intel

XENIX* 286
C LIBRARY GUIDE

*XENIX is a trademark of Microsoft Corporation.

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 174542-001

XENIX* 286
C LIBRARY GUIDE

Order Number: 174542-001

*XENIX is a trademark of Microsoft Corporation.

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

REV.	REVISION HISTORY	DATE
-001	Original issue	11/84

intel®

TABLE OF CONTENTS

CONTENTS

PAGE

PAGE

PAGE

intel®

CHAPTER 1
INTRODUCTION

This manual describes functions provided by the C libraries of the XENIX system. Each library contains multiple functions organized into a single library file. These functions support device-independent input/output (I/O), display and window I/O, and operating system functions such as process control, signaling events, interprocess communication, dynamic storage allocation, process synchronization, and error processing. Miscellaneous functions are provided to support numeric computation, data base programming, encryption, character and string manipulation, and other applications.

Included in the functions described in this manual are all the *system calls* that define the XENIX kernel. These functions can be called by the C programmer like all the others and are not distinguished from the other library functions.

Prerequisites

This manual presumes that you understand the C programming language and basic programming concepts. This manual also presumes some knowledge of XENIX or UNIX. You should be familiar with the shell **sh**; all programming examples using a shell use **sh**.

Manual Organization

This manual contains eight tutorial chapters (2-9) covering different groups of functions. Appendix C contains reference descriptions for all the C library functions, including many that are not mentioned in the tutorial chapters. Chapters in this manual are

1. **Introduction:** manual overview, prerequisites, organization, and notation.

2. **Standard I/O Library:** functions that allow a program to read and write files and devices in the XENIX system.

3. **Screen Processing:** a library of screen processing functions including support for windows.

4. **Character and String Processing:** functions to classify characters and to copy, compare, and search strings.

5. **Process Control:** functions that enable a program to execute other programs or create multiple copies of itself.

6. **Pipes:** functions for efficient interprocess communication.

7. **Signals:** functions that allow programs to control the handling of various system events, including some caused by program errors.

8. **System Resources:** functions for dynamic memory allocation, sharing memory between processes, locking file regions, and using semaphores to synchronize use of other resources.

9. **Error Processing:** functions that support program handling of errors returned by other system functions.

Appendixes in this manual are

A. **Assembly Language Interface:** how to call C functions from assembly language and how to write assembly language routines that can be called from C.

B. **XENIX 286 Release 3 Programming Differences:** some of the differences between versions of XENIX and UNIX.

C. **System Functions:** descriptions of all the C library functions, including many that are not mentioned in the tutorial chapters. Also lists error codes used by the functions.

D. **File Formats:** some information on some of the file formats used in XENIX.

E. **Related Publications:** Descriptions and ordering information for all XENIX 286 Release 3 manuals and any other publications referenced by this manual.

Notation

These notational conventions are used in this manual:

- Literal names are bolded where they occur in text, e.g., **/sys/include, printf, dev_tab, EOF.**
- Syntactic categories are italicized where they occur and indicate that you must substitute an instance of the category, e.g., *filename.*
- In examples of dialogue with the XENIX 286 system, characters entered by the user are bolded.
- In syntax descriptions, optional items are enclosed in brackets (**[ ]**).
- Items that can be repeated one or more times are followed by an ellipsis (**...**).
- Items that can be repeated zero or more times are enclosed in brackets and followed by an ellipsis (**[ ]...**).
- A choice between items is indicated by separating the items with vertical bars (**|**).

Using the C Library Functions

To use the C library functions you must include the proper function call and definitions in the program and make sure the corresponding library is given when the program is compiled. The standard C library, contained in the file **libc.a**, is automatically given when you compile a C language program. Other libraries, including the screen updating and cursor movement library in the file **libcurses.a**, must be explicitly specified when you compile a program using the -l option of the **cc** command (see "cc: a C Compiler" in the *XENIX 286 Programmer's Guide*).

intel®

CHAPTER 2
STANDARD I/O LIBRARY

Nearly all programs use some form of input and output (I/O). Some programs read from or write to files stored on disk. Others write to devices such as line printers. Many programs read from and write to the user's terminal. For this reason, the standard C library provides several predefined I/O functions that a programmer can use in programs. These I/O functions are typically implemented in all C programming environments, not just XENIX or UNIX systems. Thus programs using these functions are highly portable.

This chapter explains how to use the I/O functions in the standard C library. In particular, it describes

- Command line arguments
- Standard input and standard output files
- Stream functions for ordinary files
- Low-level functions for ordinary files
- Random access functions

Preparing for the I/O Functions

To use the standard I/O functions a program must include the file **stdio.h**, which defines the needed constants, macros, and data types. To include this file, place the following line at the beginning of the program:

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

The actual functions are contained in the library file **libc.a**. This file is automatically read whenever you compile a program, so no special argument is needed when you invoke the compiler.

Special Names

The standard I/O library uses many names for special purposes. In general, these names can be used in any program that has included the **stdio.h** file. The following is a list of the special names:

stdin The name of the standard input file

stdout The name of the standard output file

stderr The name of the standard error file

EOF The value returned by the read routines on end-of-file or error

NULL The null pointer, returned by pointer-valued functions to indicate an error

FILE The name of the file type used to declare pointers to streams

BSIZE The size in bytes suitable for an I/O buffer supplied by the user

Special Macros

The functions **getc, getchar, putc, putchar, feof, ferror,** and **fileno** are actually macros, not functions. This means that you cannot redeclare them or use them as targets for a breakpoint when debugging.

Using Command Line Arguments

The XENIX system lets you pass information to a program at the same time you invoke it for execution. You can do this with command line arguments.

A XENIX command line is the line you type to invoke a program. A command line argument is anything you type in a XENIX command line. A command line argument can be a file name, an option, or a number. The first argument in any command line must be the file name of the program you wish to execute.

When you type a command line, the system reads the first argument and loads the corresponding program. It also counts the other arguments, stores them in memory in the same order in which they appear on the line, and passes the count and the locations to the main function of the program.

To access the arguments, the main function must have two parameters: **argc**, an integer variable containing the argument count, and **argv**, an array of pointers to the argument values. You can define the parameters by using the lines

```
main(argc, argv)
int argc;
char *argv[];
```

at the beginning of the main program function. When a program begins execution, **argc** contains the count, and each element in **argv** contains a pointer to one argument.

An argument is stored as a null-terminated string (i.e., a string ending with a null character). The first string (referenced by **argv[0]**) is the program name. The argument count is never less than 1, since the program name is always considered the first argument.

In the following example, command line arguments are read and then echoed on the terminal screen. This program is similar to the XENIX **echo** command.

```
main(argc, argv) /* echo arguments */
int argc;
char *argv[];
{
        int i;

        for (i = 1; i < argc; i++)
                printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

In the example above, an extra space character is added at the end of each argument to separate it from the next argument. This is required, since the system automatically removes leading and trailing white space characters (i.e., spaces and tabs) when it reads the arguments from the command line. Adding a newline character to the last argument is for convenience only; it causes the shell prompt to appear on the next line after the program terminates.

When typing arguments on a command line, make sure each argument is separated from the others by one or more white space characters. If an argument must contain white space characters, enclose that argument in double quotation marks. For example, in the command line

```
display 3 4 "echo hello"
```

the string "echo hello" is treated as a single argument. Also enclose in double quotation marks any argument that contains characters recognized by the shell (e.g., <, >, |, and ↑).

Using the Standard Files

Whenever you invoke a program for execution, the XENIX system automatically creates a standard input, a standard output, and a standard error file to handle a program's input and output needs. Since the bulk of input and output of most programs is through the user's own terminal, the system normally assigns the user's terminal keyboard and screen as the standard input and output, respectively. The standard error file, which receives any error messages generated by the program, is also assigned to the terminal screen.

A program can read the standard input file and write the standard output file with the **getchar, gets, scanf, putchar, puts,** and **printf** functions. The standard error file can be accessed using the stream functions described in the section "Using Stream I/O" later in this chapter.

The XENIX system lets you redirect the standard input and output using the shell's redirection symbols. This allows a program to use other devices and files as its chief source of input and output in place of the terminal keyboard and screen.

The following sections explain how to read from and write to the standard input and output and how to redirect the standard input and output.

Reading from the Standard Input

You can read from the standard input with the **getchar, gets,** and **scanf** functions.

The **getchar** function reads one character at a time from the standard input. The function call has the form

```
int c;
...
c = getchar();
```

where **c** is the variable to receive the character. It must have **int** type, because the possible return value **EOF** is outside the range of the type **char.** The function normally returns the character read but will return the end-of-file value **EOF** if the end of the file or an error is encountered.

The **getchar** function is typically used in a conditional loop to read a string of characters from the standard input. For example, the following function reads **cnt** characters from the standard input, or until **EOF** is read. The example function returns **EOF** if the end-of-file was encountered, else it returns 0.

```
readn(p, cnt)
char p[];
int cnt;
{
        int i, c;

        i = 0;
        while (i<cnt)
                if ((p[i + +] = getchar()) = = EOF) {
                        p[i] = 0;  /* null terminator */
                        return(EOF);

                }
        p[i] = 0;  /* null terminator */
        return(0);
}
```

Note that if **getchar** is reading from the keyboard, it waits for characters to be typed before returning.

The **gets** function reads a string of characters from the standard input into a given memory location. The function call has the form

```
gets(s)
```

where **s** is a pointer to the location to receive the string. The function reads characters until it finds a newline character, then replaces the newline character with a null character. The function returns the null pointer value **NULL** if the end of the file or an error is encountered. Otherwise, it returns the value of s.

gets is typically used to read a full line from the standard input. For example, the following program fragment reads a line from the standard input, stores it in the character array **cmdln** and calls a function (called **parse**) if no error occurs.

```
char cmdln[SIZE];

if ( gets(cmdln) != NULL )
        parse();
```

In this case, the length of the string is assumed to be less than **SIZE**-1.

Note that **gets** does not check the length of the string it reads, so overflow can occur.

The **scanf** function reads one or more values from the standard input where a value is a character, a character string, or a decimal, octal, hexadecimal, or floating-point number. The function call has the form

```
scanf (format [, argptr]...)
```

where **format** is a pointer to a string that defines the format of the values to be read and each **argptr** is a pointer to a variable that is to receive a value. There must be one **argptr** for each assignment specified in the **format** string. Some formats are "%c" for a character, "%s" for a string, and "%d", "%o", or "%x" for a decimal, octal, or hexadecimal number respectively. (Other formats are described in **scanf** in Appendix C.) **scanf** normally returns the number of assignments successfully made but returns **EOF** if the end of the file or an I/O error is encountered. (An input format error simply causes premature return with a smaller number of successful assignments, 0 if the first input item is incorrectly formatted.)

Unlike the **getchar** and **gets** functions, **scanf** skips all white space characters, reading only those characters that make up a value. It then converts the characters, if necessary, into the appropriate string or number.

The **scanf** function is typically used whenever formatted input is required, i.e., input that must be typed in a special way or that has a special meaning. For example, in the following program fragment, **scanf** reads both a name and a number from the same line.

```
char name[20];
int number;

scanf("%s %d", name, &number);
```

In this example, the string "%s %d" defines what values are to be read (a string and a decimal number). The string is copied to the character array **name** and the number to the integer variable **number.** Note that pointers to these variables are used in the call and not the actual variables themselves.

When reading from the keyboard, **scanf** waits for values to be typed before returning. Each value must be separated from the next by one or more white space characters (such as spaces, tabs, or even newline characters). For example, for the function

```
scanf("%s %d %c", name, age, sex);
```

an acceptable input is

```
John 27
M
```

If a value is a number, it must have the appropriate digits, that is, a decimal number must have decimal digits, octal numbers octal digits, and hexadecimal numbers hexadecimal digits.

If **scanf** encounters an error, it immediately stops reading the standard input. Before **scanf** can be used again, the illegal character that caused the error must be removed from the input using the **getchar** function.

You may use the **getchar, gets,** and **scanf** functions in a single program. Just remember that each function reads the next available character, making that character unavailable to the other functions.

Note that when the standard input is the terminal keyboard, the **getchar, gets,** and **scanf** functions usually do not return a value until at least one newline character has been typed. This is true even if only one character is desired.

Writing to the Standard Output

You can write to the standard output with the **putchar, puts,** and **printf** functions.

The **putchar** function writes a single character to the standard output. The function call has the form

```
int putchar(c)
```

where **c** is the character to be written. The function normally returns the same character it wrote, but will return **EOF** if an error is encountered.

The function is typically used in a conditional loop to write a string of characters to the standard output. For example, the function

```
writen(p, cnt)
char p[];
int cnt;
{
        int i;

        while (--cnt > = 0)
                putchar( *p + + );
        putchar('\n');
```

writes **cnt** characters plus a newline character to the standard output.

The **puts** function copies the string found at a given memory location to the standard output. The function call has the form

```
puts(s)
```

where **s** is a pointer to the location containing the string. The string may be any number of characters but must end with a null character. The function writes each character in the string to the standard output (not including the null character) and then writes a newline character.

Since the function automatically appends a newline character, it is typically used when writing full lines to the standard output. For example, the following program fragment writes one of three strings to the standard output.

```
char c;

switch(c) {
        case('1'):
                puts("Continuing...");
                break;

        case('2'):
                puts("All done.");
                break;

        default:
                puts("Sorry, there was an error.");
}
```

The string to be written depends on the value of **c**.

The **printf** function writes one or more values to the standard output where a value is a character, character string, or a decimal, octal, hexadecimal, or floating-point number. The function automatically converts numbers into the proper display format. The function call has the form

```
printf(format [, arg]...)
```

where **format** is a pointer to a string that describes the format of each value to be written and each **arg** is either a value to be written or, in the case of a character string, a pointer to the string. There must be one **arg** for each format in the **format** string. Some formats are "%c" for a character, "%s" for a string, and "%d", "%o", or "%x" for a decimal, octal, or hexadecimal number respectively. Other formats are described in **printf** in Appendix C. **printf** normally returns the number of characters actually printed but returns a negative value if an error was encountered.

The **printf** function is typically used when formatted output is required, i.e., when the output must be displayed in a certain way. For example, you may use the function to display a name and number on the same line as in the following example.

```
char name[];
int number;

printf("person = %s, age = %d", name, number);
```

In this example, the format indicators "%s" and "%d" define how the output values are to be converted to characters, as a string and a decimal number respectively. The output values are copied from the character array **name** and the integer variable **number.** Note that characters in the format string other than format indicators, i.e., "person = " and ", age = ", are copied to the output of **printf** unchanged.

You may use the **putchar, puts,** and **printf** functions in a single program. Just remember that the output appears in the same order as it is written to the standard output.

Redirecting the Standard Input

You can change the standard input from the terminal keyboard to an ordinary file by using the normal shell redirection symbol, <. This symbol directs the shell to open the file named following the symbol for reading as the standard input. For example, the following command line opens the file **phonelist** as the standard input to the program **dial.**

```
dial <phonelist
```

The **dial** program may then use the **getchar, gets,** and **scanf** functions to read characters and values from this file. Note that if the file does not exist, the shell displays an error message and stops the program.

Whenever **getchar, gets,** or **scanf** are used to read from an ordinary file, they return the value EOF if the end of the file or an error is encountered. You may want to check for this value to make sure you do not continue to read characters after an error has occurred.

Redirecting the Standard Output

You can change the standard output of a program from the terminal screen to an ordinary file by using the shell redirection symbol, >. The symbol directs the shell to open for writing the file whose name immediately follows the symbol. For example, the command line

```
dial >savephone
```

opens the file **savephone** as the standard output of the program **dial** instead of the terminal screen. You may use the **putchar, puts,** and **printf** functions to write to the file.

If the file does not exist, the shell automatically creates it. If the file exists, but the program does not have permission to write the file, the shell displays an error message and does not execute the program.

Piping the Standard Input and Output

Another way to redefine the standard input and output is to create a pipe. A pipe simply connects the standard output of one program to the standard input of another. The programs may then use the standard output and input to pass information from one to the other. You can create a pipe by using the standard shell pipe symbol, |.

For example, the command line

```
dial | wc
```

connects the standard output of the program **dial** to the standard input of the program **wc.** (The standard input of **dial** and standard output of **wc** are not affected.) If **dial** writes to its standard output with the **putchar, puts,** or **printf** functions, **wc** can read this output with the **getchar, gets,** or **scanf** functions.

Note that when the program on the output side of a pipe terminates, the system automatically places the value **EOF** in the standard input of the program on the input side. Pipes are described in more detail in Chapter 6, "Pipes."

Program Example

This section shows how you may use the standard input and output files to perform a useful task. The **ccstrip** (for "control character strip") program defined below strips out all ASCII control characters from its input except for newline and tab. You may use this program to display text or data files that contain characters that may disrupt your terminal screen.

```
#include <stdio.h>

main() /* ccstrip: strip control characters */
{
        int c;
        while ((c = getchar()) != EOF)
                if ((c >= ' ' && c < 0177) ||
                        c == '\t' || c == '\n')
                        putchar(c);
        exit(0);
}
```

You can strip and display the contents of a single file by changing the standard input of the **ccstrip** program to the desired file. The command line

```
ccstrip <doc.t
```

reads the contents of the file **doc.t,** strips out control characters, then writes the stripped file to the standard output.

If you wish to strip several files at the same time, you can create a pipe between the **cat** command and **ccstrip.**

To read and strip the contents of the files **file1, file2,** and **file3** and then display them on the standard output, use the command

```
cat file1 file2 file3 | ccstrip
```

If you wish to save the stripped files, you can redirect the standard output of **ccstrip.** For example, the following command line writes the stripped files to the file **clean.**

```
cat file1 file2 file3 | ccstrip >clean
```

Note that the **exit** function is used at the end of the program to ensure that any program that executes the **ccstrip** program will receive a normal termination status (typically 0) from the program when it completes. An explanation of the **exit** function and how to execute one program under control of another is given in Chapter 5, "Process Control."

Using the Stream I/O Functions

The functions described so far have all read from the standard input and written to the standard output. The next step is to show functions that access files not already connected to the program. One set of standard I/O functions allows a program to open and access ordinary files as if they were a "stream" of characters. For this reason, these functions are called "stream functions."

Unlike the standard input and output files, a file to be accessed by a stream function must be explicitly opened with the **fopen** function. The function can open a file for reading, writing, or appending. A program can read from a file with the **getc, fgetc, fgets, fgetw, fread,** and **fscanf** functions. It can write to a file with the **putc, fputc, fputs, fputw, fwrite,** and **fprintf** functions. A program can test for the end of a file or for an error with the **feof** and **ferror** functions. A program can close a file with the **fclose** function.

Using File Pointers

Every file opened for access by the stream functions has a unique pointer associated with it called a file pointer. This pointer, defined with the predefined type **FILE** found in the **stdio.h** file, points to a structure that contains information about the file, such as the location of the buffer (the intermediate storage area between the actual file and the program), the current character position in the buffer, and whether the file is being read or written. A file pointer is returned by the **fopen** function as described in the next section. Thereafter, the file pointer may be used to refer to that file until the file is explicitly closed by calling **fclose.**

A file pointer is defined with a statement such as

```
FILE *infile;
```

The standard input, output, and error files, like other opened files, have corresponding file pointers. These file pointers are named **stdin** for standard input, **stdout** for standard output, and **stderr** for standard error. Unlike other file pointers, the standard file pointers are predefined in the **stdio.h** file. This means a program may use these pointers to read and write from the standard files without first using the **fopen** function to open them.

The predefined file pointers are typically used when a program needs to alternate between the standard input or output file and an ordinary file. Although the predefined file pointers have the type **FILE,** they are constants, not variables. They cannot be assigned values.

Opening a File

The **fopen** function opens a given file and returns a pointer (called a file pointer) to a structure containing the data necessary to access the file. The pointer may then be used in subsequent stream functions to read from or write to the file.

The function call has the form

```
FILE *fp;
...
fp = fopen(filename, type)
```

where **fp** is the pointer variable to receive the file pointer, **filename** is a pointer to the name of the file to be opened, and **type** is a pointer to a string that defines how the file is to be opened. The type string may be "r" for reading, "w" for writing, or "a" for appending, i.e., open for writing at the end of the file.

A file may be opened for different operations at the same time if separate file pointers are used. For example, the following program fragment opens the file named **/usr/accounts** for both reading and appending.

```
FILE *rp, *wp;

rp = fopen("/usr/accounts", "r");
wp = fopen("/usr/accounts", "a");
```

Opening an existing file for writing destroys the old contents. Opening an existing file for appending leaves the old contents unchanged and causes any data written to the file to be appended to the end.

Trying to open a nonexistent file for reading causes an error. Trying to open a nonexistent file for writing or appending causes a new file to be created. Trying to open any file for which the program does not have appropriate permission causes an error.

fopen normally returns a valid file pointer but returns **NULL** if an error is encountered. Check for **NULL** after each call to **fopen** to prevent reading or writing after an error.

Reading a Single Character

The **getc** and **fgetc** functions return a single character read from a given file and return the value **EOF** if the end of the file or an error is encountered. The function calls have the form

```
int c;
...
c = getc(stream)

c = fgetc(stream)
```

where **stream** is the file pointer to the file to be read and **c** is the **int** variable to receive the character. The return value is always an integer.

The functions are typically used in conditional loops to read a string of characters from a file. For example, the following program fragment reads characters into a buffer from the file given to it by **infile** until the end of the file or an error is encountered, or the buffer is filled.

```
int i, c;
char buf[MAX];
FILE *infile;
...
i = 0;
while ((c = getc(infile)) != EOF && i < MAX)
        buf[i++] = c;
```

The only difference between the functions is that **getc** is defined as a macro and **fgetc** as a true function. This means that, unlike **getc**, **fgetc** may be passed as an argument in another function, used as a target for a breakpoint when debugging, or used to avoid any side effects of macro processing.

Reading a String from a File

The **fgets** function reads a string of characters from a file to a given memory location. The function call has the form

```
fgets(s, n, stream)
```

where **s** is a pointer to the location to receive the string, **n** is a count of the maximum number of characters to be stored (including the terminating null), and **stream** is a file pointer for the file to be read. The function reads **n**-1 characters or through the first newline character, whichever occurs first. The function writes a null character at the end of the string. **fgets** returns **NULL** if the end of file or an error is encountered. Otherwise, it returns the pointer **s.**

fgets is typically used to read a full line from a file. For example, the following program fragment reads a string of characters from the file given by **myfile.**

```
char cmdln[MAX];
FILE *myfile;

if ( fgets( cmdln, MAX, myfile ) != NULL)
        parse( cmdln );
```

In this example, **fgets** reads the string into the character array **cmdln.**

Reading Records from a File

The **fread** function reads one or more fixed-length records from a file and copies them to a given memory location. The function call has the form

```
fread(ptr, size, nitems, stream)
```

where **ptr** is a pointer to the location to receive the records, **size** is the size (in bytes) of each record to be read, **nitems** is the number of records to be read, and **stream** is the file pointer of the file to be read. **ptr** can be a pointer to a variable of any type (from a single character to a structure). **size,** an integer, should give the number of bytes in each item you wish to read. One way to ensure this is to use the **sizeof** function (see the example below). **fread** always returns the number of records it read, regardless of whether or not the end of file or an error was encountered.

fread is typically used to read binary data from a file. For example, the following program fragment reads one record from the file given by **database** and copies the records into the structure **person.**

```
FILE *database;
struct record {
        char name[20];
        int age;
} person;

fread(&person, sizeof(person), 1, database);
```

Note that since **fread** does not explicitly indicate errors, the **feof** and **ferror** functions should be used to detect end of file or error conditions. These functions are described later in this chapter.

Reading Formatted Data from a File

fscanf reads formatted input from a given file and copies it to the memory location given by the respective argument pointers, just as the **scanf** function reads from the standard input. The function call has the form

```
fscanf(stream, format [, argptr]...)
```

where **stream** is the file pointer of the file to be read, **format** is a pointer to the string that defines the format of the input to be read, and each **argptr** is a pointer to a variable that is to receive some formatted input. There must be one **argptr** for each assignment specified in the **format** string. Some formats are "%c" for a character, "%s" for a string, and "%d", "%o", or "%x" for a decimal, octal, or hexadecimal number respectively. Other formats are described in **scanf** in Appendix C. **fscanf** normally returns the number of successful assignments made but returns **EOF** if it encounters end of file or an I/O error.

fscanf can be used to read files that contain both numbers and text. For example, this program fragment reads a name and a decimal number from the file given by **file.**

```
FILE *file;
char name[20];
int pay;

fscanf(file,"%s %d\n", name, &pay);
```

This program fragment copies the name to the character array **name** and the number to the integer variable **pay.**

Writing a Single Character

The **putc** and **fputc** functions write single characters to a given file. The function calls have the forms

```
int putc(c, stream)
```

and

```
int fputc(c, stream)
```

where **c** is the character to be written (of type **char**) and **stream** is a file pointer to the file to receive the character. The functions normally return the character written but return **EOF** if an error is encountered.

These functions are typically used in conditional loops to write a string of characters to a file. For example, the following program fragment writes characters from the array **name** to the file specified by **out** until the end of the array or a null character is encountered.

```
FILE *out;
char name[MAX];
int i;

for (i = 0; i<MAX && name[i]! = '\0'; i + +)
        fputc(name[i], out);
```

The only difference between the **putc** and **fputc** functions is that **putc** is defined as a macro and **fputc** as a true function. This means that **fputc,** unlike **putc,** may be used as an argument to another function, as the target of a breakpoint when debugging, or to avoid the side effects of macro processing.

Writing a String to a File

The **fputs** function writes a string to a given file. The function call has the form

```
fputs(s, stream)
```

where **s** is a pointer to the string to be written and **stream** is a file pointer to the file.

fputs can be used with **gets** to copy strings from one file to another. For example, in the following program fragment, **gets** and **fputs** are combined to copy strings from the standard input to the file specified by **out.**

```
FILE *out;
char cmdln[MAX];

if ( gets(cmdln) ! = EOF )
        fputs(cmdln, out);
```

fputs returns 0 if successful, **EOF** if an error is encountered.

Writing Records to a File

The **fwrite** function writes one or more records to a given file. The function call has the form

```
fwrite(ptr, size, nitems, stream)
```

where **ptr** is a pointer to the first record to be written, **size** is the size (in bytes) of each record, **nitems** is the number of records to be written, and **stream** is the file pointer of the file. **ptr** may point to a variable of any type (from a single character to a structure). **size** should give the number of bytes in each item to be written. One way to ensure this is to use the **sizeof** function (see the example below). **fwrite** always returns the number of items actually written to the file whether or not an error is encountered.

fwrite is typically used to write binary data to a file. For example, the following program fragment writes one record to the file given by **database.**

```
FILE *database;
struct record {
        char name[20];
        int age;
} person;

fwrite(&person, sizeof(person), 1, database);
```

Because the function does not report errors, the **ferror** function should be used for error checking.

Writing Formatted Output

The **fprintf** function writes formatted output to a given file, just as the **printf** function writes to the standard output. The function call has the form

```
fprintf(stream, format [, arg]...)
```

where **stream** is a file pointer to the file to be written, **format** is a pointer to a string that defines the format of the output, and each **arg** is an argument to be written. There must be one **arg** for each format conversion in the format string. Some formats are "%c" for a character, "%s" for a string, and "%d", "%o", or "%x" for a decimal, octal, or hexadecimal number respectively. Other formats are described in **printf** in Appendix C. If a string is to be written, the corresponding **arg** is a pointer to the string. For all other formats, the value to be converted is supplied directly as an **arg.** **fprintf** returns the number of characters written, returning **EOF** if an I/O error is encountered.

fprintf can be used to write output that contains both numbers and text. For example, to write a name and a decimal number to the file specified by **outfile**, use the following program fragment.

```
FILE *outfile;
char name[20];
int pay;

fprintf(outfile,"%s %d\n", name, pay);
```

The name is copied from the character array **name** and the number from the integer variable **pay**.

Testing for the End of a File

The **feof** function returns the value -1 if a given file has reached its end. The function call has the form

```
feof(stream)
```

where **stream** is the file pointer of the file. The function returns -1 only if the file has reached its end, otherwise it returns 0. The return value is always an integer.

The **feof** function is typically used after calling functions with return values that are not a clear indicator of an end-of-file condition. For example, the following program fragment checks for end of file after each record is read. The reading stops as soon as **feof** returns -1.

```
char name[10];
FILE *stream;

do
        fread( name, sizeof(name), 1, stream );
while (!feof( stream ));
```

Testing for File Errors

The **ferror** function tests a given stream file for an error. The function call has the form

```
ferror(stream)
```

where **stream** is the file pointer of the file to be tested. The function returns a nonzero (true) value if an error is detected, otherwise it returns zero (false). The function returns an integer value.

ferror is typically used to test for errors before a subsequent read or write to the file. For example, in the following program fragment **ferror** tests the file given by **stream.**

```
FILE *stream;
char x[5];
...
while ( !ferror(stream) )
        fread(x, sizeof(x), 1, stream);
```

If **ferror** returns zero, the next item in the file specified by **stream** is copied to **x.** Otherwise, execution passes to the next statement.

Further use of a stream after an error is detected may cause undesirable results.

Closing a File

The **fclose** function closes a file by breaking the connection between the file pointer and the structure created by **fopen.** Closing a file empties the contents of the corresponding buffer and frees the file pointer for use by another file. The function call has the form

```
fclose(stream)
```

where **stream** is the file pointer of the file to close. The function normally returns 0 but will return -1 if an error is encountered.

The **fclose** function is typically used to free file pointers when they are no longer needed. This is important because usually no more than 20 files can be open at the same time. For example, the following program fragment closes the file given by **infile** when the file has reached its end.

```
FILE *infile;
...
if ( feof(infile) )
        fclose( infile );
```

Note that whenever a program terminates normally, the **fclose** function is automatically called for each open file, so no explicit call is required unless the program must close a file before its end. Also, the function automatically calls **fflush** to ensure that everything written to the file's buffer actually gets to the file.

Program Example

This section shows you how to use the stream functions for a useful task. The following program, which counts the characters, words, and lines found in one or more files, uses the **fopen, getc, fprintf,** and **fclose** functions to open, read, write, and close the given files. The program incorporates a basic design common to other XENIX programs, namely, it uses the file names found in the command line as the files to open and read, or if no names are present, it uses the standard input. This allows the program to be invoked on its own or to be the receiving end of a pipe.

```
#include <stdio.h>

main(argc, argv) /* wc: count lines, words, chars */
int argc;
char *argv[];
{
        int c, i, inword;
        FILE *fp, fopen();
        long linect, wordct, charct;
        long tlinect = 0, twordct = 0, tcharct = 0;

        i = 1;
        fp = stdin;
        do
        {
                if (argc > 1 &&
                        (fp = fopen(argv[i], "r")) = = NULL) {
                        fprintf (stderr, "wc: can't open %s\n",
                          argv[i]);
                        continue;

                }
                linect = wordct = charct = inword = 0;
                while ((c = getc(fp)) ! = EOF) {
                        charct + + ;
                        if (c = = '\n')
                          linect + + ;
                        if (c = = ' ' || c = = '\t' || c = = '\n')
                          inword = 0;
                        else if (inword = = 0) {
                          inword = 1;
                          wordct + + ;
                        }
                }
```

```
                printf("%7ld %7ld %7ld", linect, wordct, charct);
                printf(argc > 1 ? " %s\n" : "\n", argv[i]);
                fclose(fp);
                tlinect + = linect;
                twordct + = wordct;
                tcharct + = charct;
        } while ( + + i < argc);
        if (argc > 2)
                printf("%7ld %7ld %7ld total\n", tlinect,
                        twordct, tcharct);
        exit(0);
}
```

The program uses **fp** as the pointer to receive the current file pointer. Initially this is set to **stdin** in case no file names are present in the command line. If a file name is present, the program calls **fopen** and assigns the file pointer to **fp.** If the file cannot be opened (in which case **fopen** returns **NULL),** the program writes an error message to the standard error file **stderr** with the **fprintf** function. The function prints the format string "wc: can't open %s", replacing the "%s" with the name pointed to by **argv[i].**

Once a file is opened, the program uses the **getc** function to read each character from the file. As it reads characters, the program keeps a count of the number of characters, words, and lines. The program continues to read until the end of the file is encountered, that is, when **getc** returns the value **EOF.**

Once a file has reached its end, the program uses the **printf** function to display the character, word, and line counts at the standard output. The format string in this function causes the counts to be displayed as long decimal numbers with no more than seven digits. The program then closes the current file with the **fclose** function and examines the command line arguments to see if there is another file name.

When all files have been counted, the program uses the **printf** function to display a grand total at the standard output and then stops execution with the **exit** function.

Using More Stream Functions

The stream functions allow more control over a file than just opening, reading, writing, and closing. The functions also let a program take an existing file pointer and reassign it to another file (similar to redirecting the standard input and output files) as well as manipulate the buffer used for intermediate storage between the file and the program.

Using Buffered Input and Output

Buffered I/O is an input and output technique used by the XENIX system to cut down the time needed to read from and write to files. Buffered I/O lets the system collect the characters to be read or written and then transfer them all at once rather than one character at a time. This reduces the number of times the system must access the I/O devices and consequently provides more time for running user programs. Not all files have buffers. For example, files associated with terminals, such as the standard input and output, are not buffered. This prevents unwanted delays when transferring the input and output. When a file does have a buffer, the buffer size in bytes is given by the mainfest constant BSIZE, which is defined in the **stdio.h** file.

When a file has a buffer, the stream functions read from and write to the buffer instead of the file. The system keeps track of the buffer and when necessary fills it with new characters (when reading) or flushes (copies) it to the file (when writing). Normally, a buffer is not directly accessible to a program; however, a program can define its own buffer for a file with the **setbuf** function. **setbuf** also can change a buffered file to be an unbuffered one. The **ungetc** function takes a character that has been read and pushes it back into the buffer. The **fflush** function flushes an output buffer, sending its contents on to the receiving process or device driver.

Reopening a File

The **freopen** function closes the file associated with a given file pointer, then opens a new file and gives it the same file pointer as the old file. The function call has the form

```
freopen(newfile, type, stream)
```

where **newfile** is a pointer to the name of the new file, **type** is a pointer to the string that defines how the file is to be opened ("r" for reading, "w" for writing, and "a" for appending), and **stream** is the file pointer of the old file. The function returns the file pointer **stream** if the new file is opened. Otherwise, it returns **NULL.**

freopen is most often used to attach the predefined file pointers **stdin, stdout,** and **stderr** to other files. For example, the following program fragment opens the file named by **newfile** as the new standard output file.

```
char *newfile;
FILE *nfile;
...
nfile = freopen(newfile, "w", stdout);
```

This has the same effect as using the redirection symbols in the command line of the program.

Setting the Buffer

The **setbuf** function changes the buffer associated with a given file to the program's own buffer. It can also change the access to the file to no buffering. The function call has the form

```
setbuf(stream, buf)
```

where **stream** is a file pointer and **buf** is a pointer to the new buffer or is **NULL** if no buffering is desired. If a buffer is given, it must be **BSIZE** bytes in length, where **BSIZE** is a constant defined in **stdio.h.**

setbuf is typically used to to create a buffer for the standard output when it is assigned to the user's terminal, improving execution time by eliminating the need to write one character to the screen at a time. For example, the following program fragment changes the buffer of the standard output to the location referenced by **p.**

```
char *p;

p = malloc( BSIZE );
setbuf( stdout, p );
```

The new buffer is **BSIZE** bytes long.

setbuf can also be used to change a file from buffered to unbuffered input or output. Unbuffered input and output generally increase the total time needed to transfer large numbers of characters to or from a file but give the fastest transfer speed for individual characters.

setbuf should be called immediately after opening a file and before reading or writing to it. Furthermore, **fclose** or **fflush** must be called to flush the buffer before terminating the program. Otherwise, some data written to the buffer may not be written to the file.

Putting a Character Back into a Buffer

The **ungetc** function puts a character back into the buffer of a given file. The function call has the form

```
ungetc(c, stream)
```

where **c** is the character to put back and **stream** is the file pointer of the file. The function normally returns the same character it put back but returns **EOF** if an error is encountered.

ungetc is typically used when scanning a file for the first character of a string of characters. For example, the following program fragment puts the first character that is not a white space character back into the buffer of the file given by **infile,** allowing the subsequent call to **gets** to read that character as the first character in the string.

```
FILE *infile;
char name[20];
...
while( isspace( c = getc(infile) ) )
        ;
ungetc( c, stdin );
gets( name );
```

Putting a character back into the buffer does not change the corresponding file; it only changes the next character to be read.

Note that the function can put a character back only if one has been previously read. The function cannot put more than one character back at a time. For example, if three characters are read, then only the last character can be put back, never the first two.

Note that the value **EOF** must never be put back in the buffer.

Flushing a File Buffer

The **fflush** function empties the buffer of a given file by immediately writing the buffer contents to the file. The function call has the form

```
fflush(stream)
```

where **stream** is the file pointer of the file. The function normally returns zero but returns **EOF** if an error is encountered.

fflush is used to guarantee that the contents of a partially filled buffer are written to the file. For example, the following program fragment empties the buffer for the file given by **outtty** if the variable **errflag** is 0.

```
FILE *outtty;
int errflag;
...
if (errflag = = 0)
        fflush( outtty );
```

Note that **fflush** is automatically called by the **fclose** function to empty the buffer before closing the file. This means that no explicit call to **fflush** is required if the file is being closed.

fflush ignores any attempt to empty the buffer of a file opened for reading.

Using the Low-Level I/O Functions

The low-level I/O functions provide direct access to files and peripheral devices. They are actually direct calls to the routines used in the XENIX operating system to read from and write to files and peripheral devices. The low-level functions give a program the same control over a file or device as the system, letting it access the file or device in ways that the stream functions do not. However, low-level functions, unlike stream functions, do not provide buffering or any other useful services of the stream functions. This means that any program that uses the low-level functions has the complete burden of handling input and output.

The low-level functions, like the stream functions, cannot be used to read from or write to a file until the file has been opened. A program may use the **open** function to open an existing or a new file. A file can be opened for reading, writing, or appending.

Once a file is opened for reading, a program can read bytes from it with the **read** function. A program can write to a file opened for writing or appending with the **write** function. A program can close a file with the **close** function.

Using File Descriptors

Each file that has been opened for access by the low-level functions has a unique integer called a "file descriptor" associated with it. A file descriptor is similar to a file pointer in that it identifies the file. A file descriptor is unlike a file pointer in that it does not point to any specific structure. Instead, the descriptor is used internally by the system to access the necessary information. Since the system maintains all information about a file, the only access to a file for a program is through the file descriptor.

There are three predefined file descriptors (just as there are three predefined file pointers) for the standard input, output, and error files. The descriptors are 0 for the standard input, 1 for the standard output, and 2 for the standard error file. As with predefined file pointers, a program may use the predefined file descriptors without explicitly opening the associated files.

Note that if the standard input and output files are redirected, the system changes the default assignments for the file descriptors 0 and 1 to the named files. This is also true if the input or output is associated with a pipe. File descriptor 2 can be redirected but normally remains attached to the terminal.

Opening a File

The **open** function opens an existing or a new file and returns a file descriptor for that file. The function call has the form

```
fd = open(name, access [,mode] );
```

where **fd** is the integer variable to receive the file descriptor, **name** is a pointer to a string containing the file name, **access** is an integer expression giving the type of file access, and **mode** is an integer number giving a new file's permissions. **open** normally returns a file descriptor (a positive integer), but returns -1 if an error is encountered.

The **access** expression is formed by using one or more of the following constants: **O_RDONLY** for reading, **O_WRONLY** for writing, **O_RDWR** for both reading and writing, **O_APPEND** for appending to the end of an existing file, and **O_CREAT** for creating a new file. (Other constants are described in **open** in Appendix C.) The constants are defined in the **fcntl.h** include file. The logical OR operator (|) can be used to combine the constants. The argument **mode** is required only if **O_CREAT** is specified. For example, in the following program fragment, **open** is used to open the existing file named **/usr/accounts** for reading and open the new file named **/usr/tmp/scratch** for reading and writing.

