1

530128

UNIX System Overview

1.1 Introduction

All operating systems provide services for programs they run. Typical services include executing a new program, opening a file, reading a file, allocating a region of memory, getting the current time of day, and so on. The focus of this text is to describe the services provided by various versions of the UNIX operating system.

Describing the UNIX System in a strictly linear fashion, without any forward references to terms that haven't been described yet, is nearly impossible (and would probably be boring). This chapter provides a whirlwind tour of the UNIX System from a programmer's perspective. We'll give some brief descriptions and examples of terms and concepts that appear throughout the text. We describe these features in much more detail in later chapters. This chapter also provides an introduction to and overview of the services provided by the UNIX System for programmers new to this environment.

1.2 UNIX Architecture

In a strict sense, an operating system can be defined as the software that controls the hardware resources of the computer and provides an environment under which programs can run. Generally, we call this software the *kernel*, since it is relatively small and resides at the core of the environment. Figure 1.1 shows a diagram of the UNIX System architecture.

The interface to the kernel is a layer of software called the *system calls* (the shaded portion in Figure 1.1). Libraries of common functions are built on top of the system call

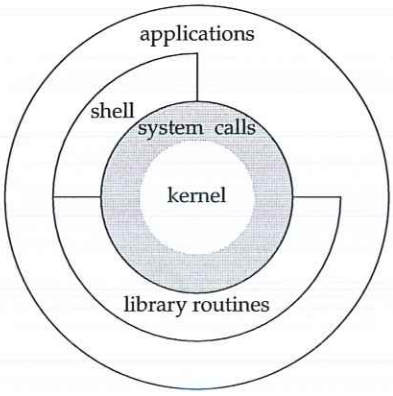

Figure 1.1 Architecture of the UNIX operating system

interface, but applications are free to use both. (We talk more about system calls and library functions in Section 1.11.) The shell is a special application that provides an interface for running other applications.

In a broad sense, an operating system consists of the kernel and all the other software that makes a computer useful and gives the computer its personality. This other software includes system utilities, applications, shells, libraries of common functions, and so on.

For example, Linux is the kernel used by the GNU operating system. Some people refer to this combination as the GNU/Linux operating system, but it is more commonly referred to as simply Linux. Although this usage may not be correct in a strict sense, it is understandable, given the dual meaning of the phrase *operating system*. (It also has the advantage of being more succinct.)

1.3 Logging In

Login Name

When we log in to a UNIX system, we enter our login name, followed by our password. The system then looks up our login name in its password file, usually the file `/etc/passwd`. If we look at our entry in the password file, we see that it's composed of seven colon-separated fields: the login name, encrypted password, numeric user ID (205), numeric group ID (105), a comment field, home directory (`/home/sar`), and shell program (`/bin/ksh`).

```
sar:x:205:105:Stephen Rago:/home/sar:/bin/ksh
```

All contemporary systems have moved the encrypted password to a different file. In Chapter 6, we'll look at these files and some functions to access them.

Shells

Once we log in, some system information messages are typically displayed, and then we can type commands to the shell program. (Some systems start a window management program when you log in, but you generally end up with a shell running in one of the windows.) A *shell* is a command-line interpreter that reads user input and executes commands. The user input to a shell is normally from the terminal (an interactive shell) or sometimes from a file (called a *shell script*). The common shells in use are summarized in Figure 1.2.

Name	Path	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Bourne shell	`/bin/sh`	•	•	copy of `bash`	•
Bourne-again shell	`/bin/bash`	optional	•	•	•
C shell	`/bin/csh`	link to `tcsh`	optional	link to `tcsh`	•
Korn shell	`/bin/ksh`	optional	optional	•	•
TENEX C shell	`/bin/tcsh`	•	optional	•	•

Figure 1.2 Common shells used on UNIX systems

The system knows which shell to execute for us based on the final field in our entry in the password file.

The Bourne shell, developed by Steve Bourne at Bell Labs, has been in use since Version 7 and is provided with almost every UNIX system in existence. The control-flow constructs of the Bourne shell are reminiscent of Algol 68.

The C shell, developed by Bill Joy at Berkeley, is provided with all the BSD releases. Additionally, the C shell was provided by AT&T with System V/386 Release 3.2 and was also included in System V Release 4 (SVR4). (We'll have more to say about these different versions of the UNIX System in the next chapter.) The C shell was built on the 6th Edition shell, not the Bourne shell. Its control flow looks more like the C language, and it supports additional features that weren't provided by the Bourne shell: job control, a history mechanism, and command-line editing.

The Korn shell is considered a successor to the Bourne shell and was first provided with SVR4. The Korn shell, developed by David Korn at Bell Labs, runs on most UNIX systems, but before SVR4 was usually an extra-cost add-on, so it is not as widespread as the other two shells. It is upward compatible with the Bourne shell and includes those features that made the C shell popular: job control, command-line editing, and so on.

The Bourne-again shell is the GNU shell provided with all Linux systems. It was designed to be POSIX conformant, while still remaining compatible with the Bourne shell. It supports features from both the C shell and the Korn shell.

The TENEX C shell is an enhanced version of the C shell. It borrows several features, such as command completion, from the TENEX operating system (developed in 1972 at Bolt Beranek and Newman). The TENEX C shell adds many features to the C shell and is often used as a replacement for the C shell.

