3

SYSTEM PROGRAMMING CONCEPTS

This chapter covers various topics that are prerequisites for system programming. We begin by introducing system calls and detailing the steps that occur during their execution. We then consider library functions and how they differ from system calls, and couple this with a description of the (GNU) C library.

Whenever we make a system call or call a library function, we should always check the return status of the call in order to determine if it was successful. We describe how to perform such checks, and present a set of functions that are used in most of the example programs in this book to diagnose errors from system calls and library functions.

We conclude by looking at various issues related to portable programming, specifically the use of feature test macros and the standard system data types defined by SUSv3.

3.1 System Calls

A *system call* is a controlled entry point into the kernel, allowing a process to request that the kernel perform some action on the process's behalf. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a

new process, performing I/O, and creating a pipe for interprocess communication. (The *syscalls(2)* manual page lists the Linux system calls.)

Before going into the details of how a system call works, we note some general points:

- A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory.
- The set of system calls is fixed. Each system call is identified by a unique number. (This numbering scheme is not normally visible to programs, which identify system calls by name.)
- Each system call may have a set of arguments that specify information to be transferred from user space (i.e., the process's virtual address space) to kernel space and vice versa.

From a programming point of view, invoking a system call looks much like calling a C function. However, behind the scenes, many steps occur during the execution of a system call. To illustrate this, we consider the steps in the order that they occur on a specific hardware implementation, the x86-32. The steps are as follows:

1. The application program makes a system call by invoking a wrapper function in the C library.
2. The wrapper function must make all of the system call arguments available to the system call trap-handling routine (described shortly). These arguments are passed to the wrapper via the stack, but the kernel expects them in specific registers. The wrapper function copies the arguments to these registers.
3. Since all system calls enter the kernel in the same way, the kernel needs some method of identifying the system call. To permit this, the wrapper function copies the system call number into a specific CPU register (`%eax`).
4. The wrapper function executes a *trap* machine instruction (`int 0x80`), which causes the processor to switch from user mode to kernel mode and execute code pointed to by location `0x80` (128 decimal) of the system's trap vector.

   > More recent x86-32 architectures implement the `sysenter` instruction, which provides a faster method of entering kernel mode than the conventional `int 0x80` trap instruction. The use of `sysenter` is supported in the 2.6 kernel and from *glibc* 2.3.2 onward.

5. In response to the trap to location `0x80`, the kernel invokes its *system_call()* routine (located in the assembler file `arch/i386/entry.S`) to handle the trap. This handler:
   a) Saves register values onto the kernel stack (Section 6.5).
   b) Checks the validity of the system call number.
   c) Invokes the appropriate system call service routine, which is found by using the system call number to index a table of all system call service routines (the kernel variable *sys_call_table*). If the system call service routine has any arguments, it first checks their validity; for example, it checks that addresses point to valid locations in user memory. Then the service

routine performs the required task, which may involve modifying values at addresses specified in the given arguments and transferring data between user memory and kernel memory (e.g., in I/O operations). Finally, the service routine returns a result status to the *system_call()* routine.

d) Restores register values from the kernel stack and places the system call return value on the stack.

e) Returns to the wrapper function, simultaneously returning the processor to user mode.

6. If the return value of the system call service routine indicated an error, the wrapper function sets the global variable *errno* (see Section 3.4) using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

> On Linux, system call service routines follow a convention of returning a nonnegative value to indicate success. In case of an error, the routine returns a negative number, which is the negated value of one of the *errno* constants. When a negative value is returned, the C library wrapper function negates it (to make it positive), copies the result into *errno*, and returns −1 as the function result of the wrapper to indicate an error to the calling program.
>
> This convention relies on the assumption that system call service routines don't return negative values on success. However, for a few of these routines, this assumption doesn't hold. Normally, this is not a problem, since the range of negated *errno* values doesn't overlap with valid negative return values. However, this convention does cause a problem in one case: the `F_GETOWN` operation of the *fcntl()* system call, which we describe in Section 63.3.

Figure 3-1 illustrates the above sequence using the example of the *execve()* system call. On Linux/x86-32, *execve()* is system call number 11 (`__NR_execve`). Thus, in the *sys_call_table* vector, entry 11 contains the address of *sys_execve()*, the service routine for this system call. (On Linux, system call service routines typically have names of the form *sys_xyz()*, where *xyz()* is the system call in question.)

The information given in the preceding paragraphs is more than we'll usually need to know for the remainder of this book. However, it illustrates the important point that, even for a simple system call, quite a bit of work must be done, and thus system calls have a small but appreciable overhead.

> As an example of the overhead of making a system call, consider the *getppid()* system call, which simply returns the process ID of the parent of the calling process. On one of the author's x86-32 systems running Linux 2.6.25, 10 million calls to *getppid()* required approximately 2.2 seconds to complete. This amounts to around 0.3 microseconds per call. By comparison, on the same system, 10 million calls to a C function that simply returns an integer required 0.11 seconds, or around one-twentieth of the time required for calls to *getppid()*. Of course, most system calls have significantly more overhead than *getppid()*.

Since, from the point of view of a C program, calling the C library wrapper function is synonymous with invoking the corresponding system call service routine, in the remainder of this book, we use wording such as "invoking the system call *xyz()*" to mean "calling the wrapper function that invokes the system call *xyz()*."