```
int in, out;

in = open( "/usr/accounts", O_RDONLY );
out = open( "/usr/tmp/scratch", O_RDWR | O_CREAT, 0754 );
```

In the XENIX system, each file has 9 bits of protection information that control read, write, and execute permission for the owner of the file, for the owner's group, and for all others. A three-digit octal number is the most convenient way to specify the permissions. For instance, in the example above, the octal number 0754 specifies read, write, and execute permission for the owner, read and execute permission for the group, and read-only permission for all others.

Note that if **O_CREAT** is given and the file already exists, then **open** destroys the file's old contents.

There is a limit (usually 20) to the number of files that a program can have open simultaneously. Therefore, any program that intends to process many files must be prepared to reuse file descriptors by closing unneeded files.

Reading Bytes from a File

The **read** function reads one or more bytes of data from a given file and copies them to a given memory location. The function call has the form

```
int n_read, n, fd;
char *buf;

n_read = read(fd, buf, n);
```

where **n_read** is the variable to receive the count of bytes actually read, **fd** is the file descriptor of the file, **buf** is a pointer to the memory location to receive the bytes read, and **n** is the desired number of bytes to be read. The function normally returns the same number of bytes as requested but will return fewer if the file does not have that many bytes left to be read. **read** returns 0 if the file has reached its end and -1 if an error is encountered.

When the file is a terminal, **read** normally reads only through the next newline.

Writing Bytes to a File

The **write** function writes one or more bytes from memory to a file. The function call has the form

```
int n_written, n, fd;
char *buf;

n_written = write(fd, buf, n);
```

where **n_written** is the variable to receive a count of bytes actually written, **fd** is the file descriptor of the file, **buf** is a pointer to the memory location containing the bytes to be written, and **n** is the number of bytes to be written.

write always returns the number of bytes actually written. It is considered an error if the return value is not equal to the number of bytes requested to be written.

Closing a File

The **close** function breaks the connection between a file descriptor and an open file and frees the file descriptor for use with some other file. The function call has the form

```
close(fd)
```

where **fd** is the file descriptor of the file to close. **close** normally returns **0** but will return -1 if an error is encountered.

close is typically used to close files that are no longer needed. For example, the following program fragment closes the standard input if the argument count is greater than 1.

```
int fd;

if (argc > 1)
        close( 0 );
```

Note that all open files in a program are closed when a program terminates normally or when the **exit** function is called, so no explicit calls to close are required in these cases.

Program Examples

This section shows how to use the low-level I/O functions to perform useful tasks. It presents three examples that use the low-level I/O functions as the sole method of input and output.

The first program copies its standard input to its standard output.

```
#define     BUFSIZE BSIZE

main() /* copy input to output */
{
        char  buf[ BUFSIZE ];
        int n;

        while ((n = read( 0, buf, BUFSIZE )) > 0)
                write(1, buf, n);
        exit(0);
}
```

The program uses the **read** function to read BUFSIZE bytes from the standard input (file descriptor 0). It then uses **write** to write the same number of bytes it read to the standard output (file descriptor 1). If the standard input file size is not a multiple of BUFSIZE, the last **read** returns a smaller number of bytes to be written by **write,** and the next call to **read** returns zero.

This program can be used like a copy command to copy the content of one file to another. You can do this by redirecting the standard input and output files.

The second example shows how the **read** and **write** functions can be used to construct higher level functions like **getchar** and **putchar.** For example, the following is a version of **getchar,** which performs unbuffered input:

```
#define     CMASK  0377   /* for making chars > 0 */

int getchar()     /* unbuffered single character input */
{
        char c;
        return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

The variable **c** must be declared **char,** because **read** accepts a character pointer. In this case, the character being returned must be masked with octal 0377 to ensure that it is positive; otherwise sign extension may make it negative.

The second version of **getchar** reads input in large blocks but hands out the characters one at a time:

```
#define CMASK  0377 /* for making characters > 0 */
#define BUFSIZE BSIZE

getchar()     /* buffered version */
{
        static char buf[BUFSIZE];
        static char *bufp = buf;
        static int n = 0;
```

```
        if (n = = 0) { /* buffer is empty */
                n = read(0, buf, BUFSIZE);
                bufp = buf;
        }
        return((--n > = 0) ? *bufp + + & CMASK : EOF);
}
```

Again, each character must be masked with the octal constant 0377 to ensure that it is positive.

The final example is a simplified version of the XENIX utility, **cp**, a program that copies one file to another. The main simplifications are that this version copies only one file and does not permit the second argument to be a directory.

```
#define BUFSIZE BSIZE
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)     /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
        int   f1, f2, n;
        char  buf[ BUFSIZE ];

        if (argc ! = 3)
                error("Usage: cp from to", NULL);
        if ((f1 = open(argv[1], O__RDONLY)) = = -1)
                error("cp: can't open %s", argv[1]);
        if ((f2 = open(argv[2], O__CREAT | O__WRONLY, PMODE)) = = -1)
                error("cp: can't create %s", argv[2]);

        while ((n = read(f1, buf, BUFSIZE)) > 0)
                if (write(f2, buf, n) ! = n)
                        error("cp: write error", NULL);
        exit(0);
}

error(s1, s2)  /* print error message and die */
char *s1, *s2;
{
        printf(s1, s2);
        printf("\n");
        exit(1);
}
```

Using Random Access I/O

Input and output operations on any file are normally sequential. This means each read or write takes place at the character position immediately after the last character read or written. The standard library, however, provides a number of stream and low-level functions that allow a program to access a file randomly, that is, to exactly specify the position it wishes to read from or write to next.

The functions that provide random access operate on a file's "character pointer." Every open file has a character pointer that points to the next character to be read from that file, or the next place in the file to receive a character. Normally, the character pointer is maintained and controlled by the system, but the random access functions let a program move the pointer to any position in the file.

Moving the Character Pointer

The **lseek** function, a low-level function, moves the character pointer in a file opened for low-level access to a given position. The function call has the form

```
long lseek(fd, offset, origin)
int fd; long offset; int origin;
```

where **fd** is the file descriptor of the file, **offset** is the number of bytes to move the character pointer, and **origin** is the number that gives the starting point for the move. **origin** can be 0 for the beginning of the file, 1 for the current position, and 2 for the end. **lseek** returns the new character pointer value as measured in bytes from the beginning of the file.

For example, the following call forces the current position in the file whose descriptor is 3 to move to the 512th byte from the beginning of the file.

```
lseek( 3, (long)512, 0 )
```

Subsequent reading or writing will begin at that position. Note that **offset** must be a long integer and **fd** and **origin** must be integers.

The function may be used to move the character pointer to the end of a file to allow appending, or to the beginning as in a rewind function. For example, the call

```
lseek(fd, (long)0, 2);
```

prepares the file for appending, and

```
lseek(fd, (long)0, 0);
```

rewinds the file (moves the character pointer to the beginning). Notice the "(long)0" argument; it could also be written as

```
0L
```

Using **lseek** it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file:

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
        lseek(fd, pos, 0);    /* get to pos */
        return(read(fd, buf, n));
}
```

Moving the Character Pointer in a Stream

The **fseek** function, a stream function, moves the character pointer in a file to a given location. The function call has the form

```
fseek(stream, offset, origin)
```

where **stream** is the file pointer of the file, **offset** is the number of characters to move to the new position (it must be a long integer), and **origin** is the starting position in the file of the move (it must be 0 for beginning, 1 for current position, or 2 for end of the file). **fseek** normally returns zero but will return **EOF** if an error is encountered.

For example, the following program fragment moves the character pointer to the end of the file specified by **stream.**

```
FILE *stream;

fseek(stream, (long)0, 2);
```

fseek can be used on either buffered or unbuffered files.

Rewinding a File

The **rewind** function, a stream function, moves the character pointer to the beginning of a given file. The function call has the form

```
rewind(stream)
```

where **stream** is the file pointer of the file. The function is equivalent to the following function call

```
fseek(stream, 0L, 0);
```

It is chiefly used as a more readable version of the call.

Getting the Current Character Position

The **ftell** function, a stream function, returns the current position of the character pointer in the given file. The returned position is always relative to the beginning of the file. The function call has the form

```
long p;
p = ftell(stream);
```

where **stream** is the file pointer of the file and **p** is the variable to receive the position. The return value is always a long integer. The function returns the value -1 if an error is encountered.

ftell is typically used to save the current location in the file so that the program can later return to that position. For example, the following program fragment first saves the current character position in **oldp**, then later restores the file to this position if the current character position is greater than 800.

```
FILE *outfile;
long oldp;

oldp = ftell( outfile );
...
if ((ftell( outfile )) > 800)
        fseek(outfile, oldp, 0);
```

ftell is identical to the function call

```
lseek( fd, (long)0, 1)
```

where **fd** is the file descriptor of the given stream file.

intel®

CHAPTER 3
SCREEN PROCESSING

This chapter explains how to use the screen updating and cursor movement library named **curses.** The library provides functions to create and update screen windows, get input from the screen or windows, and move the cursor within the screen or within a window.

Screen Processing Overview

The purpose of screen processing is to give a program a simple and efficient way to use the capabilities of a given terminal. The terminal must be connected to the standard input and output files of a screen processing program. Screen processing does not rely on the given terminal's type. Instead, the screen processing functions use the XENIX terminal capability file **/etc/termcap** to tailor their actions for any given terminal. This makes a screen processing program terminal-independent. The program can be run with any terminal that is described in the **/etc/termcap** file.

The screen processing functions access a terminal screen by working through intermediate "screens" and "windows" in memory. A screen is a representation of what the entire terminal screen should look like. A window is a representation of what some portion of the terminal screen should look like. A screen can be made up of one or more windows, and a window can be as small as a single character or as large as an entire screen.

Before a screen or window can be used, it must be created using the **newwin** or **subwin** functions. These function define the size of the screen or window in terms of lines and columns. Each position in a screen or window represents a place for a single character and corresponds to a similar place on the terminal screen. Positions are numbered according to their line and column. For example, the position in the upper left corner of a screen or window is numbered (0,0) and the position immediately to its right is (0,1). A typical screen has 24 lines and 80 columns. Its upper left corner corresponds to the upper left corner of the terminal screen. A window, on the other hand, may be any size and its upper left corner can correspond to any position on the terminal screen. For convenience, the **initscr** function that initializes a program for screen processing also creates a default screen, **stdscr** (for "standard screen"), which may be used without first creating it. The function also creates **curscr** (for "current screen"), which contains a copy of what is currently on the terminal screen.

To display characters at the terminal screen, a program must write these characters to a screen or window using screen processing functions such as **addch** and **waddch.** If necessary, a program can move to the desired position in the screen or window by using the **move** or **wmove** functions. Once characters are in a screen or window, the program can copy the characters to the terminal screen by using the **refresh** or **wrefresh** function. These functions update the terminal screen according to what has changed in the given screen or window. Since the terminal screen is not changed until a program calls **refresh** or **wrefresh,** a program can maintain several different windows, each containing different characters for the same portion of the terminal screen. The program can choose which window should actually be displayed before updating.

A program can continue to add new characters to a screen or window as needed and edit these characters by using functions such as **insertln, deleteln,** and **clear.** A program can also combine windows to make a composite screen using the **overlay** and **overwrite** functions. In each case, the **refresh** or **wrefresh** function is used to copy the changes to the terminal screen.

Using the Library

To use the library in a program, you must add the line

```
#include <curses.h>
```

to the beginning of your program. The **curses.h** file contains definitions for types and constants used by the library.

The actual screen processing functions are in the library files **libcurses.a** and **libtermlib.a.** These files are not automatically read when you compile your program, so you must include the appropriate library switches in your invocation of the compiler. The command line must have the form

```
cc file ... -lcurses -ltermlib
```

where *file* is the name of the source file you wish to compile. You may give more than one filename if desired. You may also use other compiler options in the command line. For example, the command

```
cc main.c intf.c -lcurses -ltermlib -o sample
```

compiles the files **main.c** and **intf.c** and copies the executable program to the file **sample** after linking the screen processing library files to the program.

Note that the **curses.h** file automatically includes the file **sgtty.h** in your program. This file must not be included twice.

The screen processing library has a variety of predefined names. These names refer to variables, constants, and types that can be used with the library functions. The following is a list of these names:

curscr A pointer to the current version of the terminal screen. It has type pointer to **WINDOW.**

stdscr A pointer to the default screen used for updating when no explicit screen is defined. It has type pointer to **WINDOW.**

def_term A pointer to the default terminal type if the type cannot be determined. It has type pointer to **char.**

my_term Terminal type flag. If **TRUE,** causes the terminal specification in **def_term** to be used, regardless of the real terminal type. It has type **bool.**

ttytype A pointer to the full name of the current terminal, with type pointer to **char.**

LINES Number of lines on the terminal, of type **int.**

COLS Number of columns on the terminal, of type **int.**

ERR Error flag. Returned by functions on an error, of type **int.**

OK Okay flag. Returned by functions on successful operation, of type **int.**

reg A storage class; identical to **register** storage class.

bool A type; identical to type **char.**

TRUE The Boolean true value (1).

FALSE The Boolean false value (0).

Preparing for the Screen Functions

The **initscr** and **endwin** functions perform the operations required to initialize and terminate screen processing in a program. The following sections describe these functions and how they affect the terminal.

Initializing the Screen

The **initscr** function initializes screen processing for a program by allocating the required memory space for the screen processing functions and variables and by setting the terminal to the proper modes. The function call has the form

```
initscr()
```

No arguments are required.

inintscr must be used to prepare the program for subsequent calls to other screen processing functions and for using screen processing variables. For example, in the following program fragment **initscr** initializes the screening processing functions.

```
#include <curses.h>

main()
{

        initscr();
        if ( strcmp(ttytype, "dumb") = = 0 )
                fprintf(stderr, "Terminal type can't display screen.");
        ...
```

In this example, the predefined variable **ttytype** is checked for the currrent terminal type.

The function returns **ERR** if memory allocation causes an overflow.

Using Terminal Capability and Type

The **initscr** function uses the terminal capability descriptions given in the XENIX system's **/etc/termcap** file to prepare the screen processing functions for creating and updating terminal screens. The descriptions define the character sequences required to perform a given operation on a given terminal. These sequences are used by the screen processing functions to add, insert, delete, and move characters on the screen. The descriptions are automatically read from the file when screen processing is initialized, so direct access by a program is not required.

The **initscr** function uses the shell's **TERM** variable to determine which terminal capability description to use. The **TERM** variable is usually assigned an identifier when a user logs in. This identifier defines the terminal type and is associated with a terminal capability description in the **/etc/termcap** file.

If the **TERM** variable has no value, the functions use the default terminal type in the library's predefined variable **def_term**. This variable initially has the value "dumb" (for "dumb terminal"), but the user may change it to any desired value. This must be done before calling the **initscr** function.

In some cases, it is desirable to force the screen processing functions to use the default terminal type. This can be done by setting the library's predefined variable **my_term** to the value **TRUE**. The full name of the current terminal is stored in the predefined variable **ttytype**.

Terminal capabilities, types, and identifiers are described in detail in **termcap** in "Files" in the *XENIX 286 Reference Manual*.

Using Default Terminal Modes

The **initscr** function automatically sets a terminal to default operation modes. These modes define how the terminal displays characters sent to the screen and how it responds to characters typed at the keyboard. **initscr** sets the terminal to ECHO mode, which causes characters typed at the keyboard to be displayed on the screen, and RAW mode, which causes characters to be used as direct input (no editing or signal processing is done).

The default terminal modes can be changed, if desired, by using the appropriate functions described in the section "Setting a Terminal Mode" near the end of this chapter. If the modes are changed, they must be changed immediately after calling **initscr**. Terminal modes are described in **tty** in "Devices" in the *XENIX 286 Reference Manual*.

Using Default Window Flags

The **initscr** function automatically clears the **cursor, scroll,** and **clear** flags of the standard screen to their default values. These flags, called the window flags, define how the **refresh** function affects the terminal screen when refreshing from the standard screen. When clear, the **cursor** flag prevents the terminal's cursor from moving back to its original location after the screen is updated, the **scroll** flag prevents scrolling on the screen, and the **clear** flag prevents the characters on the screen from being cleared before being updated. The flags may be changed by using the functions described in the section "Setting Window Flags" later in this chapter.

Using the Default Terminal Size

The **initscr** function sets the terminal screen size to a default number of lines and columns. The default values are given in the predefined variables **LINES** and **COLS.** If desired, you can change the default size of a terminal by setting the variables to new values. This should be done before the first call to **initscr.** If done after the first call, a second call to **initscr** must be made to delete the existing standard screen and create a new one.

Terminating Screen Processing

The **endwin** function terminates the screen processing in a program by freeing all memory resources allocated by the screen processing functions and restoring the terminal to the state it was in before screen processing began. The function call has the form

```
endwin()
```

No arguments are required.

The function must be used before leaving a program that has called the **initscr** function to restore the terminal to its previous state. The function is generally the last function call in the program. For example, in the following program fragment **initscr** and **endwin** form the beginning and end of the program.

```
#include <curses.h>

main()
{

        initscr();
        /* Program body. */
        endwin();
}
```

endwin should not be called in your program until after calling **initscr, gettmode,** or **setterm.**

Using the Standard Screen

The following sections explain how to use the standard screen to display and edit characters on the terminal screen.

Adding a Character

The **addch** function adds a given character to the standard screen and moves the character pointer one position to the right. The function call has the form

```
addch( ch )
```

where **ch** gives the character to be added; it must have **char** type. For example, if the current position is (0,0), the function call

```
addch('A')
```

places the letter 'A' at this position and moves the pointer to (0,1).

If a newline ('\n') character is given, the function deletes all characters from the current position to the end of the line and moves the pointer one line down, or if the **newline** flag is set, the function deletes the characters and moves the pointer to the beginning of the next line. If a return ('\r') is given, the function moves the pointer to the beginning of the current line. If a tab ('\t') is given, the function moves the pointer to the next tab stop, adding enough spaces to fill gap between the current position and the stop. Tab stops are placed at every eight character positions.

The function returns **ERR** if it encounters an error, such as illegal scrolling.

Adding a String

The **addstr** function adds a string of characters to the standard screen, placing the first character of the string at the current position and moving the pointer one position to the right for each character in the string. The function call has the form

```
addstr( str )
```

where **str** is a character pointer to the given string. For example, if the current position is (0,0), the function call

```
addstr("line");
```

places the beginning of the string "line" at this position and moves the pointer to (0,4).

If the string contains newline, return, or tab characters, the function performs the same actions as described for the **addch** function. If the string is longer than can fit on the current line, the string is truncated.

The function returns **ERR** if it encounters an error such as illegal scrolling.

Printing Formatted Output

The **printw** function prints one or more values on the standard screen, where a value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form

```
printw( fmt [, arg ]...)
```

where **fmt** is a pointer to a string that defines the format of the values, and **arg** is a value to be printed. If more than one **arg** is given, each must be separated from the preceding item with a comma. For each **arg** given, a corresponding format must be given in **fmt**. A format may be "%s" for string, "%c" for character, and "%d", "%o", or "%x" for a decimal, octal, or hexadecimal number respectively. (Other formats are described in **printf** in Appendix C, "System Functions." If "%s" is given, the corresponding **arg** must be a character pointer. For other formats, the actual value or a variable containing the value may be given.

The function is typically used to copy both numbers and strings to the standard screen at the same time. For example, if the current position is (0,0), the function call

```
printw("%s %d", name, 15);
```

prints a name given by the variable "name" at position (0,0) and the number "15" immediately after the name.

The function returns **ERR** if it encounters an error such as illegal scrolling.

Reading a Character from the Keyboard

The **getch** function reads a single character from the terminal keyboard and returns the character as a value. The function call has the form

```
c = getch()
```

where **c** is the variable to receive the character.

The function is typically used to read a series of individual characters. For example, in the following program fragment characters are read and stored until a newline character is encountered.

```
char c, p[MAX];
int i;

i = 0;
while ((c = getch()) != '\n' && i < MAX-1)
        p[i++] = c;
p[i] = '\0';  /* terminating null */
```

If the terminal is set to ECHO mode, the function copies the character to the standard screen. If the terminal is not set to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character. Terminal modes are described later in this chapter, in the section "Controlling the Terminal."

The function returns **ERR** if it encounters an error such as illegal scrolling.

Reading a String from the Keyboard

The **getstr** function reads a string of characters from the terminal keyboard and copies the string to a given location. The function call has the form

```
getstr( str )
```

where **str** is a character pointer to the location to receive the string. When typed at the keyboard, the string must end with a newline character or with the end-of-file character. The extra character is replaced by a null character when the string is stored.

The function is typically used to read names and other text from the keyboard. For example, the following program fragment reads a file name from the keyboard and stores it in the array **name.**

```
char name[20];

getstr(name);
```

If the terminal is set to ECHO mode, the function copies the string to the standard screen. If the terminal is not set to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the characters. Terminal modes are described later in this chapter, in the section "Controlling the Terminal."

The function returns **ERR** if it encounters an error such as illegal scrolling.

Reading Formatted Input

The **scanw** function reads one or more values from the terminal keyboard and copies the values to given locations. A value may be a string, character, or decimal, octal, or hexadecimal number. The function call has the form

```
scanw( fmt [, argptr]... )
```

where **fmt** is a pointer to a string defining the format of the values to be read and **argptr** is a pointer to the variable to receive a value. If more than one **argptr** is given, they are separated by commas. For each **argptr** given, a corresponding format must be given in **fmt.** A format may be "%s" for string, "%c" for character, and "%d", "%o", or "%x" for decimal, octal, or hexadecimal number respectively. (Other formats are described in **scanf** in Appendix C, "System Functions.")

scanw is typically used to read a combination of strings and numbers from the keyboard. For example, in the following program fragment **scanw** reads a name and a number from the keyboard.

```
char name[20];
int id;

scanw("%s %d", name, &id);
```

In this example, the input values are stored in the character array **name** and the integer variable **id.**

If the terminal is set to ECHO mode, the function copies the string to the standard screen. If the terminal is not set to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the characters. Terminal modes are described later in this chapter, in the section "Controlling the Terminal."

The function returns **ERR** if it encounters an error such as illegal scrolling.

Moving the Current Position

The **move** function moves the pointer to the given position. The function call has the form

```
move(y, x)
```

where **y** is an integer value giving the new row position, and **x** is an integer value giving the new column position. For example, if the current position is (0,0), the function call

```
move(4, 4)
```

moves the pointer to (4,4).

The function returns **ERR** if it encounters an error such as illegal scrolling.

Inserting a Character

The **insch** function inserts a character at the current position and shifts the existing character (and all characters to its right) one position to the right. The function call has the form

```
insch( c )
```

where **c** is the character to be inserted.

The function is typically used to insert a series of characters into an existing line. For example, in the following program fragment **insch** is used to insert the number of characters given by **cnt** into the standard screen at the current position.

```
int cnt;
char *string;

while ( cnt ! = 0 ) {
        insch(string[--cnt]);
}
```

The function returns **ERR** if it encounters an error such as illegal scrolling.

Inserting a Line

The **insertln** function inserts a blank line at the current position and moves the existing line (and all lines below it) down one line, causing the last line to move off the bottom of the screen. The function call has the form

```
insertln()
```

No arguments are required.

The function is used to insert additional lines of text into the standard screen. For example, in the following program fragment **insertln** is used to insert a blank line when **cnt** is equal to 79.

```
int cnt;

if ( cnt = = 79 )
        insertln();
```

The function returns **ERR** if it encounters an error such as illegal scrolling.

Deleting a Character

The **delch** function deletes the character at the current position and shifts the character to the right of the deleted character (and all characters to its right) one position to the left. The function call has the form

```
delch()
```

No arguments are required.

The function is typically used to delete a series of characters from the standard screen. For example, in the following program fragment **delch** deletes **cnt** characters beginning at the current position.

```
int cnt;

while ( cnt ! = 0 ) {
        delch();
        cnt-- ;
}
```

Deleting a Line

The **deleteln** function deletes the current line and shifts the line below the deleted line (and all lines below it) one line up, leaving the last line in the screen blank. The function call has the form

```
deleteln()
```

No arguments are required.

The function is used to delete existing lines from the standard screen. For example, in the following program fragment **deleteln** is used to delete a line from the standard screen if **cnt** is 79.

```
int cnt;

if ( cnt = = 79 )
        deleteln();
```

Clearing the Screen

The **clear** and **erase** functions clear all characters from the standard screen by replacing them with spaces. These functions are typically used to prepare the screen for new text.

The **clear** function clears all characters from the standard screen, moves the pointer to (0,0), and sets the standard screen's **clear** flag. The flag causes the next call to the **refresh** function to clear all characters from the terminal screen.

The **erase** function clears the standard screen, moves the pointer to (0,0), but does not set the **clear** flag. For example, in the following program fragment **clear** clears the screen if the input value is 12.

```
char c;

if ((c = getch()) = = 12)
        clear();
```

Clearing a Part of the Screen

The **clrtobot** and **clrtoeol** functions clear one or more characters from the standard screen by replacing the characters with spaces. The functions are typically used to prepare a part of the standard screen for new characters.

The **clrtobot** function clears the screen from the current position to the bottom of the screen. For example, if the current position is (10,0), the function call

```
clrtobot();
```

clears all characters on line 10 and all lines below line 10.

The **clrtoeol** function clears the standard screen from the current position to the end of the current line. For example, if the current position is (10,10), the function call

```
clrtoeol();
```

clears all characters from (10,10) to (10,79). The characters at the beginning of the line remain unchanged.

Note that both the **clrtobot** and **clrtoeol** functions do not change the current position.

Refreshing from the Standard Screen

The **refresh** function updates the terminal screen by copying one or more characters from the standard screen to the terminal. The function effectively changes the terminal screen to reflect the new contents of the standard screen. The function call has the form

```
refresh()
```

No arguments are required.

The function is used solely to display changes to the standard screen. The function copies only those characters that have changed since the last call to **refresh** and leaves any existing text on the terminal screen. For example, in the following program fragment **refresh** is called twice.

```
addstr("The first time.\n");
refresh();
addstr("The second time.\n");
refresh();
```

In this example, the first call to **refresh** copies the string "The first time." to the terminal screen. The second call copies only the string "The second time." to the terminal, since the original string has not changed.

The function returns **ERR** if it encounters an error such as illegal scrolling. If an error is encountered, the function attempts to update as much as the screen as possible without causing the scroll.

Creating and Using Windows

The following sections explain how to create and use windows to display and edit text on the terminal screen.

Creating a Window

The **newwin** function creates a window and returns a pointer that may be used in subsequent screen processing functions. The function call has the form

```
win = newwin( lines, cols, begin_y, begin_x )
```

where **win** is the pointer variable to receive the return value, **lines** and **cols** are integer values that give the total number of lines and columns in the window, and **begin_y** and **begin_x** are integer values that give the line and column positions, respectively, of the upper left corner of the window when displayed on the terminal screen. The **win** variable must have type pointer to **WINDOW.**

The function is typically used in programs that maintain a set of windows, displaying different windows at different times or alternating between windows as needed. For example, in the following program fragment **newwin** creates a new window and assigns the pointer to this window to the variable **midscreen.**

```
WINDOW *midscreen;

midscreen = newwin(5, 10, 9, 35);
```

The window has 5 lines and 10 columns. The upper left corner of the window is placed at position (9,35) on the terminal screen.

If either **lines** or **cols** is zero, the function automatically creates a window that has **LINES - begin_y** lines or **COLS - begin_x** columns, where **LINES** and **COLS** are the predefined constants giving the total number of lines and columns on the terminal screen. For example, the function call

```
newwin(0, 0, 0, 0)
```

creates a new window with its upper left corner at position (0,0) and with **LINES** lines and **COLS** columns.

Creating a Subwindow

The **subwin** function creates a subwindow and returns a pointer to the new window. A subwindow is a window that shares all or part of the character space of another window and provides an alternate way to access the characters in that space. The function call has the form

```
swin = subwin( win, lines, cols, begin_y, begin_x)
```

where **swin** is the pointer variable to receive the return value, **win** is the pointer to the window to contain the new subwindow, **lines** and **cols** are integer values that give the total number of lines and columns in the subwindow, and **begin_y** and **begin_x** are integer values that give the line and column position, respectively, of the upper left corner of the subwindow when dislayed on the terminal screen. The **swin** variable must have **WINDOW** type.

The function is typically used to divide a large window into separate regions. For example, in the following program fragment **subwin** creates the subwindow named **cmdmenu** in the lower part of the standard screen.

```
WINDOW *cmdmenu;

cmdmenu = subwin(stdscr, 5, 80, 19, 0);
```

In this example, changes to **cmdmenu** affect the standard screen as well.

Adding and Printing to a Window

The **waddch, waddstr,** and **wprintw** functions add and print characters, strings, and numbers to a given window.

The **waddch** function adds a given character to the given window and moves the character pointer one position to the right. The function call has the form

```
waddch( win, ch )
```

where **win** is a pointer to the window to receive the character and **ch** gives the character to be added; it must have **char** type. For example, if the current position in the window "midscreen" is (0,0), the function call

```
waddch(midscreen, 'A')
```

places the letter 'A' at this position and moves the pointer to (0,1).

The **waddstr** function adds a string of characters to the given window, placing the first character of the string at the current position and moving the pointer one position to the right for each character in the string. The function call has the form

```
waddstr( win, str )
```

where **win** is a pointer to the window to receive the string and **str** is a character pointer to the given string. For example, if the current position is (0,0), the function call

```
waddstr(midscreen, "line");
```

places the beginning of the string "line" at this position and moves the pointer to (0,4).

The **wprintw** function prints one or more values on the given window, where a value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form

```
wprintw( win, fmt [, arg ]...)
```

where **win** is a pointer to the window to receive the values, **fmt** is a pointer to a string that defines the format of the values, and **arg** is a value to be printed. If more than one **arg** is given, each must be separated from the preceding with a comma. For each **arg** given, a corresponding format must be given in **fmt**. A format may be "%s" for string, "%c" for character, and "%d", "%o", or "%x" for decimal, octal, or hexadecimal number respectively. (Other formats are described in **printf** in Appendix C, "System Functions.") If "%s" is given, the corresponding **arg** must be a character pointer. For other formats, the actual value or a variable containing the value may be given.

The function is typically used to copy both numbers and strings to the standard screen at the same time. For example, in the following program fragment **wprintw** prints a name and then the number 15 at the current position in the window **midscreen.**

```
char *name;

wprintw(midscreen, "%s %d", name, 15);
```

Note that when a newline, return, or tab character is given to a **waddch, waddstr,** or **wprintw** function, the functions perform the same actions as described for the **addch** function. The functions return **ERR** if they encounter errors such as illegal scrolling.

Reading and Scanning for Input

The **wgetch, wgetstr,** and **wscanw** functions read characters, strings, and numbers from the standard input and usually echo the values by copying them to the given window.

The **wgetch** function reads a single character from the standard input and returns the character as a value. The function call has the form

```
c = wgetch( win )
```

where **win** is a pointer to a window, and **c** is the character variable to receive the character.

The function is typically used to read a series of characters from the keyboard. For example, in the following program fragment **wgetch** reads characters until a colon (:) is found.

```
char c, dir[MAX];
int i;

i = 0;
while ((c = wgetch(cmdmenu)) ! = ':' && i < MAX)
        dir[i + +] = c;
```

The **wgetstr** function reads a string of characters from the terminal keyboard and copies the string to a given location. The function call has the form

```
wgetstr( win, str )
```

where **win** is a pointer to a window, and **str** is a character pointer to the location to receive the string. When typed at the keyboard, the string must end with a newline character or with the end-of-file character. The extra character is replaced by a null character when the string is stored.

The function is typically used to read names and other text from the keyboard. For example, in the following program fragment **wgetstr** reads a string from the keyboard and stores it in the array **filename.**

```
char filename[20];

wgetstr(cmdmenu, filename);
```

The **wscanw** function reads one or more values from the standard input and copies the values to given locations. A value may be a string, character, or decimal, octal, or hexadecimal number. The function call has the form

```
wscanw( win, fmt [, argptr ]... )
```

where **win** is a pointer to a window, **fmt** is a pointer to a string defining the format of the values to be read, and **argptr** is a pointer to the variable to receive a value. If more than one **argptr** is given, each must be separated from the preceding by a comma. For each **argptr** given, a corresponding format must be given in **fmt.** A format may be "%s" for string, "%c" for character, and "%d", "%o", or "%x" for decimal, octal, or hexadecimal number respectively. (Other formats are described in **scanf** in Appendix C, "System Functions.")

The function is typically used to read a combination of strings and numbers from the keyboard. For example, in the following program fragment **wscanw** reads a name and a number from the keyboard.

```
char name[20];
int id;

wscanw(midscreen, "%s %d", name, &id);
```

In this example, the name is stored in the character array **name** and the number in the integer variable **id.**

If the terminal is set to ECHO mode, the function copies the string to the given window. If the terminal is not set to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character.

The functions return **ERR** if they encounter errors such as illegal scrolling.

Moving a Window's Current Position

The **wmove** function moves the current position of a given window. The function call has the form

```
wmove(win, y, x)
```

where **win** is a pointer to a window, **y** is an integer value giving the new line position, and **x** is an integer value giving the new column position. For example, the function call

```
wmove(midscreen, 4, 4)
```

moves the current position in the window **midscreen** to (4,4).

The function returns **ERR** if it encounters an error such as illegal scrolling.

Inserting Characters

The **winsch** and **winsertln** functions insert characters and lines into a given window.

The **winsch** function inserts a character at the current position and shifts the existing character (and all characters to its right) one position to the right. The function call has the form

```
winsch( win, c )
```

where **win** is a pointer to a window, and **c** is the character to be inserted.

The function is typically used to edit the existing contents of the given window. For example, the function call

```
winsch(midscreen, 'X');
```

inserts the character 'X' at the current position in the window **midscreen.**

The **winsertln** function inserts a blank line at the current position and moves the existing line (and all lines below it) down one line, causing the last line to move off the bottom of the screen. The function call has the form

```
winsertln( win )
```

where **win** is a pointer to the window to receive the blank line.

The function is used to insert lines into a window. For example, in the following program fragment **winsertln** inserts a blank line at the top of the window **cmdmenu,** preparing it for a new line.

```
char line[80];

wmove(cmdmenu, 3, 0);
winsertln(cmdmenu);
waddstr(cmdmenu, line);
```

Both functions return **ERR** if they encounter errors such as illegal scrolling.

Deleting Characters

The **wdelch** and **wdeleteln** functions delete characters and lines from the given window.

The **wdelch** function deletes the character at the current position and shifts the character to the right of the deleted character (and all characters to its right) one position to the left. The function call has the form

```
wdelch( win )
```

where **win** is a pointer to a window.

The function is typically used to edit the existing contents of the standard screen. For example, the function call

```
wdelch(midscreen);
```

deletes the character at the current position in the window **midscreen.**

The **wdeleteln** function deletes the current line and shifts the line below the deleted line (and all lines below it) one line up, leaving the last line in the screen blank. The function call has the form

```
wdeleteln( win )
```

where **win** is a pointer to a window.

The function is typically used to delete existing lines from a given window. For example, in the following program fragment **wdeleteln** deletes the lines in **midscreen** until **cnt** is equal to zero.

```
int cnt;

while ( cnt ! = 0 ) {
        wdeleteln(midscreen);
        cnt--;
}
```

Clearing the Screen

The **wclear, werase, wclrtobot,** and **wclrtoeol** functions clear all or part of the characters from the given window by replacing them with spaces. The functions are typically used to prepare the window for new text.

The **wclear** function clears all characters from the window, moves the pointer to (0,0), and sets the standard screen's **clear** flag. The flag causes the next **refresh** function call to clear all characters from the terminal screen. The function call has the form

```
wclear( win )
```

where **win** is a pointer to the window to be cleared.

The **werase** function clears the given window, moves the pointer to (0,0), but does not set the **clear** flag. It is used whenever the contents of the terminal screen must be preserved. The function call has the form

```
werase( win )
```

where **win** is a pointer to the window to be cleared.

The **wclrtobot** function clears the window from the current position to the bottom of the screen. The function call has the form

```
wclrtobot( win )
```

where **win** is a pointer to the window to be cleared. For example, if the current position in the window **midscreen** is (10,0), the function call

```
wclrtobot( midscreen );
```

clears all characters on line 10 and all lines below line 10.

The **wclrtoeol** function clears the standard screen from the current position to the end of the current line. The function call has the form

```
wclrtoeol( win )
```

where **win** is a pointer to the window to be cleared. For example, if the current position in **midscreen** is (10,10), the function call

```
clrtoeol( midscreen );
```

clears all characters from (10,10) to the end of the line. The characters at the beginning of the line remain unchanged.

Note that the **wclrtobot** and **wclrtoeol** functions do not change the current position.

Refreshing from a Window

The **wrefresh** function updates the terminal screen by copying one or more characters from the given window to the terminal. The function effectively changes the terminal screen to reflect the new contents of the window. The function call has the form

```
wrefresh( win )
```

where **win** is a pointer to a window.

The function is used solely to display changes to the window. The function copies only those characters that have changed since the last call to **wrefresh** and leaves any existing text on the terminal screen. For example, in the following program fragment **wrefresh** is called twice.

```
waddstr(cmdmenu, "Type a command name\n");
wrefresh(cmdmenu);
waddstr(cmdmenu, "Command: ");
wrefresh(cmdmenu);
```

In this example, the first call to **wrefresh** copies the string "Type a command name" to the terminal screen. The second call copies only the string "Command: " to the terminal, since the original string has not changed.

The function returns **ERR** if it encounters an error such as illegal scrolling. If an error is encountered, the function attempts to update as much as the screen as possible without causing the scroll.

Overlaying Windows

The **overlay** function copies all characters except spaces from one window to another, moving characters from their original positions in the first window to identical positions in the second. The function effectively lays the first window over the second, letting characters in the second window that would otherwise be covered by spaces remain unchanged. The function call has the form

```
overlay( win1, win2)
```

where **win1** is a pointer to the window to be copied, and **win2** is a pointer to the window to receive the copied text. The starting positions of **win1** and **win2** must match, otherwise an error occurs. If **win1** is larger than **win2**, the function copies only those lines and columns in **win1** that fit in **win2**.

The function is typically used to build a composite screen from overlapping windows. For example, in the following program fragment **overlay** is used to build the standard screen from two different windows.

```
WINDOW *info, *cmdmenu;

overlay(info, stdscr);
overlay(cmdmenu, stdscr);
refresh();
```

Overwriting a Screen

The **overwrite** function copies all characters, including spaces, from one window to another, moving characters from their positions in the first window to identical positions in the second. The function effectively writes the contents of the first window over the second, destroying the previous contents of the second window. The function call has the form

```
overwrite( win1, win2 )
```

where **win1** is a pointer to the window to be copied, and **win2** is a pointer to the window to receive the copied text. If **win1** is larger than **win2**, the function copies only those lines and columns in **win1** that fit in **win2**.

The function is typically used to display the contents of a temporary window in the middle of a larger window. For example, in the following program fragment **overwrite** is used to copy the contents of a work window to the standard screen.

```
WINDOW *work;

overwrite(work, stdscr);
refresh();
```

Moving a Window

The **mvwin** function moves a given window to a new position on the terminal screen, causing the upper left corner of the window to occupy a given line and column position. The function call has the form

```
mvwin( win, y, x)
```

where **win** is a pointer to the window to be moved, **y** is an integer value giving the line to which the corner is to be moved, and **x** is an integer value giving the column to which the corner is to be moved.

The function is typically used to move a temporary window when an existing window under it contains information to be viewed. For example, in the following program fragment **mvwin** moves the window named **work** to the upper left corner of the terminal screen.