The shell was standardized in the POSIX 1003.2 standard. The specification was based on features from the Korn shell and Bourne shell.

The default shell used by different Linux distributions varies. Some distributions use the Bourne-again shell. Others use the BSD replacement for the Bourne shell, called `dash` (Debian Almquist shell, originally written by Kenneth Almquist and later ported to Linux). The default user shell in FreeBSD is derived from the Almquist shell. The default shell in Mac OS X is the Bourne-again shell. Solaris, having its heritage in both BSD and System V, provides all the shells shown in Figure 1.2. Free ports of the shells are available on the Internet.

Throughout the text, we will use parenthetical notes such as this to describe historical notes and to compare different implementations of the UNIX System. Often the reason for a particular implementation technique becomes clear when the historical reasons are described.

Throughout this text, we'll show interactive shell examples to execute a program that we've developed. These examples use features common to the Bourne shell, the Korn shell, and the Bourne-again shell.

1.4 Files and Directories

File System

The UNIX file system is a hierarchical arrangement of directories and files. Everything starts in the directory called *root*, whose name is the single character `/`.

A *directory* is a file that contains directory entries. Logically, we can think of each directory entry as containing a filename along with a structure of information describing the attributes of the file. The attributes of a file are such things as the type of file (regular file, directory), the size of the file, the owner of the file, permissions for the file (whether other users may access this file), and when the file was last modified. The `stat` and `fstat` functions return a structure of information containing all the attributes of a file. In Chapter 4, we'll examine all the attributes of a file in great detail.

We make a distinction between the logical view of a directory entry and the way it is actually stored on disk. Most implementations of UNIX file systems don't store attributes in the directory entries themselves, because of the difficulty of keeping them in synch when a file has multiple hard links. This will become clear when we discuss hard links in Chapter 4.

Filename

The names in a directory are called *filenames*. The only two characters that cannot appear in a filename are the slash character (`/`) and the null character. The slash separates the filenames that form a pathname (described next) and the null character terminates a pathname. Nevertheless, it's good practice to restrict the characters in a filename to a subset of the normal printing characters. (If we use some of the shell's special characters in the filename, we have to use the shell's quoting mechanism to reference the filename, and this can get complicated.) Indeed, for portability, POSIX.1 recommends restricting filenames to consist of the following characters: letters (`a-z`, `A-Z`), numbers (`0-9`), period (`.`), dash (`-`), and underscore (`_`).

Two filenames are automatically created whenever a new directory is created: `.` (called *dot*) and `..` (called *dot-dot*). Dot refers to the current directory, and dot-dot refers to the parent directory. In the root directory, dot-dot is the same as dot.

The Research UNIX System and some older UNIX System V file systems restricted a filename to 14 characters. BSD versions extended this limit to 255 characters. Today, almost all commercial UNIX file systems support at least 255-character filenames.

Pathname

A sequence of one or more filenames, separated by slashes and optionally starting with a slash, forms a *pathname*. A pathname that begins with a slash is called an *absolute pathname*; otherwise, it's called a *relative pathname*. Relative pathnames refer to files relative to the current directory. The name for the root of the file system (/) is a special-case absolute pathname that has no filename component.

Example

Listing the names of all the files in a directory is not difficult. Figure 1.3 shows a bare-bones implementation of the `ls`(1) command.

```
#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Figure 1.3 List all the files in a directory

The notation `ls`(1) is the normal way to reference a particular entry in the UNIX system manuals. It refers to the entry for `ls` in Section 1. The sections are normally numbered 1 through 8, and all the entries within each section are arranged alphabetically. Throughout this text, we assume that you have a copy of the manuals for your UNIX system.

Historically, UNIX systems lumped all eight sections together into what was called the *UNIX Programmer's Manual*. As the page count increased, the trend changed to distributing the sections among separate manuals: one for users, one for programmers, and one for system administrators, for example.

Some UNIX systems further divide the manual pages within a given section, using an uppercase letter. For example, all the standard input/output (I/O) functions in AT&T [1990e] are indicated as being in Section 3S, as in `fopen`(3S). Other systems have replaced the numeric sections with alphabetic ones, such as C for commands.

Today, most manuals are distributed in electronic form. If your manuals are online, the way to see the manual pages for the `ls` command would be something like

```
man 1 ls
```

or

```
man -s1 ls
```

Figure 1.3 is a program that just prints the name of every file in a directory, and nothing else. If the source file is named `myls.c`, we compile it into the default `a.out` executable file by running

```
cc myls.c
```

Historically, `cc`(1) is the C compiler. On systems with the GNU C compilation system, the C compiler is `gcc`(1). Here, `cc` is usually linked to `gcc`.

Some sample output is

