than their values and use only the set of signals defined here. Specific implementations may have additional signals.

The signal SIGSEGV is only included for compatibility with existing applications. It is not part of the SVID and should not be used in programs that wish to be portable.

If signals are being caught, then *errno* may be changed by errors that occur in the signal-catching funtion. Its value cannot be relied upon if there is any possibility of a signal arriving between the setting of *errno* and its use.

## FUTURE DIRECTIONS

A macro SIG_ERR will be defined in <**signal.h**> to represent the return value *(int(*)())*-1 by *signal* in the case of error.

## RELATIONSHIP TO SVID

Identical to the SVID entry, except for the inclusion of the two extra signals, SIGABRT and SIGSEGV. (N.B. The addition of SIGABRT is forecast in the FUTURE DIRECTIONS section of the SVID.)

## NAME

stat, fstat — get file status

## SYNOPSIS

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

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

## DESCRIPTION

*Path* points to a path name naming a file. 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. *Stat* obtains information about the named file.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system 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:

```
ushort   st_mode;   /* File mode, see mknod(2) */
ino_t    st_ino;    /* Inode number */
dev_t    st_dev;    /* ID of device containing */
                    /* a directory entry for this */
                    /* file */
dev_t    st_rdev;   /* ID of device */
                    /* This entry is defined only */
                    /* for character special or /*/
                    /* block special files */
short    st_nlink;  /* Number of links */
ushort   st_uid;    /* User ID of the file's owner */
ushort   st_gid;    /* Group ID of the file's group */
off_t    st_size;   /* File size in bytes */
```

```
time_t    st_atime;    /* Time of last access */
time_t    st_mtime;    /* Time data last modified */
time_t    st_ctime;    /* Time of last file status */
                       /* change */
                       /* Times in seconds since */
                       /* 00:00:00 GMT, Jan. 1, 1970 */
```

**st_atime** Time when file data was last accessed. Changed by the following system calls: *creat*(2), *fcntl*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).

**st_mtime** Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).

**st_ctime** Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

## ERRORS

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

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

[ENOENT] The named file does not exist.

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

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

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

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

[EFAULT] *Buf* points to an invalid 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

chmod(2), chown(2), creat(2), link(2), fcntl(2), mknod(2), pipe(2), read(2), time(2), unlink(2), utime(2), write(2), types(5), stat(5).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

stime — set time

## SYNOPSIS

```
int stime (tp)
long *tp;
```

## DESCRIPTION

*Stime* sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

## ERRORS

*Stime* will fail if the following is true:

[EPERM] the effective user ID of the calling process is not 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.

## SEE ALSO

time(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

The SVID reads "... will fail if one or more of the following are true ..." in the ERRORS section, yet there is only one possible error.

## NAME

sync — update super-block

## SYNOPSIS

**void sync ( )**

## DESCRIPTION

*Sync* causes all information in kernel buffers † that updates a file system to be written out to the file system. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

‡ The term "kernel buffers" refers to system buffers which are invisible to the user and are only present to improve performance. It does *not* mean buffering provided by the *stdio*(3S) functions.

## APPLICATION USAGE

Application programs are not expected to use *sync*.

## RELATIONSHIP TO SVID

Identical to the SVID entry, except:
† In the first line of the DESCRIPTION section, the SVID has the term "transient memory" in place of "kernel buffers". The X/OPEN definition has included the last paragraph of the DESCRIPTION section to achieve the same purpose.
‡ Explanatory text added.

## NAME

time — get time

## SYNOPSIS

```
long time ((long *) 0)

long time (tloc)
long *tloc;
```

## DESCRIPTION

*Time* returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

As long as *tloc* is not zero, the return value is also stored in the location to which *tloc* points.

## ERRORS

[EFAULT] *tloc* points to an invalid address.

## RETURN VALUE

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

## APPLICATION USAGE

Some implementations of *time* fail to check the validity of *tloc* and give undefined behaviour if *tloc* points to an invalid address.

## SEE ALSO

stime(2).

## RELATIONSHIP TO SVID

Functionally identical to the SVID entry. The warning above in APPLICATION USAGE is implicit in the SVID definition, which assumes that no systems check *tloc*. The SVID does not mention [EFAULT] or *errno*.

## NAME

times — get process and child elapsed process times

## SYNOPSIS

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

long times (buffer)
struct tms *buffer;
```

## DESCRIPTION

*Times* fills the structure pointed to by *buffer* with time-accounting information. The structure *tms*, which is defined in <**sys/times.h**>, contains the following elements:

```
time_t   tms_utime;
time_t   tms_stime;
time_t   tms_cutime;
time_t   tms_cstime;
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*(2). All times are defined in {CLK_TCK}ths of a second.