```
WINDOW *work;

mvwin(work, 0, 0);
```

The function returns **ERR** if it encounters an error such as an attempt to move part of a window off the edge of the screen.

Reading a Character from a Window

The **inch** and **winch** functions read a single character from the current pointer position in a window or screen.

The **inch** function reads a character from the standard screen. The function call has the form

```
c = inch()
```

where **c** is the character variable to receive the character read.

The **winch** function reads a character from a given window or screen. The function call has the form

```
c = winch( win )
```

where **win** is the pointer to the window containing the character to be read.

The functions are typically used to compare the actual contents of a window with what is assumed to be there. For example, in the following program fragment **inch** and **winch** are used to compare the characters at position (0,0) in the standard screen and in the window named **altscreen.**

```
char c1, c2;

c1 = inch();
c2 = winch(altscreen);
if (c1 != c2)
        error();
```

Note that reading a character from a window does not alter the contents of the window.

Touching a Window

The **touchwin** function makes the entire contents of a given window appear to be modified, causing a subsequent **refresh** call to copy all characters in the window to the terminal screen. The function call has the form

```
touchwin( win )
```

where **win** is a pointer to the window to be touched.

The function is typically used when two or more overlapping windows make up the terminal screen. For example, the function call

```
touchwin(leftscreen);
```

is used to touch the window named **leftscreen.** A subsequent **refresh** copies all characters in **leftscreen** to the terminal screen.

Deleting a Window

The **delwin** function deletes a given window from memory, freeing the space previously occupied by the window for other windows or for dynamically allocated variables. The function call has the form

```
delwin( win )
```

where **win** is the pointer to the window to be deleted.

The function is typically used to remove temporary windows from a program or to free memory space for other uses. For example, the function call

```
delwin(midscreen);
```

removes the window **midscreen.**

Note that you must delete subwindows from a window before deleting the window. If you do not, the memory space held by the subwindow becomes inaccessible.

Using Other Window Functions

The following sections explain how to perform a variety of operations on existing windows, such as setting window flags and drawing boxes around the window.

Drawing a Box

The **box** function draws a box around a window using the given characters to form the horizontal and vertical sides. The function call has the form

```
box( win, vert, hor )
```

where **win** is the pointer to the desired window, **vert** is the vertical character, and **hor** is the horizontal character. Both **ver** and **hor** must have **char** type.

The function is typically used to distinguish one window from another when combining windows on a single screen. For example, in the following program fragment **box** creates a box around the window in the lower half of the screen.

```
WINDOW *cmdmenu;

cmdmenu = subwin(stdscr, 5, 80, 19, 0);
box(cmdmenu, '|', '-');
```

If necessary, the function will leave the corners of the box blank to prevent illegal scrolling.

Displaying Bold Characters

The **wstandout** function sets the standout character attribute, causing characters subsequently added to the given window to be displayed as bold characters. The function call has the form

```
wstandout( win )
```

where **win** is a pointer to a window.

wstandout is typically used to make error messages or instructions clearly visible when displayed at the terminal screen.

Note that the actual appearance of characters with the standout attribute depends on the given terminal. This attribute is defined by the SO and SE (or US and UE) sequences given in the terminal's **termcap** entry (see **termcap** in "Files" in the *XENIX 286 Reference Manual*).

Restoring Normal Characters

The **wstandend** function restores the normal character attribute, causing characters subsequently added to a specified window to be displayed as normal characters. The function call has the form

```
wstandend( win )
```

where **win** is a pointer to a window.

Setting Window Flags

The **leaveok, scrollok,** and **clearok** functions set or clear the **cursor, scroll,** and **clear-screen** flags. The flags control the action of the **refresh** function when called for the given window.

The **leaveok** function sets or clears the **cursor** flag that defines how the **refresh** function places the terminal cursor and the window pointer after updating the screen. If the flag is set, **refresh** leaves the cursor after the last character to be copied and moves the pointer to the corresponding position in the window. If the flag is cleared, **refresh** moves the cursor to the same position on the screen as the current pointer position in the window. The function call has the form

```
leaveok( win, state )
```

where **win** is a pointer to the window containing the flag to be set, and **state** is a Boolean value defining the state of the flag. If **state** is **TRUE** the flag is set; if **FALSE,** the flag is cleared. For example, the function call

```
leaveok(stdscr, TRUE);
```

sets the **cursor** flag.

The **scrollok** function sets or clears the **scroll** flag for the given window. If the flag is set, scrolling through the window is allowed. If the flag is clear, then no scrolling is allowed. The function call has the form

```
scrollok( win, state )
```

where **win** is a pointer to a window, and **state** is a Boolean value defining how the flag is to be set. If **state** is **TRUE,** the flag is set; if **FALSE,** the flag is cleared. The flag is initially clear, making scrolling illegal.

The **clearok** function sets and clears the **clear** flag for a given screen. The function call has the form

```
clearok( win, state )
```

where **win** is a pointer to the desired screen, and **state** is a Boolean value. The function sets the flag if **state** is **TRUE**, and clears the flag if **FALSE**. For example, the function call

```
clearok(stdscr, TRUE)
```

sets the **clear** flag for the standard screen.

When the **clear** flag is set, each **refresh** call to the given screen automatically clears the screen by passing a clear-screen sequence to the terminal. This sequence affects the terminal only; it does not change the contents of the screen.

If **clearok** is used to set the **clear** flag for the current screen **curscr**, each call to **refresh** automatically clears the screen, regardless of which window is specified in the call.

Scrolling a Window

The **scroll** function scrolls the contents of a given window upward by one line. The function call has the form

```
scroll( win )
```

where **win** is a pointer to the window to be scrolled.

Combining Movement with Action

Many screen operations move the current position of a given window before performing an action on the window. For convenience, you can combine a number of functions with the movement prefix. This combination has the form

```
mvfunc( [ win, ] y, x [, arg ] ... )
```

where **func** is the name of a function, **win** is a pointer to the window to be operated on (if necessary), **y** is an integer value giving the line to move to, **x** is an integer value giving the column to move to, and **arg** is a required argument for the given function. If more than one argument is required, they must be separated with commas. For example, the function call

```
mvaddch(10, 5, 'X');
```

moves the position to (10,5) and adds the character 'X'. The operation is the same as moving the position with the **move** function and then adding a character with **addch**.

Controlling the Terminal

The following sections explain how to set the terminal modes, how to move the terminal's cursor, and how to access other aspects of the terminal.

Setting a Terminal Mode

The **crmode, echo, nl,** and **raw** functions set a terminal to the corresponding mode, causing subsequent input from the terminal's keyboard to be processed accordingly.

The **crmode** function sets the CBREAK mode for the terminal. The mode preserves the function of editing and signal keys, allowing input to be edited as it is typed and allowing signals to be sent to a program from the keyboard. The function call has the form

```
crmode()
```

No arguments are required.

The **echo** function sets the ECHO mode for the terminal, causing each character typed at the keyboard to be displayed at the terminal screen. The function call has the form

```
echo()
```

No arguments are required.

The **nl** function sets a terminal to NEWLINE mode, causing all newline characters to be mapped to a corresponding newline and return character combination. The function call has the form

```
nl()
```

No arguments are required.

The **raw** function sets the RAW mode for the terminal, causing each character typed at the keyboard to be sent as direct input. The mode disables the function of the editing and signal keys and disables the mapping of newline characters into newline and return combinations. The function call has the form

```
raw()
```

No arguments are required.

Clearing a Terminal Mode

The **nocrmode, noecho, nonl,** and **noraw** functions clear a terminal from the corresponding mode, allowing input to be processed according to a previous mode.

The **nocrmode** function clears a terminal from the CBREAK mode. The function call has the form

```
nocrmode()
```

No arguments are required.

The **noecho** function clears a terminal from the ECHO mode. This mode prevents characters typed at the keyboard from being displayed on the terminal screen. The function call has the form

```
noecho()
```

No arguments are required.

The **nonl** function clears a terminal from NEWLINE mode, causing newline characters to be mapped into themselves. This allows the screen processing functions to perform better optimization. The function call has the form

```
nonl()
```

No arguments are required.

The **noraw** function clears a terminal form RAW mode, restoring normal editing and signal generating function to the keyboard. The function call has the form

```
noraw()
```

No arguments are required.

Moving the Terminal's Cursor

The **mvcur** function moves the terminal's cursor from one position to another. The function call has the form

```
mvcur(last_y, last_x, new_y, new_x)
```

where **last_y** and **last_x** are integer values giving the last line and column position of the cursor, and **new_y** and **new_x** are integer values giving the new line and column position of the cursor. For example, the function call

```
mvcur(10, 5, 3, 0)
```

moves the cursor from (10,5) to (3,0) on the terminal screen.

The function can only be used to perform special tasks in programs that do not use other screen processing functions.

Getting the Terminal Mode

The **gettmode** function returns the current tty mode. The function call has the form

```
s = gettmode()
```

where **s** is the variable to receive the status.

The function is normally called by the **initscr** function.

Setting a Terminal Type

The **setterm** function sets the terminal type to the given type. The function call has the form

```
setterm( name )
```

where **name** is a pointer to a string containing the terminal type identifier. The function is normally called by the **initscr** function but may be used in special cases.

Reading the Terminal Name

The **longname** function converts a given **termcap** identifier into the full name of the corresponding terminal. The function call has the form

```
longname( termbuf, name )
```

where **termbuf** is a pointer to the string containing the terminal type identifier, and **name** is a character pointer to the location to receive the long name. The terminal type identifier must exist in the **/etc/termcap** file.

The function is typically used to get the full name of the terminal currently being used. Note that the current terminal's identifier is stored in the variable **ttytype**, which may be used to receive a new name.

intel®

CHAPTER 4
CHARACTER AND STRING PROCESSING

Character and string processing are an important part of many programs. Programs regularly assign, manipulate, and compare characters and strings to complete their tasks. For this reason, the standard library provides a variety of character and string processing functions. These functions are a convenient way to test, translate, assign, or compare characters and strings.

To use the character functions in a program, the file **ctype.h**, which provides the definitions for special character macros, must be included in the program. The line

```
#include <ctype.h>
```

must appear at the beginning of the program.

To use the string functions, no special action is required. These functions are defined in the standard C library and are available whenever you compile a C program.

Using the Character Functions

The character functions test and convert characters. Many character functions are defined as macros and thus cannot be redefined or used as targets for breakpoints when debugging.

Testing for an ASCII Character

The **isascii** function tests for characters in the ASCII character set, i.e., characters whose values range from 0 to 127. The function call has the form

```
isascii(c)
```

where **c** is the character to be tested. The function returns a nonzero (true) value if the character is ASCII, otherwise it returns zero (false). For example, in the following program fragment **isascii** determines whether or not the value in **c** read from the file given by **data** is in the acceptable ASCII range.

```
FILE *data;
int c;

c = fgetc(data);
if (!isascii(c))
        notext();
```

In this example, a function **notext** is called if the character is not in range.

Converting to ASCII Characters

The **toascii** function converts non-ASCII characters to ASCII. The function call has the form

```
c = toascii(i);
```

where **c** is the variable to receive the character, and **i** is the integer value to be changed. The function creates an ASCII character by masking out all but the low 7 bits of the non-ASCII value. If **i** is already an ASCII character, no change takes place. For example, the function call

```
ascii = toascii(160);
```

converts value 160 to 32, the ASCII value of the space character.

The function is typically used to prepare non-ASCII characters for display at the standard output. For example, in the following program fragment **toascii** converts each character read from the file given by the variable **oddstrm.**

```
FILE *oddstrm;
int c;

c = toascii( getc(oddstrm) );
if ( isprint(c) || isspace(c) )
        putchar(c);
```

If the resulting character is printable or is white space, it is written to the standard output.

Testing for Alphanumerics

The **isalnum** function tests for letters and decimal digits, i.e., the alphanumeric characters. The function call has the form

```
isalnum(c)
```

where **c** is the character to test. The function returns a nonzero (true) value if the character is alphanumeric, otherwise it returns zero (false). For example, the function call

```
isalnum('1')
```

returns a nonzero value, but the call

```
isalnum('@')
```

returns zero.

Testing for a Letter

The **isalpha** function tests for uppercase or lowercase letters, i.e., alphabetic characters. The function call has the form

```
isalpha(c)
```

where **c** is the character to be tested. The function returns a nonzero (true) value if the character is a letter, otherwise it returns zero. For example, the function call

```
isalpha('a')
```

returns a nonzero value, but the call

```
isalpha('2')
```

returns zero.

Testing for Control Characters

The **iscntrl** function tests for control characters, i.e., characters with ASCII values in the range 0 to 31 or are 127. The function call has the form

```
iscntrl(c)
```

where **c** is the character to be tested. The function returns a nonzero (true) value if the character is a control character, otherwise it returns zero (false). For example, in the following program fragment **iscntrl** determines whether or not the character in **c** read from the file given by **infile** is a control character.

```
FILE *infile, *outfile;
int c;

c = fgetc(infile);
if ( !iscntrl(c) )
        fputc(c, outfile);
```

The **fputc** function is ignored if the character is a control character.

Testing for a Decimal Digit

The **isdigit** function tests for decimal digits. The function call has the form

```
isdigit(c)
```

where **c** is the character to be tested. The function returns a nonzero value if the character is a digit, otherwise it returns zero. For example, in the following program fragment a sequence of digits read from a file is interpreted as a decimal unsigned integer.

```
FILE *infile;
int c, num;
num = 0;
while ( isdigit( c = getc(infile) ) )
        num = num*10 + (c - '0');
```

Testing for a Hexadecimal Digit

The **isxdigit** function tests for a hexadecimal digit, that is, a character that is either a decimal digit or an uppercase or lowercase letter in the range A to F. The function call has the form

```
isxdigit(c)
```

where **c** is the character to be tested. The function returns a nonzero value if the character is a hexadecimal digit, otherwise it returns zero. For example, in the following program fragment **isxdigit** tests whether a hexadecimal digit is read from the standard input.

```
int c;

c = getchar();
if ( isxdigit(c) )
        hexmode();
```

In this example, a function named **hexmode** is called if a hexadecimal digit is read.

Testing for Printable Characters

The **isprint** function tests for printable characters, i.e., characters whose ASCII values range from 32 to 126. The function call has the form

```
isprint(c)
```

where **c** is the character to be tested. The function returns a nonzero value if the character is printable, otherwise it returns zero. Note that the space character (ASCII value 32) is considered printable.

Testing for Punctuation

The **ispunct** function tests for punctuation characters, i.e., characters that are printable characters but not alphanumeric characters. The function call has the form

```
ispunct(c)
```

where **c** is the character to be tested. The function returns a nonzero value if the character is a punctuation character, otherwise it returns zero.

Testing for White Space

The **isspace** function tests for white-space characters, i.e, the space, horizontal tab, vertical tab, carriage return, formfeed, and linefeed (newline) characters. The function call has the form

```
isspace(c)
```

where **c** is the character to be tested. The function returns a nonzero value if the character is a white-space character, otherwise it returns zero.

Testing for Case in Letters

The **isupper** and **islower** functions test for uppercase and lowercase letters respectively. The function calls have the form

```
isupper(c)
```

and

```
islower(c)
```

where **c** is the character to be tested. Each function returns a nonzero value if the character is the proper case, otherwise it returns zero. For example, the function call

```
isupper('b')
```

returns zero (false), but the call

```
islower('b')
```

returns a nonzero (true) value.

Converting the Case of a Letter

The **tolower** and **toupper** functions convert the case of a given letter. The function calls have the form

```
c = tolower(i);
```

and

```
c = toupper(i);
```

where **c** is the variable to receive the converted letter, and **i** is the letter to be converted. For example, the statement

```
lower = tolower('B');
```

converts 'B' to 'b' and assigns it to the variable **lower,** and the statement

```
upper = toupper('b');
```

converts 'b' to 'B' and assigns it to the variable **upper.**

The **tolower** function returns the character unchanged if it is not an uppercase letter. Similarly, the **toupper** function returns the character unchanged if it is not a lowercase letter.

These functions are typically used to make the case of the characters read from a file or standard input consistent. For example, in the following program fragment **tolower** changes the character read from the standard input to lowercase before it is compared.

```
if (tolower( getchar() ) != 'y')
        exit(0);
```

This conversion allows the user to type either 'Y' or 'y' to prevent the program from executing the **exit** function.

Using the String Functions

The string functions concatenate, compare, copy, or count the number of characters in a string. Two special string functions, **sscanf** and **sprintf,** let a program read from and write to a string in the same way the standard input and output can be read and written. These functions are convenient when reading or writing whole lines containing values of several different formats.

Many string functions have two forms: a form that manipulates all characters in the string and one that manipulates a given number of characters. This gives programs very fine control over all or parts of strings.

Concatenating Strings

The **strcat** function concatenates two strings by appending the characters of one string to the end of another. The function call has the form

```
strcat(dst, src)
```

where **dst** is a pointer to the string to receive the new characters, and **src** is a pointer to the string containing the new characters. The function appends the new characters in the same order as they appear in **src**, then appends a null character to the last character in the new string. The function always returns the pointer **dst**.

The function is typically used to build a string such as a full path name from two smaller strings. For example, in the following program fragment **strcat** concatenates the string "temp" to the contents of the character array **dir**.

```
char dir[MAX] = "/usr/";

strcat(dir, "temp");
```

Note that the destination region must be large enough to hold the concatenated string. **strcat** does not check for overflow.

Comparing Strings

The **strcmp** function compares the characters in one string to those in another and returns an integer value showing the result of the comparison. The function call has the form

```
strcmp(s1, s2)
```

where **s1** and **s2** are pointers to the strings to be compared. The function returns zero if the strings are equal (i.e., have the same characters in the same order). If the strings are not equal, the function returns the difference between the ASCII values of the first unequal pair of characters. The value of the second string character is always subtracted from the first. For example, the function call

```
strcmp("Character A", "Character A");
```

returns zero since the strings are identical in every way, but the function call

```
strcmp("Character A", "Character B");
```

returns -1 because the ASCII value of 'A' is one less than the ASCII value of 'B'.

Note that the function continues to compare characters until a mismatch is found. If one string is shorter than the other, the function stops at the end of the shorter string. For example, the function call

```
strcmp("Character A", "Character ")
```

returns 65, i.e., the difference between the null character at the end of the second string and the 'A' at the end of the first string.

Copying a String

The **strcpy** function copies a given string to a given location. The function call has the form

```
strcpy(dst, src)
```

where **src** is a pointer to the string to be copied, and **dst** is a pointer to the location to receive the string. The function copies all characters in the source string **src** to the destination **dst** and appends a null character to the end of the new string. If **dst** contained a string before the copy, that string is destroyed. The function always returns a pointer to the new string, **dst.**

For example, in the following program fragment **strcpy** copies the string "not available" to the location given by **name.**

```
char na[] = "not available";
char name[20];

strcpy(name, na);
```

Note that the location to receive a string must be large enough to contain the string. The function does not detect overflow.

Getting a String's Length

The **strlen** function returns the number of characters contained in a given string. The function call has the form

```
strlen(s)
```

where **s** is a pointer to a string. The count includes all characters up to, but not including, the first null character. The return value is always an integer.

In the following program fragment, **strlen** is used to determine whether or not the contents of **inname** are short enough to be stored in **name.** Note that the source length must be *less* than the number of characters available in the destination, to allow for the null character appended to the copied string.

```
char *inname;
char name[MAX];

if ( strlen(inname) < MAX )
        strcpy(name, inname);
```

Concatenating Characters to a String

The **strncat** function appends one or more characters to the end of a given string. The function call has the form

```
strncat(dst, src, n)
```

where **dst** is a pointer to the string to receive the new characters, **src** is a pointer to the string containing the new characters, and **n** is an integer value giving the number of characters to be concatenated. The function appends the given number of characters to the end of the **dst** string, then returns the pointer **dst.** Unlike **strcat,** the number of characters to be concatenated by **strncat** is explicitly specified.

In the following program fragment, **strncat** copies the first three characters in "letter" to the end of "cover".

```
char cover[] = "cover";
char letter[] = "letter";

strncat(cover, letter, 3);
```

This example creates the new string "coverlet" in "cover".

Comparing Characters in Strings

The **strncmp** function compares one or more pairs of characters in two given strings and returns an integer giving the result of the comparison. The function call has the form

```
strncmp(s1, s2, n)
```

where **s1** and **s2** are pointers to the strings to be compared, and **n** is an integer value giving the number of characters to compare. The function returns zero if the first **n** characters are identical or if **n** is zero. Otherwise, the function returns the difference between the ASCII values of the first unequal pair of characters. The function generates the difference by subtracting the second string character from the first.

For example, the function call

```
strncmp("Character A", "Character B", 5)
```

returns zero because the first five characters are identical, but the function call

```
strncmp("Character A", "Character B", 11)
```

returns -1 because the value of 'A' is one less than 'B'.

Note that the function continues to compare characters until a mismatch or the end of a string is found. The function returns zero if an end of string is encountered before a mismatch (which only occurs if the strings are identical and **n** is greater than their length).

Copying Characters to a String

The **strncpy** function copies a given number of characters to a given string. The function call has the form

```
strncpy (dst, src, n)
```

where **dst** is a pointer to the string to receive the characters, **src** is a pointer to the string containing the characters, and **n** is an integer value giving the number of characters to be copied. The function either copies the first **n** characters in **src** to **dst**, or if **src** has fewer than **n** characters, copies all characters up to the first null character. The function always returns the pointer **dst**.

In the following program fragment, **strncpy** copies the first three characters in **date** to **day**.

```
char date [29] = "Fri Dec 29 09:35:44 EDT 1982";
char day[MAX];

strncpy( day, date, 3);
```

In this example, **day** receives the string "Fri".

Reading Values from a String

The **sscanf** function reads one or more values from a given character string and stores the values at specified memory locations. The function is similar to the **scanf** function, which reads values from the standard input. The function call has the form

```
sscanf (s, format [, argptr ]...);
```

where **s** is a pointer to the string to be read, **format** is a pointer to the string defining the format of the values to be read, and **argptr** is a pointer to the variable that is to receive the values read. If more than one **argptr** is given, they must be separated by commas. The **format** string may contain the same formats as given for **scanf** (see **scanf** in Appendix C). The function always returns the number of values read.

The function is typically used to read values from a string containing several values of different formats or to read values from a program's own input buffer. For example, in the following program fragment **sscanf** reads two values from the string pointed to by **datestr**.

```
char datestr[] = "THU MAR 29 11:04:40 EST 1983";
char month[4];
char year[5];

sscanf(datestr, "%*3s%3s%*2s%*8s%*3s%4s", month, year);
printf("%s, %s\n", month, year);
```

The first value (a three-character string) is stored at the location pointed to by **month**. The second value (a four-character string) is stored at the location pointed to by **year**.

Writing Values to a String

The **sprintf** function writes one or more values to a given string. The function call has the form

```
sprintf(s, format [, arg]...)
```

where **s** is a pointer to the string to receive the value, **format** is a pointer to the string defining the format of the values to be written, and **arg** is the variable or value to be written. If more than one **arg** is given, they must be separated by commas. The **format** string may contain the same formats as given for **printf** (see **printf** in Appendix C). After all values are written to the string, the function adds a null character to the end of the string. The function normally returns the number of characters written (not including the terminating null). Fewer than the expected number of characters may be written if a formatting error is encountered.

The function is typically used to build a large string from several values of different formats. For example, in the following program fragment **sprintf** writes three values to the string pointed to by **cmd.**

```
char cmd[100];
char *doc = "/usr/src/cmd/cp.c"
int width = 50;
int length = 60;

sprintf(cmd,"pr -w%d -l%d %s\n", width, length, doc);
system(cmd);
```

In this example, the string created by **sprintf** is used in a call to the **system** function. The first two values are the decimal numbers given by **width** and **length.** The last value is a string (a file name) and is pointed to by **doc.** The final string has the form

```
pr -w50 -l60 /usr/src/cmd/cp.c
```

Note that the string to receive the values must have sufficient length to store those values. The function does not check for overflow.

intel®

CHAPTER 5
PROCESS CONTROL

This chapter describes the process control functions of the standard C library. These functions let a program call other programs, using a method similar to calling functions.

There are a variety of process control functions. The **system** and **exit** functions provide the highest level of execution control and are used by most programs that need a straightforward way to call another program or terminate the current one. The **execl, execv, fork,** and **wait** functions provide low-level control of execution and are for those programs that must have very fine control over their own execution and the execution of other programs. Other process control functions such as **abort** and **exec** are described in detail in Appendix C, "System Functions."

The process control functions are part of the standard C library. Since this library is automatically read when compiling a C program, no special library argument is required when invoking the compiler.

Using Processes

"Process" is the term used to describe a program executed by the XENIX system. A process consists of instructions and data, and a table of information about the program, such as its allocated memory, open files, and current execution status.

You create a process whenever you invoke a program through a shell. The system assigns a unique process ID to a program when it is invoked and uses this ID to control and manage the program. The unique IDs are needed in a system running several processes at the same time.

You can also create a process by directing a program to call another program. This causes the system to perform the same functions as when it invokes a program through a shell. In fact, these two methods are actually the same method--invoking a program through a shell is nothing more than directing a program (the shell) to call another program.

Calling a Program

The **system** function calls the given program, executes it, and then returns control to the original program. The function call has the form

```
system(cmdline)
```

where **cmdline** is a pointer to a string containing a shell command line. The command line must be exactly as it would be typed at the terminal, that is, it must begin with the program name followed by any required or optional arguments. For example, the call

```
system("date");
```

causes the system to execute the **date** command, which displays the current time and date at the standard output. The call

```
system("cat >response");
```

causes the system to execute the **cat** command. In this case, the standard output is redirected to the file **response,** so the command reads from the standard input and copies this input to the file **response.**

The **system** function is typically used in the same way as a function call to execute a program and return to the original program. For example, in the following program fragment **system** calls a program whose name is given in the string **cmd.**

```
char name[20], cmd[40];
...
printf("Enter filename: ");
scanf("%s", name);
sprintf(cmd, "cat %s ", name);
system(cmd);
```

Note that the string in **cmd** is built using the **sprintf** function and contains the program name **cat** and an argument (the file name read by **scanf).** The effect is to execute the **cat** command with the given file name.

When using the **system** function, it is important to remember that buffered input and output functions, such as **getc** and **putc,** do not change the contents of their buffer until it is ready to be read or flushed. If a program uses one of these functions, then executes a command with the **system** function, that command may read or write data not intended for its use. To avoid this problem, the program should clear all buffered input and output before making a call to the **system** function. You can do this for output with the **fflush** function and for input with the **setbuf** function described in the section "More Stream Functions" in Chapter 2, "Standard I/O Library."

Stopping a Program

The **exit** function stops the execution of a program by returning control to the system. The function call has the form

```
exit(status)
```

where **status** is the integer value to be sent to the system as the termination status.

The function is typically used to terminate a program before its normal end, such as after a serious error. For example, in the following program fragment **exit** stops the program and sends the integer value 2 to the system if the **fopen** function returns the null pointer value **NULL.**

```
FILE *ttyout;

if ( fopen(ttyout,"r") = = NULL )
        exit(2);
```

Note that the **exit** function automatically closes each open file in the program before returning to the system. Thus, no explicit calls to the **fclose** or **close** functions are required before an exit.

Overlaying a Program

The **execl** and **execv** functions cause the system to overlay the calling program with the given one, allowing the calling program to terminate while the new program continues execution.

The **execl** function call has the form

```
execl(pathname, command-name, argptr...)
```

where **pathname** is a pointer to a string containing the full path name of the command you want to execute, **command-name** is a pointer to a string containing the name of the program you want to execute, and **argptr** is one or more pointers to strings that contain the program arguments. Each **argptr** must be separated from any other argument by a comma. The last **argptr** in the list must be the null pointer value **NULL.** For example, in the call

```
execl("/bin/date", "date", NULL);
```

the **date** command, whose full path name is "/bin/date", takes no arguments, and in the call

```
execl("/bin/cat", "cat", file1, file2, NULL);
```

the **cat** command, whose full path name is "/bin/cat", takes the pointers **file1** and **file2** as arguments.

The **execv** function call has the form

```
execv(pathname, ptr);
```

where **pathname** is the full path name of the program you want to execute, and **ptr** is pointer to an array of pointers. Each element in the array must point to a string. The array may have any number of elements, but the first element must point to a string containing the program name, and the last must be the null pointer, **NULL.**

The **execl** and **execv** functions are typically used in programs that execute in two or more phases and communicate through temporary files (for example a two-pass compiler). The first part of such a program can call the second part by giving the name of the second part and the appropriate arguments. For example, the following program fragment checks the status of **errflag,** then either overlays the current program with the program **pass2,** or displays an error message and quits.

```
char *tmpfile;
int errflag;

if (errflag = = 0)
        execl("/usr/bin/pass2", "pass2", tmpfile, NULL);
else {
        fprintf(stderr, "Error %d: Quitting", errflag);
        exit(2);
}
```

The **execv** function is typically used to pass arguments to a program when the precise number of arguments is not known beforehand. For example, the following program fragment reads arguments from the command line (beginning with the third one), copies the pointer of each to an element in **cmd,** sets the last element in **cmd** to **NULL,** and executes the **cat** command.

```
char *cmd[];

cmd[0] = "cat";
for (i = 3; i<argc; i + + )
        cmd[i] = argv[i];
cmd[argc] = NULL;

execv("/bin/cat", cmd);
```

The **execl** and **execv** functions return control to the original program only if there is an error in finding the given program (e.g., a misspelled path name or no execute permission). This allows the original program to check for errors and display an error message if necessary. For example, the following program fragment searches for the program **display** in the **/usr/bin** directory.

```
execl("/usr/bin/display", "display", NULL);
fprintf(stderr, "Can't execute 'display' \n");
```

If the program **display** is not found or lacks the necessary permissions, the original program resumes control and displays an error message.

Note that the **execl** and **execv** functions will not expand metacharacters (e.g., <, >, *, ?, [, and]) given in the argument list. If a program needs these features, it can use **execl** or **execv** to call a shell as described in the next section.

Executing a Program through a Shell

One drawback of the **execl** and **execv** functions is that they do not provide the metacharacter features of a shell. One way to overcome this problem is to use **execl** to execute a shell and let the shell execute the command you want.

The function call has the form

```
execl("/bin/sh", "sh", "-c", cmdline, NULL);
```

where **cmdline** is a pointer to the string containing the command line needed to execute the program. The string must be exactly as it would appear if typed at the terminal.

For example, a program can execute the command

```
cat *.c
```

(which contains the metacharacter *) with the call

```
execl("/bin/sh", "sh", "-c", "cat *.c", NULL);
```

In this example, the full path name **/bin/sh** and command name **sh** start the shell. The argument **-c** causes the shell to treat the argument "cat *.c" as a whole command line. The shell expands the metacharacter and displays all files that end with **.c**, something that the **cat** command cannot do by itself.

Duplicating a Process

The **fork** function splits an executing program into two independent and fully-functioning processes. The function call has the form

```
int fork()
```

No arguments are required.

The function is typically used to make multiple copies of any program that must take divergent actions as a part of its normal operation, e.g., a program that must use the **execl** function yet still continue to execute. The original program, called the "parent" process, continues to execute normally, just as it would after any other function call. The new process, called the "child" process, starts its execution at the same point, that is, just after the **fork** call. (The child never goes back to the beginning of the program to start execution.) The two processes now execute as independent programs.

The **fork** function returns a different value to each process. To the parent process, the function returns the process ID of the child. The child process ID is always a positive integer and is always different than the parent ID. To the child, the function returns 0. All other variables and values remain exactly as they were in the parent.

The return value is typically used to determine which steps the child and parent should take next. For example, in the program segment

```
char *cmd;

if (fork() = = 0)
        execl("/bin/sh", "sh", "-c", cmd, NULL);
```

The child's return value, 0, causes the expression "fork() == 0", to be true, and therefore the **execl** function is called. The parent's return value, on the other hand, causes the expression to be false, and the function call is skipped. Executing the **execl** function causes the child to be overlayed by the program given by **cmd.** This does not affect the parent.

If **fork** encounters an error and cannot create a child, it will return the value -1. It is a good idea to check for this value after each call.

Waiting for a Process

The **wait** function causes a parent process to wait until its child processes have completed their execution before continuing its own execution. The function call has the form

```
wait(ptr)
```

where **ptr** is a pointer to an integer variable. It receives the termination status of the child from both the system and the child itself. The function normally returns the process ID of the terminated child, so the parent may check it against the value returned by **fork.**

The function is typically used to synchronize the execution of a parent and its child and is especially useful if the parent and child processes access the same files. For example, the following program fragment causes the parent to wait while the program named by **pathname** (which has overlaid the child process) finishes its execution.

```
int status;
char *pathname;
char *cmd[];

if (fork() = = 0)
        execv(pathname, cmd);
wait(&status);
```

The **wait** function always copies a status value to its argument. The status value is actually two 8-bit values combined into one. The low-order 8 bits is the termination status of the child as defined by the system. This status is zero for normal termination and nonzero for other kinds of termination, such as termination by an interrupt, quit, or hangup signal (see **signal** in Appendix C for a description of the various kinds of termination). The next 8 bits is the termination status of the child as defined by its own call to **exit**. If the child did not explicitly call **exit**, the status is zero.

Inheriting Open Files

Any program called by another program or created as a child process to a program automatically inherits the original program's open files and standard input, output, and error files. This means if the file was open in the original program, it will be open in the new program or process.

A new program also inherits the contents of the input and output buffers used by the open files of the original program. To prevent a new program or process from reading or writing data that is not intended for its use, these buffers should be flushed before calling the program or creating the new process. A program can flush an output buffer with the **fflush** function and an input buffer with **setbuf**.

Program Example

This section shows how to use the process control functions to control a simple process. The following program starts a shell on the terminal given in the command line. The terminal is assumed to be connected to the system through a line that has not been enabled for multiuser operation.

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
        int status;
```

```
        if (argc < 2) {
            fprintf(stderr,"No tty given.\n");
            exit(1);
        }
        if (fork() = = 0) {
            if (freopen(argv[1],"r",stdin) = = NULL)
                exit(2);
            if (freopen(argv[1],"w",stdout) = = NULL)
                exit(2);
            if (freopen(argv[1],"w",stderr) = = NULL)
                exit(2);
            execl("/bin/sh","sh",NULL);
        }
        wait(&status);
        if (status = = 512)
            fprintf("Bad tty name: %s\n", argv[1]);
}
```

In this example, the **fork** function creates a duplicate copy of the program. The child changes the standard input, output, and error files to the new terminal by closing and reopening them with the **freopen** function. The terminal name pointed to by **argv** must be the name of the device special file associated with the terminal, e.g., **/dev/tty03.** The **execl** function then calls the shell, which uses the new terminal as its standard input, output, and error files.

The parent process waits for the child to terminate. The **exit** function terminates the process if an error occurs when reopening the standard files. Otherwise, the process continues until the CONTROL-D key is pressed at the new terminal.

intel®

CHAPTER 6
PIPES

A pipe is an artifical file that a program may create and use to pass information to other programs. A pipe is similar to a file in that it has a file pointer and/or a file descriptor and can be read from or written to using the input and output functions of the standard library. Unlike a file, a pipe does not represent a specific file or device. Instead a pipe represents temporary storage in memory that is independent of the program's own memory and is controlled entirely by the system.

Pipes are chiefly used to pass information between programs, just as the shell pipe symbol (|), is used to pass the output of one program to the input of another. This eliminates the need to create temporary files to pass information to other programs. A pipe can also be used as a temporary storage place for a single program. A program can write to the pipe, then read that information back at a later time.

The standard library provides several pipe functions. The **popen** and **pclose** functions control both a pipe and a process. **popen** opens a pipe and creates a new process at the same time, making the new pipe the standard input or output of the new process. **pclose** closes a pipe and waits for termination of the corresponding process. The **pipe** function, on the other hand, gives low-level access to a pipe. The function is similar to the **open** function, but opens the pipe for both reading and writing, returning two file descriptors instead of one. The program can either use both sides of the pipe or close the one it does not need. The low-level input and output functions **read** and **write** can be used to read from and write to a pipe. Pipe file descriptors are used in the same way as other file descriptors.

Opening a Pipe to a New Process

The **popen** function creates a new process and then opens a pipe to the standard input or output file of that new process. The function call has the form

```
popen(command, type)
```

where **command** is a pointer to a string that contains a shell command line, and **type** is a pointer to a string that defines whether the pipe is to be opened for reading or writing by the original process. The string may be "r" for reading or "w" for writing. The function normally returns the **FILE** pointer to the open pipe but will return the null pointer value **NULL** if an error is encountered.

popen is typically used by programs that need to call another program and pass substantial amounts of data to that program. For example, in the following program fragment **popen** creates a new process for the **cat** command and opens a pipe for writing.

```
FILE *pstrm;

pstrm = popen("cat >response","w");
```

The new file pointer given by **pstrm** links the standard input of the command with data written by the program. Data written to the pipe using **pstrm** will be used as input by the **cat** command.

Reading and Writing to a Process

fscanf, fprintf, and other stream functions may be used to read from or write to a pipe opened by the **popen** function. These functions have the same form as described in Chapter 2.

The **fscanf** function can be used to read from a pipe opened for reading. For example, in the following program fragment **fscanf** reads from the pipe given by **pstrm.**

```
FILE *pstrm;
char name[20];
int number;

pstrm = popen("cat","r");
fscanf(pstrm, "%s %d", name, &number);
```

This pipe is connected to the standard output of the **cat** command, so **fscanf** reads the first name and number written by **cat** to its standard output.

The **fprintf** function can be used to read from a pipe opened for writing. For example, in the following program fragment **fprintf** writes the string pointed to by **buf** to the pipe given by **pstrm.**

```
FILE *pstrm;
char buf[MAX];
...
pstrm = popen("wc", "w");
fprintf(pstrm, "%s", buf);
```

This pipe is connected to the standard input of the **wc** command, so the command reads and counts the contents of **buf.**

Closing a Pipe

The **pclose** function closes the pipe opened by the **popen** function. The function call has the form

```
pclose(stream)
```

where **stream** is the file pointer of the pipe to be closed. The function normally returns the exit status of the command issued as the first argument of its corresponding **popen** but will return the value -1 if the pipe was not opened by **popen.**

For example, in the following program fragment **pclose** closes the pipe given by **pstrm** if the end-of-file value **EOF** has been found in the pipe.

```
FILE *pstrm;

if (feof(pstrm))
        pclose (pstrm);
```

Opening a Low-Level Pipe

The **pipe** function opens a pipe for both reading and writing. The function call has the form

```
pipe(fd)
```

where **fd** is a pointer to an array of two **int** elements. Each element receives one file descriptor. The first element receives the file descriptor for the reading side of the pipe; the second element receives the file descriptor for the writing side. The function normally returns 0 but will return -1 if an error is encountered. For example, in the following program fragment **pipe** creates two file descriptors if no error is encountered.

```
int chan[2];

if (pipe(chan) = = -1)
        exit(2);
```

The array element **chan[0]** receives the file descriptor for the reading side of the pipe, and **chan[1]** receives the descriptor for the writing side.

pipe is typically used to open a pipe in preparation for linking it to a child process. For example, the following program fragment creates a child process if it successfully creates a pipe.

```
int fd[2];

if (pipe(fd) ! = -1)
        if ( fork() = = 0 )
                close(fd[1]);
```

Note that the child process closes the writing side of the pipe. The parent can now pass data to the child by writing to the pipe; the child can retrieve the data by reading the pipe.

Reading from and Writing to a Low-Level Pipe

The **read** and **write** input and output functions can be used to read and write characters to a low-level pipe. These functions have the same form and operation described in Chapter 2, "Standard I/O Library."

read can be used to read from the read side of an open pipe. For example, in the following program fragment **read** reads **MAX** characters from the read side of the pipe given by **chan.**

```
int chan[2];
char buf[MAX];
int number;

pipe(chan);
number = read(chan[0], buf, MAX);
```

In this example, **read** stores the characters in the array **buf.**

Note that unless the end-of-file character **EOF** is encountered, a **read** call waits for the given number of characters to be read before returning.

write can be used to write to the write side of a pipe. For example, in the following program fragment **write** writes **MAX** characters from the character array **buf** to the writing side of the pipe given by **chan.**

```
int chan[2];
char buf[MAX];
int number;
...
pipe(chan);
number = write(chan[1], buf, MAX);
```

If the **write** function finds that a pipe is too full, it waits until some characters have been read before completing its operation.

Closing a Low-Level Pipe

The **close** function can be used to close the reading or the writing side of a pipe. The function has the same form and operation as described in Chapter 2, "Standard I/O Library." For example, the function call

```
close(chan[0])
```

closes the reading side of the pipe given by **chan**, and the call

```
close(chan[1])
```

closes the writing side.

The system copies the end-of-file value **EOF** to a pipe when the process that made the original pipe and every process created by that process has closed the writing side of the pipe. This means, for example, that if a parent process is sending data to a child process through a pipe and closes the pipe to signal the end of the file, the child process will not receive the **EOF** value unless or until it has closed its own write side of the pipe.

Program Examples

This section shows how to use the process control functions with the low-level **pipe** function to create functions similar to the **popen** and **pclose** functions.