```
$ ./a.out /dev
.
..
cdrom
stderr
stdout
stdin
fd
sda4
sda3
sda2
sda1
sda
tty2
tty1
console
tty
zero
null
                    many more lines that aren't shown
mem
$ ./a.out /etc/ssl/private
can't open /etc/ssl/private: Permission denied
$ ./a.out /dev/tty
can't open /dev/tty: Not a directory
```

Throughout this text, we'll show commands that we run and the resulting output in this fashion: Characters that we type are shown in **`this font`**, whereas output from programs is shown `like this`. If we need to add comments to this output, we'll show

the comments in *italics*. The dollar sign that precedes our input is the prompt that is printed by the shell. We'll always show the shell prompt as a dollar sign.

Note that the directory listing is not in alphabetical order. The `ls` command sorts the names before printing them.

There are many details to consider in this 20-line program.

- First, we include a header of our own: `apue.h`. We include this header in almost every program in this text. This header includes some standard system headers and defines numerous constants and function prototypes that we use throughout the examples in the text. A listing of this header is in Appendix B.
- Next, we include a system header, `dirent.h`, to pick up the function prototypes for `opendir` and `readdir`, in addition to the definition of the `dirent` structure. On some systems, the definitions are split into multiple header files. For example, in the Ubuntu 12.04 Linux distribution, `/usr/include/dirent.h` declares the function prototypes and includes `bits/dirent.h`, which defines the `dirent` structure (and is actually stored in `/usr/include/x86_64-linux-gnu/bits`).
- The declaration of the `main` function uses the style supported by the ISO C standard. (We'll have more to say about the ISO C standard in the next chapter.)
- We take an argument from the command line, `argv[1]`, as the name of the directory to list. In Chapter 7, we'll look at how the `main` function is called and how the command-line arguments and environment variables are accessible to the program.
- Because the actual format of directory entries varies from one UNIX system to another, we use the functions `opendir`, `readdir`, and `closedir` to manipulate the directory.
- The `opendir` function returns a pointer to a `DIR` structure, and we pass this pointer to the `readdir` function. We don't care what's in the `DIR` structure. We then call `readdir` in a loop, to read each directory entry. The `readdir` function returns a pointer to a `dirent` structure or, when it's finished with the directory, a null pointer. All we examine in the `dirent` structure is the name of each directory entry (`d_name`). Using this name, we could then call the `stat` function (Section 4.2) to determine all the attributes of the file.
- We call two functions of our own to handle the errors: `err_sys` and `err_quit`. We can see from the preceding output that the `err_sys` function prints an informative message describing what type of error was encountered ("Permission denied" or "Not a directory"). These two error functions are shown and described in Appendix B. We also talk more about error handling in Section 1.7.
- When the program is done, it calls the function `exit` with an argument of 0. The function `exit` terminates a program. By convention, an argument of 0 means OK, and an argument between 1 and 255 means that an error occurred. In Section 8.5, we show how any program, such as a shell or a program that we write, can obtain the `exit` status of a program that it executes. □

Working Directory

Every process has a *working directory*, sometimes called the *current working directory*. This is the directory from which all relative pathnames are interpreted. A process can change its working directory with the `chdir` function.

For example, the relative pathname `doc/memo/joe` refers to the file or directory `joe`, in the directory `memo`, in the directory `doc`, which must be a directory within the working directory. From looking just at this pathname, we know that both `doc` and `memo` have to be directories, but we can't tell whether `joe` is a file or a directory. The pathname `/usr/lib/lint` is an absolute pathname that refers to the file or directory `lint` in the directory `lib`, in the directory `usr`, which is in the root directory.

Home Directory

When we log in, the working directory is set to our *home directory*. Our home directory is obtained from our entry in the password file (Section 1.3).

1.5 Input and Output

File Descriptors

File descriptors are normally small non-negative integers that the kernel uses to identify the files accessed by a process. Whenever it opens an existing file or creates a new file, the kernel returns a file descriptor that we use when we want to read or write the file.

Standard Input, Standard Output, and Standard Error

By convention, all shells open three descriptors whenever a new program is run: standard input, standard output, and standard error. If nothing special is done, as in the simple command

```
ls
```

then all three are connected to the terminal. Most shells provide a way to redirect any or all of these three descriptors to any file. For example,

```
ls > file.list
```

executes the `ls` command with its standard output redirected to the file named `file.list`.

Unbuffered I/O

Unbuffered I/O is provided by the functions `open`, `read`, `write`, `lseek`, and `close`. These functions all work with file descriptors.

Example

If we're willing to read from the standard input and write to the standard output, then the program in Figure 1.4 copies any regular file on a UNIX system.

```
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Figure 1.4 Copy standard input to standard output

The <unistd.h> header, included by apue.h, and the two constants STDIN_FILENO and STDOUT_FILENO are part of the POSIX standard (about which we'll have a lot more to say in the next chapter). This header contains function prototypes for many of the UNIX system services, such as the read and write functions that we call.

The constants STDIN_FILENO and STDOUT_FILENO are defined in <unistd.h> and specify the file descriptors for standard input and standard output. These values are 0 and 1, respectively, as required by POSIX.1, but we'll use the names for readability.

In Section 3.9, we'll examine the BUFFSIZE constant in detail, seeing how various values affect the efficiency of the program. Regardless of the value of this constant, however, this program still copies any regular file.

The read function returns the number of bytes that are read, and this value is used as the number of bytes to write. When the end of the input file is encountered, read returns 0 and the program stops. If a read error occurs, read returns –1. Most of the system functions return –1 when an error occurs.

If we compile the program into the standard name (a.out) and execute it as

```
./a.out > data
```

standard input is the terminal, standard output is redirected to the file data, and standard error is also the terminal. If this output file doesn't exist, the shell creates it by default. The program copies lines that we type to the standard output until we type the end-of-file character (usually Control-D).