*Tms_utime* is the CPU time used while executing instructions in the user space of the calling process.

*Tms_stime* is the CPU time used by the system on behalf of the calling process.

*Tms_cutime* is the sum of the *tms_utimes* and *tms_cutimes* of the child processes.

*Tms_cstime* is the sum of the *tms_stimes* and *tms_cstimes* of the child processes.

## ERRORS

[EFAULT] *Times* will fail if *buffer* points to an illegal address.

## RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in {CLK_TCK}ths of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

exec(2), fork(2), time(2), wait(2), types(5) times(5).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

ulimit — get and set user limits

## SYNOPSIS

```
long ulimit (cmd, newlimit)
int cmd;
long newlimit;
```

## DESCRIPTION

*Ulimit* provides for control over process limits. The *cmd* values available are:

1 Get the file size limit of the process. 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 file size limit of the process 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.

3 Get the maximum possible *brk* value see *brk*(2). *Brk*(2) is *optional*.

## ERRORS

[EPERM] *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit.

## 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

write(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that the *cmd* value 3 case has been added. This arises because *brk*(2) is included as an *optional* facility by X/OPEN, although it is omitted from the SVID.

## NAME

umask — set and get file creation mask

## SYNOPSIS

```
int umask (cmask)
int cmask;
```

## DESCRIPTION

*Umask* sets the process's file mode creation mask (see *creat*(2)) to *cmask* and returns the previous value of the mask. Only the owner, group, other permission bits of *cmask* and the file mode creation mask are used.

## RETURN VALUE

The previous value of the file mode creation mask is returned.

## SEE ALSO

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

umount — unmount a file system

## SYNOPSIS

```
int umount (spec)
char *spec;
```

## DESCRIPTION

*Umount* requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

*Umount* may be invoked with an effective user ID equal to super-user.

## ERRORS

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

| | |
|---|---|
| [EPERM] | The process's effective user ID is not super-user. |
| [ENXIO] | The device associated with *spec* does not exist. |
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENOENT] | The named file does not exist. |
| [ENOTBLK] | *Spec* is not a block special device. |
| [EINVAL] | *Spec* is not mounted. |
| [EBUSY] | A file on *spec* is busy. |
| [EFAULT] | *Spec* 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

mount(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry, except for the last paragraph of the DESCRIPTION section, which in the SVID reads: "*Umount* may be invoked only by the super-user."

## NAME

uname — get name of current system

## SYNOPSIS

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

int uname (name)
struct utsname *name;
```

## DESCRIPTION

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

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

```
char    sysname[{SYS_NMLN}];
char    nodename[{SYS_NMLN}];
char    release[{SYS_NMLN}];
char    version[{SYS_NMLN}];
char    machine[{SYS_NMLN}];
```

*Uname* returns a null-terminated character string naming the current system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Machine* contains a standard name that identifies the hardware that the system is running on.

## ERRORS

[EFAULT] *Uname* will fail if *name* points to an invalid address.

## RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, —1 is returned and *errno* is set to indicate the error.

## SEE ALSO

utsname(5).

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that the term "UNIX system" has been changed to "system".

## NAME

unlink — remove directory entry

## SYNOPSIS

```
int unlink (path)
char *path;
```

## DESCRIPTION

*Unlink* removes the directory entry named by the path name pointed to by *path*. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, space occupied by the file is *not* released until all references to the file have been closed.

## ERRORS

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

| | |
|---|---|
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [ENOENT] | The named file does not exist. |
| [EACCES] | Search permission is denied for a component of the path prefix. |
| [EACCES] | Write permission is denied on the directory containing the link to be removed. |
| [EPERM] | The named file is a directory and the effective user ID of the process is not super-user. |
| [EBUSY] | The entry to be unlinked is the mount point for a mounted file system. |
| [ETXTBSY] | The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. |
| [EROFS] | The directory entry to be unlinked is part of a read-only file system. |
| [EFAULT] | *Path* points outside the process's allocated address space. |

## 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

close(2), link(2), open(2).

RELATIONSHIP TO SVID

Identical to the SVID entry, except that the last sentence of the DESCRIPTION section has been clarified. The SVID reads: "...removed, the removal is postponed until all references to the file have been closed".

## NAME

ustat — get file system statistics

## SYNOPSIS

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

int ustat (dev, buf)
int dev;
struct ustat *buf;
```

## DESCRIPTION

*Ustat* returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure that includes the following elements:

```
daddr_t     f_tfree;        /* Total free blocks */
ino_t       f_tinode;       /* Number of free inodes */
char        f_fname[6];     /* Filsys name or NULL */
char        f_fpack[6];     /* Filsys pack name or NULL */
```

The last two fields, *f_fname* and *f_fpack*, may not have meaningful information on all systems, and, in that case, will contain the NULL character.