The first example is a modified version of the **popen** function. The modified function identifies the new pipe with a file descriptor rather than a file pointer. It also requires a **mode** argument rather than a **type** argument, where the mode is 0 for reading or 1 for writing.

```
#include <stdio.h>

#define READ  0
#define WRITE  1
#define tst(a, b)    (mode = = READ ? (b) : (a))
static int popen_pid;

popen(cmd, mode)
char  *cmd;
int   mode;
{
        int p[2];

        if (pipe(p) < 0)
                return(NULL);
```

```
        if ((popen_pid = fork()) = = 0) {
                close(tst(p[WRITE], p[READ]));
                close(tst(0, 1));
                dup(tst(p[READ], p[WRITE]));
                close(tst(p[READ], p[WRITE]));
                execl("/bin/sh", "sh", "-c", cmd, 0);
                exit(1);  /* sh cannot be found */
        }
        if (popen_pid = = -1)
                return(NULL);

        close(tst(p[READ], p[WRITE]));
        return(tst(p[WRITE], p[READ]));
}
```

The function creates a pipe with the **pipe** function first. It then uses the **fork** function to create two copies of the original process. Each process has its own copy of the pipe. The child process decides whether it is supposed to read or write through the pipe, then closes the other side of the pipe and uses **execl** to overlay the new process and execute the desired program. The parent closes the side of the pipe it does not use.

The sequence of **close** functions in the child process is a trick used to link the standard input or output of the child process to the pipe. The first **close** determines which side of the pipe should be closed and closes it. If **mode** is **WRITE,** the writing side is closed; if **READ,** the reading side is closed. The second **close** closes the standard input or output depending on the mode. If the mode is **WRITE,** the input is closed; if **READ,** the output is closed. **dup** creates a duplicate of the side of the pipe still open. Since the standard input or output was closed immediately before this call, this duplicate receives the same file descriptor as the standard file. The system always chooses the lowest available file descriptor for a newly opened file. Since the duplicate pipe has the same file descriptor as the standard file, it becomes the standard input or output file for the process. Finally, the last **close** closes the original pipe, leaving only the duplicate.

The following example is a modified version of the **pclose** function. The modified version requires a file descriptor as an argument rather than a file pointer.

```
#include <signal.h>

pclose(fd)   /* close pipe fd */
int fd;
{
        int r, status;
        int (*hstat)(), (*istat)(), (*qstat)();
        extern int popen_pid;

        close(fd);

        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        hstat = signal(SIGHUP, SIG_IGN);
```

```
        while ((r = wait(&status)) != popen_pid && r != -1)
                ;
        if (r == -1)
                status = -1;

        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        signal(SIGHUP, hstat);

        return(status);
}
```

The function closes the pipe first. It then uses a **while** statement to wait for the child process given by **popen_pid.** If other child processes terminate while it waits, it ignores them and continues to wait for the given process. It stops waiting as soon as the given process terminates or if no child process exists. The function returns the termination status of the child or returns -1 if an error was detected.

The **signal** function calls used in this example ensure that no interrupts interfere with the waiting process. The first set of functions causes the process to ignore the interrupt, quit, and hang up signals. The last set restores the signals to their original status. The **signal** function is described in detail in Chapter 7, "Signals."

Note that both example functions use the external variable **popen_pid** to store the process ID of the child process. If more than one pipe is to be opened, the **popen_pid** value must be saved in another variable before each call to **popen,** and this value must be restored before calling **pclose** to close the pipe. The functions can be modified to support more than one pipe by changing the **popen_pid** variable to an array indexed by the file descriptor.

FIFOs

XENIX Release 3 supports another kind of pipe, called a *FIFO,* also known as a "named pipe." FIFOs are created as special files in the file system, using the **mknod** function or the **mknod** command. Like other files, FIFOs can be opened, written, read, and closed. Like pipes, data written to a FIFO by one process can be read (and removed) by another. Only data that has been written but not yet read is stored in a FIFO, in strict first-in-first-out order. Reading data from a FIFO removes it from the FIFO. Because FIFOs are implemented in the file system, data in a FIFO is better protected from system crashes, and also can be independent of any particular process. A good example of using a FIFO is transaction logging for a data base system. Transaction records can be written to the FIFO and periodically read by an archiving process that writes them to tape.

The **O_NDELAY** flag in the open flags modifies the behavior of the **open, read,** and **write** functions for FIFOs. See these functions in Appendix C, "System Functions," for more detail. Appendix C also describes the **mknod** function used to create a FIFO. The corresponding **mknod** command is described in "Commands" in the *XENIX 286 Reference Manual.*

intel®

CHAPTER 7
SIGNALS

This chapter explains how to use C library functions to process signals sent to a program by the XENIX system. A signal is the system's response to an unusual condition that occurs during execution of a program, such as a user pressing the INTERRUPT key or the system detecting an illegal operation. A signal interrupts normal execution of the program and initiates an action such as terminating the program or displaying an error message.

The **signal** function of the standard C library lets a program define the action of a signal. The function can be used to disable a signal to prevent it from affecting the program. It can also be used to give a signal a user-defined action.

The **signal** function is often used with the **setjmp** and **longjmp** functions to redefine and reshape the action of a signal. These functions allow programs to save and restore the execution state of a program, giving a program a means to jump from one state of execution to another without a complex assembly language interface.

To use the **signal** function, you must add the line

```
#include <signal.h>
```

to the beginning of the program. The **signal.h** file defines the various constants used as arguments by the function. To use the **setjmp** and **longjmp** functions you must add the line

```
#include <setjmp.h>
```

to the beginning of the program. The **setjmp.h** file contains the declaration for the type **jmp_buf**, a template for saving a program's current execution state.

Using the signal Function

The **signal** function changes the action of a signal from its current action to a given action. The function has the form

```
signal(sigtype, ptr)
```

where **sigtype** is an integer or constant that defines the signal to be changed, and **ptr** is a pointer to the function defining the new action or a constant giving a predefined action. The function always returns a pointer value. This pointer defines the signal's previous action and may be used in subsequent calls to restore the signal to its previous value.

ptr may be **SIG_IGN** to indicate no action (ignore the signal) or **SIG_DFL** to indicate the default action. The **sigtype** may be **SIGINT** for interrupt signal, caused by pressing the INTERRUPT key, **SIGQUIT** for quit signal, caused by pressing the QUIT key, or **SIGHANG** for hangup signal, caused by hanging up the line when connected to the system by a modem. (Other constants for other signals are given in **signal** in Appendix C.)

For example, the function call

```
signal(SIGINT, SIG_IGN)
```

changes the action of the interrupt signal to no action. The signal will have no effect on the program. The default action is usually to terminate the program.

The following sections show how to use the **signal** function to disable, change, and restore signals.

Disabling a Signal

You can disable a signal, i.e., prevent it from affecting a program, by using the **SIG_IGN** constant with **signal.** The function call has the form

```
signal(sigtype, SIG_IGN)
```

where **sigtype** is the manifest constant of the signal you wish to disable. For example, the function call

```
signal(SIGINT, SIG_IGN);
```

disables the interrupt signal.

The function call is typically used to prevent a signal from terminating a program executing in the background (e.g., a child process that is not using the terminal for input or output). The system passes signals generated from keystrokes at a terminal to all programs that have been invoked from that terminal. This means that pressing the INTERRUPT key to stop a program running in the foreground will also stop a program running in the background if it has not disabled that signal. For example, in the following program fragment **signal** is used to disable the interrupt signal for the child.

```
#include <signal.h>
main ()
{
        if ( fork() = = 0) {
                signal(SIGINT, SIG_IGN);
                /* Child process. */
                }
else {
        /* Parent process. */
        }
}
```

This call does not affect the parent process, which continues to receive interrupts as before. Note that if the parent process is interrupted, the child process continues to execute until it reaches its normal end.

Restoring a Signal's Default Action

You can restore a signal to its default action by using the **SIG_DFL** constant with **signal.** The function call has the form

```
signal(sigtype, SIGDFL)
```

where **sigtype** is the constant defining the signal you wish to restore. For example, the function call

```
signal(SIGINT, SIG_DFL)
```

restores the interrupt signal to its default action.

The function call is typically used to restore a signal after it has been temporarily disabled to keep it from interrupting critical operations. For example, in the following program fragment the second call to **signal** restores the signal to its default action.

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

main ()
{
        FILE *fp;
        char record[BUF];

        signal (SIGINT, SIG_IGN);
        fp = fopen("my_file", "r");
        fread(record, BUF, 1, fp);
        signal (SIGINT, SIG_DFL);

}
```

In this example, the interrupt signal is ignored while a record is read from the file **my_file.**

Catching a Signal

You can catch a signal and define your own action for it by providing a function that defines the new action and giving the function as an argument to **signal.** The function call has the form

```
signal(sigtype, newptr)
```

where **sigtype** is the constant defining the signal to be caught, and **newptr** is a pointer to the function defining the new action. For example, the function call

```
signal(SIGINT, catch)
```

changes the action of the interrupt signal to the action defined by the function named **catch.**

The function call is typically used to let a program do additional processing before terminating. In the following program fragment, the function **catch** defines the new action for the interrupt signal.

```
#include <signal.h>

main ()
{
        int catch();

        printf("Press INTERRUPT key to stop.\n");
        signal (SIGINT, catch);
        while (1) {
                /* Body */
        }
}

catch ()
{
        printf("Program terminated.\n");
        exit(1);
}
```

The **catch** function prints the message "Program terminated" before stopping the program with the **exit** function.

A program may redefine the action of a signal at any time. Thus, many programs define different actions for different conditions. For example, in the following program fragment the action of the interrupt signal depends on the return value of a function named **keytest.**

```
#include <signal.h>

main()
{
        int catch1(), catch2();

        if (keytest() = = 1)
                signal(SIGINT, catch1);
        else
                signal(SIGINT, catch2);

}
```

Later the program may change the signal to the other action or even a third action.

When using a function pointer in the **signal** call, you must make sure that the function name is defined before the call. In the program fragment shown above, **catch1** and **catch2** are explicitly declared at the beginning of the main program function. Their formal definitions are assumed to appear after the **signal** call.

Restoring a Signal

You can restore a signal to its previous value by saving the return value of a **signal** call, then using this value in a subsequent call. The function call has the form

```
signal(sigtype, oldptr)
```

where **sigtype** is the constant defining the signal to be restored and **oldptr** is the pointer value returned by a previous **signal** call.

The function call is typically used to restore a signal when its previous action may be one of many possible actions. For example, in the following program fragment the previous action depends solely on the return value of a function **keytest.**

```
#include <signal.h>
main()
{
        int catch1(), catch2();
        int (*savesig)();
        if (keytest() = = 1)
                signal(SIGINT, catch1);
        else
                signal(SIGINT, catch2);
        savesig = signal (SIGINT, SIG_IGN);
        compute();
        signal(SIGINT, savesig);
}
```

In this example, the old pointer is saved in the variable **savesig.** This value is restored after the function **compute** returns.

Program Example

This section shows how to use the **signal** function to create a modifed version of the **system** function. In this version, **system** disables all interrupts in the parent process until the child process has completed its operation. It then restores the signals to their previous actions.

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

system(s)    /* run command string s */
char *s;
{
        int status, pid, w;
        register int (*istat)(), (*qstat)();

        if ((pid = fork()) = = 0) {
                execl("/bin/sh", "sh", "-c", s, NULL);
                exit(127);
        }
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        while ((w = wait(&status)) != pid && w != -1) ;

        if (w = = -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        return(status);
}
```

Note that the parent uses the **while** statement to wait until the child's process ID **pid** is returned by **wait.** If **wait** returns the error code -1 no more child processes are left, so the parent returns the error code as its own status.

Controlling Execution with Signals

Signals need not be used only to terminate programs. Many signals can be redefined to delay their actions or even cause actions that terminate a portion of a program without terminating the entire program. The following sections describe how signals can be caught and used to provide control of a program.

Delaying a Signal's Action

You can delay the action of a signal by catching the signal and redefining its action to be nothing more than setting a globally-defined flag. Such a signal does nothing to the current execution of the program. Instead, the program continues uninterrupted until it can test the flag to see if a signal has been received. It can then respond according to the value of the flag.

The key to a delayed signal is that all functions return execution to the exact point at which the program was interrupted. If the function returns normally, the program continues execution just as if no signal occurred.

Delaying a signal is especially useful in programs that must not be stopped at an arbitrary point. If, for example, a program updates a linked list, the action of a signal can be delayed to prevent the signal from interrupting the update and destroying the list. For example, in the following program fragment the function **delay** used to catch the interrupt signal sets the globally-defined flag **sigflag** and returns immediately to the point of interruption.

```
#include <signal.h>
int sigflag;

main ()
{
        int delay ();
        int (*savesig)();
        extern int sigflag;

        signal(SIGINT, delay); /* Delay the signal. */
        updatelist();
        savesig = signal(SIGINT, SIG_IGN); /* Disable the signal. */
        if (sigflag)
                /* Process delayed signals if any. */

}

delay ()
{
        extern int sigflag;

        sigflag = 1;
}
```

In this example, if the signal is received while **updatelist** is executing, it is delayed until after **updatelist** returns. Note that the interrupt signal is disabled before processing the delayed signal to prevent a change to **sigflag** when it is being tested.

Note that the system automatically resets a signal to its default action immediately after the signal is processed. If your program delays a signal, make sure that the signal is redefined after each interrupt. Otherwise, the default action will be taken on the next occurrence of the signal.

Using Delayed Signals with System Functions

When a delayed signal is used to interrupt the execution of a XENIX system function, such as **read** or **wait**, the system forces the function to stop and return an error code. This action, unlike actions taken during execution of other functions, causes all processing performed by the system function to be discarded. A serious error can occur if a program interprets a system function error caused by delayed signals as a normal error. For example, if a program receives a signal when reading the terminal, all characters read before the interruption are lost, making it appear as though no characters were typed.

Whenever a program intends to use delayed signals during calls to system functions, the program should include a check of the function return values to ensure that an error was not caused by an interruption. In the following program fragment, the program checks the current value of the interrupt flag **intflag** to make sure that the value EOF returned by **getchar** actually indicates the end of the file.

```
if (getchar() = = EOF)
        if (intflag)
                /* EOF caused by interrupt */
        else
                /* true end-of-file */
```

Using Signals in Interactive Programs

Signals can be used in interactive programs to control the execution of the program's various commands and operations. For example, a signal may be used in a text editor to interrupt the current operation (e.g., displaying a file) and return the program to a previous operation (e.g., waiting for a command).

To provide this control, the function that redefines the signal's action must be able to return execution of the program to a meaningful location, not just the point of interruption. The standard C library provides two functions to do this: **setjmp** and **longjmp.** The **setjmp** function saves a copy of a program's execution state. The **longjmp** function changes the current execution state to a previously saved state. The functions cause a program to continue execution at an old location with old register values and status as if no operations had been performed between the time the state was saved and the time it was restored.

The **setjmp** function has the form

```
setjmp(buffer)
```

where **buffer** is the variable to receive the execution state. It must be explicitly declared with type **jmpbuf** before it is used in the call. For example, in the following program fragment **setjmp** copies the execution state of the program to the variable **oldstate** defined with type **jmpbuf.**

```
jmpbuf oldstate;

setjmp(oldstate);
```

Note that after a **setjmp** call, the **buffer** variable contains values for the program counter, the data and address registers, and the process status. These values must not be modified in any way.

The **longjmp** function has the form

```
longjmp(buffer)
```

where **buffer** is the variable containing the execution state. It must contain values previously saved with a **setjmp** function. The function copies the values in the buffer **variable** to the program counter, data and address registers, and the process status table. Execution continues as if it had just returned from the **setjmp** function that saved the previous execution state. For example, in the following program fragment **setjmp** saves the execution state of the program at the location just before the main processing loop and **longjmp** restores it on an interrupt signal.

```
#include <signal.h>
#include <setjmp.h>

main()
{
        jmpbuf sjbuf;
        int onintr();

        setjmp(sjbuf);
        signal(SIGINT, onintr);

        /* main processing loop */
}

onintr()
{
        printf("\nInterrupt\n");
        longjmp(sjbuf);
}
```

In this example, the action of the interrupt signal as defined by **onintr** is to print the message "Interrupt" and restore the old execution state. When an interrupt signal is received in the main processing loop, execution passes to **onintr,** which prints the message and then passes execution back to the main program function, making it appear as though control is returning from the **setjmp** function.

Using Signals in Multiple Processes

The XENIX system passes all signals generated at a given terminal to all programs invoked at that terminal. This means that a program has potential access to a signal even if that program is executing in the background or as a child to some other program. The following sections explain how signals may be used in multiple processes.

Protecting Background Processes

Any program that has been invoked using the shell's background symbol (&) is executed as a background process. Such programs usually do not use the terminal for input or output and complete their tasks silently. Since these programs do not need additional input, the shell automatically disables the signals before executing the program. This means signals generated at the terminal do not affect execution of the program. This is how the shell protects the program from signals intended for other programs invoked from the same terminal.

In some cases, a program that has been invoked as a background process may also attempt to catch its own signals. If it succeeds, the protection from interruption given to it by the shell is defeated, and signals intended for other programs will interrupt the program. To prevent this, any program intended to be executed as a background process should test the current state of a signal before redefining its action. A program should redefine a signal only if the signal is enabled. For example, in the following program fragment the action of the interrupt signal is changed only if the signal is not currently being ignored.

```
#include <signal.h>

main()
{
        int catch();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, catch);

        /* Program body. */
}
```

This step lets a program continue to ignore signals if it is already doing so and change the signal if it is not.

Protecting Parent Processes

A program can create and wait for a child process that catches its own signals only if the program protects itself by disabling all signals before calling the **wait** function. By disabling the signals, the parent process prevents signals intended for the child processes from terminating its call to **wait**. This prevents serious errors that may result if the parent process continues execution before the child processes are finished.

For example, in the following program fragment the interrupt signal is disabled in the parent process immediately after the child is created.

```
#include <signal.h>

main ()
{
        int (*saveintr)();

        if (fork () = = 0)
                execl( ... );

        saveintr = signal (SIGINT, SIG_IGN);
        wait( &status );
        signal (SIGINT, saveintr);
}
```

The signal's action is restored after the **wait** function returns normal control to the parent.

intel®

CHAPTER 8
USING SYSTEM RESOURCES

This chapter describes the standard C library functions that let programs share the resources of the XENIX system. The functions give a program the means to queue for the use and control of a given resource and to synchronize its use with use by other programs.

In particular, this chapter explains how to

- Allocate memory for dynamically required storage
- Lock a file to ensure exclusive use by a program
- Use semaphores to control access to a resource
- Share data space to allow interaction between programs

Allocating Space

Some programs require significant changes to the size of their allocated memory space during different phases of their execution. The memory allocation functions of the standard C library let programs allocate space dynamically. This means a program can request a given number of bytes of storage for its exclusive use at the moment it needs the space and free this space after it has finished using it.

There are four memory allocation functions: **malloc, calloc, realloc,** and **free.** The **malloc** and **calloc** functions are used to allocate space for the first time, allocating a given number of bytes and returning a pointer to the new space. The **realloc** function reallocates an existing space, allowing it to be used in a different way. The **free** function returns allocated space to the system.

Allocating Space for a Variable

The **malloc** function allocates space for a variable containing a given number of bytes. The function call has the form

```
malloc(size)
```

where **size** is an unsigned number that gives the number of bytes to be allocated. For example, the function call

```
table = malloc(4)
```

allocates four bytes of storage. The function normally returns a pointer to the starting address of the allocated space but will return **NULL** if there is not enough space to allocate.

The function can be used to allocate storage for strings that vary in length. For example, in the following program fragment **malloc** is used to allocate space for ten different strings, each of a different length.

```
int i;
char *temp[100];
char *strings[10];
unsigned isize;

for ( i = 0; i<10; i + + ) {
        scanf("%s", temp);
        isize = strlen(temp) + 1; /* Allow for null terminator. */
        string[i] = malloc(isize);
        if (string[i] ! = NULL)
                strcpy(string[i], temp);
        else
                /* error processing */
}
```

In this example, the strings are read from the standard input. The function **strlen** is called to get the size in bytes of each string, and the **strcpy** function is called to copy each string from **temp** to **string[i]**.

Allocating Space for an Array

The **calloc** function allocates storage for a given array and initializes each element in the new array to zeros. The function call has the form

```
calloc(n, size)
```

where **n** is the number of elements in the array, and **size** is the number of bytes in each element. The function normally returns a pointer to the starting address of the allocated space but will return **NULL** if there is not enough memory. For example, the function call

```
table = calloc(10, 4);
```

allocates space for a 10-element array. Each element has 4 bytes.

The function is typically used in programs that must process large arrays without knowing the size of an array in advance. For example, in the following program fragment **calloc** is used to allocate storage for an array of values read from the standard input.

```
int i;
long table[];
unsigned inum;

scanf("%d", &inum);
table = calloc(inum, sizeof(long));
if (table = = NULL)
        /* error processing */
else
        for (i = 0; i<inum; i + + )
                scanf("%D", table[i]);
```

Note that the number of elements is read from the standard input before the elements are read. Also note the use of the C operator **sizeof** to specify the element size rather than using a machine-dependent number.

Reallocating Space

The **realloc** function changes the size of a block and returns a pointer to the (possibly moved) block. The block's contents are unchanged up to the lesser of the new and old sizes. The function call has the form

```
realloc(ptr, size)
```

where **ptr** is a pointer to the starting address of the space to be reallocated, and **size** is an unsigned number giving the new size in bytes of the reallocated space. The function normally returns a pointer to the starting address of the allocated space but will return **NULL** if there is not enough space to allocate. If NULL is returned, the block referenced by **ptr** may be destroyed.

realloc can be used to expand or contract a dynamic storage block as needed. For example, a block used for a compiler's symbol table can be reallocated with a larger size if it becomes full, retaining its previous contents at the start of the new larger block.

Freeing Unused Space

The **free** function frees unused memory space previously allocated by a **malloc, calloc,** or **realloc** function call. The function call has the form

```
free(ptr)
```

where **ptr** is the pointer to the starting address of the space to be freed. This pointer must be a non-**NULL** value returned by a **malloc, calloc,** or **realloc** function.

The function is used exclusively to free space that is no longer used or to free space to be used for other purposes. For example, in the following program fragment **free** frees the space pointed to by **table** if the first element is equal to zero.

```
extern
        int table[];

if (table[0] = = 0)
        free(table);
```

Locking Files

Locking a file is a way to synchronize file use when several processes may require access to a single file. The standard C library provides one file locking function, the **locking** function. This function locks or unlocks a specified section of a file, preventing all other processes that wish to use the section from gaining access. A process may lock an entire file or only a small portion. In any case, only the locked section is protected; all other sections may be accessed by other processes as usual.

File locking protects a file from the damage that may be caused if several processes try to read or write to the file at the same time. It also provides unhindered access to any portion of a file for a controlling process. Before a file can be locked, however, it must be prepared using the **open** and **lseek** functions described in Chapter 2, "Standard I/O Library." To use the **locking** function, you must add the line

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

to the beginning of your program. The file **sys/locking.h** contains definitions for the modes used with the function.

Preparing a File for Locking

Before a file can be locked, it must first be opened using the **open** function, then properly positioned by using the **lseek** function to move the file's character pointer to the first byte to be locked.

The **open** function is used once at the beginning of the program to open the file. The **lseek** function may be used any number of times to move the character pointer to each new section to be locked. For example, the following statements prepare a region beginning at byte position 1,024 in the file **reservations** for locking.

```
int fd;
fd = open("reservations", O__RDONLY);
lseek(fd, 1024, 0);
```

Locking a File

The **locking** function locks one or more bytes of a specified file, beginning at the current character position in the file. The function call has the form

```
locking(filedes, mode, size)
```

where **filedes** is the file descriptor of the file to be locked, **mode** is an integer value that defines the type of lock to be applied to the file, and **size** is a long integer value giving the size in bytes of the file section to be locked or unlocked. If **size** is zero, or extends beyond the end of file, then all the file from the pointer position to the end of file is locked or unlocked. **mode** may be **LK_LOCK** for locking the given bytes, or **LK_UNLCK** for unlocking them. For example, in the following program fragment **locking** locks 100 bytes at the current character pointer position in the file given by **fd.**

```
fd = open("data", O_RDWR);
locking(fd, LOCK, 100);
```

The function normally returns the number of bytes locked or unlocked but will return -1 if it encounters an error.

For more information on this subject, see the entry **locking** in Appendix C.

Program Example

This section shows how to lock and unlock a small section in a file using the **locking** function. In the following program, the function locks 100 bytes in the file **data,** which is opened for reading and writing. The locked portion of the file is accessed, then **locking** is used again to unlock the file.

```
#include <sys/locking.h>
main()
{
int fd, err;
char *data;
        fd = open("data", O_RDWR);   /* Open data for R/W */
        if (fd = = -1)
                perror("");
        else {
                lseek(fd, 100L, 0);   /* Seek to pos 100 */
                err = locking(fd, LK_LOCK, 100L); /* Lock bytes 100-200 */
                if (err = = -1) {
                        /* process error return */
                        }
                else {
                /* read or write bytes 100 - 200 in the file */
                        lseek(fd, 100L, 0); /* Seek to pos 100 */
                        locking(fd, LK_UNLCK, 100L); /* Lock bytes 100-199 */
        }
}
```

Using Semaphores

The standard C library provides a group of functions, called the semaphore functions, that can be used to control access to given system resources. The functions create and use semaphores. Semaphores are special files that have names and entries in the file system but contain no data. The semaphore functions restrict access to a semaphore to one process at a time; all other processes wishing to access the semaphore must wait. This means a semaphore can be used to control a process's access to a system resource by requiring that process to obtain control of the semaphore before performing any tasks with the resource.

There are five semaphore functions: **creatsem, opensem, waitsem, nbwaitsem,** and **sigsem.** The **creatsem** function creates a semaphore. The semaphore may then be opened and used by other processes. Another process can open a semaphore with the **opensem** function. Processes request control of a semaphore with the **waitsem** or **nbwaitsem** function. Once a process has control of a semaphore, it can carry out tasks using the associated resource. All other processes must wait. When a process has finished using the resource, it can relinquish control of the semaphore with the **sigsem** function. This lets other processes obtain control of the semaphore and use the corresponding resource.

Creating a Semaphore

The **creatsem** function creates a semaphore, returning a semaphore number that can be used in subsequent semaphore functions. The function call has the form

```
creatsem(sem_name, mode)
```

where **sem_name** is a character pointer to the path name of the semaphore, and **mode** is an integer value that defines the access mode of the semaphore. Semaphore names have the same syntax as regular file names. The names must be unique. The function normally returns an integer semaphore number that can be used in subsequent semaphore functions to refer to the semaphore. The function returns -1 if it encounters an error, such as attempting to create a semaphore that already exists or using the name of an existing regular file.

The function is typically used at the beginning of one process to clearly define the semaphores it intends to share with other processes. For example, in the following program fragment **creatsem** creates a semaphore named "tty1" before preceding with its tasks.

```
main()
{
int tty1;
FILE *ftty1;

        tty1 = creatsem("tty1", 0777);
        ftty1 = fopen("/dev/tty01", "w");
        /* Program body. */
}
```

Note that **fopen** is used immediately after **creatsem** to open the file **/dev/tty01** for writing. This is one way to make the association between a semaphore and a device clear.

The mode 0777 defines the semaphore's access permissions. The permissions are similar to the permissions of a regular file. A semaphore may have read permission for the owner, for users in the same group as the owner, and for all other users. The write and execution permissions have no meaning. Thus, 0777 means read permission for all users.

No more than one process ever need create a given semaphore; all other processes simply open the semaphore with the **opensem** function. Once created or opened, a semaphore may only be accessed using the **waitsem, nbwaitsem,** or **sigsem** functions. The **creatsem** function may be used more than once during execution of a process. In particular, it can be used to reset a semaphore if a process fails to relinquish control before terminating.

Opening a Semaphore

The **opensem** function opens an existing semaphore for use by the calling process. The function call has the form

```
opensem(sem_name)
```

where **sem_name** is a pointer to the path name of the semaphore. The function returns a semaphore number that may be used in subsequent semaphore functions to refer to the semaphore. The function returns -1 if it encounters an error, such as trying to open a semaphore that does not exist or using the name of an existing regular file.

The function is typically used by a process just before it requests control of a given semaphore. A process need not use the function if it also created the semaphore. For example, in the following program fragment **opensem** is used to open the semaphore named **semaphore1.**

```
int sem1;

if ( (sem1 = opensem("semaphore1")) != -1)
        waitsem(sem1);
```

In this example, the semaphore number is assigned to the variable **sem1.** If the number is not -1, then **sem1** is used in the semaphore function **waitsem,** which requests control of the semaphore.

A semaphore must not be opened more than once during execution of a process.

Requesting Control of a Semaphore

The **waitsem** function requests control of a given semaphore for the calling process. If the semaphore is available, control is given immediately. Otherwise, the process waits. The function call has the form

```
waitsem(sem_num)
```

where **sem_num** is the semaphore number of the semaphore to be controlled. If the semaphore is not available (if it is under control of another process), the function forces the requesting process to wait. If other processes are already waiting for control, the request is placed next in a queue of requests. When the semaphore becomes available, the first waiting process receives it. When this process relinquishes control, the next process receives control, and so on. The function returns -1 if it encounters an error such as requesting a semaphore that does not exist or requesting a semaphore that is locked by a dead process.

The function is used whenever a given process wishes to access the device or system resource associated with the semaphore. For example, in the following program fragment **waitsem** signals the intention to write to the file given by "tty1".

```
int tty1;
FILE *ftty1;
...
waitsem(tty1);
fprintf(ftty1, "Changing tty driver\n");
```

Checking the Status of a Semaphore

The **nbwaitsem** (for "non-blocking wait") function checks the current status of a semaphore. If the semaphore is not available, the function returns the error value -1. Otherwise, it gives immediate control of the semaphore to the calling process. The function call has the form

```
nbwaitsem(sem_num)
```

where **sem_num** is the semaphore number of the semaphore to be checked. The function returns -1 if it encounters an error such as requesting a semaphore that does not exist. The function also returns -1 if the process controlling the requested semaphore terminated without relinquishing control of the semaphore. The case of the semaphore being held by another process is distinguished by an **errno** value of **ENAVAIL.**

nbwaitsem is typically used in place of **waitsem** to take control of a semaphore only if it is available.

Relinquishing Control of a Semaphore

The **sigsem** function causes a process to relinquish control of a semaphore and to signal this fact to the next process waiting for the semaphore. The function call has the form

```
sigsem(sem_num)
```

where **sem_num** is the semaphore number of the semaphore to relinquish. The semaphore must have been previously created or opened by the process. Furthermore, the process must have previously taken control of the semaphore with the **waitsem** or **nbwaitsem** function. The function returns -1 if it encounters an error such as trying to relinquish a semaphore that does not exist or that it does not control.

sigsem is typically used after a process has finished accessing the corresponding device or system resource. This allows waiting processes to take control. For example, in the following program fragment **sigsem** signals the end of control of the semaphore **tty1.**

```
main ()
{
int tty1;
FILE *temp, *ftty1;
...
waitsem(tty1);
while ((c = fgetc(temp)) != EOF)
        fputc(c, ftty1);
sigsem(tty1);
```

This example also signals the end of the copy operation to the semaphore's corresponding device, given by **ftty1.**

Note that a semaphore can become locked to a dead process if the process fails to signal the semaphore before terminating. In such a case, the semaphore must be reset by using the **creatsem** function.

Using Shared Memory

Shared memory is a method by which one process shares its allocated data space with another process. Shared memory allows processes to pool information in a central location and directly access that information without the burden of creating pipes or temporary files.

The standard C library provides several functions to access and control shared memory. The **sdget** function creates and/or adds a shared memory segment to a given process's data space. To access a segment, a process must signal its intention with the **sdenter** function. Once a segment has completed its access, it can signal that it is finished using the segment with the **sdleave** function. The **sdfree** function is used to remove a segment from a process's data space. The **sdgetv** and **sdwaitv** functions are used to synchronize processes when several are accessing the segment at the same time.

To use the shared data functions, you must add the line

```
#include <sd.h>
```

at the beginning of the program. The **sd.h** file contains definitions for the constants and macros used by the functions.

Creating a Shared Data Segment

The **sdget** function creates a shared data segment for the current process, or if the segment already exists, attaches the segment to the data space of the current process. The function call has the form

```
sdget(path, flag [, size, mode ] )
```

where **path** is a character pointer, **flag** is an integer value that defines how the segment should be created or attached, **size** is an integer value that defines the size in bytes of the segment to be created, and **mode** is an integer value that defines the access permissions to be given to the segment if created. The **size** and **mode** values are used only when creating a segment. **flag** may be **SD_RDONLY** for attaching the segment for reading only, **SD_WRITE** for attaching the segment for reading and writing, **SD_CREAT** for creating the segment given by **path** if it does not already exist, or **SD_UNLOCK** for allowing simultaneous access by multiple processes. The values can be combined by logically ORing them. The **SD_UNLOCK** value is used only if the segment is created. The function returns the address of the segment if it has been successfully created or attached. Otherwise, the function returns -1 if it encounters an error.

The function is most often used to create a segment to be shared by another process. The function may then be used in the other process to attach the segment to its data space. For example, in the following program fragment **sdget** creates a segment and assigns the address of the segment to the variable **shared.**

```
#include <sd.h>

main()
{
char *shared, *spath;

        shared = sdget( spath, SD__CREAT, 512, 0777 );
}
```

When the segment is created, the size 512 and the mode 0777 are used to define the segment's size in bytes and access permissions. Access permissions are similar to permissions given to regular files. A segment may have read or write permission for the owner of the process, for users belonging to the same group as the owner, and for all other users. Execute permission for a segment has no meaning. For example, the mode 0666 means read and write permission for everyone, but 0660 means read and write permissions for the owner and group processes only. When first created, a segment is filled with zeros.

Note that the **SD_UNLOCK** flag used on systems without hardware support for shared data may severely degrade the execution performance of the program.

Entering Shared Data Segment

The **sdenter** function signals a process's intention to access the contents of a shared data segment. A process cannot effectively access the contents of the segment unless it enters the segment. The function call has the form

```
sdenter(addr [, flag])
```

where **addr** is a pointer to the segment to be accessed, and **flag** is an optional integer value that defines how the segment is to be accessed. The **flag** may be **SD_RDONLY** for indicating read-only access to the segment, or **SD_NOWAIT** for returning an error if the segment is locked and another process is currently accessing it. These values may also be combined by logically ORing them.

The function normally waits for the segment to become available before allowing access to it. A segment is not available if the segment has been created without the **SD_UNLOCK** flag and another process is currently accessing it.

In general, it is unwise to stay in a shared data segment any longer than it takes to examine or modify the desired location. The **sdleave** function should be called after each access. When in a shared data segment, a program should avoid using system functions. System functions can disrupt the normal operations required to support shared data and may cause some data to be lost. In particular, if a program creates a shared data segment that cannot be shared simultaneously, the program must not call the **fork** function when it is accessing the segment.

Leaving a Shared Data Segment

The **sdleave** signals a process's intention to leave a shared data segment after reading or modifying its contents. The function call has the form

```
sdleave(addr)
```

where **addr** is a pointer to the desired segment. The function returns -1 if it encounters an error, otherwise it returns 0.

sdleave should be called after each access to shared data to terminate the access. If the segment's lock flag is set, the function must be called after each access to allow other processes to access the segment. For example, in the following program fragment **sdleave** terminates each access to the segment given by **shared.**

```
#include <sd.h>

main()
{
char *shared;

        ...
        sdenter(shared);
        /* write to segment */
        sdleave(shared);
        ...
}
```

Getting the Current Version Number

The **sdgetv** function returns the current version number of the given data segment. The function call has the form

```
sdgetv(addr)
```

where **addr** is a character pointer to the desired segment. A segment's version number is initially zero, but is incremented by one whenever a process leaves the segment using the **sdleave** function. Thus, the version number is a record of the number of times the segment has been accessed. The function's return value is always an integer. It returns -1 if it encounters an error.

The function is typically used to choose an action based on the current version number of the segment. For example, in the following program fragment **sdgetv** determines whether or not **sdenter** should be used to enter the segment given by **shared.**

```
#include <sd.h>

main ()
{
char *shared;
...
if (sdgetv(shared) > 10)
        sdenter(shared);
```

In this example, the segment is entered if the current version number of the segment is greater than 10.

Waiting for a Version Number

The **sdwaitv** function causes a process to wait until the version number for the given segment is no longer equal to a given version number. The function call has the form

```
sdwaitv(addr, vnum)
```

where **addr** is a character pointer to the desired segment, and **vnum** is an integer value that defines the version number to wait on. The function normally returns the new version number; it returns -1 if it encounters an error. The return value is always an integer.

The function is typically used to synchronize the actions of two separate processes. For example, in the following program fragment the program waits while the process corresponding to version number 3 performs its operations in the segment.

```
if ( sdwaitv(sdseg, 3) = = -1 )
        fprintf(stderr, "Cannot find segment\n");
```

If an error occurs while the program is waiting, the example code prints an error message.

Freeing a Shared Data Segment

The **sdfree** function detaches the current process from the given shared data segment. The function call has the form

```
sdfree(addr)
```

where **addr** is a pointer to the segment to be freed. The function returns the integer value 0 if the segment is freed. Otherwise, it returns -1.

If the process is currently accessing the segment, **sdfree** automatically calls **sdleave** to leave the segment before freeing it.

Segments that have been freed by all attached processes are destroyed by the system.

intel®

CHAPTER 9
ERROR PROCESSING

A variety of errors can occur when a program attempts to access the XENIX operating system through the standard C library functions. Errors range from problems with accessing files to allocating memory. In most cases, the system simply reports the error and lets the program decide how to respond. The XENIX system terminates a program only if a serious error has occurred, such as a memory addressing error.

This chapter explains how to process errors, describes the functions and variables a program may use to detect and respond to errors, and shows some ways to handle errors.

Using the Standard Error File

The standard error file is a special output file that can be used by a program to display error messages. The standard error file is one of three standard files (standard input, output, and error) automatically created for the program when it is invoked.

The standard error file, like the standard output, is normally assigned to the user's terminal screen. Thus, error messages written to the file are displayed at the screen. The file can also be redirected by using the shell's redirection symbol (>). For example, the following command redirects the standard error file to the file **errorlist.**

```
% dial 2>errorlist
```

In this case, subsequent error messages are written to the given file.

The standard error file, like the standard input and standard output, has predefined file pointer and file descriptor values. The file pointer **stderr** can be used with stream I/O functions to write data to the error file. The file descriptor **2** may be used with low-level I/O functions to write data to the error file. For example, in the following program fragment, **stderr** is used to write the message "Unexpected end of file." to the standard error file.

```
if ( (c = getchar()) = = EOF)
        fprintf(stderr, "Unexpected end of file.\n");
```

The standard error file is not affected by the shell's pipe symbol (|); even if the standard output of a program is piped to another program, errors generated by the program will still appear at the terminal screen (or in the appropriate file if the standard error output is redirected).

Using the errno Variable

The **errno** variable is a predefined external variable that contains the error number of the most recent XENIX system function error. Errors detected by system functions, such as access permission errors and lack of space, cause the system to assign a distinct number to **errno** and return control to the program. The error number identifies the error condition. **errno** can be read by subsequent statements that process the error.

errno is typically read immediately after a system function has returned an error. In the following program fragment, **errno** is read to determine the course of action after an unsuccessful call to the **open** function.

```
if ( (fd = open("accounts", O_RDONLY)) = = -1 )
        switch (errno) {
                case(EACCES):
                        fd = open("/usr/tmp/accounts", O_RDONLY);
                        break;
                default:
                        exit(errno);
        }
```

In this example, if **errno** is equal to the constant **EACCES,** permission to open the file **accounts** in the current directory is denied, so the file is opened in the directory **/usr/tmp** instead. If **errno** is any other value, the program terminates.

To use **errno** in a program, it must be explicitly defined as an external variable with **int** type. Note that the file **errno.h** contains manifest constant definitions for each error number. These constants may be used in any program in which the line

```
#include <errno.h>
```

is placed at the beginning of the program. The meaning of each error constant is described in the introduction to Appendix C.

Printing Error Messages

The **perror** function copies a short error message describing the most recent system function error to the standard error file. The function call has the form

```
perror(s)
```

where **s** is a pointer to a string containing additional information about the error.

The **perror** function places the given string before the error message and separates the two with a colon (:). The error message corresponds to the current value of the **errno** variable. For example, in the following program fragment **perror** displays the message

```
accounts: Permission denied.
```

if **errno** is equal to the constant **EACCES.**

```
if ( errno = = EACCES ) {
        perror("accounts");
        fd = open("/usr/tmp/accounts", O__RDONLY);
}
```

The error messages displayed by **perror** are stored in an array named **sys_errno,** an external array of character strings. The **perror** function uses the variable **errno** as the index to the array element containing the desired message.