If we run

```
./a.out < infile > outfile
```

then the file named infile will be copied to the file named outfile. □

In Chapter 3, we describe the unbuffered I/O functions in more detail.

Standard I/O

The standard I/O functions provide a buffered interface to the unbuffered I/O functions. Using standard I/O relieves us from having to choose optimal buffer sizes, such as the `BUFFSIZE` constant in Figure 1.4. The standard I/O functions also simplify dealing with lines of input (a common occurrence in UNIX applications). The `fgets` function, for example, reads an entire line. The `read` function, in contrast, reads a specified number of bytes. As we shall see in Section 5.4, the standard I/O library provides functions that let us control the style of buffering used by the library.

The most common standard I/O function is `printf`. In programs that call `printf`, we'll always include `<stdio.h>`—normally by including `apue.h`—as this header contains the function prototypes for all the standard I/O functions.

Example

The program in Figure 1.5, which we'll examine in more detail in Section 5.8, is like the previous program that called `read` and `write`. This program copies standard input to standard output and can copy any regular file.

```
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Figure 1.5 Copy standard input to standard output, using standard I/O

The function `getc` reads one character at a time, and this character is written by `putc`. After the last byte of input has been read, `getc` returns the constant `EOF` (defined in `<stdio.h>`). The standard I/O constants `stdin` and `stdout` are also defined in the `<stdio.h>` header and refer to the standard input and standard output. □

1.6 Programs and Processes

Program

A *program* is an executable file residing on disk in a directory. A program is read into memory and is executed by the kernel as a result of one of the seven `exec` functions. We'll cover these functions in Section 8.10.

Processes and Process ID

An executing instance of a program is called a *process*, a term used on almost every page of this text. Some operating systems use the term *task* to refer to a program that is being executed.

The UNIX System guarantees that every process has a unique numeric identifier called the *process ID*. The process ID is always a non-negative integer.

Example

The program in Figure 1.6 prints its process ID.

```
#include "apue.h"

int
main(void)
{
    printf("hello world from process ID %ld\n", (long)getpid());
    exit(0);
}
```

Figure 1.6 Print the process ID

If we compile this program into the file `a.out` and execute it, we have

```
$ ./a.out
hello world from process ID 851
$ ./a.out
hello world from process ID 854
```

When this program runs, it calls the function `getpid` to obtain its process ID. As we shall see later, `getpid` returns a `pid_t` data type. We don't know its size; all we know is that the standards guarantee that it will fit in a long integer. Because we have to tell `printf` the size of each argument to be printed, we have to cast the value to the largest data type that it might use (in this case, a long integer). Although most process IDs will fit in an `int`, using a `long` promotes portability. □

Process Control

There are three primary functions for process control: `fork`, `exec`, and `waitpid`. (The `exec` function has seven variants, but we often refer to them collectively as simply the `exec` function.)

Example

The process control features of the UNIX System are demonstrated using a simple program (Figure 1.7) that reads commands from standard input and executes the commands. This is a bare-bones implementation of a shell-like program.

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

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

Figure 1.7 Read commands from standard input and execute them

There are several features to consider in this 30-line program.

- We use the standard I/O function `fgets` to read one line at a time from the standard input. When we type the end-of-file character (which is often Control-D) as the first character of a line, `fgets` returns a null pointer, the loop stops, and the process terminates. In Chapter 18, we describe all the special terminal characters—end of file, backspace one character, erase entire line, and so on—and how to change them.
- Because each line returned by `fgets` is terminated with a newline character, followed by a null byte, we use the standard C function `strlen` to calculate the length of the string, and then replace the newline with a null byte. We do this because the `execlp` function wants a null-terminated argument, not a newline-terminated argument.
- We call `fork` to create a new process, which is a copy of the caller. We say that the caller is the parent and that the newly created process is the child. Then `fork` returns the non-negative process ID of the new child process to the parent,

and returns 0 to the child. Because `fork` creates a new process, we say that it is called once—by the parent—but returns twice—in the parent and in the child.

- In the child, we call `execlp` to execute the command that was read from the standard input. This replaces the child process with the new program file. The combination of `fork` followed by `exec` is called spawning a new process on some operating systems. In the UNIX System, the two parts are separated into individual functions. We'll say a lot more about these functions in Chapter 8.
- Because the child calls `execlp` to execute the new program file, the parent wants to wait for the child to terminate. This is done by calling `waitpid`, specifying which process to wait for: the `pid` argument, which is the process ID of the child. The `waitpid` function also returns the termination status of the child—the `status` variable—but in this simple program, we don't do anything with this value. We could examine it to determine how the child terminated.
- The most fundamental limitation of this program is that we can't pass arguments to the command we execute. We can't, for example, specify the name of a directory to list. We can execute `ls` only on the working directory. To allow arguments would require that we parse the input line, separating the arguments by some convention, probably spaces or tabs, and then pass each argument as a separate parameter to the `execlp` function. Nevertheless, this program is still a useful demonstration of the UNIX System's process control functions.

If we run this program, we get the following results. Note that our program has a different prompt—the percent sign—to distinguish it from the shell's prompt.