## ERRORS

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

[EINVAL] *Dev* is not the device number of a device containing a mounted file system.

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

## 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

stat(2), types(5), ustat(5).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

utime — set file access and modification times

## SYNOPSIS

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

int utime (path, times)
char *path;
struct utimbuf *times;
```

## DESCRIPTION

*Path* points to a path name naming a file. *Utime* sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the superuser may use *utime* this way.

The times in the following structure *utimbuf* are measured in seconds since 00:00:00 GMT, Jan. 1, 1970:

```
struct utimbuf{
        time_t  actime;
        time_t  modtime;
};
```

*Utime* will also cause the time of the last file status change (*st_ctime*) to be updated, see *stat*(5).

## ERRORS

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

| | |
|---|---|
| [ENOENT] | The named file does not exist. |
| [ENOTDIR] | A component of the path prefix is not a directory. |
| [EACCES] | Search permission is denied by a component of the path prefix. |
| [EPERM] | The effective user ID is not super-user and not the owner of the file and *times* is not NULL. |
| [EACCES] | The effective user ID is not super-user and not the owner of the file and *times* is NULL and write access is denied. |
| [EROFS] | The file system containing the file is mounted read-only. |

[EFAULT] *Times* is not NULL and points outside the process's allocated address space.

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

## 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

stat(2), types(5).

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that the SVID does not give an explicit declaration of *struct utimbuf* anywhere.

## NAME

wait — wait for child process to stop or terminate

## SYNOPSIS

```
int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)
```

## DESCRIPTION

*Wait* suspends the calling process until one of the immediate children terminates or until a child that is being traced stops because it has hit a break point. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

If the child process stopped, the low order 8 bits of status will be set to 0177 and the next 8 bits of status will contain the number of the signal that caused the process to stop.

If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the next 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit*(2).

If the child process terminated due to a signal, the low order 8 bits will contain the number of the signal that caused the termination and the next 8 bits of status will be zero. In addition, if abnormal process termination routines (see *signal*(2)) were successfully completed, then the low order seventh bit (i.e. bit 0200) will be set.

If a parent process terminates without waiting for its child processes to terminate, a special system process inherits the child processes, see *exit*(2).

*Wait* will fail and its actions are undefined if *stat_loc* points to an illegal address.

## ERRORS

*Wait* will fail and return immediately if the following is true:

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

## RETURN VALUE

If *wait* 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 *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.

## SEE ALSO

exec(2), exit(2), fork(2), pause(2), signal(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that in the ERRORS section the SVID reads "... if one or more of the following are true:" even though only one error is given.

## NAME

write — write on a file

## SYNOPSIS

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

## DESCRIPTION

*Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

*Write* attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device 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 a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* see *ulimit* (2)) or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO), and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

## ERRORS

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

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

[EPIPE] and **SIGPIPE** signal
An attempt is made to write to a pipe that is not open for reading by any process.

| | |
|---|---|
| [EFBIG] | An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2). |
| [EINTR] | A signal was caught during the *write* system call. |
| [EFAULT] | *Buf* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent. |
| [ENOSPC] | There is no free space remaining on the device containing the file. |
| [EIO] | An I/O error occurred on a special file. |
| [ENXIO] | A request was made of a non-existent special file, or the request was outside the capabilities of the device. |

## RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, —1 is returned and *errno* is set to indicate the error.

## SEE ALSO

creat(2), dup(2), lseek(2), open(2), pipe(2), ulimit(2).

## APPLICATION USAGE

Normally, applications should use *stdio*(3S) library routines to open, close, read and write files. Thus, if an application had used the *stdio* routine *fopen*(3S) to open a file, it would use the *stdio* routine *fwrite*(3S) rather than *write*(2).

Warning: *write* errors to I/O devices give implementation defined results.

## FUTURE DIRECTIONS

[EAGAIN] will be returned in *errno* if the no delay mode is in use on the file and the process will be delayed in the write operation.
*Write* will be enhanced to provide enforcement-mode record and file locking features.

## RELATIONSHIP TO SVID

Identical to the SVID entry, with the addition of the [EIO] and [ENXIO] errors.