Using Error Signals

Some program errors cause the XENIX system to generate error signals. These signals are passed back to the program that caused the error and normally terminate the program. The most common error signals are **SIGBUS,** the bus error signal; **SIGFPE,** the floating point exception signal; **SIGSEGV,** the segment violation signal; **SIGSYS,** the system call error signal; and **SIGPIPE,** the pipe error signal. Other signals are described in **signal** in Appendix C.

A program can, if necessary, catch an error signal and perform its own error processing by using the **signal** function. This function, as described in Chapter 7, "Signals", can set the action of a signal to a user-defined action. For example, the function call

```
signal(SIGBUS, fixbus);
```

sets the action of the bus error signal to the action defined by the user-supplied function **fixbus.** Such a function usually attempts to remedy the problem or at least display detailed information about the problem before terminating the program.

For details about how to catch, redefine, and restore these signals, see Chapter 7, "Signals."

Encountering System Errors

Programs that encounter serious errors, such as hardware failures or internal errors, generally do not receive detailed reports on the cause of the errors. Instead, the XENIX system treats these errors as "system errors" and reports them by displaying a system error message on the system console. This section briefly describes some aspects of XENIX system errors and how they relate to user programs. For a complete list and description of XENIX system error messages, see the section "Messages" in the *XENIX 286 Reference Manual.*

Most system errors occur during calls to system functions. If the system error is recoverable, the system will return an error value to the program and set the **errno** variable to an appropriate value. No other information about the error is available.

Although the system lets two or more programs share a given resource, such as an I/O device, it does not keep close track of which program is using the resource at any given time. When an error occurs, the system returns an error value to all programs regardless of which caused the error. No information about which program caused the error is available.

System errors that occur during routine I/O operations initiated by the XENIX system itself generally do not affect user programs. Such errors cause the system to display appropriate system error messages on the system console.

Some system errors are not detected by the system until after the corresponding function has returned successfully. Such errors occur when data written to a file by a program has been queued for writing to disk at a more convenient time, or when a portion of data to be read from disk is found to already be in memory and the remaining portion is not read until later. In such cases, the system assumes that the subsequent read or write operation will be carried out successfully and passes control back to the program along with a successful return value. If the subsequent operation is not carried out successfully, it causes a delayed error.

When a delayed error occurs, the system usually attempts to return an error on the next call to a system function that accesses the same file or resource. If the program has already terminated or does not make a suitable call, then the error is not reported.

intel®

APPENDIX A
ASSEMBLY LANGUAGE INTERFACE

This appendix explains how to use 8086/286 assembly language routines with C language programs and functions. In particular, it explains how to call assembly language routines from C language programs and how to call C language functions from an assembly language routine. This gives the assembly language programmer access to all the library functions described in this manual.

C Calling Sequence

To receive values from C language function calls or to pass values to C functions, assembly language routines must follow the C argument passing conventions. C language function calls pass their arguments to the given functions by pushing the value of each argument onto the stack. The call pushes the value of the last argument first and the first argument last. If an argument is an expression, the call computes the expression's value before pushing it onto the stack.

Arguments with **char, int,** or **unsigned** type occupy a single word (16 bits) on the stack. Arguments with **long, float,** or **double** type occupy a double word (32 bits) with the value's low order word occupying the first word. Note that **char** type arguments are sign-extended to **int** type before being pushed on the stack. Similarly, **float** type arguments are sign-extended to **double** type.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word in pushed.

Entering an Assembly Routine

Assembly language routines that receive control from C function calls should preserve the contents of the **bp, si,** and **di** registers and set the **bp** register to the current **sp** register value before proceeding with their tasks. The following example illustrates the recommended instruction sequence for entering an assembly language routine called from C:

```
entry:  push bp
        mov bp,sp
        push di
        push si
```

This is the same sequence used by the C compiler.

If this sequence is used, the last argument pushed by the function call (which is the first argument given in the call's argument list) is at address **4(bp)** in small model programs or address 6(bp) in middle or large model programs (because a long call is used and both CS and IP are pushed).

The above instruction sequence is recommended even for entering a routine that does not modify the **si** or **di** registers, because it allows backtracking with the **adb** program during program debugging.

Return Values

Assembly language routines that return values to C language callers must follow the C return value conventions. C functions place return values with type **int, char,** or **unsigned** in register **ax;** return values with type **long** are placed in **ax** (low word) and **dx** (high word).

To return a structure or a floating point value, C functions place the address of the structure or value in register **ax.** The structure or floating-point value must be in a static memory area.

Exiting a Routine

Assembly language routines that return control to C programs should restore the values of registers **bp, si,** and **di** before returning control. The following instruction sequence can be used:

```
pop si
pop di
leave
ret
```

Program Example

To illustrate the assembly language interface, consider the following example of a C function.

```
add(i, j)
int i, j;
{
        return(i + j);
}
```

If written as an assembly language routine, this function must save the proper registers, retrieve the the arguments from the stack, add the arguments, place the return value in the **ax** register, then restore registers and return control. The following is an example of how the routine can be written (in small model).

```
_add:
        push  bp
        mov   bp,sp
        push  di
        push  si
        mov   ax,*4(bp)
        add   ax,*6(bp)
        pop   si
        pop   di
        leave
        ret
```

If, on the other hand, the C function is to be called by an assembly language routine, the routine must contain instructions that push the arguments on the stack in the proper order, call the function, and clear the stack. It may then use the return value in the **ax** register. The following is an example of the instructions that can do this.

```
push    <j value>
push    <i value>
call    _add
add     sp,*4
```

intel®

APPENDIX B
XENIX 286 RELEASE 3
PROGRAMMING DIFFERENCES

This appendix lists some of the differences between XENIX Release 3, XENIX Release 1, and UNIX System III. It is intended to aid users who wish to convert system calls in existing application programs for use on other systems.

Executable File Format

XENIX Release 3 uses a new **a.out** executable file format. This format is similar to the old **a.out** format but contains additional information about the executable file such as text and data relocation bases, file segment information, target machine identification, word and byte ordering, and symbol table and relocation table format. The **a.out** file also contains the revision number of the kernel, used during execution to control access to system functions. To execute existing programs in **a.out** format, you must first convert to the new format.

Revised System Calls

Some system calls in XENIX Release 3 have been revised and do not perform the same tasks as the corresponding calls in UNIX System III. The following table lists the revised system calls.

System Call #	XENIX R2 function	System III function
35	ftime	unused
38	unused	clocal
39	unused	setpgrp
40	unused	cxenix
57	unused	utssys
62	clocal	fcntl
63	cxenix	ulimit

The **cxenix** function provides access to system calls unique to XENIX Release 3. The **clocal** function provides access to all calls unique to an OEM.

ioctl Function

XENIX Release 3 supports some XENIX Release 1 **ioctl** calls. The available XENIX Release 1 **ioctl** calls are: **TIOCSETP, TIOCSETN, TIOCGETP, TIOCSETC, TIOCGETC, TIOCEXCL, TIOCNXCL, TIOCHPCL, TIOCFLUSH, TIOCGETD,** and **TIOCSETD.**

Version 7 Additions

XENIX Release 3 maintains a number of UNIX V7 features that were dropped from UNIX System III. In particular, XENIX Release 3 continues to support the **dup2** and **ftime** functions. The **ftime** function, used with the **ctime** function, provides the default value for the time zone when the TZ environment variable has not been set. This means a binary configuration program can be used to change the default time zone. No source license is required.

Path Name Resolution

If a null path name is given, XENIX Release 1 interprets the name to be the current directory, but UNIX System III considers the name to be an error. XENIX Release 3 uses the version number in the **a.out** header to determine what action to take.

If the symbol ".." is given as a path name when in a root directory defined using the **chroot** function, XENIX Release 1 moves to the next higher directory. XENIX Release 3 also allows the ".." symbol in a root directory but restricts its use to the super-user.

Using the mount and chown Functions

Both XENIX Release 3 and UNIX System III restrict the use of the **mount** system call to the super-user. Also, both allow the owner of a file to use **chown** to change the file ownership.

Super-Block and File System Format

XENIX Release 3 uses a new internal format for its super-blocks and file systems. File systems from XENIX Release 1 or UNIX systems should be transferred using the **tar** command, described in the *XENIX 286 Reference Manual*.

Change in Word Order within Double-Words

XENIX Release 3 has changed the order in which words are stored within double-words, as compared to XENIX Release 1. The least significant word of a double-word quantity is now stored at the lower memory address; the most significant word of a double-word quantity is now stored at the higher memory address. Users transferring binary data between Release 1 and Release 3 systems, especially floating-point or **long** numbers, may have to use special programs to convert their data.

intel®

APPENDIX C
SYSTEM FUNCTIONS

This appendix describes all system functions provided by XENIX. System functions include all system calls that can be made to the XENIX kernel, as well as many other useful routines. All system calls and some other routines are included in the standard library **libc**. **libc** is searched whenever any C program is linked. Other routines are contained in a variety of libraries. For each library, three different library files are provided, for linking with small-, middle-, and large-model programs respectively.

To use routines in a library other than the standard library, the appropriate library must be linked by specifying *-lname* to the compiler or linker, where *name* is one of the following library names:

- **curses** screen, windows, and cursor manipulation functions
- **dbm** data base management functions
- **m** math functions
- **termcap** functions to access the **termcap** terminal capabilities file

Finding Functions

Most functions in this appendix are listed in alphabetical order. However, some related functions are listed together out of alphabetical order. If you cannot find a function, refer to the Index at the back of this manual; it includes *all* function names as indexed terms in alphabetical order.

Each function description in this appendix references any related functions, commands, files, or file formats under the heading "See Also" in the function description. Referenced functions can be found in this appendix and are simply listed by name. References to other entries in this manual are specified by name and section. References to entries in other manuals are specified by name, section, and manual.

Error Codes

Many functions, especially system calls, have error returns. An error condition is indicated by an otherwise impossible return value, frequently -1 if a number is being returned, or **NULL** if a pointer is being returned. An error code is stored in the external variable **errno.** **errno** should be checked immediately after an error return is detected, to better determine the cause of the error.

The system call descriptions do not list all possible error codes, because many errors are possible for most of the calls. The error codes are defined in the include file **<error.h>**. The following alphabetical list describes all these codes:

EACCES "Permission denied"
An attempt was made to access a file in a way forbidden by the protection system.

EAGAIN "No more processes"
A **fork** failed because the system's process table is full or the user is not allowed to create more processes.

EBADF "Bad file number"
Either a file descriptor refers to no open file, or a read request is made to a file open only for writing, or a write request is made to a file open only for reading.

EBUSY "Mount device busy"
An attempt was made to mount an already-mounted device, or an attempt was made to dismount a device that contains an active file (an open file, a current directory, a mounted-on file, or an active text segment), or an attempt was made to enable accounting when it was already enabled.

ECHILD "No child processes"
A **wait** was executed by a process with no existing or unwaited-for child processes.

EDEADLOCK "Would deadlock"
A process's attempt to lock a file region would cause a deadlock between processes contending for control of that region.

EDOM "Math argument"
The argument of a math library function is outside the domain of the function.

EEXIST "File exists"
An existing file was referenced in an inappropriate context, e.g., **link.**

EFAULT "Bad address"
An address fell outside the bounds of the calling process's address space. This error is usually due to a bad pointer argument to a function.

EFBIG "File too large"
The size of a file exceeded the maximum file size (1,082,201,088 bytes) or the process's file size limit (see **ulimit**).

EINTR "Interrupted system call"
An asynchronous signal (such as **SIGINT** or **SIGQUIT**) that the user has elected to catch occurred during a system call. If execution is resumed after processing the signal, it appears as if the interrupted system call returned this error condition.

EINVAL "Invalid argument"
Some invalid argument (e.g., dismounting an unmounted device or specifying an undefined signal to **signal**).

EIO "I/O error"
Some physical I/O error. This error may, in some cases, occur on a call following the one to which it actually applies.

EISDIR "Is a directory"
Attempt to write on a directory.

EMFILE "Too many open files"
The calling process tried to open a file when it had 20 file descriptors open already.

EMLINK "Too many links"
An attempt was made to make more than the maximum number of links (1,000) to a file.

ENAVAIL "Not available"
An **opensem, waitsem**, or **sigsem** call was issued to a semaphore that had not been initialized by calling **creatsem**; or, **sigsem** was called out of sequence, before the calling process called **waitsem** on the semaphore; or, **nbwaitsem** was called on a semaphore that was being used by another process; or, a semaphore on which a process was waiting has been left in an inconsistent state when the process controlling the semaphore exited without relinquishing control by calling **sigsem**; or, a name file (semaphore, shared data, etc.) was specified when not expected.

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

ENODEV "No such device"
An attempt was made to apply an inappropriate system call to a device, e.g., read a write-only device.

ENOENT "No such file or directory"
The named file or directory does not exist, or a null path name was specified.

ENOEXEC "Exec format error"
An attempt was made to execute a file that does not start with a valid magic number (see **a.out** in Appendix D, "File Formats").

ENOMEM "Not enough space"
An **exec** or **sbrk** call requested more space than the system can supply (not a temporary condition; the maximum space available is a system parameter); or, the arrangement of text, data, and stack segments requires too many segmentation registers; or, there is not enough swap space during a **fork** call.

ENOSPC "No space left on device"
During a **write** to an ordinary file, there is no free space left on the device.

ENOTBLK "Block device required"
A nonblock file was mentioned where a block device is required, e.g., in calling **mount.**

ENOTDIR "Not a directory"
A nondirectory was specified where a directory is required, e.g., as an argument to **chdir.**

ENOTNAM "Not a semaphore"
A semaphore function was called with an invalid semaphore name or invalid semaphore identifier.

ENOTTY "Not a typewriter"
An attempt was made to perform a terminal function on a device that is not a terminal.

ENXIO "No such device or address"
An attempt was made to open a device that does not exist or is not on-line.

EPERM "Not owner"
An attempt was made to access a file in a way forbidden except to its owner or the super-user; or, an ordinary user attempted an action allowed only to the super-user.

EPIPE "Broken pipe"
An attempt was made to write to a pipe for which there is no process waiting to read the data written. (This error normally generates a signal; this error code results if the signal is ignored.)

ERANGE "Result too large"
The result of a math function exceeds the range that can be represented in the destination data type.

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

ESPIPE "Illegal seek"
lseek was called on a pipe.

ESRCH "No such process"
A process specified in a call to **kill** or **ptrace** cannot be found.

ETXTBSY "Text file busy"
An attempt was made to execute a pure-procedure program file that was open; or, an attempt was made to open for writing a pure-procedure program file that was being executed.

EUCLEAN "Structure needs cleaning"
An attempt was made to **mount** a file system with a super-block that is not flagged as "clean."

EXDEV "Cross-device link"
An attempt was made to link to a file on another device.

E2BIG "Arg list too long"
An argument list longer than 5120 bytes was presented in a call to **execl, execle, execlp, execv, execve,** or **execvp.**

Definitions

This section defines some of the terms used in the function descriptions.

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 30,000.

Parent Process ID

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

Process Group ID

Each active process is a member of a process group identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see **kill.**

tty Group ID

Each active process can be a member of a terminal group 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** and **signal.**

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 creating the process.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID used to determine file access permissions. 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.**

Super-User

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

Special Processes

The processes with process IDs of 0 and 1 are special processes and are referred to as **proc0** and **proc1**.

proc0 is the scheduler. **proc1** is the initialization process (**init**). **proc1** is the ancestor of every other process in the system and is used to control the process structure.

File Name

Names with up to 14 characters can name an ordinary file, special file, or directory.

These characters can be selected from the set of all character values excluding null and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, [, or] in file names because of the special meaning attached to these characters by the shell. Likewise, the high-order bit of the characters should not be set.

Path Name and Path Prefix

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. A file name is a string of 1 to 14 characters other than the ASCII slash and null; a directory name is a string of 1 to 14 characters (other than the ASCII slash and null) naming 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.

Unless specifically stated otherwise, the null path name is treated as if it named a nonexistent file.

Directory

Directory entries are called links. By convention, a directory contains at least two links, . ("dot") and .. ("dot dot"). Dot refers to the directory itself and dot dot refers to its parent directory.

Root Directory and Current Working Directory

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

File Access Permissions

Read, write, or execute/search permissions on a file are granted to a process if one or more of the following are true:

- The process's effective user ID is super-user.
- The process's effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.
- The process's effective user ID does not match the user ID of the owner of the file, but the process's group ID matches the group of the file and the appropriate access bit of the "group" portion (070) of the file mode is set.
- The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID does not match the group ID of the file, but the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied. See **chmod.**

Function Descriptions

Subsequent sections of this appendix are function descriptions. For each function, the appendix specifies syntax, operation, information about error returns, and references to related functions, commands, or files.

a64l, l64a - Convert between long integer and base-64 ASCII.

Syntax

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

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

Description

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

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

a64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. **l64a** takes a **long** argument and returns a pointer to the corresponding null-terminated base-64 representation.

Notes

The value returned by **l64a** points to static data that is overwritten by each call.

abort - Generate an IOT fault.

Syntax

abort()

Description

abort sends an IOT signal to the calling process. This usually results in termination with a core dump.

It is possible for **abort** to return control if **SIGIOT** is caught or ignored.

See Also

exit, signal

adb in "Programming Commands" in the *XENIX 286 Programmer's Guide*

Diagnostics

Usually returns "abort - core dumped" from the shell.

abs - Integer absolute value.

Syntax

```
int abs(i)
int i;
```

Description

abs returns the absolute value of its integer operand.

See Also

fabs

Notes

If the argument to **abs** is the largest negative integer supported by the hardware, then **abs** returns the largest negative integer as its result.

access - Determine accessibility of a file.

Syntax

```
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 in place of the effective group ID. The bit pattern contained in **amode** is constructed as follows:

04	Read
02	Write
01	Execute (search)
00	Check existence of file

The file is not accessible if one or more of the following are true:

- **path** is an illegal address. [**EFAULT**]
- The path name is null or the named file does not exist. [**ENOENT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- A component of the path prefix denies search permission, or the file mode does not permit the requested access. [**EACCES**]
- Write access is requested for a file on a read-only file system. [**EROFS**]
- Write access is requested for a pure-procedure (shared text) file that is being executed. [**ETXTBSY**]

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits, members of the owner'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.

If the super-user calls **access,** the file's permission bits are not checked and accessibility is granted (if there are no errors in referencing the file).

Return Value

If the requested access is permitted, 0 is returned. If access is denied or some other error is encountered, -1 is returned and **errno** is assigned an error code.

See Also

chmod

acct - Enable or disable process accounting.

Syntax

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

Description

acct is used to enable or disable the system's process accounting routine. If accounting is enabled, an accounting record is written on an accounting file for each process that terminates. Termination can be caused by a signal or by calling **exit.** The effective user ID of the calling process must be super-user to use this call.

path points to the path name of the accounting file. (The accounting file format is given in the entry **acct** in Appendix D, "File Formats.") Accounting is enabled if **path** is not **NULL** and no errors occur during the system call. Accounting is disabled if **path** is **NULL** and no errors occur during the system call.

acct fails if one or more of the following are true:

- The effective user ID of the calling process is not super-user. [**EPERM**]
- **path** is an illegal address. [**EFAULT**]
- The path name is null or the named file does not exist. [**ENOENT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- A component of the path prefix denies search permission, or the file mode does not permit the requested access, or the named file is not an ordinary file. [**EACCES**]
- The named file is in a read-only file system. [**EROFS**]
- An attempt was made to enable accounting when it is already enabled. [**EBUSY**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** assigned an error code.

See Also

acct in Appendix D, "File Formats"

accton, acctcom in "Commands" in the *XENIX 286 Reference Manual*

alarm - Set process alarm clock.

Syntax

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

Description

alarm instructs the calling process's alarm clock to send the signal **SIGALRM** to the calling process after the number of real-time seconds specified by **sec** have elapsed.

Alarm requests are not stacked; successive calls reset the calling process's alarm clock, replacing any previously set alarm.

If **sec** is 0, any previously made alarm request is canceled without setting a new request.

Return Value

alarm returns the amount of time previously remaining in the calling process's alarm clock.

See Also

pause, signal

assert - Help verify validity of program.

Syntax

```
#include <assert.h>
...
assert(expression);
```

Description

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false, it prints "Assertion failed: file *xyz*, line *nnn*" on the standard error file and exits. *xyz* is the source file and *nnn* the source line number of the **assert** statement. Compiling with the preprocessor option **-DNDEBUG** causes **assert** to be ignored.

See Also

cc in "Programming Commands" in the *XENIX 286 Programmer's Guide*

atof, atoi, atol - Convert ASCII to numbers.

Syntax

```
double atof(nptr)
char *nptr;

int atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

Description

These functions convert a string pointed to by **nptr** to floating, integer, or long integer representation respectively. The first unrecognized character ends the string.

atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of decimal digits optionally containing a decimal point, then an optional **e** or **E** followed by an optionally signed integer.

atoi and **atol** recognize an optional string of tabs and spaces, then an optional sign, then a string of decimal digits.

See Also

scanf

Notes

There are no provisions for overflow.

BESSEL: j0, j1, jn, y0, y1, yn - Bessel functions.

Syntax

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x);
int n;
double x;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
int n;
double x;
```

Description

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

Diagnostics

Negative arguments cause **y0**, **y1**, and **yn** to return a huge negative value.

bsearch - Binary search.

Syntax

```
char *bsearch(key, base, nel, width, compar)
char *key;
char *base;
int nel, width;
int (*compar)();
```

Description

bsearch returns a pointer into a table indicating the location at which a datum can be found. **NULL** is returned if the key cannot be found in the table. The table must be sorted in increasing order. **key** is a pointer to the datum to be searched for. **base** is a pointer to the base of the table. **nel** is the number of (fixed-size) elements in the table. **width** is the size of an element in bytes. **compar** is a pointer to the comparison routine. It is called with two arguments that are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 depending on whether the first argument is to be considered less than, equal to, or greater than the second.

Note that the format of the key and the table elements and how they are ordered are determined by the caller via the caller-supplied comparison routine.

See Also

lsearch, qsort

chdir - Change the working directory.

Syntax

```
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 '/'.

chdir fails and the current working directory is unchanged if one or more of the following are true:

- **path** is an illegal address. [**EFAULT**]
- The path name is null or the named directory does not exist. [**ENOENT**]
- A component of the path name is not a directory. [**ENOTDIR**]
- A component of the path name denies search permission. [**EACCES**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

chroot

chmod - Change mode of a file.

Syntax

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

Description

Path points to a path name naming a file. The file can be an ordinary file, directory, or special file (e.g., a device). **chmod** sets the access permission portion of the named file's mode according to the bit pattern contained in **mode.**

Access permission bits are interpreted as follows:

04000	Set user ID on execution.
02000	Set group ID on execution.
01000	Save text image after execution.
00400	Read by owner.
00200	Write by owner.
00100	Execute (or search if a directory) by owner.
00070	Read, write, execute (search) by group.
00007	Read, write, execute (search) by others.

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

If the effective user ID of the process is not super-user, **mode** bit 01000 (save text image after execution) is cleared.

If the effective user ID of the process is not super-user or the effective group ID of the process does not match the group ID of the file, **mode** bit 02000 (set group ID on execution) is cleared.

If an executable file is prepared for sharing, then mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus, when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time. Many systems have relatively small amounts of swap space, and the save-text bit should be used sparingly, if at all.

chmod fails and the file mode is unchanged if one or more of the following are true:

- The effective user ID does not match the owner of the file and the effective user ID is not super-user. [**EPERM**]
- **path** is an illegal address. [**EFAULT**]
- The path name is null or the named file does not exist. [**ENOENT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- A component of the path prefix denies search permission. [**EACCES**]
- The named file is in a read-only file system. [**EROFS**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

chown, mknod

chown - Change the owner and group of a file.

Syntax

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

Description

path points to a path name naming a file. The file can be an ordinary file, directory, or special file (e.g., a device). The owner ID and group ID of the named file are set to the numeric values contained in **owner** and **group** respectively.

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

If **chown** is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, are cleared.

chown fails and the owner and group of the named file are unchanged if one or more of the following are true:

- The effective user ID does not match the owner of the file and the effective user ID is not super-user. **[EPERM]**
- **path** is an illegal address. **[EFAULT]**
- The path name is null or the named file does not exist. **[ENOENT]**
- A component of the path prefix is not a directory. **[ENOTDIR]**
- A component of the path prefix denies search permission. **[EACCES]**
- The named file is in a read-only file system. **[EROFS]**

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

chmod

chroot - Change the root directory.

Syntax

```
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 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.

chroot fails and the root directory is unchanged if one or more of the following are true:

- The effective user ID is not super-user. **[EPERM]**
- **path** is an illegal address. **[EFAULT]**
- The path name is null or the named file does not exist. **[ENOENT]**
- A component of the path name is not a directory. **[ENOTDIR]**
- A component of the path prefix denies search permission. **[EACCES]**

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

chdir

chroot in "Commands" in the *XENIX 286 Reference Manual*

chsize - Change the size of a file.

Syntax

```
int chsize(fildes, size)
int fildes;
long size;
```

Description

This routine changes the size of the file associated with the file descriptor **fildes** to be exactly **size** bytes in length, by either truncating the file or padding it with an appropriate number of bytes. If **size** is less than the initial size of the file, then all allocated disk blocks between **size** and the initial file size are freed.

The maximum file size as set by **ulimit** is enforced when **chsize** is called, rather than on subsequent writes. Thus **chsize** fails, and the file size remains unchanged if the new changed file size would exceed the process's file size limit.

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

creat, dup, lseek, open, pipe, ulimit

Notes

In general, if **chsize** is used to expand the size of a file, when data is written to the end of the file, intervening blocks are filled with zeros.

close - Close a file descriptor.

Syntax

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

Description

fildes is a file descriptor obtained from a **creat, dup, fcntl, open,** or **pipe** system call. **close** closes the file indicated by **fildes.**

close fails if **fildes** is not a valid open file descriptor for the calling process. **[EBADF]**

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

creat, dup, exec, fcntl, open, pipe

CONV: toupper, tolower, toascii - Translate characters.

Syntax

```
#include <ctype.h>

int toupper(c)
int c;

int tolower(c)
int c;

int _toupper(c)
int c;

int _tolower(c)
int c;

int toascii(c)
int c;
```

Description

toupper and **tolower** have as domain the range of **getc:** the integers from -1 through 255. If the argument of **toupper** represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of **tolower** represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

_toupper and **_tolower** are macros that accomplish the same thing as **toupper** and **tolower** but have restricted domains and are faster. **_toupper** requires a lowercase letter as its argument; its result is the corresponding uppercase letter. **_tolower** requires an uppercase letter as its argument; its result is the corresponding lowercase letter. Arguments outside these domains cause undefined results.

toascii returns its argument with all bits cleared that are not part of a standard ASCII character.

Because **toupper** and **tolower** are implemented as macros, they should not be used where unwanted side effects may occur. If **toupper** and **tolower** are "undefined" with **#undef,** then the corresponding library functions are linked instead, allowing any arguments to be used without worry about side effects.

See Also

CTYPE

creat - Create a new file or rewrite an existing one.

Syntax

```
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, owner, and group are unchanged. For a new file, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of **mode** modified as follows:

- All bits set in the process's file mode creation mask are cleared. See **umask.**
- The "save text image after execution" bit of the mode is cleared. See **chmod.**

Upon successful completion, a nonnegative integer, namely 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).**

creat fails and the file is not created or rewritten if one or more of the following are true:

- **path** is an illegal address. [**EFAULT**]
- The path name is null or the named file does not exist. [**ENOENT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- A component of the path prefix denies search permission, or the file does not already exist and the directory in which the file is to be created does not permit writing, or the file exists and write permission is denied. [**EACCES**]
- The named file is in a read-only file system. [**EROFS**]
- The named file is a pure procedure (shared text) file that is being executed. [**ETXTBSY**]
- The named file is an existing directory. [**EISDIR**]
- The process has twenty file descriptors open. [**EMFILE**]

Return Value

If successful, the new file descriptor (always nonnegative) is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

close, dup, lseek, open, read, umask, write

creatsem - Create an instance of a binary semaphore.

Syntax

```
int creatsem(sem_name, mode)
char *sem_name;
int mode;
```

Description

creatsem creates a binary semaphore, identified by a distinct integer, to be used by **waitsem** and **sigsem** to manage mutually exclusive access to a resource, shared variable, or critical section of a program. **creatsem** returns a unique semaphore number, which is then used as the parameter in **waitsem** and **sigsem** calls. Semaphores are special files with length zero. The file name space is used to provide unique identifiers for semaphores. **mode** sets the accessibility of the semaphore using the same format as file access bits. Access to a semaphore is granted only on the basis of the read access bit; the write and execute bits are ignored.

A semaphore can be operated on only by a synchronizing primitive (**waitsem** or **sigsem**), or by a function that initializes it to some value (**creatsem**), or by a function that opens the semaphore for use by a process (**opensem**). Synchronizing primitives are guaranteed to execute without interruption once started. These primitives are used by associating a semaphore with each resource (including critical code sections) to be protected.

The process controlling the semaphore should issue

```
sem_num = creatsem("semaphore", mode);
```

to create, initialize, and open the semaphore for that process. All other processes using the semaphore should issue

```
sem_num = opensem("semaphore")
```

to access the semaphore's identification value. Note that a process cannot open or use a semaphore that has not been initialized by a call to **creatsem**, nor should a process open a semaphore more than once in one period of execution. Both the creating and opening processes use **waitsem(sem_num)** and **sigsem(sem_num)** to use the semaphore **sem_num**.

creatsem fails and the semaphore is not created if one or more of the following are true:

- **sem_name** is an illegal address. [**EFAULT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- A component of the path prefix denies search permission. [**EACCES**]
- Write access is requested for a semaphore in a read-only file system. [**EROFS**]
- **sem_name** names an existing file that is not a semaphore. [**ENOTNAM**]
- **sem_name** names an existing semaphore that is open for use by other processes. [**EEXIST**]

Return Value

If successful, the new semaphore number (always nonnegative) is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

open, waitsem, sigsem

Notes

After calling **creatsem,** a program must still call **waitsem** to gain control of a given resource.

crypt, setkey, encrypt - Encryption functions.

Syntax

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

setkey(key)
char *key;

encrypt(block, edflag)
char *block;
int edflag;
```

Description

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

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

The **setkey** and **encrypt** functions provide access to the actual DES algorithm. The argument of **setkey** is a character array of length 64 containing only bytes with numeric value 0 or 1. If this string is divided into groups of 8, the low-order byte in each group is ignored, leading to a 56-bit key that is set into the machine.

The argument to **encrypt** is likewise a character array of length 64 containing only bytes with numeric value 0 or 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by **setkey**. If **edflag** is 0, the argument is encrypted; if nonzero, it is decrypted.

See Also

getpass

passwd in "Commands" in the *XENIX 286 Reference Manual*

Notes

The return value from **crypt** points to static data that is overwritten by each call.

ctermid - Generate a file name for a terminal.

Syntax

```
#include <stdio.h>

char *ctermid(s)
char *s;
```

Description

ctermid generates a null-terminated string that refers to the controlling terminal for the current process when used as a file name.

If **s** is **NULL**, the string is stored in an internal static area, the contents of which are overwritten at the next call to **ctermid**, and the address of which is returned. Otherwise, **s** is assumed to point to a character array of at least **L_ctermid** elements; the string is placed in this array and the value of **s** is returned. The constant **L_ctermid** is defined in **<stdio.h>**.

Notes

The difference between **ctermid** and **ttyname** is that **ttyname** must be given a file descriptor and returns the actual name of the terminal associated with that file descriptor, while **ctermid** returns a string ("/dev/tty") that refers to the terminal if used as a file name. Thus **ttyname** is useless unless the process already has at least one file open to a terminal.

See Also

ttyname

ctime, localtime, gmtime, asctime, tzset - Convert date and time to ASCII.

Syntax

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

#include<time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

tzset()

extern long timezone;
extern int daylight;
extern char tzname;
```

Description

ctime converts a time pointed to by **clock** (such as returned by **time**) into ASCII and returns a pointer to a 26-character string in the following form. All the fields have constant width:

```
Sun Sep 16 01:03:52 1972\n
```

localtime and **gmtime** return pointers to structures containing the time broken down into fields, of type **tm**. **localtime** corrects for the time zone and possible daylight savings time; **gmtime** converts directly to GMT (Greenwich Meridian Time), which is the time the XENIX system uses. **asctime** converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration for **tm** is defined in **/usr/include/time.h.**

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday = 0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight savings time is in effect.

The external **long** variable **timezone** contains the difference, in seconds, between GMT and local standard time (in EST, **timezone** is 5*60*60); the external integer variable **daylight** is nonzero only if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975.

If an environment variable named **TZ** is present, **asctime** uses the contents of the variable to override the default time zone. The value of **TZ** must be a three-letter time zone name, followed by a number representing the difference between local time (with optional sign) and Greenwich time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for New Jersey would be EST5EDT. The effects of setting **TZ** are thus to change the values of the external variables **timezone** and **daylight;** in addition, the time zone names contained in the external array **tzname** are set from the environment variable, e.g.:

```
char *tzname[2] = {"EST", "EDT"};
```

The function **tzset** sets the external variables from **TZ;** it is called by **asctime** and may also be called explicitly by the user.

See Also

getenv, time

Notes

The return values point to static data that is overwritten by each call.

CTYPE: isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii - Classify characters.

Syntax

```
#include <ctype.h>

int isalpha(c)
int c;

...
```

Description

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. **isascii** is defined on all integer values; the rest are defined only where **isascii** is true and on the single non-ASCII value **EOF** (see STDIO).

isalpha **c** is a letter

isupper **c** is an uppercase letter

islower **c** is a lowercase letter

isdigit **c** is a decimal digit [0-9]

isxdigit **c** is a hexadecimal digit [0-9], [A-F] or [a-f]

isalnum **c** is an alphanumeric (letter or digit)

isspace **c** is a space, tab, carriage return, line feed (newline), vertical tab, or form feed

ispunct **c** is a punctuation character (neither control nor alphanumeric)

isprint **c** is a printing character, octal 040 (space) through octal 0176 (tilde)

isgraph **c** is a graphic printing character, like **isprint** except false for space

iscntrl **c** is a delete character (octal 0177) or ordinary control character (octal 0 - 037)

isascii **c** is an ASCII character (octal 0 - 0177)

curses - Perform screen, window, and cursor functions.

Syntax

cc [flags] files -lcurses -ltermlib [libraries]

Description

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the **refresh** function is called to make the current screen look like the new one. In order to initialize the routines, **initscr** must be called before any of the other routines that deal with windows and screens are used.

The screen routines are linked with the loader options **-lcurses** and **-ltermlib.** The screen functions and data structures are described in detail in Chapter 3, "Screen Processing." These major functions are provided:

addch(ch)	Adds a character to **stdscr**
addstr(str)	Adds a string to **stdscr**
box(win,vert, hor)	Draws a box around a window
clear()	Clears **stdscr**
clearok(scr, boolf)	Sets clear flag for **scr**
clrtobot()	Clears to bottom on **stdscr**
clrtoeol()	Clears to end of line on **stdscr**
crmode()	Sets CBREAK mode
delwin(win)	Deletes **win**
echo()	Sets ECHO mode
erase()	Erases **stdscr**
getch()	Gets a char through **stdscr**
getstr(str)	Gets a string through **stdscr**
gettmode()	Gets tty modes
inch()	Gets char at current (y, x) coordinates
initscr()	Initializes screens
leaveok(win, boolf)	Sets leave flag for **win**
longname(termbuf, name)	Gets long name from **termbuf**

move(y, x)	Moves to (y, x) on **stdscr**
mvcur(lasty, lastx, newy, newx)	Actually moves cursor
newwin(lines, cols, begin_y, begin_x)	Creates a new window
nl()	Sets newline mapping
nocrmode()	Unsets CBREAK mode
noecho()	Unsets ECHO mode
nonl()	Unsets newline mapping
noraw()	Unsets RAW mode
overlay(win1, win2)	Overlays **win1** on **win2**
overwrite(win1, win2)	Overwrites **win1** on top of **win2**
printw(fmt [, arg]...)	Sends formatted output to **stdscr**
raw()	Sets RAW mode
refresh()	Makes current screen look like **stdscr**
scanw(fmt [,argptr]...)	Sends formatted input from **stdscr**
scroll(win)	Scrolls **win** one line
scrollok(win, boolf)	Sets scroll flag for **win**
setterm(name)	Sets term variables for **name**
unctrl(ch)	Returns printable version of **ch**
waddch(win, ch)	Adds char to **win**
waddstr(win, str)	Adds string to **win**
wclear(win)	Clears **win**
wclrtobot(win)	Clears to bottom of **win**
wclrtoeol(win)	Clears to end of line on **win**
werase(win)	Erases **win**
wgetch(win)	Gets a char through **win**
wgetstr(win, str)	Gets a string through **win**
winch(win)	Gets char at current (y, x) in **win**
wmove(win, y, x)	Sets current (y, x) coordinates on **win**
wprintw(win, fmt [, arg]...)	Sends formatted output to **win**
wrefresh(win)	Makes screen look like **win**
wscanw(win, fmt [, argptr]...)	Sends formatted input from **win**

Chapter 3, "Screen Processing," describes these functions and some additional screen processing functions.

See Also

stty, setenv

termcap in "Files" in the *XENIX 286 Reference Manual*

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

cuserid - Get the login name of the user.

Syntax

```
#include <stdio.h>

char *cuserid(s)
char *s;
```

Description

cuserid generates a character representation of the login name of the owner of the current process. If **s** is **NULL**, this representation is generated in an internal static area, the address of which is returned. Otherwise, **s** is assumed to point to an array of at least **L_cuserid** characters and the representation is left in this array. The constant **L_cuserid** is defined in **<stdio.h>**.

Diagnostics

If the login name cannot be found, **cuserid** returns **NULL**; if **s** is non-**NULL** in this case, a single null character is stored at ***s**.

See Also

getlogin, getpwuid

Notes

cuserid uses **getpwnam**; thus the results of a user's call to the latter are overwritten by a subsequent call to the former.

DBM: dbminit, fetch, store, delete, firstkey, nextkey - Perform data base functions.

Syntax

```
typedef struct { char *dptr; int dsize; } datum;

int dbminit(file)
char *file;

datum fetch(key)
datum key;

int store(key, content)
datum key, content;

int delete(key)
datum key;

datum firstkey();

datum nextkey(key);
datum key;
```

Description

These functions maintain key/content pairs in a data base. The functions can handle very large (one billion blocks) data bases and can access a keyed item in one or two file system accesses. The functions are obtained with the loader option **-ldbm.**

keys and **contents** are described by the **datum** data type. A **datum** specifies a string of **dsize** bytes pointed to by **dptr.** Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has ".dir" as its suffix. The second file contains all data and has ".pag" as its suffix.

Before a data base can be accessed, it must be opened by **dbminit.** At the time of this call, the files **file.dir** and **file.pag** must exist. (An empty data base is created by creating zero-length **.dir** and **.pag** files.)