```
$ ./a.out
% date
Sat Jan 21 19:42:07 EST 2012
% who
sar      console  Jan  1 14:59
sar      ttys000  Jan  1 14:59
sar      ttys001  Jan 15 15:28
% pwd
/home/sar/bk/apue/3e
% ls
Makefile
a.out
shell1.c
% ^D                                   type the end-of-file character
$                                      the regular shell prompt
```

□

The notation `^D` is used to indicate a control character. Control characters are special characters formed by holding down the control key—often labeled `Control` or `Ctrl`—on your keyboard and then pressing another key at the same time. Control-D, or `^D`, is the default end-of-file character. We'll see many more control characters when we discuss terminal I/O in Chapter 18.

Threads and Thread IDs

Usually, a process has only one thread of control—one set of machine instructions executing at a time. Some problems are easier to solve when more than one thread of control can operate on different parts of the problem. Additionally, multiple threads of control can exploit the parallelism possible on multiprocessor systems.

All threads within a process share the same address space, file descriptors, stacks, and process-related attributes. Each thread executes on its own stack, although any thread can access the stacks of other threads in the same process. Because they can access the same memory, the threads need to synchronize access to shared data among themselves to avoid inconsistencies.

Like processes, threads are identified by IDs. Thread IDs, however, are local to a process. A thread ID from one process has no meaning in another process. We use thread IDs to refer to specific threads as we manipulate the threads within a process.

Functions to control threads parallel those used to control processes. Because threads were added to the UNIX System long after the process model was established, however, the thread model and the process model have some complicated interactions, as we shall see in Chapter 12.

1.7 Error Handling

When an error occurs in one of the UNIX System functions, a negative value is often returned, and the integer `errno` is usually set to a value that tells why. For example, the open function returns either a non-negative file descriptor if all is OK or –1 if an error occurs. An error from open has about 15 possible `errno` values, such as file doesn't exist, permission problem, and so on. Some functions use a convention other than returning a negative value. For example, most functions that return a pointer to an object return a null pointer to indicate an error.

The file `<errno.h>` defines the symbol `errno` and constants for each value that `errno` can assume. Each of these constants begins with the character `E`. Also, the first page of Section 2 of the UNIX system manuals, named `intro`(2), usually lists all these error constants. For example, if `errno` is equal to the constant `EACCES`, this indicates a permission problem, such as insufficient permission to open the requested file.

On Linux, the error constants are listed in the `errno`(3) manual page.

POSIX and ISO C define `errno` as a symbol expanding into a modifiable lvalue of type integer. This can be either an integer that contains the error number or a function that returns a pointer to the error number. The historical definition is

```
extern int errno;
```

But in an environment that supports threads, the process address space is shared among multiple threads, and each thread needs its own local copy of `errno` to prevent one thread from interfering with another. Linux, for example, supports multithreaded access to `errno` by defining it as

```
extern int *__errno_location(void);
#define errno (*__errno_location())
```

There are two rules to be aware of with respect to `errno`. First, its value is never cleared by a routine if an error does not occur. Therefore, we should examine its value only when the return value from a function indicates that an error occurred. Second, the value of `errno` is never set to 0 by any of the functions, and none of the constants defined in `<errno.h>` has a value of 0.

Two functions are defined by the C standard to help with printing error messages.

```
#include <string.h>

char *strerror(int errnum);
```

Returns: pointer to message string

This function maps *errnum*, which is typically the `errno` value, into an error message string and returns a pointer to the string.

The `perror` function produces an error message on the standard error, based on the current value of `errno`, and returns.

```
#include <stdio.h>

void perror(const char *msg);
```

It outputs the string pointed to by *msg*, followed by a colon and a space, followed by the error message corresponding to the value of `errno`, followed by a newline.

Example

Figure 1.8 shows the use of these two error functions.

```
#include "apue.h"
#include <errno.h>

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

Figure 1.8 Demonstrate `strerror` and `perror`

If this program is compiled into the file `a.out`, we have

```
$ ./a.out
EACCES: Permission denied
./a.out: No such file or directory
```

Note that we pass the name of the program—`argv[0]`, whose value is `./a.out`—as the argument to `perror`. This is a standard convention in the UNIX System. By doing this, if the program is executed as part of a pipeline, as in

```
prog1 < inputfile | prog2 | prog3 > outputfile
```

we are able to tell which of the three programs generated a particular error message. □

Instead of calling either `strerror` or `perror` directly, the examples in this text use the error functions shown in Appendix B. These functions let us use the variable argument list facility of ISO C to handle error conditions with a single C statement.

Error Recovery

The errors defined in `<errno.h>` can be divided into two categories: fatal and nonfatal. A fatal error has no recovery action. The best we can do is print an error message on the user's screen or to a log file, and then exit. Nonfatal errors, on the other hand, can sometimes be dealt with more robustly. Most nonfatal errors are temporary, such as a resource shortage, and might not occur when there is less activity on the system.

Resource-related nonfatal errors include `EAGAIN`, `ENFILE`, `ENOBUFS`, `ENOLCK`, `ENOSPC`, `EWOULDBLOCK`, and sometimes `ENOMEM`. `EBUSY` can be treated as nonfatal when it indicates that a shared resource is in use. Sometimes, `EINTR` can be treated as a nonfatal error when it interrupts a slow system call (more on this in Section 10.5).