Once open, the data stored under a key is accessed by **fetch** and data is placed under a key by **store.** A key (and its associated contents) is deleted by **delete.** A linear pass through all keys in a data base may be made, in an (apparently) random order, by use of **firstkey** and **nextkey. firstkey** returns the first key in the data base. Given any key, **nextkey** returns the next key in the data base. The following code can be used to traverse a data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key)) {
        /* code to access/change record goes here */
}
```

Diagnostics

All functions that return an **int** indicate errors with negative values. A zero return value indicates success. Routines that return a **datum** indicate errors with a **dptr** value of **NULL.**

Notes

Every **.pag** file contains holes so that its apparent size is about four times its actual content. Older XENIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (**cp, cat, tp, tar, ar**) without filling in the holes.

dptr pointers returned by these functions point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the block size **BSIZE** (1024 bytes in XENIX Release 3). Moreover all key/content pairs that hash together must fit in a single block. **store** returns an error if a disk block fills with inseparable data.

delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys returned by **firstkey** and **nextkey** depends on a hashing function, not on anything interesting.

These routines are not re-entrant, so they should not be used on more than one data base at a time.

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

defopen, defread - Read default entries.

Syntax

```
int defopen(filename)
char *filename;

char *defread(pattern)
char *pattern;
```

Description

defopen and **defread** are a pair of routines designed to allow easy access to default definition files. XENIX is normally distributed in binary form; the use of default files allows OEMs or site administrators to customize utility defaults without having the source code.

A program first calls **defopen** with the path name of a file containing the default entries. **defopen** returns 0 if it is successful in opening the file. If **defopen** fails, it returns the (nonzero) error code assigned to **errno** by **fopen.**

The program then calls **defread** with a character string, **pattern. defread** reads the previously opened file from the beginning until it encounters a line beginning with **pattern. defread** then returns a pointer to the first character in the line after the initial **pattern.** This line has been read into static storage; the newline at the end of the line is replaced by a null character. The next call to **defread** may overwrite the line.

defread returns **NULL** if a default file is not open, if **pattern** could not be found, or if it encounters any line in the file longer than 128 characters.

When all items of interest have been extracted from the opened file, the program may call **defopen** with the name of another file to be searched, or it may call **defopen** with **NULL,** which closes the default file without opening another.

Files

The XENIX convention is for a system program **xyz** to store its defaults (if any) in the file **/etc/default/xyz.**

dup, dup2 - Duplicate an open file descriptor.

Syntax

```
int dup(fildes)
int fildes;

dup2(fildes, fildes2)
int fildes, fildes2;
```

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.**

dup returns the lowest available file descriptor. **dup2** causes **fildes2** to refer to the same file as **fildes.** If **fildes2** already referred to an open file, it is closed first.

dup fails if one or more of the following are true:

- **fildes** is not a valid open file descriptor. [**EBADF**]
- Twenty file descriptors are currently open. [**EMFILE**]

Return Value

If successful, **dup** returns the (nonnegative) file descriptor. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

creat, close, exec, fcntl, open, pipe

ecvt, fcvt, gcvt - Numeric output conversions.

Syntax

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

Description

ecvt converts **value** to a null-terminated string of **ndigit** ASCII digits and returns a pointer thereto. The low-order digit of the string produced is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through **decpt** (negative means to the left of the returned digits). If the sign of the result is negative, the **int** pointed to by **sign** is nonzero, otherwise it is zero.

fcvt operates in the same way as **ecvt,** except that the correct digit is rounded for FORTRAN F format output of the number of digits specified by **ndigits.**

gcvt converts **value** to a null-terminated ASCII string in **buf** and returns a pointer to **buf.** It attempts to produce **ndigit** significant digits in FORTRAN F format if possible, otherwise it uses FORTRAN E format, ready for printing. Trailing zeros may be suppressed.

See Also

printf

Notes

The return values of **ecvt** and **fcvt** point to a static data area that is overwritten by each call.

EXEC: execl, execv, execle, execve, execlp, execvp - Execute a file.

Syntax

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

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

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

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

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

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

Description

exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the "new process file." There can be no return from a successful **exec** because the calling process is overlaid by the new process.

path points to a path name that identifies the new process file.

file points to the new process file path name. The path prefix for this file is obtained (if it is a relative path name) by a search of the directories passed as the environment line "PATH =".

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** must be present and point 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** must have at least one member, and it must point to a string that is the same as **path** (or its last component). **argv** is terminated by a **NULL** element.

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** element.

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.** For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process are set to terminate the new process. Signals set to be ignored by the calling process are set to be ignored by the new process. Signals set to be caught by the calling process are set to terminate the new process; see **signal.**

If the set-user-ID mode bit of the new process file is set (see **chmod**), **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.

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

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

Nice value (see **nice**)

Process ID

Parent process ID

Process group ID

tty group ID (see **exit** and **signal**)

Trace flag (see **ptrace**)

Time left until an alarm clock signal (see **alarm**)

Current working directory

Root directory

File mode creation mask (see **umask**)

File size limit (see **ulimit**)

utime, stime, cutime, and **cstime** (see **times**)

From C, two interfaces are available. **execl** is useful when a known file with known arguments is being called; the arguments to **execl** are the character strings constituting the file and the arguments. The first argument is conventionally the same as the file name (or its last component). A **NULL** argument must end the argument list.

The **execv** version is useful when the number of arguments is unknown in advance. The arguments to **execv** are the name of the file to be executed and a vector of strings containing the arguments. The last argument must be **NULL** to terminate the list.

When a C program is executed, it is called as follows:

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

where **argc** is the argument count and **argv** is an array of character pointers to the arguments themselves. As indicated, **argc** is conventionally at least one and the first member of the array **argv** points to a string containing the name of the file.

argv is directly usable in another **execv** because **argv[argc]** is 0.

envp is a pointer to an array of strings that constitute the **environment** of the process. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a **NULL** element. The shell passes an environment entry for each global shell variable defined when the program is called. The C run-time start-off routine places a copy of **envp** in the global cell **environ**, which is used by **execv** and **execl** to pass the environment to any subprograms executed by the current program. The **exec** routines use lower-level routines as follows to pass an environment explicitly:

```
execle(file, arg0, arg1, . . . , argn, 0, environ);

execve(file, argv, environ);
```

execlp and **execvp** are called with the same arguments as **execl** and **execv** but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

exec fails and returns to the calling process if one or more of the following are true:

- A pointer argument is an illegal address, or the new process file is not as long as indicated by the size values in its header. **[EFAULT]**
- The path name for the new process file is null or the named file does not exist. **[ENOENT]**
- A component of the path prefix is not a directory. **[ENOTDIR]**
- A component of the path prefix denies search permission, or the new process file is not an ordinary file, or the new process file mode denies execute permission. **[EACCES]**
- The new process file has the appropriate access permission but has an invalid magic number in its header. **[ENOEXEC]**
- The new process file is a pure procedure (shared text) file currently open for writing by some process. **[ETXTBSY]**
- The new process requires more memory than is allowed by the system-imposed limit **MAXMEM**. **[ENOMEM]**
- The new process's argument list exceeds the size limit of 5120 bytes. **[E2BIG]**

Return Value

exec returns to the calling process only if an error has occurred; -1 is returned and **errno** is assigned an error code.

See Also

exit, fork

exit - Terminate a process.

Syntax

```
exit(status)
int status;
```

Description

exit terminates the calling process with the following consequences:

- All file descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a **wait**, it is notified of the calling process's termination and the low-order 8 bits (i.e., bits 0377) of **status** are made available to it; see **wait**.
- If the parent process of the calling process is not executing a **wait**, the calling process is transformed into a zombie process. A zombie process is a process that only occupies a slot in the process table; it has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by **times**.
- The parent process ID of all of the calling process' existing child processes and zombie processes is set to 1. This means the initialization process inherits each of these processes.
- An accounting record is written on the accounting file if the system's accounting routine is enabled; see **acct**.
- If the process ID, tty group ID, and process group ID of the calling process are equal, the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

See Also

signal, wait

Warning

See **Warning** in **signal**.

exp, log, log10, pow, sqrt - Exponential, logarithm, power, square root functions.

Syntax

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;
```

Description

exp returns the exponential function of **x**, which is the number *e* raised to the **x** power, where *e* is the base of the natural logarithms.

log returns the natural logarithm of **x**.

log10 returns the base 10 logarithm of **x**.

pow returns **x**^{y}.

sqrt returns the square root of **x**.

Diagnostics

exp and **pow** return a huge value when the correct value would overflow. Some overflows can cause **errno** to be set to **ERANGE**. **log** and **log10** return a huge negative value and set **errno** to **EDOM** when **x** is negative or zero. **pow** returns a huge negative value and sets **errno** to **EDOM** when **x** is negative or zero and **y** is not an integer, or when **x** and **y** are both zero. **sqrt** returns 0 and sets **errno** to **EDOM** when **x** is negative.

See Also

hypot, sinh

fclose, fflush - Close or flush a stream.

Syntax

```
#include <stdio.h>

int fclose(stream)
FILE *stream;

int fflush(stream)
FILE *stream;
```

Description

fclose causes any buffers for the named **stream** to be emptied and the file to be closed. Buffers allocated by the standard input/output functions are freed.

fclose is performed automatically for all open streams of a process on calling **exit.**

fflush causes any buffered output data for the named output **stream** to be written to the file. The stream remains open.

These functions return 0 for success and **EOF** if any errors are detected.

See Also

close, fopen, setbuf

fcntl - Control open files.

Syntax

```
#include <fcntl.h>

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

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.

The **cmds** 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** 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 remains open across **exec,** otherwise the file is closed on 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.

F_SETFL Set **file** status flags to **arg.** Only certain flags can be set.

fcntl fails if one or more of the following are true:

- **fildes** is not a valid open file descriptor for the calling process. **[EBADF]**
- **cmd** is **F_DUPFD** and 20 file descriptors are currently open. **[EMFILE]**
- **cmd** is **F_DUPFD** and **arg** is less than zero or greater than 19. **[EINVAL]**

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

Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

close, exec, open

ferror, feof, clearerr, fileno - Determine stream status.

Syntax

```
#include <stdio.h>

int feof(stream)
FILE *stream;

int ferror(stream)
FILE *stream

clearerr(stream)
FILE *stream

int fileno(stream)
FILE *stream;
```

Description

feof returns nonzero when end-of-file is read on the named input **stream,** otherwise zero.

ferror returns nonzero when an error has occurred reading or writing the named **stream,** otherwise zero. Unless cleared by **clearerr,** the error indicator lasts until the stream is closed.

clearerr resets the error indicator on the named **stream.**

fileno returns the integer file descriptor associated with the **stream;** see **open.**

feof, ferror, and **fileno** are implemented as macros; they cannot be redeclared.

See Also

open, fopen

fabs, floor, ceil, fmod - Absolute value, floor, ceiling, remainder functions.

Syntax

```
#include <math.h>

double fabs(x)
double x;

double floor(x)
double x;

double ceil(x)
double x;

double fmod(x, y)
double x, y;
```

Description

fabs returns the absolute value of **x.**

floor returns the largest integer not greater than **x** (as a double precision number).

ceil returns the smallest integer not less than **x** (as a double precision number).

fmod returns the number **f** such that **x = iy + f,** for some integer **i,** and **0 <= f < y.**

See Also

abs

fopen, freopen, fdopen - Open a stream.

Syntax

```
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;

FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen(fildes, type)
int fildes;
char *type;
```

Description

fopen opens the file named by **filename** and associates a stream with it. **fopen** returns a pointer used to identify the stream in subsequent operations.

type is a character string having one of the following values:

r Open for reading

w Create for writing

a Append; open for writing at end of file, or create for writing

r+ Open for update (reading and writing)

w+ Create for update (reading and writing)

a+ Append; open or create for update at end of file

freopen substitutes the named file in place of the open **stream.** It returns the original value of **stream.** The original stream is closed, regardless of whether the open ultimately succeeds.

freopen is typically used to attach the preopened streams **stdin, stdout,** and **stderr** to specified files.

fdopen associates a stream with a file descriptor obtained from **open, dup, creat,** or **pipe.** The **type** of the stream must agree with the mode of the open file. **type** must be provided because the standard I/O library has no way to query the type of an open file descriptor. **fdopen** returns the new stream.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening **fseek** or **rewind,** and input may not be directly followed by output without an intervening **fseek, rewind,** or an input operation that encounters end of file.

Diagnostics

fopen and **freopen** return **NULL** if **filename** cannot be opened.

See Also

open, fclose

fork - Create a new process.

Syntax

```
int fork();
```

Description

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

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.
- The child process' **utime, stime, cutime,** and **cstime** are set to 0; see **times.**
- The time left on the parent's alarm clock is not passed on to the child.

fork fails and no child process is created if one or more of the following are true:

- The system's limit on the total number of processes being executed would be exceeded, or the system's limit on the total number of processes associated with a single user would be exceeded. [**EAGAIN**]
- Not enough memory is available to create the forked image. [**ENOMEM**]

Return Value

If successful, **fork** returns 0 to the child process and returns the (nonzero) process ID of the child process to the parent process. Otherwise, -1 is returned to the parent process, no child process is created, and **errno** is assigned an error code.

See Also

exec, wait

fread, fwrite - Buffered binary input and output.

Syntax

```
#include <stdio.h>

int fread((char *) ptr, sizeof(*ptr), nitems, stream)
FILE *stream;

int fwrite((char *) ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

Description

fread reads, into a block beginning at **ptr, nitems** of data each of size **sizeof(*ptr)** from the named input **stream.** It returns the number of items actually read.

fwrite writes, from a block beginning at **ptr, nitems** of data each of size **sizeof(*ptr)** to the named output **stream. fwrite** returns the number of items actually written.

Each item read or written is **sizeof(*ptr)** bytes long.

See Also

fopen, getc, gets, printf, putc, puts, read, scanf, write

frexp, ldexp, modf - Split floating-point number into a mantissa and an exponent.

Syntax

```
double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;
int exp;

double modf(value, iptr)
double value, *iptr;
```

Description

frexp returns the mantissa of **value** as a double quantity **x,** of magnitude less than 1, and stores an integer **n** such that **value** = $x*2^n$ in the **int** referenced by **eptr.**

ldexp returns the double quantity: **value*2exp**

modf returns the positive fractional part of **value** and stores the integer part in the **int** referenced by **iptr.**

fseek, ftell, rewind - Reposition a stream.

Syntax

```
#include <stdio.h>

int fseek(stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

long ftell(stream)
FILE *stream;

rewind(stream)
FILE *stream;
```

Description

fseek sets the position of the next input or output operation on the **stream.** The new position is at the signed distance **offset** bytes from the beginning, the current position, or the end of the file, depending on whether **ptrname** has the value 0, 1, or 2.

fseek undoes any effects of **ungetc.**

ftell returns the current value of the file pointer offset relative to the beginning of the file associated with the named **stream.** The offset is measured in bytes.

rewind(stream) is equivalent to **fseek(stream,** 0L, 0).

After **fseek** or **rewind,** the next operation on an update file may be either input or output.

Diagnostics

fseek returns nonzero for improper seeks (in which case the file pointer is not moved), and zero if successful.

See Also

fopen, lseek

gamma - Log gamma function.

Syntax

```
#include <math.h>
extern int signgam;

double gamma(x)
double x;
```

Description

gamma returns the natural logarithm of the absolute value of the gamma function of the absolute value of x. The sign of the gamma function is returned in the external integer **signgam**. To calculate the gamma function, use C code like the following:

```
y = gamma(x);
if (y > 88.0)
        error();
else
        y = exp(y) * signgam;
```

Diagnostics

For negative integer arguments, a huge value is returned, and **errno** is set to **EDOM**.

getc, getchar, fgetc, getw - Get character or word from a stream.

Syntax

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar();

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

Description

getc returns the next character from the named input **stream.** Note that because **getc** is implemented as a macro, it may handle arguments with side effects incorrectly. In particular, **getc(*f++)** does not operate correctly.

getchar() is identical to **getc(stdin).**

fgetc behaves like **getc,** but is a genuine function, not a macro; it may therefore be used as an argument. **fgetc** runs more slowly than **getc,** but takes less space per invocation.

getw returns the next word from the named input **stream. getw** returns the constant **EOF** upon end-of-file or error, but since that is a valid integer value, **feof** and **ferror** should be used to check the success of **getw. getw** assumes no special alignment in the file.

Diagnostics

These functions return **EOF** on end-of-file or on a read error. A stop with the message "Reading bad file" written to **stderr** means that an attempt has been made to read from a stream that has not been opened for reading by **fopen.**

See Also

ferror, fopen, fread, gets, putc, scanf

getcwd - Get path name of current working directory.

Syntax

```
int getcwd(pnbuf, maxlen)
char *pnbuf;
int maxlen;
```

Description

Getcwd determines the path name of the current working directory and places it in the buffer **pnbuf.** The length excluding the terminating null character is returned. **maxlen** is the length of the buffer in bytes. If the length of the (null-terminated) path name exceeds **maxlen,** no path name is stored and a length <= 0 is returned.

Notes

maxlen (and the buffer referenced by **pnbuf)** must be one more than the maximum length of the path name, to allow for the terminating null.

getenv - Get value for environment name.

Syntax

```
char *getenv(name)
char *name;
```

Description

Getenv searches the environment list for a string of the form **name=value** and returns a pointer to a null-terminated string containing **value** if such a string is present, otherwise returns **NULL.**

See Also

exec

sh in "Commands" in the *XENIX 286 Reference Manual*

getgrent, getgrgid, getgrnam, setgrent, endgrent - Get group file entry.

Syntax

```
#include <grp.h>

struct group *getgrent();

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(name)
char *name;

int setgrent();

int endgrent();
```

Description

getgrent, getgrgid and **getgrnam** each return pointers to **group** structures. The format of a **group** structure is defined in **/usr/include/grp.h.**

The fields of this structure are

gr_name The name of the group

gr_passwd The encrypted password of the group

gr_gid The numerical group ID

gr_mem **NULL**-terminated vector of pointers to the individual member names

getgrent reads the next line of the system's group file, so successive calls to **getgrent** can be used to search the entire file. **getgrgid** and **getgrnam** search from the beginning of the file until a matching **gr_gid** or **gr_name** field is found, or until end-of-file is encountered.

setgrent rewinds the group file to allow repeated searches. **endgrent** closes the group file when processing is complete.

Diagnostics

NULL is returned by **getgrent, getgrid,** or **getgrnam** on end-of-file or error.

Files

/etc/group

See Also

getlogin, getpwent

Notes

All group structures returned are static data that is overwritten by each call.

getlogin - Get login name.

Syntax

```
char *getlogin();
```

Description

getlogin returns a pointer to the login name as found in **/etc/utmp.** **getlogin** can be used in conjunction with **getpwnam** to locate the correct password file entry when the same user ID is shared by several login names.

If **getlogin** is called within a process not attached to a terminal, it returns **NULL.** The correct procedure for determining the login name is to call **cuserid,** or to call **getlogin** and if it fails, to call **getpwuid.**

Diagnostics

Return **NULL** if a login name is not found.

Files

/etc/utmp

See Also

cuserid, getgrent, getpwent

Notes

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

getopt - Get option letter from argument vector.

Syntax

```
#include <stdio.h>

int getopt(argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
```

Description

getopt returns the next option letter in **argv** that matches a letter in **optstring. optstring** is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. **optarg** is set to point to the start of the option argument (if any) on return from **getopt.**

getopt places in **optind** the **argv** index of the next argument to be processed. **optind** should be zero before the first call to **getopt.**

When all options have been processed (i.e., up to the first nonoption argument), **getopt** returns **EOF.** The special option — can be used to delimit the end of the options; **EOF** is returned, and — is skipped.

Diagnostics

getopt prints an error message on **stderr** and returns a question mark (?) if it encounters an option letter not included in **optstring.**

Examples

The following program fragment shows how to process the arguments for a command that can take the mutually exclusive options **a** and **b,** and the options **f** and **o,** both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        int aflg = 0;
        int bflg = 0;
        int errflg = 0;
        char *ffile = NULL;
        char *ofile = NULL;

        extern int optind;
        extern char *optarg;

        while ((c = getopt (argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bflg++;
                        break;
                case 'f':
                        ffile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit(1);
        }

...
}
```

getpass - Read a password.

Syntax

```
char *getpass(prompt)
char *prompt;
```

Description

getpass reads a password from the file **/dev/tty**, or if that cannot be opened, from the standard input, after prompting with the null-terminated string **prompt** and disabling echoing. A pointer is returned to a null-terminated string of at most eight characters.

Files

/dev/tty

See Also

crypt

Notes

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

getpid, getpgrp, getppid - Get process, process group, and parent process IDs.

Syntax

```
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, fork, setpgrp, signal

getpw - Get name from unique identifier.

Syntax

```
getpw(uid, buf)
int uid;
char *buf;
```

Description

getpw searches the password file for the (numerical) **uid,** and fills in **buf** with the corresponding line, null-terminated; **getpw** returns zero if successful, nonzero if **uid** cannot be found.

Files

/etc/passwd

See Also

getpwent

Notes

This routine is included only for compatibility with prior systems and should not be used; see **getpwent** for routines to use instead.

getpwent, getpwuid, getpwnam, setpwent, endpwent - Get password file entry.

Syntax

```
#include <pwd.h>

struct passwd *getpwent();

struct passwd *getpwnam(name)
char *name;

struct passwd *getpwuid(uid)
int uid;

int setpwent();

int endpwent();
```

Description

getpwent, getpwnam, and **getpwuid** each returns a pointer to a structure containing the fields of an entry line in the password file. All return **NULL** on end-of-file or error. The structure of a password entry is defined in **/usr/include/pwd.h.** The **pw_comment** field is unused.

getpwent reads the next line in the file, so successive calls can be used to search the entire file. **getpwuid** and **getpwnam** search from the beginning of the file until a matching **uid** or **name** is found, or end-of-file is encountered.

setpwent rewinds the password file to allow repeated searches. **endpwent** closes the password file when processing is complete.

Files

/etc/passwd

See Also

getgrent, getlogin

Notes

The return values of **getpwent, getpwnam,** and **getpwuid** point to static data that is overwritten by each call.

gets, fgets - Get a string from a stream.

Syntax

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;
```

Description

gets reads a string into **s** from the standard input stream **stdin.** The string is terminated by a newline character, which is replaced in **s** by a null character. **gets** returns its argument. **gets** cannot check for string overflow.

fgets reads **n**-1 characters, or up to a newline character (which is retained), whichever comes first, from **stream** into the string **s.** The last character read into **s** is followed by a null character. **fgets** returns its first argument.

Diagnostics

gets and **fgets** return **NULL** on end-of-file or error.

See Also

ferror, fopen, fread, getc, puts, scanf

Notes

gets deletes the newline ending its input, but **fgets** keeps it.

getuid, geteuid, getgid, getegid - Get real user, effective user, real group, and effective group IDs.

Syntax

```
int getuid();

int geteuid();

int getgid();

int 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

hypot - Determine Euclidean distance.

Syntax

```
#include <math.h>

double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y; } z;
```

Description

Both **hypot** and **cabs** return

```
sqrt(x*x + y*y)
```

Both take precautions against unwarranted overflows.

cabs is used instead of **hypot** when computing the distance from the origin of a complex number **z**, represented as a structure containing the real (**x**) and imaginary (**y**) components of the complex number.

See Also

sqrt

ioctl - Control character devices.

Syntax

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

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

Description

ioctl performs a variety of functions on character special files (devices). The writeups of various devices in the section "Devices" in the *XENIX 286 Reference Manual* discuss how **ioctl** applies to them.

ioctl fails if one or more of the following are true:

- **fildes** is not a valid open file descriptor. [**EBADF**]
- **fildes** is not associated with a character special device. [**ENOTTY**]
- **request** or **arg** is not valid. (See **tty** in "Devices" in the *XENIX 286 Reference Manual.*) [**EINVAL**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

tty in "Devices" in the *XENIX 286 Reference Manual*

kill - Send a signal to a process or a group of processes.

Syntax

```
int kill(pid, sig)
int pid, 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** 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 effective user ID of the sending process must match the effective user ID of the receiving process unless the effective user ID of the sending process is super-user or the process is sending to itself.

The processes with process IDs of 0 and 1 are special processes and are referred to as **proc0** and **proc1** respectively.

If **pid** is greater than zero, **sig** is sent to the process with process ID equal to **pid. pid** may equal 1.

If **pid** is 0, **sig** is sent to all processes excluding **proc0** and **proc1** with process group ID 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** is sent to all processes excluding **proc0** and **proc1** with real user ID 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** is sent to all processes excluding **proc0** and **proc1.**

If **pid** is negative but not -1, **sig** is sent to all processes with process group ID equal to the absolute value of **pid.**

kill fails and no signal is sent if one or more of the following are true:

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

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

getpid, setpgrp, signal

kill in "Commands" in the *XENIX 286 Reference Manual*

l3tol, ltol3 - Convert between 3-byte integers and long integers.

Syntax

```
l3tol(lp, cp, n)
long *lp;
char *cp;
int n;

ltol3(cp, lp, n)
char *cp;
long *lp;
int n;
```

Description

l3tol converts a list of **n** 3-byte integers packed into a character string pointed to by **cp** into a list of long integers pointed to by **lp.**

ltol3 performs the reverse conversion from long integers (**lp**) to 3-byte integers (**cp**).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

See Also

file system in Appendix D, "File Formats"

link - Link a new directory entry to an existing file.

Syntax

```
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.

link fails and no link is created if one or more of the following are true:

- **path1** or **path2** is an illegal address. [**EFAULT**]
- Either path name is null or the file named by **path1** does not exist. [**ENOENT**]
- A component of either path prefix is not a directory. [**ENOTDIR**]
- A component of either path prefix denies search permission, or the directory that will contain the new link denies write permission. [**EACCES**]
- **path2** names an existing file or link. [**EEXIST**]
- **path1** names a directory, and the effective user ID is not super-user. [**EPERM**]
- The file named by **path1** and the link named by **path2** would be on different devices (different file systems). [**EXDEV**]
- The requested link is in a directory on a read-only file system. [**EROFS**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

link in "Commands" in the *XENIX 286 Reference Manual*

lock - Lock a process in primary memory.

Syntax

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

Description

If **flag** is nonzero, the process executing this call will not be swapped except if the process is required to grow. If **flag** is zero, the process is unlocked and can again be swapped normally. **lock** can only be called by the super-user.

Notes

Locked processes interfere with the compaction of primary memory and can cause deadlock. Systems with small memory configurations should avoid using this call. It is best to lock processes soon after booting because this tends to lock them into one end of memory, avoiding fragmentation.

locking - Lock or unlock a file region for reading or writing.

Syntax

```
#include <locking.h>

locking(fildes, mode, size);
int fildes, mode;
long size;
```

Description

locking allows a specified number of bytes in a file to be controlled by the locking process. Other processes that attempt to read or write a portion of the file containing the locked region may sleep until the area becomes unlocked depending on the mode in which the file region is locked. A process that attempts to write to or read a file region that has been locked against reading and writing by another process (using the **LK_LOCK** or **LK_NBLCK** mode) sleeps until that region of the file has been released by the locking process. A process that attempts to write to a file region that has been locked against writing by another process (using the **LK_RLCK** or **LK_NBRLCK** mode) sleeps until that region of the file has been released by the locking process, but a read request for that region proceeds normally.

A process that attempts to lock a region of a file that contains areas that have been locked by other processes sleeps if it has specified the **LK_LOCK** or **LK_RLCK** mode in its lock request, but returns with the error **EACCES** in **errno** if it specified **LK_NBLCK** or **LK_NBRLCK.**

fildes is the file descriptor returned by a successful **creat, open, dup,** or **pipe** system call.

mode specifies the type of lock operation to be performed on the file region. The available values for **mode** are

LK_UNLCK 0
: Unlocks the specified region. The calling process releases a region of the file it had previously locked.

LK_LOCK 1
: Locks the specified region. The calling process will sleep until the entire region is available if any part of it has been locked by a different process. The region is then locked for the calling process and no other process may read or write in any part of the locked region (lock against read and write).

LK_NBLCK 2
: Locks the specified region. If any part of the region is already locked by a different process, return the error **EACCES** in **errno** instead of waiting for the region to become available for locking (nonblocking lock request).

LK_RLCK 3
Same as **LK_LOCK** except that the locked region may be read by other processes (read permitted lock).

LK_NBRLCK 4
Same as **LK_NBLCK** except that the locked region may be read by other processes (nonblocking, read permitted lock).

size is the number of contiguous bytes to be locked or unlocked. The region to be locked starts at the current offset in the file. If **size** is 0, the entire file (up to a maximum of 2^{30} bytes) is locked or unlocked. **size** may extend beyond the end of the file, in which case only the process issuing the lock call may access or add information to the file within the boundary defined by size.

The potential for a deadlock occurs when a process controlling a locked area is put to sleep by accessing another process's locked area. Thus calls to **locking, read,** or **write** scan for a deadlock prior to sleeping on a locked region. An error return is made if sleeping on the locked region would cause a deadlock (and **errno** is assigned the value **EDEADLOCK).**

Lock requests may, in whole or part, contain or be contained by a previously locked region for the same process. When this occurs, or when adjacent regions are locked, the regions are combined into a single area if the mode of the lock is the same (i.e., either read permitted or regular lock). If the mode of the overlapping locks differ, the locked areas will be assigned assuming that the *most recent request* must be satisfied. Thus if a read-only lock is applied to a region, or part of a region, that had been previously locked by the same process against both reading and writing, the area of the file specified by the new lock will be locked for read only, while the remaining region, if any, will remain locked against reading and writing. There is no arbitrary limit to the number of regions that can be locked in a file. However, there is a system-wide limit on the total number of locked regions. This limit is 200 for XENIX systems.

Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process. Release of the center section of a locked area requires an additional locked element to hold the separated section. If the lock table is full, an error is returned, and the requested region is not released. Only the process that locked a file region can unlock it. An unlock request for a region that the process does not have locked, or that is already unlocked, has no effect. When a process terminates, all locked regions controlled by that process are unlocked.

If a process has done more than one open on a file, *all* locks put on the file by that process are released on the first close of the file.

Although no error is returned if locks are applied to special files or pipes, read/write operations on these types of files ignore any locks. Locks cannot be applied to a directory.

Diagnostics

locking returns -1 if any of the following errors occur: if any portion of the region has been locked by another process for the **LK_LOCK** and **LK_RLCK** actions and the lock request is to test only (**errno** = **EACCES**), if the file specified is a directory (**errno** = **EACCES**), if locking the region would cause a deadlock (**errno** = **EDEADLOCK**), or if there are no more free internal locks (**errno** = **EDEADLOCK**).

See Also

close, creat, dup, lseek, open, read, write

logname - Find login name of user.

Syntax

```
char *logname();
```

Description

logname returns a pointer to the null-terminated login name. It extracts the **$LOGNAME** variable from the user's environment.

Files

/etc/profile

See Also

env, login in "Commands", and **profile** in "Files" in the *XENIX 286 Reference Manual*

lsearch - Linear search and update.

Syntax

```
char *lsearch(key, base, nelp, width, compar)
char *key;
char *base;
int *nelp;
int width;
int(*compar)();
```

Description

lsearch returns a pointer into a table indicating the location at which a datum can be found. If the item was not in the table, **lsearch** adds it at the end of the table and still returns its address. **key** points to the item to be searched for. **base** points to the base of the table. **nelp** points to an integer containing the number of elements in the table. This number is incremented by **lsearch** if **lsearch** adds the element at the end of the table. Table entries have a fixed size of **width** bytes. **compar** is the comparison routine. The routine is called with pointers to the two elements being compared. The routine must return zero if the items are equal and nonzero otherwise.

Note that the format of the items and how they are tested for equality are determined by the caller via the caller-supplied comparison routine.

Notes

Unpredictable events can occur if there is not enough room in the table to add a new item.

See Also

bsearch, qsort

lseek - Move read/write file pointer.

Syntax

```
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 0, the pointer is set to **offset** bytes.
- If **whence** is 1, the pointer is set to its current location plus **offset.**
- If **whence** is 2, the pointer is set to the size of the file minus **offset.**

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

lseek fails and the file pointer is not changed if one or more of the following are true:

- **fildes** is not an open file descriptor. [**EBADF**]
- **fildes** refers to a pipe or **FIFO.** [**ESPIPE**]
- **whence** is not 0, 1, or 2. [**EINVAL** and **SIGSYS** signal]
- The resulting file pointer would be negative. [**EINVAL**]

If **lseek** fails, -1 is returned and **errno** is assigned an error code.

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

See Also

creat, dup, fcntl, open

malloc, free, realloc, calloc - Allocate main memory.

Syntax

```
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

Description

malloc and **free** provide a simple, general-purpose memory allocation package. **malloc** returns a pointer to a block of at least **size** bytes beginning on a word boundary.

The argument to **free** is a pointer to a block previously allocated by **malloc, realloc,** or **calloc;** the space is returned to the free list for further allocation.

It is evident that grave disorder can result if the free memory managed by **malloc** is overwritten or if **free** is called with an incorrect value.

malloc allocates the first contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls **sbrk** to get more memory from the system when there is no suitable space already free.

realloc changes the size of the block pointed to by **ptr** to **size** bytes and returns a pointer to the (possibly moved) block. The block contents are unchanged up to the lesser of the new and old sizes.

realloc also works if **ptr** points to a block freed since the last call of **malloc, realloc,** or **calloc;** thus sequences of **free, malloc,** and **realloc** can exploit the search strategy of **malloc** to do storage compaction.

calloc allocates space for an array of **nelem** elements of size **elsize** in bytes. The allocated space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

Diagnostics

malloc, realloc, and **calloc** return **NULL** if there is no available memory or if the free memory structure has been detectably corrupted by some process storing outside the bounds of a block. If **realloc** returns **NULL,** the block pointed to by **ptr** may be destroyed.

mknod - Make a directory, or a special or ordinary file.

Syntax

```
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 **mode** is interpreted as follows:

```
0170000    File type; one of the following:
     0010000    Named pipe special
     0020000    Character special
     0040000    Directory
     0050000    Name special file
     0060000    Block special
     0100000 or
     0000000    Ordinary file

0004000    Set user ID on execution

0002000    Set group ID on execution

0001000    Save text image after execution

0000777    Access permissions; constructed from the following
     0000400    Read by owner
     0000200    Write by owner
     0000100    Execute (search on directory) by owner
     0000070    Read, write, execute (search) by group
     0000007    Read, write, execute (search) by others
```

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

The file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's effective group ID.

The low-order 9 bits of **mode** are modified by the process' file mode creation mask: all bits set in the process' file mode creation mask are cleared. See **umask.** If **mode** indicates a block, character, or name special file, then **dev** is a configuration-dependent specification of a character or block I/O device or of a name file type. If **mode** does not indicate a block, character, or name special file, then **dev** is ignored. For block and character special files, **dev** is the special file's device number. For name special files, **dev** is the type of the name file, either a shared memory file or a semaphore.

mknod may be invoked only by the super-user for file types other than named pipe special.

mknod fails and the new file is not created if one or more of the following are true:

- The process's effective user ID is not super-user. [**EPERM**]
- **path** is an illegal address. [**EFAULT**]
- The path name is null or a component of the path prefix does not exist. [**ENOENT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- The directory in which the node is to be created is in a read-only file system. [**EROFS**]
- The named file exists. [**EEXIST**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

chmod, creatsem, exec, sdget, umask

file system in Appendix D, "File Formats"

mkdir, mknod in "Commands" in the *XENIX 286 Reference Manual*

Notes

Semaphore files should be created with **creatsem.**

Shared data files should be created with **sdget.**

mktemp - Make a unique file name.

Syntax

```
char *mktemp(template)
char *template;
```

Description

mktemp replaces **template** with a unique file name and returns the address of the template. The template should look like a file name with six trailing **X**'s; the **X**'s are replaced with a zero followed by the current process ID.

See Also

getpid

Notes

It is possible to run out of letters.

monitor - Prepare execution profile.

Syntax

```
monitor(lowpc, highpc, buffer, bufsize, nfunc)
int(*lowpc)(), (*highpc)();
short buffer[];
int bufsize, nfunc;
```

Description

An executable program created by **cc** with the **-p** option automatically includes calls for **monitor** with default parameters; **monitor** needn't be called explicitly except to gain fine control over profiling.

monitor is an interface to **profil. lowpc** and **highpc** are the addresses of two functions; **buffer** is the address of a user-supplied array of **bufsize** short integers. **monitor** arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of **lowpc** and the highest is just below **highpc.** At most **nfunc** call counts can be kept; only calls of functions compiled with the profiling option **-p** of **cc** are recorded. For the results to be significant, especially where there are small, heavily used routines, the buffer should be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext();
...
monitor(2, etext, buf, bufsize, nfunc);
```

etext lies just above all the program text.

To stop execution monitoring and write the results on the file **mon.out,** use

```
monitor(0);
```

The programming command **prof** can then be used to examine the results.

Files

mon.out

See Also

profil

cc, prof in "Programming Commands" in the *XENIX 286 Programmer's Guide*

mount - Mounts a file system.

Syntax

```
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 path name **dir** refer to the root directory of 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 by the super-user.

mount fails if one or more of the following are true:

- The effective user ID is not super-user. [**EPERM**]
- Either **spec** or **dir** is an illegal address. [**EFAULT**]
- A path name is null, or a named file does not exist. [**ENOENT**]
- A component of a path prefix is not a directory, or **dir** is not a directory. [**ENOTDIR**]
- **spec** is not a block special device. [**ENOTBLK**]
- The device associated with **spec** is not on line or not in the system. [**ENXIO**]
- The device associated with **spec** is currently mounted or the directory **dir** is currently mounted on, is someone's working directory, or is otherwise busy. [**EBUSY**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

umount

mount in "Commands" in the *XENIX 286 Reference Manual*

nap - Suspend execution for a short interval.

Syntax

```
long nap(period)
long period;
```

Description

The current process is suspended from execution for at least the number of milliseconds specified by **period**, or until a signal is received.

Return Value

If successful, a long integer giving the number of milliseconds actually slept is returned. If the process received a signal while napping, the return value is -1 and **errno** is assigned **EINTR**.

See Also

sleep

Notes

This function is driven by the system clock, which in most cases has a granularity of tens of milliseconds.

nice - Change priority of a process.

Syntax

```
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 higher 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.

nice fails and does not change the nice value if **incr** is negative and the effective user ID of the calling process is not super-user. **[EPERM]**

Return Value

If successful, **nice** returns the new nice value - 20. E.g., a return value of 3 indicates a new nice value of 23. Unlike most system calls, **nice** can return a negative value, including -1, even if successful.

See Also

exec

nice in "Commands" in the *XENIX 286 Reference Manual*

nlist - Get entries from name list.

Syntax

```
#include <a.out.h>
nlist(filename, nl)
char *filename;
struct nlist nl[];
```

Description

nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types, and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See **a.out** for a discussion of the symbol table structure.

Diagnostics

All type entries are set to 0 if the file cannot be found or if it is not a valid name list.

See Also

a.out in Appendix D, "File Formats"

open - Open file for reading or writing.

Syntax

```
#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 ORing flags from the following list (only one of the first three flags below should 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** and **write.**

When opening a FIFO with **O_RDONLY** or **O_WRONLY** set:

If **O_NDELAY** is set:

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

If **O_NDELAY** is clear:

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

When opening a file associated with a communication line:

If **O_NDELAY** is set:

The **open** returns without waiting for carrier.

If **O_NDELAY** is clear:

The **open** blocks until carrier is present.

O_APPEND If set, the file pointer is set to the end of the file when it is opened. Otherwise, the file pointer is set to the beginning of the file when it is opened.

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

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

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

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** fails if the file exists.

O_SYNCW Every write via this file descriptor will be synchronous, i.e., when the **write** system call returns, data is guaranteed to have been written to the device.

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

open fails and no file descriptor is returned if one or more of the following are true:

- **path** is an illegal address. **[EFAULT]**
- The path name is null, or **O_CREAT** is not set and the named file does not exist. **[ENOENT]**
- A component of the path prefix is not a directory. **[ENOTDIR]**
- A component of the path prefix denies search permission, or requested access to the named file is denied. **[EACCES]**
- The named file is a directory, and write access is requested. **[EISDIR]**
- Write access is requested for a file on a read-only file system. **[EROFS]**
- Write access is requested for a pure procedure (shared text) file that is being executed. **[ETXTBSY]**
- The named file is a device special file, and the device is not on line or not in the system; or, the named file is a FIFO, write-only access is requested, **O_NDELAY** is set, and no process has the file open for reading. **[ENXIO]**
- **O_CREAT** and **O_EXCL** are set, and the named file exists. **[EEXIST]**
- Twenty file descriptors are currently open in the calling process. **[EMFILE]**

Return Value

If successful, the (nonnegative) file descriptor is returned. Otherwise, -1 is returned, and **errno** is assigned an error code.

See Also

chmod, close, creat, dup, fcntl, lseek, read, umask, write

opensem - Open a semaphore.

Syntax

```
int = opensem(sem_name);
char *sem_name;
```

Description

opensem opens a semaphore file named by **sem_name** and returns a unique semaphore identification number used by **waitsem** and **sigsem. creatsem** should always be called to initialize the semaphore before the first attempt to open it, or to reset the semaphore if it has become inconsistent due to an exiting process neglecting to do a **sigsem** after issuing a **waitsem.**

Diagnostics

opensem returns -1 if one or more of the following errors occur; **errno** is assigned an error code:

- **sem_name** is an illegal address. **[EFAULT]**
- **sem_name** is null or the named semaphore does not exist. **[ENOENT]**
- A component of the path prefix is not a directory. **[ENOTDIR]**
- A component of the path prefix denies search permission or the semaphore file mode does not permit the requested access. **[EACCES]**
- **sem_name** specifies a file that is not a semaphore file. **[ENOTNAM]**
- The semaphore has become invalid due to inappropriate use. **[ENOTAVAIL]**

See Also

creatsem, waitsem, sigsem

pause - Suspend a process until a signal occurs.

Syntax

```
int pause();
```

Description

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

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

If the signal is "caught" by the calling process and control is returned from the signal catching function (see **signal**), the calling process resumes execution from the point of suspension, with a return value of -1 from **pause** and **errno** set to **EINTR**.

See Also

alarm, kill, signal, wait

perror, errno, sys_errlist, sys_nerr - Print system error messages.

Syntax

```
perror(s)
char *s;

extern int errno;
extern char *sys_errlist[];
extern int sys_nerr;
```

Description

perror produces a short error message on the standard error, describing the last error encountered during a system call from a C program. First the argument string **s** is printed, then a colon, then the message and a newline. To be of most use, the argument string should be the name of the program that incurred the error. The error number is taken from the external variable **errno**, which is set when errors occur but not cleared when correct calls are made.

To simplify variant formatting of messages, the vector of message strings **sys_errlist** is provided; **errno** can be used as an index in this table to get the message string without the newline. **sys_nerr** is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

pipe - Create an interprocess channel.

Syntax

```
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.

Writes of up to 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read-only file descriptor **fildes**[0] accesses the data written to **fildes**[1] on a first-in-first-out basis.

No process may have more than 20 file descriptors open simultaneously.

pipe fails if 19 or more file descriptors are already open. [**EMFILE**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

read, write

sh in "Commands" in the *XENIX 286 Reference Manual*

popen, pclose - I/O to or from a process.

Syntax

```
#include <stdio.h>

FILE *popen(command, type)
char *command, *type;

int pclose(stream)
FILE *stream;
```

Description

The arguments to **popen** are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either "r" for reading or "w" for writing. **popen** creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by **popen** should be closed by **pclose**, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command can be used as an input filter and a type "w" command as an output filter.

Diagnostics

popen returns **NULL** if files or processes cannot be created or if the shell cannot be accessed.

pclose returns -1 if **stream** is not associated with a **popen**ed command.

See Also

fclose, fopen, pipe, system, wait

Notes

Only one stream opened by **popen** can be in use by a process at once.

Buffered reading before opening an input filter can leave the standard input of that filter mispositioned. Similar problems with output filters can be forestalled by careful buffer flushing, e.g., with **fflush;** see **fclose.**

printf, fprintf, sprintf - Format output.

Syntax

```
#include <stdio.h>

int printf(format [ , arg ] ... )
char *format;

int fprintf(stream, format [ , arg ] ... )
FILE *stream;
char *format;

int sprintf(s, format [ , arg ] ... )
char *s, format;
```

Description

printf places output on the standard output stream **stdout. fprintf** places output on the named output **stream. sprintf** places output, followed by the null character ('\0') in consecutive bytes starting at **s;** the caller must ensure that enough storage is available. Each function returns the number of characters transmitted (not including the terminating null in the case of **sprintf**) or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its **args** under control of the **format.** The **format** is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more **args.** The results are undefined if there are insufficient **args** for the format. If the format is exhausted while **args** remain, the excess **args** are simply ignored.

Each conversion specification is introduced by the character **%.** After the **%,** the following appear in sequence:

- Zero or more **flags,** which modify the meaning of the conversion specification.
- An optional decimal digit string specifying a minimum **field width.** If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (see below) has been given) to the field width.
- A **precision** that gives the minimum number of digits to appear for the **d, o, u, x,** or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string: a null digit string is treated as zero.

- An optional **l** specifying that a following **d, o, u, x,** or **X** conversion character applies to a long integer **arg.**

- A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer **arg** supplies the field width or precision. The **arg** actually converted is not fetched until any **arg** that supplies width or precision is fetched, so the **arg**s specifying field width or precision must appear **before** the **arg** (if any) to be converted.

The flag characters and their meanings are

- The result of the conversion is left-justified within the field.

+ The result of a signed conversion is always formatted with a sign (+ or -).

blank If the first character of a signed conversion is not a sign, a blank is prepended to the result. This implies that if the blank and + flags both appear, the blank flag is ignored.

This flag specifies that the value is to be converted to an "alternate form." For **c, d, s,** and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x (X)** conversion, a nonzero result has **0x (0X)** prepended to it. For **e, E, f, g,** and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeros are *not* removed from the result. (They normally are.)

The conversion characters and their meanings are

d,o,u,x,X The integer **arg** is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**) respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is a null string (unless the conversion is **o, x,** or **X** and the **#** flag is present).

f The float or double **arg** is converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double **arg** is converted in the style "[-]d.ddde+-dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains exactly two digits.

g,G The float or double **arg** is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit.

c The character **arg** is printed.

s The **arg** is taken to be a string (character pointer) and characters from the string are printed until a null character ('\0') is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed.

% Print a **%**; no argument is converted.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by **printf** and **fprintf** are printed as if **putchar** had been called (see **putc**).

Examples

To print a date and time in the form "Sunday, July 3, 10:02", where **weekday** and **month** are pointers to null-terminated strings:

```
printf("%s, %s %d, %2d:%2d", weekday, month, day, hour, min);
```

To print pi to five decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
  /* The angle with tangent 1.0 is 45 degrees, or pi/4 radians. */
```

See Also

ecvt, putc, scanf

profil - Execution time profile.

Syntax

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

Description

buff points to an area of core whose length (in bytes) is given by **bufsiz.** After this call, the user's program counter is examined each clock tick, where a clock tick is some fraction of a second. **offset** is subtracted from it and the result multiplied by **scale.** If the resulting number corresponds to a word inside **buff,** that word is incremented.

scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1:1 mapping of program counter values 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 noninterrupting core clock).

Profiling is turned off by giving a **scale** of 0 or 1. It is rendered ineffective by giving a **bufsiz** of 0. Profiling is turned off when an **exec** is executed but remains on in child and parent both after a **fork.** Profiling is turned off if an update in **buff** would cause a memory fault.

See Also

monitor

prof in "Commands" in the *XENIX 286 Reference Manual*

ptrace - Trace a process.

Syntax

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

Description

ptrace provides a means by which a parent process can control the execution of a child process. Its primary use is in the implementation of breakpoint debugging; see **adb** in "Programming Commands" in the *XENIX 286 Programmer's Guide*. The child process behaves normally until it encounters a signal (see **signal** for the list), at which time it enters a stopped state and its parent is notified via **wait**. When the child is in the stopped state, its parent can examine and modify its "memory image" using **ptrace**. Also, the parent can cause the child either to terminate or to continue, with the possibility of ignoring the signal that caused it to stop.

The **addr** argument is dependent on the underlying machine type, specifically the process memory model. On systems where the memory management mechanism provides a uniform and linear address space to user processes, the argument is declared as

```
int *addr;
```

which is sufficient to address any location in the process's memory. On machines where the user address space is segmented (even if the particular program being traced has only one segment allocated), the form of the **addr** argument is

```
struct {
        int offset;
        int segment;
} *addr;
```

which allows the caller to specify segment and offset in the process address space.

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**. 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 The word at location **addr** in the address space of the child is returned to the parent process. If I and D spaces (Instruction and Data spaces) are separated, request **1** returns a word from I space, and request **2** returns a word from D space. If I and D spaces are not separated, either request **1** or request **2** can be used with equal results. The **data** argument is ignored. These two requests 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 (see **<sys/user.h>**) is returned to the parent process. The **data** argument is ignored. This request fails 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 spaces are separated, request 4 writes a word into I space, and request 5 writes a word into D space. If I and D spaces 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 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 to be written and **addr** is the location of the entry. The few entries that can be written are the general registers, any floating-point status registers, and certain bits of the processor status.

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. In a linear address space memory model, the value of **addr** must be **(int** *)1, or in a segmented address space the segment part of **addr** must be zero and the offset part of **addr** must be **(int** *)1. On successful completion, the value of **data** is returned to the parent. This request fails 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.**

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is **SIGTRAP.** This is part of the mechanism for implementing breakpoints. The exact implementation and behavior are CPU-dependent.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The **wait** system call is used to determine when a process stops; in such a case the termination status returned by **wait** has the value 0177 to indicate stoppage rather than genuine termination.

To prevent security violations, **ptrace** inhibits the set-user-ID facility on subsequent **exec** calls. If a traced process calls **exec,** it stops before executing the first instruction of the new image, raising signal **SIGTRAP.**

Errors

ptrace also fails if one or more of the following are true:

- **request** is an illegal number. **[EIO]**
- **pid** identifies a child that does not exist or has not executed a **ptrace** with request 0. **[ESRCH]**

See Also

exec, signal, wait

adb in "Programming Commands" in the *XENIX 286 Programmer's Guide*

Notes

The implementation and precise behavior of the **ptrace** system call depend on the specific CPU and on the process/memory model used. Code using **ptrace** is not likely to be portable across all implementations without some change.

putc, putchar, fputc, putw - Put a character or word on a stream.

Syntax

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

putchar(c)

int fputc(c, stream)
FILE *stream;

int putw(w, stream)
int w;
FILE *stream;
```

Description

putc appends the character **c** to the named output **stream.** **putc** returns the character written.

putchar(c) is defined as **putc(c, stdout).**

fputc behaves like **putc** but is a genuine function rather than a macro; it may therefore be used as an argument. **fputc** runs more slowly than **putc** but takes less space per invocation.

putw appends the word (i.e., integer) **w** to the output **stream.** **putw** neither assumes nor causes special alignment in the file.

The standard stream **stdout** is normally buffered only if the output does not refer to a terminal; this default may be changed by **setbuf.** The standard stream **stderr** is by default unbuffered unconditionally, but use of **freopen** will cause it to become unbuffered; **setbuf,** again, will set the state to whatever is desired. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. See also **fflush.**

Diagnostics

These functions return the constant **EOF** upon error. Since this is a good integer, **ferror** should be used to detect **putw** errors.

See Also

ferror, fopen, fwrite, getc, printf, puts

Notes

Because it is implemented as a macro, **putc** treats incorrectly a **stream** argument with side effects.

putpwent - Write a password file entry.

Syntax

```
#include <pwd.h>

int putpwent(p, f)
struct passwd *p;
FILE *f;
```

Description

putpwent is the inverse of **getpwent.** Given a pointer to a **passwd** structure created by **getpwent** (or **getpwuid** or **getpwnam**), **putpwuid** writes a line on the stream **f** that matches the format of **/etc/passwd.**

Diagnostics

putpwent returns nonzero if an error is detected during its operation, otherwise zero.

See Also

getpwent

passwd in "Files" in the *XENIX 286 Reference Manual*

puts, fputs - Put a string on a stream.

Syntax

```
#include <stdio.h>

int puts(s)
char *s;

int fputs(s, stream)
char *s;
FILE *stream;
```

Description

puts copies the null-terminated string **s** to the standard output stream **stdout** and appends a newline character.

fputs copies the null-terminated string **s** to the named output **stream.**

Neither routine copies the terminating null character.

Diagnostics

Both routines return **EOF** on error.

See Also

ferror, fopen, fwrite, gets, printf, putc

Notes

puts appends a newline; **fputs** does not.

qsort - Sort.

Syntax

```
qsort(base, nel, width, compar)
char *base;
int nel, width;
int (*compar)();
```

Description

qsort implements the quicker-sort algorithm. **base** is a pointer to the base of the data. **nel** is the number of (fixed-length) elements to be sorted in ascending order. **width** is the size of an element in bytes. **compar** is the comparison routine to use. The routine is called with pointers to the two elements being compared. The routine must return an integer less than, equal to, or greater than zero, depending on whether the first element is to be considered less than, equal to, or greater than the second element.

See Also

bsearch, lsearch, strcmp

sort in "Commands" in the *XENIX 286 Reference Manual*

rand, srand - Generate a random number.

Syntax

```
srand(seed)
unsigned seed;

int rand();
```

Description

rand uses a multiplicative congruential random number generator with period 2 to return successive pseudo-random numbers in the range from 0 to 2-1.

The generator is reinitialized by calling **srand** with 1 as argument. It can be set to a random starting point by calling **srand** with an unsigned integer in argument **seed.**

rdchk - Check to see if there is data to be read.

Syntax

```
rdchk(fdes)
int fdes;
```

Description

rdchk checks to see if a process will block if it attempts to read the file designated by the file descriptor **fdes.** **rdchk** returns 1 if there is data to be read or if it is the end of the file (**EOF**). An example of calling **rdchk** before **read:**

```
if(rdchk(fildes) > 0)
        read(fildes, buffer, nbytes);
```

Diagnostics

rdchk returns -1 if an error occurs (e.g., **EBADF**), 0 if the process will block if it issues a **read,** and 1 if it can read without blocking. **errno** is assigned the value **EBADF** if a **rdchk** is done on a semaphore file or if the file specified doesn't exist.

See Also

read

read - Read from a file.

Syntax

```
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, **read** starts at the 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 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, or if the number of bytes left in the file is less than **nbyte.** Zero is returned when end-of-file is reached.

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

- If **O_NDELAY** is set, **read** returns zero.
- If **O_NDELAY** is clear, **read** blocks 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 terminal that has no data currently available:

- If **O_NDELAY** is set, **read** returns zero.
- If **O_NDELAY** is clear, **read** blocks until data becomes available.

read fails if one or more of the following are true:

- **fildes** is not a valid file descriptor open for reading. **[EBADF]**
- **buf** is an illegal address. **[EFAULT]**

Return Value

If successful, a nonnegative integer is returned giving the number of bytes actually read. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

creat, dup, fcntl, ioctl, open, pipe

tty in "Devices" in the *XENIX 286 Reference Manual*

Notes

Reading a region of a file locked by another process with **locking** causes **read** to block indefinitely until the locked region is unlocked.

regex, regcmp - Compile and execute regular expressions.

Syntax

```
char *regcmp(string1 [, string2, ...], NULL);
char *string1, *string2, ...;

char *regex(re, subject [, ret0, ...]);
char *re, *subject, *ret0, ...;
```

Description

regcmp compiles a regular expression and returns a pointer to the compiled form. **malloc** is called to allocate space for the compiled expression. It is the user's responsibility to free unneeded space so allocated. A **NULL** returned by **regcmp** indicates an incorrect argument. The programming command **regcmp** ("Programming Commands" in the *XENIX 286 Programmer's Guide*) in most cases can eliminate any need for calling the **regcmp** function from user programs.

regex executes a compiled pattern against the subject string. Additional arguments can be passed to receive values back. **regex** returns **NULL** on failure, or a pointer to the next unmatched character if successful. A global character pointer, **_loc1** points to where the match began. **regcmp** and **regex** are derived from the pattern matching capabilities of the **ed** editor program, with slight changes in syntax and semantics. The following are the valid symbols and their associated meanings:

[ ]*.↑ These symbols retain their current meaning.

$ Matches the end of the string. **\n** matches a newline.

\- Within brackets, the hyphen means *through*. For example, **[a-f]** is equivalent to **[abcdef]**. - can appear by itself only if used as a first character or a last character. For example, the character class expression **[]-]** matches the characters] and -.

\+ A regular expression followed by + means "one or more times." For example, **[0-9]+** is equivalent to **[0-9][0-9]***.

{m} {m,} {m,u}
Integer values enclosed in **{#c** indicate the number of times that the preceding regular expression is to be applied. **m** is the minimum number and **u** is a number, less than or equal to 255, that is the maximum. If only **m** is present, it indicates the exact number of times the preceding regular expression is to be applied. **{m,}** is analogous to **{m,**infinity**}**. The operator * is equivalent to **{0,}**. The operator + is equivalent to **{1,}**.

(...)$n The value of the string matched by the enclosed regular expression is returned and assigned to the (n+1)th argument following the **subject** argument. At most ten enclosed regular expressions are allowed by **regex. regex** makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator, e.g., *, +, {}, can work on a single character or a regular expression enclosed in parentheses, e.g., **(a*(cb+)*)$0.**

By necessity, all of the symbols defined above are special. They must be escaped using backslash (\) to be used as themselves.

Examples

Example 1:

```
char *cursor, *newcursor, *ptr;
...
newcursor = regex((ptr = regcmp("\n", 0)), cursor);
```

This example matches a leading newline in the subject string referenced by **cursor.**

Example 2:

```
char ret0[9], *newcursor, *name;
...
name = regcmp("([A-Za-z][A-Za-z0-9_]{0,7})$0", 0);
newcursor = regex(name, "123Testing321", ret0);
```

This example matches through the string "Testing3" and returns the address of the character after the last matched character (the address of the test string plus 11). The string "Testing3" is copied to the character array **ret0.**

See Also

malloc

ed in "Commands" in the *XENIX 286 Reference Manual*

regcmp in "Programming Commands" in the *XENIX 286 Programmer's Guide*

sbrk, brk - Change data segment space allocation.

Syntax

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

char *brk(addr)
char *addr;
```

Description

sbrk and **brk** are used to dynamically change the amount of space allocated for the calling process's data segment; see **exec.** The change is made by resetting the process's "break value." The break value is the address of the first location beyond the end of the data segment. The amount of allocated data space increases as the break value increases.

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.

In large model programs, if **incr** is greater than the number of unallocated bytes remaining in the current data segment, **sbrk** automatically allocates all the requested bytes in a new data segment. This guarantees that the requested bytes will reside entirely in one segment. If **incr** is negative and equal to the number of allocated bytes in the current data segment, that segment is automatically freed for other use. If **incr** is negative and greater in magnitude than the number of allocated bytes in the current data segment, then that segment is freed, and the additional bytes are removed from the next data segment containing space allocated by **sbrk.**

sbrk fails without making any change in the allocated space if such a change would result in more space being allocated than is allowed by the system; see **ulimit.** **[ENOMEM]**

brk sets the current break value to **addr** and changes the allocated data space accordingly. **brk** fails if the address references a data segment that does not exist, or if it references beyond the maximum possible size of the current data segment.

Return Value

On successful completion, **sbrk** and **brk** return pointers to the beginning of the allocated space. Otherwise, -1 is returned and **errno** is assigned an error code. In large model programs, if **sbrk** allocates a new data segment, the return value is the starting address of the new segment.

See Also

exec

Notes

In large model programs, the call **sbrk(0)** does not necessarily return the starting address of the next **sbrk** call. In particular, if the next call causes an additional data segment to be allocated, the break values returned by these two calls will not be the same. The return value from **sbrk(0)** should only be regarded as a marker for the original end of data.

scanf, fscanf, sscanf - Convert and format input.

Syntax

```
#include <stdio.h>

int scanf(format [ , pointer ] ... )
char *format;

int fscanf(stream, format [ , pointer ] ... )
FILE *stream;
char *format;

int sscanf(s, format [ , pointer ] ... )
char *s, *format;
```

Description

scanf reads from the standard input stream **stdin.** **fscanf** reads from the named input **stream.** **sscanf** reads from the character string **s.** Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string **format** described below, and a set of **pointer** arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications used to direct interpretation of input sequences. The control string may contain

1. Blanks, tabs, or newlines, which cause input to be read up to the next non-white-space character.

2. An ordinary character (not **%**), which must match the next character of the input stream.

3. Conversion specifications consisting of the character **%**, an optional assignment suppressing character *****, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *****. An input field is defined as a string of nonspace characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are allowed:

% A single **%** is expected in the input at this point; no assignment is done.

d A decimal integer is expected; the corresponding argument should be an integer pointer.

o An octal integer is expected; the corresponding argument should be an integer pointer.

x A hexadecimal integer is expected; the corresponding argument should be an integer pointer.

s A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which is added automatically. The input field is terminated by a space character or a newline.

c A character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next nonspace character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

e, f A floating-point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a **float**. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e** followed by an optionally signed integer.

[Indicates a string that is not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a caret (), the input field consists of all characters up to the first character that is not in the set between the brackets; if the first character after the left bracket is a caret, then the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. The corresponding argument must point to a character array.

The conversion characters **d, o,** and **x** may be capitalized and/or preceded by **l** to indicate that a pointer to **long** rather than to **int** is in the argument list. Similarly, the conversion characters **e** and **f** may be capitalized and/or preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The character **h** will, some time in the future, indicate **short** data items.

scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream. This is very important to remember, because subtle errors can occur if this is not taken into account.

scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

Note that the success of literal matches and suppressed assignments is not directly determinable. Also, trailing white space, including newlines, is left unread unless matched in the control string.

Examples

The call

```
int i; float x; char name[50];
...
scanf("%d%f%s", &i, &x, name);
```

with the input line

```
25 54.32E-1 thompson
```

assigns 25 to **i**, 5.432 to **x**, and the character sequence **thompson\0** to the character array **name**. The call

```
int i; float x; char name[50];
...
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with the input line

```
56789 0123 56a72
```

assigns 56 to **i** and 789.0 to **x**, skips **0123**, and assigns the character sequence **56\0** to the character array **name**. The next call to **getchar** will return 'a'.

See Also

atof, getc, printf

sdenter, sdleave - Synchronize access to a shared data segment.

Syntax

```
#include <sd.h>

int sdenter(addr, flags)
char *addr;
int flags;

int sdleave(addr)
char *addr;
```

Description

sdenter is used to indicate that the current process is about to access the contents of a shared data segment. The actions performed depend on the value of **flags**. **flags** values are formed by ORing together entries from the following list:

SD_NOWAIT If another process has called **sdenter** but not **sdleave** for the indicated segment, and the segment was not created with the **SD_UNLOCK** flag set, returns an error instead of waiting for the segment to become free.

SD_RDONLY Indicates that the process wants only to read the data, not modify it.

sdleave is used to indicate that the current process is done modifying the contents of a shared data segment.

Only changes made between invocations of **sdenter** and **sdleave** are guaranteed to be reflected in other processes. **sdenter** and **sdleave** are very fast; consequently, it is recommended that they be called frequently rather than leave **sdenter** in effect for any period of time. In particular, system calls should be avoided between **sdenter** and **sdleave** calls.

The **fork** system call is forbidden between calls to **sdenter** and **sdleave** if the segment was created without the **SD_UNLOCK** flag.

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** assigned an error code.

See Also

sdfree, sdgetv

sdget, sdfree - Attach or detach a shared data segment.

Syntax

```
#include <sd.h>

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

int sdfree(addr);
char *addr;
```

Description

sdget attaches a shared data segment to the data space of the current process. The actions performed are controlled by the value of **flags. flags** values are constructed by ORing flags from the following list:

SD_RDONLY Attach the segment for reading only.

SD_WRITE Attach the segment for both reading and writing.

SD_CREAT If the segment named by **path** exists, this flag has no effect. Otherwise, the segment is created according to the values of **size** and **mode.** Read and write access to the segment is granted to other processes based on the permissions passed in **mode;** permissions are encoded in the same way as for normal files. Execute permission is meaningless. The segment is initialized to contain all zeros.

SD_UNLOCK If the segment is created because of this call, the segment is made so that more than one process can be between **sdenter** and **sdleave** calls.

sdfree detaches the current process from the shared data segment attached at the specified address. If the current process has done an **sdenter** but not an **sdleave** for the specified segment, an **sdleave** is done before detaching the segment.

When no process remains attached to the segment, the contents of that segment disappear, and no process can attach to the segment without creating it by calling **sdget** with the **SD_CREAT** flag.

Return Value

If successful, the address at which the segment was attached is returned (by either function). Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

sdenter, sdgetv

Notes

Use of the **SD_UNLOCK** flag on systems without hardware support for shared data may cause severe performance degradation.

sdgetv, sdwaitv - Synchronize shared data access.

Syntax

```
#include <sd.h>

int sdgetv(addr)
int sdwaitv(addr, vnum)
char *addr;
int vnum;
```

Description

sdgetv and **sdwaitv** can be used to synchronize cooperating processes that are using shared data segments. The return value of both routines is the version number of the shared data segment attached to the process at address **addr.** The version number of a segment changes whenever some process does an **sdleave** for that segment.

sdgetv simply returns the version number of the indicated segment.

sdwaitv causes the current process to sleep until the version number for the indicated segment is no longer equal to **vnum.**

Return Value

If successful, both functions return a positive integer that is the current version number for the specified shared segment. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

sdenter, sfree

setbuf - Assign buffering to a stream.

Syntax

```
#include <stdio.h>

setbuf(stream, buf)
FILE *stream;
char *buf;
```

Description

setbuf is used after a stream has been opened but before it is read or written. It causes the character array **buf** to be used instead of an automatically allocated buffer. If **buf** is **NULL**, I/O will be completely unbuffered.

The constant **BUFSIZ** tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from **malloc** upon the first **getc** or **putc** on the file, except that output streams directed to terminals and the standard error stream **stderr** are normally not buffered.

A common error is allocating buffer space local to a function or a code block, and then failing to close the stream in the same function or block.

See Also

fopen, getc, malloc, putc

setjmp, longjmp - Perform a nonlocal "goto."

Syntax

```
#include <setjmp.h>

int setjmp(env)
jmp_buf env;

int longjmp(env, val)
jmp_buf env;
int val;
```

Description

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

setjmp saves its stack environment in **env** for later use by **longjmp.** It returns value 0.

longjmp restores the environment saved by the last call of **setjmp.** It then returns in such a way that execution continues as if the call of **setjmp** had just returned the value **val** to the corresponding call to **setjmp;** control must not have returned from the function that calls **setjmp** in the interim. **longjmp** cannot return the value 0. If **longjmp** is invoked with a second argument of 0, it returns 1. All accessible data have values as of the time **longjmp** was called.

See Also

signal

setpgrp - Set process group ID.

Syntax

```
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, fork, getpid, intro, kill, signal

setuid, setgid - Set user and group IDs.

Syntax

```
int setuid(uid)
int uid;

int setgid(gid)
int gid;
```

Description

setuid is used to set the real user ID and effective user ID of the calling process. Either the previous effective user ID of the process must be super-user, or the previous real user ID of the process must be equal to the **uid** argument. Otherwise, the user IDs are not changed. [**EPERM**]

setgid is used to set the real group ID and effective group ID of the calling process. Either the effective user ID of the process must be super-user, or the previous real group ID of the process must be equal to the **gid** argument. Otherwise, the group IDs are not changed. [**EPERM**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

getuid

shutdn - Flush block I/O and halt the CPU.

Syntax

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

shutdn(sblk)
struct filsys *sblk;
```

Description

shutdn causes all information in core memory that should be on disk to be written out. This includes modified super-blocks, modified inodes, and delayed block I/O. The super-blocks of all writable file systems are flagged as "clean," so that they can be remounted without cleaning when XENIX is rebooted. **shutdn** then prints "Normal System Shutdown" on the console and halts the CPU.

If **sblk** is nonzero, it specifies the address of a super-block that is written to the root device as the last I/O before the halt. This facility is provided to allow file system repair programs to supercede the system's copy of the root super-block with one of their own.

shutdn locks out all other processes while it is doing its work. However, it is recommended that user processes be killed before calling **shutdn** as some types of disk activity could cause file systems to not be flagged "clean."

The caller must be the super-user.

See Also

mount

fsck, shutdown in "Commands" in the *XENIX 286 Reference Manual*

signal - Specify the action to be taken when a particular signal is received.

Syntax

```
#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	01	Hangup
SIGINT	02	Interrupt
SIGQUIT	03*	Quit
SIGILL	04*	Illegal instruction (not reset when caught)
SIGTRAP	05*	Trace trap (not reset when caught)
SIGIOT	06*	I/O trap instruction
SIGEMT	07*	Emulator trap instruction
SIGFPE	08*	Floating-point exception
SIGKILL	09	Kill (cannot be caught or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	Write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGUSR1	16	User-defined signal 1
SIGUSR2	17	User-defined signal 2
SIGCLD	18	Death of a child (see **Warning** below)
SIGPWR	19	Power fail (see **Warning** below)

Signals marked with an asterisk (*) may write a core image of the signalled process as their default action.

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

- **SIG_DFL** Terminate the calling process on receiving the specified signal. When a process is terminated, there are these consequences:

  - Any open file descriptors for the process are closed.
  - If the parent process of the terminated process called **wait,** it is notified of its child process's termination and supplied the terminating signal number; see **wait.** Otherwise, if the parent process of the terminated process did not call **wait,** the terminated process is transformed into a "zombie" process. (See **exit** for a definition of zombie processes.)
  - The parent process ID of each of the terminated process's child processes is set to 1; this means that the **proc1** initialization process inherits the child processes of a terminated process.
  - If process accounting is enabled, then an accounting record is written on the accounting file; see **acct.**
  - If the terminated process's process ID, tty group ID, and process group ID are equal, then the signal **SIGHUP** is sent to all processes with a process group ID equal to the process group ID of the terminated process.
  - A "core image" of the terminated process is stored in the current working directory of the terminated process if **sig** is one of the signals marked with an asterisk in the above list, and if the effective user ID and the real user ID of the terminated process are equal, and if an ordinary file named "core" either exists and is writable or can be created in the current working directory of the terminated process. If the "core" file is created, it has a mode of octal 0666 modified by the file creation mask (see **umask**), a file owner ID equal to the effective user ID of the terminated process, and a file group ID equal to the effective group ID of the terminated process.

- **SIG_IGN** Ignore the specified signal. (Note that **SIGKILL** cannot be ignored.)

- function address

  On receiving the specified signal, call the specified function. The signal number **sig** will be passed as the only argument to the function. After the function is called, the kernel changes the action for the signal to **SIG_DFL,** unless the signal is **SIGILL, SIGTRAP, SIGCLD,** or **SIGPWR. signal** must be called again to use the function to catch the specified signal again. (Note that **SIGKILL** cannot be caught by a function.)

When a signal to be caught with a function call occurs during an interruptable system call (e.g., **pause**), then the signal-catching function is executed and the interrupted system call returns -1 to the calling process with **errno** assigned the error code **EINTR.**

Calling **signal** cancels a pending signal **sig** except for a pending **SIGKILL** signal.

signal fails if one or more of the following are true:

- **sig** is **SIGKILL** or an illegal signal number. [**EINVAL**]
- **func** is an illegal address. [**EFAULT**]

Return Value

If successful, **signal** returns the previous value of **func** for the specified signal **sig.** Otherwise, -1 is returned, and **errno** is assigned an error code.

See Also

kill, pause, ptrace, setjmp, wait

kill in "Commands" in the *XENIX 286 Reference Manual*

Warning

Two other signals that behave differently than the signals described above exist in this release of the system:

SIGCLD	18	Death of a child (not reset when caught)
SIGPWR	19	Power fail (not reset when caught)

There is no guarantee that, in future releases of XENIX, these signals will continue to behave as described below; they are included only for compatibility with other versions of XENIX. Their use in new programs is strongly discouraged.

For these signals, **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 - Ignore signal (default action)
The signal is to be ignored.

SIG_IGN - Ignore signal
The signal is to be ignored. Also, if **sig** is **SIGCLD,** the calling process's child processes will not create zombie processes when they terminate; see **exit.**

function address - catch signal
If the signal is **SIGPWR,** the action to be taken is the same as that described above for **func** equal to function address. The same is true if the signal is **SIGCLD,** except that while the process is executing the signal-catching function, any received **SIGCLD** signals will be queued and the signal-catching function will be continually re-entered until the queue is empty.

SIGCLD affects two other system calls (**wait** and **exit**) in the following ways:

wait If the func value of **SIGCLD** is set to **SIG_IGN** and a **wait** is executed, the **wait** will block until all of the calling process's child processes terminate; it will then return a value of -1 with **errno** set to **ECHILD.**

exit If, in the exiting process's parent process, the **func** value of **SIGCLD** is set to **SIG_IGN,** the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

Notes

The constant **NSIG** in **signal.h** standing for the number of signals is always at least one greater than the actual number.

sigsem - Signal a process waiting on a semaphore.

Syntax

```
sigsem(sem_num);
int sem_num;
```

Description

sigsem signals a process waiting on the semaphore **sem_num** that it may proceed and use the resource governed by the semaphore. **sigsem** is used in conjunction with **waitsem** to allow synchronization of processes wishing to access a resource. One or more processes may **waitsem** on the given semaphore and will be put to sleep until the process that currently has access to the resource issues a **sigsem** call. If there are any waiting processes, **sigsem** causes the process next in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first in first out (FIFO) order.

Diagnostics

sigsem fails and has no effect if one or more of the following are true:

- **sem_num** does not refer to a semaphore file. [**ENOTNAM**]
- **sem_num** refers to a semaphore that has not been opened with **opensem.** [**EBADF**]
- **sem_num** refers to a semaphore that is not owned by the calling process, i.e., the calling process has not called **waitsem** on the semaphore before **sigsem.** [**ENAVAIL**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

creatsem, opensem, waitsem

sinh, cosh, tanh - Hyperbolic functions.

Syntax

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

Description

These functions compute the designated hyperbolic functions for real arguments.

Diagnostics

sinh and **cosh** return huge values of appropriate sign when the correct value would overflow.

sleep - Suspend execution for an interval.

Syntax

```
unsigned sleep(seconds)
unsigned seconds;
```

Description

The current process is suspended from execution for the number of **seconds** specified by the argument. The actual suspension time may be less than that requested because scheduled wakeups occur at fixed one-second intervals, and any caught signal will terminate the **sleep** following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by **sleep** will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested **sleep** time, or was prematurely aroused due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling **sleep**; if the **sleep** time exceeds the time until such alarm signal, the process sleeps only until the alarm signal would have occurred, and the caller's alarm catch routine is executed just before the **sleep** routine returns. But if the **sleep** time is less than the time until such alarm, the prior alarm time is reset to go off at the same time it would have gone off without the intervening **sleep**.

See Also

alarm, nap, pause, signal

ssignal, gsignal - Software signals.

Syntax

```
#include <signal.h>

int(*ssignal(sig, action))()
int sig, (*action)();

int gsignal(sig)
int sig;
```

Description

ssignal and **gsignal** implement a software facility similar to **signal.** This facility is used by the standard C library to enable the user to indicate the disposition of error conditions and is also made available to the user for his own purposes.

Software signals made available to users are associated with integers in the range 1 through 15. An *action* for a software signal is *established* by a call to **ssignal,** and a software signal is *raised* by a call to **gsignal.** Raising a software signal causes the action established for that signal to be *taken.*

The first argument to **ssignal** is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action function* or one of the constants **SIG_DFL** (default) or **SIG_IGN** (ignore). **ssignal** returns the action previously established for that signal type; if no action has been established or the signal number is illegal, **ssignal** returns **SIG_DFL.**

gsignal raises the signal identified by its argument, **sig:**

- If an action function has been established for **sig,** then that action is reset to **SIG_DFL** and the action function is entered with argument **sig. gsignal** returns the value returned to it by the action function.
- If the action for **sig** is **SIG_IGN, gsignal** returns the value 1 and takes no other action.
- If the action for **sig** is **SIG_DFL, gsignal** returns the value 0 and takes no other action.
- If **sig** has an illegal value or no action was ever specified for **sig, gsignal** returns the value 0 and takes no other action.

Notes

There are some additional signals with numbers outside the range 1 through 15 used by the standard C library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the standard C library.

stat, fstat - File status.

Syntax

```
#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 for 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 fields:

```
ushort      st_mode;    /* File mode; see mknod */
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 */
                        /* 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 of last data modification */
time_t      st_ctime;   /* Time of last file status change */
                        /* Times are measured in seconds since */
                        /* 00:00:00 GMT, Jan. 1, 1970 */
```

The following list gives more information about four of these fields:

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

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

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

st_rdev Device identification. In the case of block and character special files this contains the device major and minor numbers; in the case of shared memory and semaphores, it contains the type code. The file **/usr/include/sys/types.h** contains the macros **major** and **minor** for extracting major and minor numbers from **st_rdev.** See **/usr/include/sys/stat.h** for the semaphore and shared memory type code values **S_INSEM** and **S_INSHD.**

stat fails if one or more of the following are true:

- **path** or **buf** is an illegal address. [**EFAULT**]
- The path name is null, or the named file does not exist. [**ENOENT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- A component of the path prefix denies search permission. [**EACCES**]

fstat fails if one or more of the following are true:

- **fildes** is not a valid open file descriptor. [**EBADF**]
- **buf** is an illegal address. [**EFAULT**]

Return Value

For both functions, if a call is successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

chmod, chown, creat, link, mknod, time, unlink

stdio - Standard buffered input and output.

Syntax

```
#include <stdio.h>
FILE *stdin, *stdout, *stderr;
```

Description

The **stdio** library contains an efficient, user-level I/O buffering scheme. The in-line macros **getc** and **putc** handle characters quickly. The macros **getchar, putchar,** and the higher-level routines **fgetc, fgets, fprintf, fputc, fputs, fread, fscanf, fwrite, gets, getw, printf, puts, putw,** and **scanf** all use **getc** and **putc;** they can be freely intermixed.

A file with associated buffering is called a "stream" and is declared to be a pointer to a defined type **FILE. fopen** creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, three open streams with constant pointers are declared in the "include" file and associated with the standard open files:

- **stdin** standard input file
- **stdout** standard output file
- **stderr** standard error file

An integer constant **EOF** is returned on end-of-file or error by most functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

Most of the functions and constants mentioned in this section of the manual are declared in that "include" file and are described elsewhere. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): **getc, getchar, putc, putchar, feof, ferror,** and **fileno.**

Diagnostics

Invalid stream pointers can cause grave disorder, possibly including program termination. Individual function descriptions describe possible error conditions.

See Also

close, ctermid, cuserid, fclose, ferror, fopen, fread, fseek, getc, gets, open, popen, printf, putc, puts, read, scanf, setbuf, system, tmpnam, write

stime - Set the time.

Syntax

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

time_t 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.

stime fails if the effective user ID of the calling process is not super-user. **[EPERM]**

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

time

STRING: strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok - String operations.

Syntax

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;
int n;

int strcmp(s1, s2)
char *s1, *s2;

int strncmp(s1, s2, n)
char *s1, *s2;
int n;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;
int n;

int strlen(s)
char *s;

char *strchr(s, c)
char *s, c;

char *strrchr(s, c)
char *s, c;

char *strpbrk(s1, s2)
char *s1, *s2;

int strspn(s1, s2)
char *s1, *s2;

int strcspn(s1, s2)
char *s1, *s2;

char *strtok(s1, s2)
char *s1, *s2;
```

Description

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

strcat appends a copy of string **s2** to the end of string **s1**. **strncat** copies at most **n** characters. Both return a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as **s1** is lexicographically greater than, equal to, or less than **s2**. **strncmp** makes the same comparison but examines at most **n** characters.

strcpy copies string **s2** to **s1**, stopping after the null character has been moved. **strncpy** copies exactly **n** characters, truncating or null-padding **s2**; the target may not be null-terminated if the length of **s2** is **n** or more. Both return **s1**.

strlen returns the number of nonnull characters in **s**.

strchr (strrchr) returns a pointer to the first (last) occurrence of character **c** in string **s**, or **NULL** if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.

strpbrk returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or **NULL** if no character from **s2** exists in **s1**.

strspn (strcspn) returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

strtok considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2**. The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. Subsequent calls, with **NULL** for the first argument, work through the string **s1** in this way until no tokens remain. The separator string **s2** can be different from call to call. When no token remains in **s1**, **NULL** is returned.

Notes

strcmp uses native character comparison, which is signed on some machines and unsigned on others.

All string movement is performed character by character starting at the left. Thus overlapping moves toward lower addresses work as expected, but overlapping moves toward higher addresses can yield surprises.

swab - Swap bytes.

Syntax

```
swab(from, to, nbytes)
char *from, *to;
int nbytes;
```

Description

swab copies **nbytes** pointed to by **from** to the position pointed to by **to,** exchanging adjacent even and odd bytes. It is useful for transporting binary data between machines that differ in the ordering of bytes within words. **nbytes** should be even.

sync - Update disks.

Syntax

```
sync();
```

Description

sync causes all information in memory that should be on disk to be written out. This includes modified super-blocks, modified inodes, and delayed block I/O.

It should be used by programs that examine a file system, for example **fsck, df,** etc.

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

See Also

sync in "Commands" in the *XENIX 286 Reference Manual*

system - Execute a shell command.