The typical recovery action for a resource-related nonfatal error is to delay and retry later. This technique can be applied in other circumstances. For example, if an error indicates that a network connection is no longer functioning, it might be possible for the application to delay a short time and then reestablish the connection. Some applications use an exponential backoff algorithm, waiting a longer period of time in each subsequent iteration.

Ultimately, it is up to the application developer to determine the cases where an application can recover from an error. If a reasonable recovery strategy can be used, we can improve the robustness of our application by avoiding an abnormal exit.

1.8 User Identification

User ID

The *user ID* from our entry in the password file is a numeric value that identifies us to the system. This user ID is assigned by the system administrator when our login name is assigned, and we cannot change it. The user ID is normally assigned to be unique for every user. We'll see how the kernel uses the user ID to check whether we have the appropriate permissions to perform certain operations.

We call the user whose user ID is 0 either *root* or the *superuser*. The entry in the password file normally has a login name of `root`, and we refer to the special privileges of this user as superuser privileges. As we'll see in Chapter 4, if a process has superuser privileges, most file permission checks are bypassed. Some operating system functions are restricted to the superuser. The superuser has free rein over the system.

Client versions of Mac OS X ship with the superuser account disabled; server versions ship with the account already enabled. Instructions are available on Apple's Web site describing how to enable it. See `http://support.apple.com/kb/HT1528`.

Group ID

Our entry in the password file also specifies our numeric *group ID*. This, too, is assigned by the system administrator when our login name is assigned. Typically, the password file contains multiple entries that specify the same group ID. Groups are normally used to collect users together into projects or departments. This allows the sharing of resources, such as files, among members of the same group. We'll see in Section 4.5 that we can set the permissions on a file so that all members of a group can access the file, whereas others outside the group cannot.

There is also a group file that maps group names into numeric group IDs. The group file is usually `/etc/group`.

The use of numeric user IDs and numeric group IDs for permissions is historical. With every file on disk, the file system stores both the user ID and the group ID of a file's owner. Storing both of these values requires only four bytes, assuming that each is stored as a two-byte integer. If the full ASCII login name and group name were used instead, additional disk space would be required. In addition, comparing strings during permission checks is more expensive than comparing integers.

Users, however, work better with names than with numbers, so the password file maintains the mapping between login names and user IDs, and the group file provides the mapping between group names and group IDs. The `ls -l` command, for example, prints the login name of the owner of a file, using the password file to map the numeric user ID into the corresponding login name.

Early UNIX systems used 16-bit integers to represent user and group IDs. Contemporary UNIX systems use 32-bit integers.

Example

The program in Figure 1.9 prints the user ID and the group ID.

```
#include "apue.h"

int
main(void)
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

Figure 1.9 Print user ID and group ID

We call the functions `getuid` and `getgid` to return the user ID and the group ID. Running the program yields

```
$ ./a.out
uid = 205, gid = 105
```

□

Supplementary Group IDs

In addition to the group ID specified in the password file for a login name, most versions of the UNIX System allow a user to belong to other groups. This practice started with 4.2BSD, which allowed a user to belong to up to 16 additional groups. These *supplementary group IDs* are obtained at login time by reading the file `/etc/group` and finding the first 16 entries that list the user as a member. As we shall see in the next chapter, POSIX requires that a system support at least 8 supplementary groups per process, but most systems support at least 16.

1.9 Signals

Signals are a technique used to notify a process that some condition has occurred. For example, if a process divides by zero, the signal whose name is `SIGFPE` (floating-point exception) is sent to the process. The process has three choices for dealing with the signal.

1. Ignore the signal. This option isn't recommended for signals that denote a hardware exception, such as dividing by zero or referencing memory outside the address space of the process, as the results are undefined.
2. Let the default action occur. For a divide-by-zero condition, the default is to terminate the process.
3. Provide a function that is called when the signal occurs (this is called "catching" the signal). By providing a function of our own, we'll know when the signal occurs and we can handle it as we wish.

Many conditions generate signals. Two terminal keys, called the *interrupt key*—often the DELETE key or Control-C—and the *quit key*—often Control-backslash—are used to interrupt the currently running process. Another way to generate a signal is by calling the `kill` function. We can call this function from a process to send a signal to another process. Naturally, there are limitations: we have to be the owner of the other process (or the superuser) to be able to send it a signal.

Example

Recall the bare-bones shell example (Figure 1.7). If we invoke this program and press the interrupt key, the process terminates because the default action for this signal, named `SIGINT`, is to terminate the process. The process hasn't told the kernel to do anything other than the default with this signal, so the process terminates.

To catch this signal, the program needs to call the `signal` function, specifying the name of the function to call when the `SIGINT` signal is generated. The function is named `sig_int`; when it's called, it just prints a message and a new prompt. Adding

11 lines to the program in Figure 1.7 gives us the version in Figure 1.10. (The 11 new lines are indicated with a plus sign at the beginning of the line.)

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

+ static void sig_int(int);       /* our signal-catching function */
+
  int
  main(void)
  {
      char    buf[MAXLINE];   /* from apue.h */
      pid_t   pid;
      int     status;

+     if (signal(SIGINT, sig_int) == SIG_ERR)
+         err_sys("signal error");
+
      printf("%% ");  /* print prompt (printf requires %% to print %) */
      while (fgets(buf, MAXLINE, stdin) != NULL) {
          if (buf[strlen(buf) - 1] == '\n')
              buf[strlen(buf) - 1] = 0; /* replace newline with null */

          if ((pid = fork()) < 0) {
              err_sys("fork error");
          } else if (pid == 0) {      /* child */
              execlp(buf, buf, (char *)0);
              err_ret("couldn't execute: %s", buf);
              exit(127);
          }

          /* parent */
          if ((pid = waitpid(pid, &status, 0)) < 0)
              err_sys("waitpid error");
          printf("%% ");
      }
      exit(0);
  }
+
+ void
+ sig_int(int signo)
+ {
+     printf("interrupt\n%% ");
+ }
```