Syntax

```
#include <stdio.h>

int system(string)
char *string;
```

Description

system causes the **string** to be given to the shell **sh** as input as if the string had been typed as a command at a terminal. The current process waits until the new shell invocation has completed then returns the exit status of the shell.

Diagnostics

system stops if it can't execute **sh.**

See Also

exec

sh in "Commands" in the *XENIX 286 Reference Manual*

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - Terminal functions.

Syntax

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char * tgetstr(id, area)
char *id, **area;

char * tgoto(cm, destcol, destline)
char *cm;
int destcol, destline;

tputs(cp, affcnt, outc)
char *cp;
int affcnt;
int(*outc)();
```

Description

These functions extract and use capabilities from the terminal capability data base **termcap,** described in "Files" in the *XENIX 286 Reference Manual.* These are low-level routines; see CURSES for a higher-level package.

tgetent extracts the entry for terminal **name** into the buffer at **bp. bp** should be a character buffer of size 1024 and must be retained through all subsequent calls to **tgetnum, tgetflag,** and **tgetstr. tgetent** returns -1 if it cannot open the **termcap** file, 0 if the terminal name given does not have an entry, and 1 if all goes well. **tgetent** searches the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the **termcap** file. If it does begin with a slash, the string is used as a path name rather than **/etc/termcap.** This can speed up entry into programs that call **tgetent,** as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file **/etc/termcap.**

tgetnum gets the numeric value of capability **id,** returning -1 if is not given for the terminal. **tgetflag** returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. **tgetstr** gets the string value of capability **id,** placing it in the buffer at **area,** advancing the **area** pointer. It decodes the abbreviations for this field described in **termcap,** except for cursor addressing and padding information.

tgoto returns a cursor addressing string decoded from **cm** to go to column **destcol** in line **destline.** It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n, CONTROL-D** or **NULL** in the returned string. (Programs that call **tgoto** should be sure to turn off the TAB3 bit (see **tty** in "Devices" in the *XENIX 286 Reference Manual*), since **tgoto** may now output a tab. Note that programs using **termcap** should normally turn off TAB3 anyway since some terminals use **CONTROL-I** for other functions, such as nondestructive space. If a **%** sequence is given that is not understood, then **tgoto** returns **OOPS.**

tputs decodes the leading padding information of the string **cp; affcnt** gives the number of lines affected by the operation, or 1 if this is not applicable; **outc** is a routine called with each character in turn. The external variable **ospeed** should contain the output speed of the terminal as encoded by **stty.** The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null character is inappropriate.

Files

/usr/lib/libtermcap.a	**-ltermcap** library
/etc/termcap	terminal capabilities data base

See Also

CURSES

tty in "Devices" and **termcap** in "Files" in the *XENIX 286 Reference Manual*

Notes

These routines can be linked by using the linker option **-ltermcap.**

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

time, ftime - Get time and date.

Syntax

```
time_t time(tloc)
time_t *tloc;

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

ftime(tp)
struct timeb *tp;
```

Description

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

If **tloc** (taken as an integer) is not **NULL**, the return value is also stored in the location to which **tloc** points.

ftime returns the time in a structure (see below under **Return Value**).

time fails if **tloc** is not **NULL** and is an illegal address. [**EFAULT**]

ftime fails if **tp** is an illegal address. [**EFAULT**]

Return Value

If successful, **time** returns the elapsed time in seconds. Otherwise, -1 is returned and **errno** is assigned an error code.

ftime fills in a structure pointed to by its argument, as defined by **<sys/timeb.h>**:

```
/*
* Structure returned by ftime system call
*/
struct timeb {
        time_t time;
        unsigned short millitm;
        short  timezone;
        short dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Savings time applies locally during the appropriate part of the year.

See Also

ctime, stime

date in "Commands" in the *XENIX 286 Reference Manual*

times - Get process and child process times.

Syntax

```
#include <times.h>

long times(buffer)
struct tmbuf {
        long utime;
        long stime;
        long cutime;
        long cstime;
} buffer;
```

Description

times fills the structure pointed to by **buffer** with time-accounting information. This information comes from the calling process and each of its terminated child processes for which it has executed a **wait.**

All times are in clock ticks where a tick is some fraction of a second.

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

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

cutime is the sum of the **utimes** and **cutimes** of the child processes.

cstime is the sum of the **stimes** and **cstimes** of the child processes.

times fails if **buffer** is an illegal address. **[EFAULT]**

Return Value

If successful, **times** returns the elapsed real time, in clock ticks, since an arbitrary point in the past, such as the system start-up time. This point does not change from one invocation of **times** to another (so long as the system does not shut down). Otherwise, if **times** fails, -1 is returned and **errno** is assigned an error code.

See Also

exec, fork, time, wait

tmpfile - Create a temporary file.

Syntax

```
#include <stdio.h>

FILE *tmpfile();
```

Description

tmpfile creates a temporary file and returns a corresponding **FILE** pointer. Arrangements are made so that the file is automatically deleted when the process using it terminates. The file is opened for update.

See Also

creat, fopen, mktemp, tmpnam, unlink

tmpnam - Create a name for a temporary file.

Syntax

```
#include <stdio.h>

char *tmpnam(s)
char *s;
```

Description

tmpnam generates a file name that can safely be used for a temporary file. If **s** is **NULL, tmpnam** leaves its result in an internal static area and returns a pointer to that area. The next call to **tmpnam** may overwrite the contents of the area. If **s** is not **NULL,** then **s** is assumed to be the address of an array of at least **L_tmpnam** bytes; **tmpnam** places its result in that array and returns **s** as its value.

tmpnam generates a different file name each time it is called.

Files created using **tmpnam** and either **fopen** or **creat** are only temporary in the sense that they reside in a directory intended for temporary use, and their names are unique. The user should use **unlink** to remove the file when its use is ended.

See Also

creat, fopen, mktemp, unlink

Notes

If called more than 17,576 times in a single process, **tmpnam** will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using **tmpnam** or **mktemp,** and the file names are chosen so as to render duplication by other means unlikely.

TRIG: sin, cos, tan, asin, acos, atan, atan2 - Trigonometric functions.

Syntax

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(y, x)
double x, y;
```

Description

sin, cos and **tan** return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

asin returns the arc sin in the range -pi/2 to pi/2.

acos returns the arc cosine in the range 0 to pi.

atan returns the arc tangent of **x** in the range -pi/2 to pi/2.

atan2 returns the arc tangent of **y/x** in the range -pi to pi.

Diagnostics

Arguments of magnitude greater than 1 cause **asin** and **acos** to return value 0.

Notes

These routines can be linked with the linker option **-lm**.

ttyname, isatty - Find the name of a terminal.

Syntax

```
char *ttyname(fildes)
int fildes;

int isatty(fildes)
int fildes;
```

Description

ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor **fildes.** **ttyname** returns **NULL** if **fildes** does not reference a terminal device in directory **/dev.**

isatty returns 1 if **fildes** is associated with a terminal device, 0 otherwise.

Files

/dev/*

Notes

The return value from **ttyname** points to static data that is overwritten by each call.

ulimit - Get and set user limits.

Syntax

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

Description

This function provides for control over process limits. The **cmd** values available are

1 Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. The limit applies only for writing files; files of any size can be read.

2 Set the process's file size limit to the value of **newlimit.** Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. **ulimit** fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit. [**EPERM**]

3 Get the maximum possible break value. See **sbrk.**

Return Value

If successful, a nonnegative value is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

chsize, sbrk, write

Notes

The file limit is only enforced on writes to regular files. Tapes, disks, and other devices of any size can be written.

umask - Set and get file creation mask.

Syntax

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

Description

umask sets the calling process's file mode creation mask to **cmask** and returns the previous value of the mask. Only the low-order nine 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, mknod, open

mkdir, mknod, sh in "Commands" in the *XENIX 286 Reference Manual*

umount - Unmount a file system.

Syntax

```
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 on which the file system was mounted reverts to its ordinary interpretation.

umount may be invoked only by the super-user.

umount fails if one or more of the following are true:

- The effective user ID of the calling process is not super-user. [**EPERM**]
- **spec** is an illegal address. [**EFAULT**]
- The path name is null or the named device special file does not exist. [**ENOENT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- The named node is not a block device. [**ENOTBLK**]
- The named device is not on-line or not installed in the system. [**ENXIO**]
- The named device is not mounted. [**EINVAL**]
- A file on the named device is open or otherwise busy. [**EBUSY**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

mount

mount in "Commands" in the *XENIX 286 Reference Manual*

uname - Name of current XENIX system.

Syntax

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

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

Description

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

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

```
struct utsname {
        char   sysname[9];
        char   nodename[9];
        char   release[9];
        char   version[9];
        unsigned short sysorigin;
        unsigned short sysoem;
        long   sysserial;
};
```

uname writes a null-terminated character string naming the current XENIX system in the character array field **sysname.** Similarly, **nodename** contains the name that the system is known by on a communications network. **release** and **version** further identify the operating system. **sysorigin** and **sysoem** identify the source of the XENIX version. **sysserial** is a software serial number that may be zero if unused.

uname fails if **name** is an illegal address. [**EFAULT**]

Return Value

If successful, a nonnegative value is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

uname in "Commands" in the *XENIX 286 Reference Manual*

Notes

Not all fields may be set on a particular system.

ungetc - Push character back into input stream.

Syntax

```
#include <stdio.h>

int ungetc(c, stream)
char c;
FILE *stream;
```

Description

ungetc pushes the character **c** back on an input stream. **c** is returned by the next **getc** call on that stream. **ungetc** returns **c.**

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push back **EOF** are rejected.

ungetc returns **EOF** if it cannot push a character back.

fseek erases all memory of pushed back characters.

See Also

fseek, getc, setbuf

unlink - Remove directory entry.

Syntax

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

Description

unlink removes the directory entry named by the path name pointed to by **path.**

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

- **path** is an illegal address. [**EFAULT**]
- The path name is null or the named entry does not exist. [**ENOENT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- A component of the path prefix denies search permission, or write permission is denied for the directory containing the entry to be removed, or the entry to be removed is a directory and the effective user ID of the calling process is not super-user. [**EACCES**]
- The directory containing the entry is in a read-only file system. [**EROFS**]
- The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. [**ETXTBSY**]
- The entry to be unlinked is the mount point for a mounted file system. [**EBUSY**]

When all links to a file have been removed, and when all processes that have the file open have closed it, then the space occupied by the file is freed.

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

close, link, open

rm in "Commands" in the *XENIX 286 Reference Manual*

ustat - Get file system statistics.

Syntax

```
#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 */
char     f_fpack[6];  /* Filsys pack name */
```

ustat fails if one or more of the following are true:

- **dev** is not the device number of a device containing a mounted file system. [**EINVAL**]
- **buf** is an illegal address. [**EFAULT**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

stat

file system in Appendix D, "File Formats"

utime - Set file access and modification times.

Syntax

```
#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 super-user may use **utime** this way.

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

```
struct    utimbuf    {
          time_t     actime;      /* access time */
          time_t     modtime;     /* modification time */
};
```

utime fails if one or more of the following are true:

- **times** is not **NULL** and the effective user ID is not super-user and does not match the owner of the file. [**EPERM**]
- **path** is an illegal address or **times** is not **NULL** and is an illegal address. [**EFAULT**]
- The path name is null or the named file does not exist. [**ENOENT**]
- A component of the path prefix is not a directory. [**ENOTDIR**]
- A component of the path prefix denies search permission or **times** is **NULL,** the effective user ID is not super-user and not the owner of the file, and write access is denied. [**EACCES**]
- The file is in a read-only file system. [**EROFS**]

Return Value

If successful, 0 is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

stat

wait - Wait for a child process to stop or terminate.

Syntax

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

Description

wait suspends the calling process until it receives a signal that is to be caught (see **signal**), or until any one of the calling process's child processes stops in a trace mode (see **ptrace**) or terminates. If a child process stopped or terminated prior to the call on **wait**, return is immediate.

If **stat_loc** is not **NULL**, 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 high-order 8 bits of status will be zero and the low-order 8 bits will be set equal to 0177 octal. If the child process terminated due to an **exit** call, the low-order 8 bits of status will be zero and the high-order 8 bits will contain the low-order 8 bits of the argument that the child process passed to **exit**; see **exit**. If the child process terminated due to a signal, the high-order 8 bits of status will be zero and the low-order 8 bits will contain the number of the signal that caused the termination. In addition, if the low-order seventh bit (i.e., bit 0200) is set, a "core image" of the child process will have been produced; see **signal**.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the **proc1** initialization process inherits the child processes.

wait fails and returns immediately if one or more of the following are true:

- **stat_loc** is not **NULL** and is an illegal address. **[EFAULT]**
- The calling process has no existing unwaited-for child processes. **[ECHILD]**

Return Value

If successful, **wait** returns due to a stopped or terminated child process and the process ID of the child is returned. Otherwise, if **wait** returns because the calling process has received a signal, -1 is returned and **errno** is assigned **EINTR.** Otherwise, -1 is returned and **errno** is assigned another error code.

See Also

exec, exit, fork, pause, signal

Warning

See **Warning** in **signal.**

waitsem, nbwaitsem - Await or check access to a resource governed by a semaphore.

Syntax

```
waitsem(sem_num);
int sem_num;

nbwaitsem(sem_num);
int sem_num;
```

Description

waitsem gives the calling process access to the resource governed by the semaphore **sem_num.** If the resource is in use by another process, **waitsem** puts the calling process to sleep until the resource becomes available; **nbwaitsem** returns the error **ENAVAIL. waitsem** and **nbwaitsem** are used in conjunction with **sigsem** to allow synchronization of processes wishing to access a resource. One or more processes may **waitsem** on the given semaphore and will be put to sleep until the process with current access to the resource issues **sigsem. sigsem** causes the process that is next in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first-in-first-out (FIFO) order.

waitsem and **nbwaitsem** fail if one or more of the following are true:

- **sem_num** does not refer to a semaphore file. **[ENOTNAM]**
- **sem_num** refers to a semaphore that has not been previously opened by calling **opensem** or **creatsem. [EBADF]**
- The process controlling the semaphore terminates without relinquishing control (with **signal). [ENAVAIL]**

Return Value

If successful, **waitsem** and **nbwaitsem** return a nonnegative value. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

creatsem, opensem, sigsem

write - Write to a file.

Syntax

```
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 **fildes.**

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. 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 is set to the end of the file prior to each write.

write fails and the file pointer is not changed if one or more of the following are true:

- **fildes** is not a valid file descriptor open for writing. **[EBADF]**
- **buf** is an illegal address. **[EFAULT]**
- An attempt is made to write to a pipe that is not open for reading by any process. **[EPIPE** and **SIGPIPE** signal]
- An attempt is made to write a file that exceeds the maximum file size or the process's file size limit (see **ulimit**). **[EFBIG]**

If a **write** requests that more bytes be written than there are room for (e.g., the **ulimit** or the physical end of a medium), only as many bytes as there are room for are written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes then returns 20. The next write of a nonzero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO), no partial writes are permitted. Thus, a write to a pipe or FIFO fails if a write of **nbyte** bytes would exceed a limit.

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) returns a count of 0. Otherwise (**O_NDELAY** clear), **write** to a full pipe (or FIFO) blocks until space becomes available.

Return Value

If successful, the number of bytes actually written is returned. Otherwise, -1 is returned and **errno** is assigned an error code.

See Also

creat, dup, lseek, open, pipe, ulimit

Notes

Writing a region of a file locked with **locking** causes **write** to block until the locked region is unlocked.

xlist, fxlist - Get name list entries from files.

Syntax

```
#include <a.out.h>

xlist(file name, xl)
char *file name;
struct xlist xl[];

#include <a.out.h>
#include <stdio.h>

fxlist(fp, xl)
FILE *fp;
struct xlist xl[];
```

Description

fxlist performs the same function as **xlist,** except that **fxlist** accepts a pointer to a previously opened file intead of a file name.

xlist examines the name list in the given executable output file and selectively extracts a list of values. The name list structure **xl** consists of an array of **xlist** structures containing names, types and values. The list is terminated by either a pointer to a null name or by a **NULL** pointer. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted into the next two fields. If the name is not found, both entries are set to zero. See **a.out** in Appendix D, "File Formats," for a discussion of the **xlist** structure.

If the object file is in **a.out** format, and if the symbol name given to **xlist** is longer than eight characters, then only the first eight characters are used for comparison. In all other cases, the name given to **xlist** must be the same length as a name list entry in order to match.

If two or more symbols happen to match the name given to **xlist,** then the type and value used are those of the last symbol found.

Return Value

If successful, 0 is returned. A zero return does not indicate that any or all of the given symbols were found. Otherwise, -1 is returned and all type entries are zeroed. A -1 return can be because the file cannot be read, is not an object file, or contains an invalid name list.

See Also

a.out in Appendix D, "File Formats"

intel®

APPENDIX D
FILE FORMATS

This section outlines the formats of various files. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys.**

a.out - Format of assembler and link editor output.

Description

a.out is the output file of the assembler **as** and the link editor **ld.** Both programs will make **a.out** executable if there were no errors in assembling or linking, and no unresolved external references.

See Also

as, ld, nm, strip in "Programming Commands" in the *XENIX 286 Programmer's Guide*

acct - Format of per-process accounting file.

Syntax

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

Description

Files produced as a result of calling **acct** have records in the form defined by **<sys/acct.h>**.

In **ac_flag**, the AFORK flag is turned on by each **fork** and turned off by an **exec.** The **ac_comm** field is inherited from the parent process and is reset by any **exec.** Each time the system charges the process with a clock tick, it also adds the current process size to **ac_mem** computed as follows:

(data size) + (text size) / (number of in-core processes using text)

The value of **ac_mem/ac_stime** can be viewed as an approximation to the mean process size, as modified by text-sharing.

See Also

acct, exec, fork in Appendix C, "System Functions"

acct, acctcom in "Commands" in the *XENIX 286 Reference Manual*

Notes

The **ac_mem** value for a short-lived command gives little information about the actual size of the command because **ac_mem** may be incremented while a different command (e.g., the shell) is being executed by the process.

ar - Archive file format.

Description

The archive command **ar** is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor **ld.**

A file produced by **ar** has a "magic number" at the start, followed by the constituent files, each preceded by a file header. The magic number is 0177545 octal. The header of each file is 26 bytes long and is declared in **/usr/include/ar.h.**

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless, the size given reflects the actual size of the file exclusive of padding.

Note that there is no provision for empty areas in an archive file.

See Also

ar, ld in "Programming Commands" in the *XENIX 286 Programmer's Guide*

checklist - List of file systems processed by **fsck.**

Description

Residing in the directory **/etc, checklist** contains a list of up to 15 special file names. Each special file name is contained on a separate line and corresponds to a file system. Each named file system is automatically processed by the **fsck** command when that program is invoked, but only if no file systems are given explicitly on the **fsck** command line.

See Also

fsck in "Commands" in the *XENIX 286 Reference Manual*

core - Format of core image file.

Description

XENIX writes out a core image of a terminated process when any of various errors occur. See **signal** for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated QUIT signals. The core image is called **core** and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter **usize**, defined in **/usr/include/sys/param.h.** The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the user structure of the system, defined in **/usr/include/sys/user.h.** The locations of registers are outlined in **/usr/include/sys/reg.h.**

See Also

setuid, signal in Appendix C, "System Functions"

adb in "Commands" in the *XENIX 286 Reference Manual*

cpio - Format of **cpio** archive.

Description

The header structure, when the **c** option is not used, is

```
struct {
        short  h_magic,
               h_dev,
               h_ino,
               h_mode,
               h_uid,
               h_gid,
               h_nlink,
               h_rdev,
               h_mtime[2],
               h_namesize,
               h_filesize[2];
        char   h_name[h_namesize rounded to word];
} Hdr;
```

When the **c** option is used, the header information is described by the statement

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%6o%s",
        &Hdr.h_magic,&Hdr.h_dev,&Hdr.h_ino,&Hdr.h_mode,
        &Hdr.h_uid,&Hdr.h_gid,&Hdr.h_nlink,&Hdr.h_rdev,
        &Longtime,&Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and **Longfile** are equivalent to **Hdr.h_mtime** and **Hdr.h_filesize** respectively. The contents of each file are recorded in an element of the array of varying length structures, **archive,** together with other items describing the file. Every instance of **h_magic** contains the constant 070707 (octal). The items **h_dev** through **h_mtime** have meanings explained in **stat.** The length of the null-terminated path name **h_name,** including the null byte, is given by **h_namesize.**

The last record of the archive always contains the name **TRAILER!!!.** Special files, directories, and the trailer are recorded with **h_filesize** equal to zero.

See Also

stat in Appendix C, "System Functions"

cpio, find in "Commands" in the *XENIX 286 Reference Manual*

dir - Format of a directory.

Syntax

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

Description

A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its inode entry (see **file system** in this appendix). The structure of a directory is given in the include file **/usr/include/sys/dir.h.**

By convention, the first two entries in each directory are "dot" (.) and "dotdot" (..). The first is an entry for the directory itself. The second is an entry for the parent directory. The meaning of dotdot is modified for the root directory of the master file system; there is no parent, so dotdot has the same meaning as dot.

See Also

file system

dump - Incremental dump tape format.

Description

The **dump** and **restor** commands are used to write and read incremental dump magnetic tapes.

A dump tape consists of a header record, some bit mask records, a group of records describing file system directories, a group of records describing file system files, and some records describing a second bit mask.

The header record and the first record of each description have the format described by the structure included by

```
#include <dumprestor.h>
```

Fields in the **dumprestor** structure are described below.

NTREC is the number of blocks in a physical tape record. **MLEN** is the number of bits in a bit map word. **MSIZ** is the number of bit map words.

The **TS_** entries are used in the **c_type** field to indicate what sort of header the header record is. The types and their meanings are as follows:

TS_TYPE Tape volume label.

TS_INODE A file or directory follows. The **c_dinode** field is a copy of the disk inode and contains bits telling what sort of file this is.

TS_BITS A bit mask follows. This bit mask has a one bit for each inode that was dumped.

TS_ADDR A subblock to a file (**TS_INODE**). See the description of **c_count** below.

TS_END End of tape record.

TS_CLRI A bit mask follows. This bit mask contains a 1 bit for all inodes that were empty on the file system when dumped.

MAGIC All header blocks have this number in **c_magic**.

CHECKSUM Header blocks checksum to this value.

The fields of the header structure are as follows:

c_type The type of the header.

c_date The date the dump was taken.

c_ddate The date the file system was dumped from.

c_volume The current volume number of the dump.

c_tapea The current block number of this record. This is counting blocks, not bytes.

c_inumber The number of the inode being dumped if this is of type **TS_INODE.**

c_magic Contains the value **MAGIC** above, truncated as needed.

c_checksum Contains whatever value is needed to make the block sum to **CHECKSUM.**

c_dinode A copy of the inode as it appears on the file system.

c_count Count of characters that describe the file. A character is zero if the block associated with that character was not present on the file system; otherwise, the character is nonzero. If the block was not present on the file system, no block was dumped; the block is replaced as a hole in the file. If there is insufficient space in this block to describe all of the blocks in a file, **TS_ADDR** blocks will be scattered through the file, each one picking up where the last left off.

c_addr The array of characters that is used as described above.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a **TS_END** block and then the tapemark.

The structure **idates** describes an entry of the file where dump history is kept.

See Also

file system

dump, restor in "Commands" in the *XENIX 286 Reference Manual*

file system - Format of system volume.

Syntax

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

Description

The file system is divided up into a number of 1024-byte blocks. It imposes a particular structure and usage upon these disk blocks. File system structures are located in the file **/usr/included/sys/filsys.h.**

XENIX 286 Release 3 includes a number of performance improvements that not only make its file system significantly different from Release 1 but also make the two file systems incompatible. The Release 1 file system maintained a structure in a 1K block at the beginning of the file system. The structure, or super-block, described the entire system, and data blocks were allocated from an array of 100 free data blocks maintained in the super-block. A similar free list existed for the inodes.

Release 3 replaces the free list allocation scheme with a bit map extent allocation scheme and imposes a cylinder grouping strategy on the file system. Cylinder grouping is a method of partitioning the file system into one or more groups that contain inodes and data blocks. Associated with each cylinder group is a cylinder group structure that maintains information about the cylinder group. The cylinder group structure, which resides in a 1K block at the beginning of the cylinder group, is referred to as a cylinder group block. The Release 3 file system layout is as follows:

```
S C i..i dd...dd C i..i dd...dd C i..i dd...dd
```

where **S** denotes the super-block that maintains the location of each cylinder group block denoted by **C.** Each cylinder group block, in turn, maintains the location of the inode blocks, denoted by **i,** and the data blocks, denoted by **d,** within a particular cylinder group.

Defined in the **filsys.h** include file is **struct filsys,** which is the structure for the super-block. Within this structure is an array of structures, **struct cginfo,** which contains information about the cylinder groups. Each cylinder group block contains a structure, **struct cylinder,** describing the inodes and data blocks in the cylinder group.

Within **struct cginfo,** the variable **fs_cgblk** is the disk block address of the cylinder group block. Within **struct cylinder,** the array **cg_bits** contains the bit map of the data blocks in the cylinder group. (Note that the bit map describes only data blocks, not inode blocks. Inodes are allocated from a link list of free inodes.) Each bit in the bit map array represents one data block. If the bit is set to one, the data block is free. If the bit is set to zero, the data block is allocated.

The file system's bit map extent allocation policy preallocates a number of blocks to a file. As a file grows, a number of blocks are preallocated out of the bit map and reserved for the file. When the file is closed, any unused blocks are returned to the bit map. The extent bit map allocation policy tries to force contiguous files. The size of the file system's extent (the number of blocks to preallocate) is a configuration parameter and may vary from 1 to 32 blocks. The default extent size is 8 blocks.

In addition, the **filsys.h** file contains **BMAPSIZE**, which defines the maximum size of a cylinder group bit map, and **MAXCGS**, which describes the maximum number of cylinder groups per file system. Both are crucial to file system performance.

BMAPSIZE*8 is the total number of data blocks that can exist in any one cylinder group. Since inode blocks are not described by the bit map, the file system can support cylinder group sizes up to 8 MB. With the bit map size of 994 bytes, the cylinder group can support 994 * 8, or 7952, data blocks. The default inode blocks-to-data-blocks ratio is 2 data blocks to 1 inode, so a system supporting 7952 data blocks could support 7952 / 2, or 3976, inodes. Since each 1K block can contain 16 inodes, the cylinder group would need 3976 / 16, or 248.5 blocks for inodes. (Inodes are easier to handle on block boundaries, so 248.5 would be rounded up to 249 blocks.) Counting the cylinder group block, the entire cylinder group would require a total of 7952 + 249 + 1, or 8202, blocks.

MAXCGS describes the number of cylinder groups in the file system. The file system structure will support 80 cylinder groups, each containing 8 MB, so the maximum file system partition size is 640 MB. However, since inode numbers are only 16 bits, the most inodes the file system can contain is actually 65,535. With a 2048-byte-to-1-inode ratio, file system partitions larger than 130 MB are restricted in the number of inodes they can contain. It is suggested that disks larger than 40 MB be separated into two file systems, one for the root and one for the user. For user file systems larger than 130 MB, multiple user file system partitions are recommended.

In creating a file system, keep in mind a number of variables: the size of a cylinder group, the blocks-to-inode ratio, and the extent size of the file system. As the last cylinder group in the file system will seldom be a full cylinder group, take care when choosing the size of the cylinder groups. (The last cylinder group size must be as close to the others as possible. Otherwise, the last group may have a large number of inode blocks and few data blocks.) The suggested ratio is 2 data blocks to 1 inode. For file systems larger than 130 MB, increase the ratio to 3 or 4 blocks per inode.

The extent size is application-dependent. For a normal XENIX system, an extent size of 8 or less is adequate. However, applications with large files require a larger extent size, which forces the files to be contiguous.

Files

/usr/include/sys/filsys.h
/usr/include/sys/types.h
/usr/include/sys/param.h

See Also

inode

fsck, mkfs in "Commands" in the *XENIX 286 Reference Manual*

inode - Format of an inode.

Syntax

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

Description

An inode for a plain file or directory in a file system has the structure defined by **<sys/ino.h>**. For the meaning of the defined types **off_t** and **time_t**, see **types.**

Files

/usr/include/sys/ino.h

See Also

file system, types

stat in Appendix C, "System Functions"

master - Format of master device information table.

Description

This file is used by the **config** program (in "Commands" in the *XENIX 286 Reference Manual*) to obtain device information that enables it to generate the configuration files. Note that **config** is not a normal user command.

The file consists of three parts, each separated by a line with a dollar sign (**$**) in column 1. Part 1 contains device information; part 2 contains names of devices that have aliases; and part 3 contains tunable parameter information. Any line with an asterisk (*****) in column 1 is treated as a comment.

There are 14 fields in the Part 1 lines that describe devices, but some are unused and others are redundant. Fields in Part 1 lines are free format, separated by blanks or tabs. To fill in the line for a device, you must know the answers to these questions:

1. What is the name of your device? What is the prefix used for your driver routines, if different from the device name?

2. Does your device support a block interface? If it does, then what major number is used for the block interface?

3. Does your device support a character interface? If it does, then what major number is used for the character interface?

4. Does your device use interrupts? If so, what interrupt level(s) does it use?

5. What standard driver routines are not present in your driver and should be replaced by **nulldev** in the **cdevsw** or **bdevsw** tables?

6. What is the maximum number of boards handled by your driver that can be present in a system?

Field 1, **name**, is the name of the device, beginning in column 1 and from 1 to 8 characters long. Intel devices are customarily identified as **ixxx**, e.g., **i534** for the iSBC 534 board. If limited to 4 characters, field 1 can be identical to Field 5, **hndler**.

Field 2, **vsiz**, is the size of the device's interrupt vector in words. If the device uses interrupts, this field is the number of interrupt levels used, normally 1. If the device does not use interrupts, this field is 0. A "virtual" device such as a RAM disk is an example of a device that does not use interrupts.

Field 3, **msk**, is an octal bit mask indicating which standard driver routines are present:

0100	**init** routine present
0020	**open** routine present
0010	**close** routine present
0004	**read** routine present
0002	**write** routine present
0001	**ioctl** routine present

Note that the **strategy** routine of block drivers and the **intr** routine of all drivers that handle interrupts are not listed. The **strategy** routine is mandatory for all block drivers. The **intr** routine must be provided for all drivers that use interrupt levels. You can form the bit mask to use for your device by taking the mask values for all the routines present in your driver and ORing them. For example, for a line printer driver that provided all routines except **read** and **ioctl**, the mask value would be 0132. The kernel routine **nulldev** replaces missing routines in the **cdevsw** or **bdevsw** tables. **nulldev** does nothing when called, simply returning to its caller.

Field 4, **typ**, is an octal bit mask indicating device type and some miscellaneous information:

0200	Only one specification of the device is allowed. I.e., only one line in **master**'s device table can refer to the device.
0040	The device does *not* use interrupts.
0020	The device is *required* in the configuration.
0010	The device provides a block interface.
0004	The device provides a character interface.

You can form the bit mask to use for your device by taking all the mask values that apply to your driver and ORing them. Terminals and simple character devices have type 004. Disks, which normally support a "raw" character interface as well as a block interface, have type 014. A RAM disk might not need a character interface and could have type 010.

Field 5, **hndlr**, is the prefix that is prepended to the standard routine names to produce the routine names used in your driver. For example, if **lp** is the value of the **hndlr** field for your device, your routine names *must* be **lpinit**, **lpopen**, etc. The prefix can be from 1 to 4 characters in length. The prefix must begin with a letter, and the characters in the prefix must be limited to those allowed in C identifiers. The prefix is used to generate the routine names in the switch tables **dinitsw**, **cdevsw**, **bdevsw**, and **vecintsw**. It can reduce confusion if your prefix is the same as the device name in Field 1.

Field 6, **na**, is not used and should be zero.

Field 7, **bmaj**, is the major number used for the device's block interface. If the device does not have a block interface, the field is not used but typically zero. A major number of zero is allowed.

Field 8, **cmaj**, is the major number used for the device's character interface. If the device does not have a character interface, the field is not used but typically zero. A major number of zero is allowed.

For devices that have both block and character interfaces, the same major number is typically used for both interfaces. This is not required; the block and character major numbers for a device can be different.

Field 9, **#**, is the maximum number of boards supported by the device driver that may be present in the system. This number is only used for checking against another such number in the **xenixconf** file. This number has nothing to do with the number of devices or range of minor device numbers that your driver supports.

Field 10, **na**, is not used and should be **-1**.

Fields 11, 12, 13, and 14 contain up to four octal interrupt levels used by the driver. Unused interrupt levels should be zero; zero is not allowed as a valid interrupt level. Levels should be in the range 1-0377 (1-255). The levels specified must be compatible with those used by other devices and must be the same as those actually used by the hardware! The letter **a** must immediately follow field 14.

There is no ordering of entries in the device table. You can insert the line for your device at any position in the table. The name, prefix, major numbers, and interrupt levels for your device must be distinct from those used by other devices.

Part 2 contains lines with 2 fields separated by blanks or tabs. Field 1 is the alias name of the device, from 1 to 8 characters. Field 2 is the reference name of the device, from 1 to 8 characters, as named in Field 1 of the device's line in Part 1.

Part 3 contains lines with 2 or 3 fields separated by blanks or tabs. Field 1 is the parameter name as it appears in the description file, from 1 to 20 characters. Field 2 is the parameter name as it appears in the **c.c** file, from 1 to 20 characters. Field 3 is the default parameter value of from 1 to 20 characters; parameter specification is required if this field is omitted.

See Also

config in "Commands" in the *XENIX 286 Reference Manual*

XENIX 286 Installation and Configuration Guide

"Adding Drivers to the Configuration" in the *XENIX 286 Device Driver Guide*

Notes

The **config** program is only for systems that have configurable kernels. It resides in the **/usr/sys/conf** directory.

mnttab - Format of mounted file system table.

Syntax

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

Description

The file **mnttab** resides in the **/etc** directory. It contains a table of devices mounted by use of the **mount** command.

Each table entry contains the path name of the directory on which the device is mounted, the name of the device special file, the read/write permissions of the special file, and the date when the device was mounted.

The maximum number of entries in **mnttab** is based on the system parameter **NMOUNT** located in **/usr/sys/conf/c.c**, which defines the number of allowable mounted special files.

See Also

mount in "Commands" in the *XENIX 286 Reference Manual*

sccsfile - Format of an SCCS file.

Description

An SCCS file is an ASCII file. It consists of six logical parts: the checksum, the delta table (contains information about each delta), user names (contains login names and/or numerical group IDs of users who may add deltas), flags (contains definitions of internal keywords), comments (contains arbitrary descriptive information about the file), and the body (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines that begin with the ASCII SOH (start of heading) character (octal 001). This character, hereafter referred to as the control character, will be represented graphically as @. Any line described below that is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form DDDDD represent a five-digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is

```
@hDDDDD
```

The value of the checksum is the sum of all characters except those of the first line. The @hR provides a magic number of 064001 octal.

Delta Table

The delta table consists of a variable number of entries of the form

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
  .
  .
  .
@c <comments> ...
  .
  .
  .
@e
```

The first line (**@s**) contains the number of lines inserted/deleted/unchanged respectively. The second line (**@d**) contains the type of the delta (currently, normal: D and removed: R), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored respectively. These lines are optional.

The **@m** lines (optional) each contain one MR number associated with the delta; the **@c** lines contain comments associated with the delta.

The **@e** line ends the delta table entry.

User Names

User names is a list of login names and/or numerical group IDs of users who may add deltas to the file, separated by newlines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines **@u** and **@U**. An empty list allows anyone to make a delta.

Flags

Flags are keywords used internally. (See **admin** in "Programming Commands" in the *XENIX 286 Programmer's Guide* for more information on their use.) Each flag line takes the form

```
@f <flag> <optional text>
```

The following flags are defined:

```
@f t      <type of program>
@f v      <program name>
@f i
@f b
@f m      <module name>
@f f      <floor>
@f c      <ceiling>
@f d      <default-sid>
@f n
@f j
@f l      <lock-releases>
@f q      <user defined>
```

The **t** flag defines the replacement for the **%Y%** identification keyword.

The **v** flag controls prompting for MR numbers in addition to comments; if the optional text is present, it defines an MR number validity checking program.

The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a fatal error. (The file will not be gotten, or the delta will not be made.)

When the **b** flag is present, the **-b** keyletter may be used with the **get** command to cause a branch in the delta tree.

The **m** flag defines the first choice for the replacement text of the **%M%** identification keyword.

The **f** flag defines the "floor" release, the release below which no deltas may be added.

The **c** flag defines the "ceiling" release, the release above which no deltas may be added.

The **d** flag defines the default SID to be used when none is specified on a **get** command.

The **n** flag causes **delta** to insert a "null" delta (a delta that applies no changes) in those releases that are skipped when a delta is made in a new release. (For example, when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped.) The absence of the **n** flag causes skipped releases to be completely empty.

The **j** flag causes **get** to allow concurrent edits of the same base SID.

The **l** flag defines a list of releases locked against editing (**get** with the **-e** keyletter).

The **q** flag defines the replacement for the **%Q%** identification keyword.

Comments

Comments consist of arbitrary text surrounded by the bracketing lines **@t** and **@T**. The comments section typically contains a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines do not begin with the control character; control lines do. There are three kinds of control lines: insert, delete, and end:

```
@I DDDDD
@D DDDDD
@E DDDDD
```

The digit string (DDDDD) is the serial number corresponding to the delta for the control line.

See Also

admin in "Programming Commands" in the *XENIX 286 Programmer's Guide*

types - Primitive system data types.

Syntax

#include<sys/types.h>

Description

The data types defined in the include file **<sys/types.h>** are used in XENIX system code; some data of these types are accessible to user code.

The form **daddr_t** is used for disk addresses except in an inode on disk. (See **file system.**) Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The **label_t** variables are used to save the processor state while another process is running.

See Also

file system

utmp, wtmp - Formats of utmp and wtmp entries.

Description

The files **utmp** and **wtmp** hold user and accounting information for use by commands such as **who, acctcon1,** and **login.** They have the following structure, as defined by **/usr/include/utmp.h:**

```
struct utmp
{
        char    ut-line[8];     /* tty name */
        char    ut_name[8];     /* login name */
        long    ut_time;        /* time on */

};
```

Files

/etc/utmp
/usr/adm/wtmp
/usr/include/utmp.h

See Also

acctcon, login, who, write in "Commands" in the *XENIX 286 Reference Manual*

intel®

APPENDIX E
RELATED PUBLICATIONS

Copies of the following publications can be ordered from

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Overview of the XENIX 286 Operating System, Order Number 174385 -- XENIX history, XENIX uses, basic XENIX concepts, and an overview of other XENIX manuals.

XENIX 286 Installation and Configuration Guide, Order Number 174386 -- how to install XENIX on your hardware and tailor the XENIX configuration to your needs.

XENIX 286 User's Guide, Order Number 174387 -- a tutorial on the most-used parts of XENIX, including terminal conventions, the file system, the screen editor, and the shell.

XENIX 286 Visual Shell User's Guide, Order Number 174388 -- a XENIX command interface ("shell") that replaces the standard command syntax with a menu-driven command interpreter.

XENIX 286 System Administrator's Guide, Order Number 174389 -- how to perform system administrator chores such as adding and removing users, backing up file systems, and troubleshooting system problems.

XENIX 286 Reference Manual, Order Number 174390 -- all commands in the XENIX 286 Basic System.

XENIX 286 Programmer's Guide, Order Number 174391 -- XENIX 286 Extended System commands used for developing and maintaining programs.

XENIX 286 C Library Guide, Order Number 174542 -- (this manual) standard subroutines used in programming with XENIX 286, including all system calls.

XENIX 286 Device Driver Guide, Order Number 174393 -- how to write device drivers for XENIX 286 and add them to your system.

XENIX 286 Text Formatting Guide, Order Number 174541 -- XENIX 286 Extended System commands used for text formatting.

XENIX 286 Communications Guide, Order Number 174461 -- installing, using, and administering XENIX networking software.

C is described in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. One copy is supplied with Intel's XENIX product. Additional copies can be ordered from the publisher, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

intel®

INDEX

Most bolded entries in this index are function or macro names. Other types of bolded entries are followed by a descriptive phrase, such as "include file," "variable," or "file format."

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does this publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply ☐

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Pkwy.
Hillsboro, Oregon 97123

O.M.S. Technical Publications

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.

SOFTWARE

0426/6K/0585/OSPS/AD