Figure 1.10 Read commands from standard input and execute them

In Chapter 10, we'll take a long look at signals, as most nontrivial applications deal with them. □

1.10 Time Values

Historically, UNIX systems have maintained two different time values:

1. Calendar time. This value counts the number of seconds since the Epoch: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC). (Older manuals refer to UTC as Greenwich Mean Time.) These time values are used to record the time when a file was last modified, for example.

   The primitive system data type `time_t` holds these time values.

2. Process time. This is also called CPU time and measures the central processor resources used by a process. Process time is measured in clock ticks, which have historically been 50, 60, or 100 ticks per second.

   The primitive system data type `clock_t` holds these time values. (We'll show how to obtain the number of clock ticks per second with the `sysconf` function in Section 2.5.4.)

When we measure the execution time of a process, as in Section 3.9, we'll see that the UNIX System maintains three values for a process:

- Clock time
- User CPU time
- System CPU time

The clock time, sometimes called *wall clock time*, is the amount of time the process takes to run, and its value depends on the number of other processes being run on the system. Whenever we report the clock time, the measurements are made with no other activities on the system.

The user CPU time is the CPU time attributed to user instructions. The system CPU time is the CPU time attributed to the kernel when it executes on behalf of the process. For example, whenever a process executes a system service, such as `read` or `write`, the time spent within the kernel performing that system service is charged to the process. The sum of user CPU time and system CPU time is often called the *CPU time*.

It is easy to measure the clock time, user time, and system time of any process: simply execute the `time`(1) command, with the argument to the `time` command being the command we want to measure. For example:

```
$ cd /usr/include
$ time -p grep _POSIX_SOURCE */*.h > /dev/null

real     0m0.81s
user     0m0.11s
sys      0m0.07s
```

The output format from the `time` command depends on the shell being used, because some shells don't run `/usr/bin/time`, but instead have a separate built-in function to measure the time it takes commands to run.

In Section 8.17, we'll see how to obtain these three times from a running process. The general topic of times and dates is covered in Section 6.10.

1.11 System Calls and Library Functions

All operating systems provide service points through which programs request services from the kernel. All implementations of the UNIX System provide a well-defined, limited number of entry points directly into the kernel called *system calls* (recall Figure 1.1). Version 7 of the Research UNIX System provided about 50 system calls, 4.4BSD provided about 110, and SVR4 had around 120. The exact number of system calls varies depending on the operating system version. More recent systems have seen incredible growth in the number of supported system calls. Linux 3.2.0 has 380 system calls and FreeBSD 8.0 has over 450.

The system call interface has always been documented in Section 2 of the *UNIX Programmer's Manual*. Its definition is in the C language, no matter which implementation technique is actually used on any given system to invoke a system call. This differs from many older operating systems, which traditionally defined the kernel entry points in the assembly language of the machine.

The technique used on UNIX systems is for each system call to have a function of the same name in the standard C library. The user process calls this function, using the standard C calling sequence. This function then invokes the appropriate kernel service, using whatever technique is required on the system. For example, the function may put one or more of the C arguments into general registers and then execute some machine instruction that generates a software interrupt in the kernel. For our purposes, we can consider the system calls to be C functions.

Section 3 of the *UNIX Programmer's Manual* defines the general-purpose library functions available to programmers. These functions aren't entry points into the kernel, although they may invoke one or more of the kernel's system calls. For example, the `printf` function may use the `write` system call to output a string, but the `strcpy` (copy a string) and `atoi` (convert ASCII to integer) functions don't involve the kernel at all.

From an implementor's point of view, the distinction between a system call and a library function is fundamental. From a user's perspective, however, the difference is not as critical. From our perspective in this text, both system calls and library functions appear as normal C functions. Both exist to provide services for application programs. We should realize, however, that we can replace the library functions, if desired, whereas the system calls usually cannot be replaced.

Consider the memory allocation function `malloc` as an example. There are many ways to do memory allocation and its associated garbage collection (best fit, first fit, and so on). No single technique is optimal for all programs. The UNIX system call that handles memory allocation, `sbrk`(2), is not a general-purpose memory manager. It increases or decreases the address space of the process by a specified number of bytes. How that space is managed is up to the process. The memory allocation function, `malloc`(3), implements one particular type of allocation. If we don't like its operation, we can define our own `malloc` function, which will probably use the `sbrk` system call. In fact, numerous software packages implement their own memory allocation algorithms with the `sbrk` system call. Figure 1.11 shows the relationship between the application, the `malloc` function, and the `sbrk` system call.

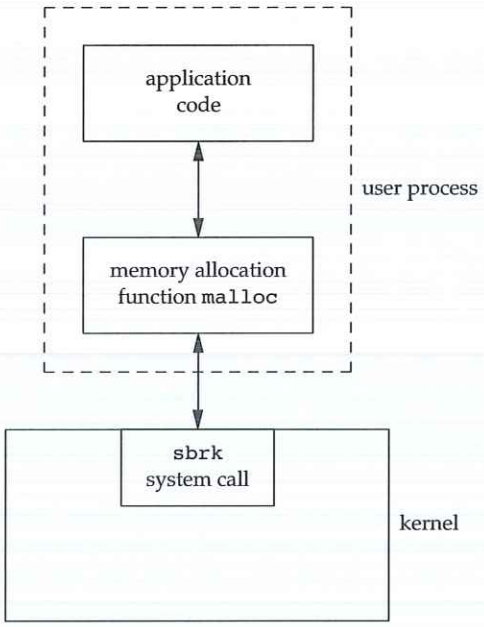

Figure 1.11 Separation of `malloc` function and `sbrk` system call

Here we have a clean separation of duties: the system call in the kernel allocates an additional chunk of space on behalf of the process. The `malloc` library function manages this space from user level.

Another example to illustrate the difference between a system call and a library function is the interface the UNIX System provides to determine the current time and date. Some operating systems provide one system call to return the time and another to return the date. Any special handling, such as the switch to or from daylight saving time, is handled by the kernel or requires human intervention. The UNIX System, in contrast, provides a single system call that returns the number of seconds since the Epoch: midnight, January 1, 1970, Coordinated Universal Time. Any interpretation of this value, such as converting it to a human-readable time and date using the local time zone, is left to the user process. The standard C library provides routines to handle most cases. These library routines handle such details as the various algorithms for daylight saving time.

An application can either make a system call or call a library routine. Also realize that many library routines invoke a system call. This is shown in Figure 1.12.

Another difference between system calls and library functions is that system calls usually provide a minimal interface, whereas library functions often provide more elaborate functionality. We've seen this already in the difference between the `sbrk` system call and the `malloc` library function. We'll see this difference again later, when

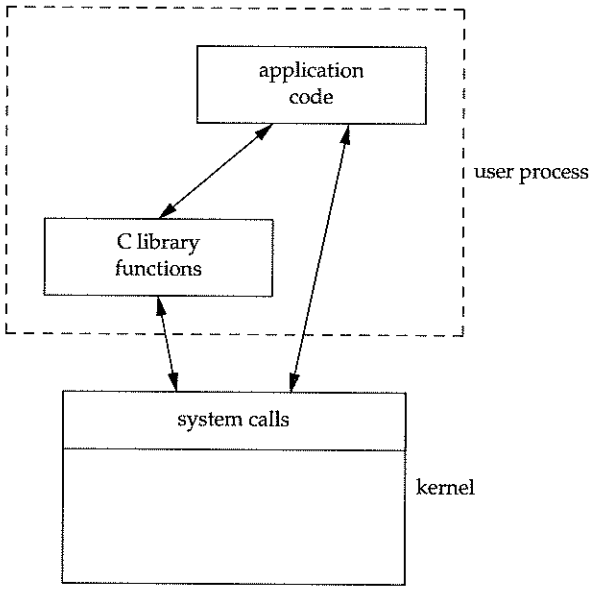

Figure 1.12 Difference between C library functions and system calls

we compare the unbuffered I/O functions (Chapter 3) and the standard I/O functions (Chapter 5).

The process control system calls (`fork`, `exec`, and `waitpid`) are usually invoked by the user's application code directly. (Recall the bare-bones shell in Figure 1.7.) But some library routines exist to simplify certain common cases: the `system` and popen library routines, for example. In Section 8.13, we'll show an implementation of the `system` function that invokes the basic process control system calls. We'll enhance this example in Section 10.18 to handle signals correctly.

To define the interface to the UNIX System that most programmers use, we have to describe both the system calls and some of the library functions. If we described only the `sbrk` system call, for example, we would skip the more programmer-friendly `malloc` library function that many applications use. In this text, we'll use the term *function* to refer to both system calls and library functions, except when the distinction is necessary.

1.12 Summary

This chapter has provided a short tour of the UNIX System. We've described some of the fundamental terms that we'll encounter over and over again. We've seen numerous small examples of UNIX programs to give us a feel for what the remainder of the text talks about.

The next chapter is about standardization of the UNIX System and the effect of work in this area on current systems. Standards, particularly the ISO C standard and the POSIX.1 standard, will affect the rest of the text.

Exercises

1.1 Verify on your system that the directories dot and dot-dot are not the same, except in the root directory.

1.2 In the output from the program in Figure 1.6, what happened to the processes with process IDs 852 and 853?

1.3 In Section 1.7, the argument to `perror` is defined with the ISO C attribute `const`, whereas the integer argument to `strerror` isn't defined with this attribute. Why?

1.4 If the calendar time is stored as a signed 32-bit integer, in which year will it overflow? How can we extend the overflow point? Are these strategies compatible with existing applications?

1.5 If the process time is stored as a signed 32-bit integer, and if the system counts 100 ticks per second, after how many days will the value overflow?