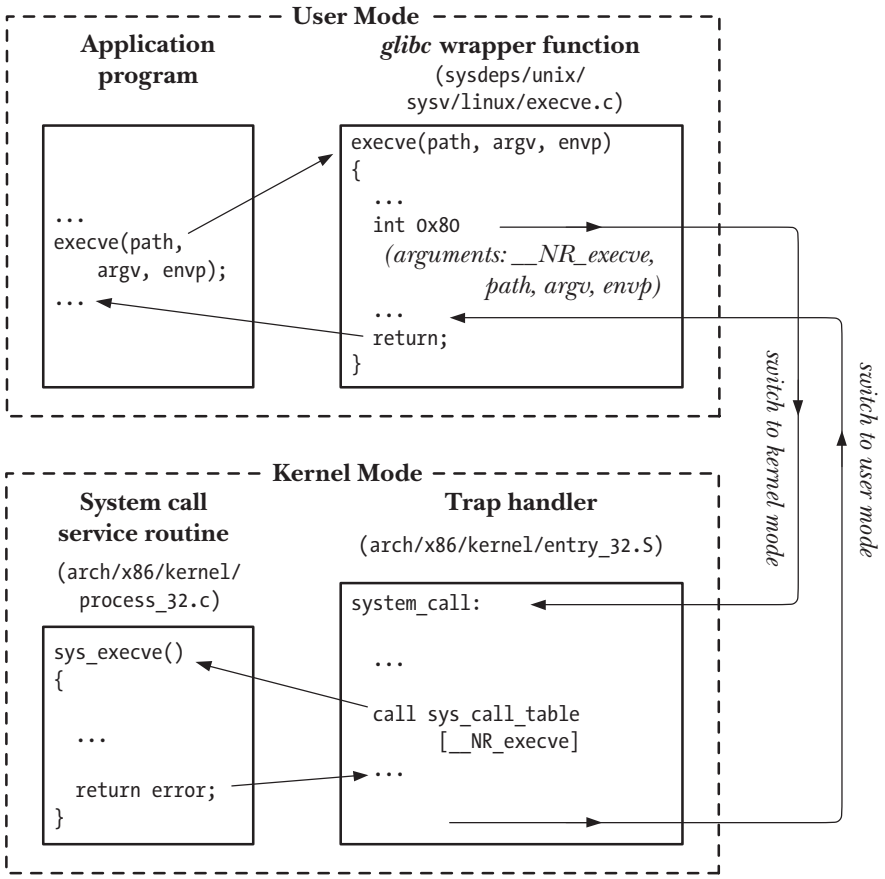

Figure 3-1: Steps in the execution of a system call

Appendix A describes the *strace* command, which can be used to trace the system calls made by a program, either for debugging purposes or simply to investigate what a program is doing.

More information about the Linux system call mechanism can be found in [Love, 2010], [Bovet & Cesati, 2005], and [Maxwell, 1999].

3.2 Library Functions

A *library function* is simply one of the multitude of functions that constitutes the standard C library. (For brevity, when talking about a specific function in the rest of the book we'll often just write *function* rather than *library function*.) The purposes of these functions are very diverse, including such tasks as opening a file, converting a time to a human-readable format, and comparing two character strings.

Many library functions don't make any use of system calls (e.g., the string-manipulation functions). On the other hand, some library functions are layered on top of system calls. For example, the *fopen()* library function uses the *open()* system call to actually open a file. Often, library functions are designed to provide a more caller-friendly interface than the underlying system call. For example, the *printf()* function provides output formatting and data buffering, whereas the *write()* system

call just outputs a block of bytes. Similarly, the *malloc()* and *free()* functions perform various bookkeeping tasks that make them a much easier way to allocate and free memory than the underlying *brk()* system call.

3.3 The Standard C Library; The GNU C Library (*glibc*)

There are different implementations of the standard C library on the various UNIX implementations. The most commonly used implementation on Linux is the GNU C library (*glibc*, *http://www.gnu.org/software/libc/*).

> The principal developer and maintainer of the GNU C library was initially Roland McGrath. Nowadays, this task is carried out by Ulrich Drepper.
>
> Various other C libraries are available for Linux, including libraries with smaller memory requirements for use in embedded device applications. Examples include *uClibc* (*http://www.uclibc.org/*) and *diet libc* (*http://www.fefe.de/dietlibc/*). In this book, we confine the discussion to *glibc*, since that is the C library used by most applications developed on Linux.

Determining the version of *glibc* on the system

Sometimes, we need to determine the version of *glibc* on a system. From the shell, we can do this by running the *glibc* shared library file as though it were an executable program. When we run the library as an executable, it displays various text, including its version number:

```
$ /lib/libc.so.6
GNU C Library stable release version 2.10.1, by Roland McGrath et al.
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.4.0 20090506 (Red Hat 4.4.0-4).
Compiled on a Linux >>2.6.18-128.4.1.el5<< system on 2009-08-19.
Available extensions:
        The C stubs add-on version 2.1.2.
        crypt add-on version 2.1 by Michael Glad and others
        GNU Libidn by Simon Josefsson
        Native POSIX Threads Library by Ulrich Drepper et al
        BIND-8.2.3-T5B
        RT using linux kernel aio
For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
```

In some Linux distributions, the GNU C library resides at a pathname other than `/lib/libc.so.6`. One way of determining the location of the library is to run the *ldd* (list dynamic dependencies) program against an executable linked dynamically against *glibc* (most executables are linked in this manner). We can then inspect the resulting library dependency list to find the location of the *glibc* shared library:

```
$ ldd myprog | grep libc
        libc.so.6 => /lib/tls/libc.so.6 (0x4004b000)
```

There are two means by which an application program can determine the version of the GNU C library present on the system: by testing constants or by calling a library function. From version 2.0 onward, *glibc* defines two constants, `__GLIBC__` and `__GLIBC_MINOR__`, that can be tested at compile time (in `#ifdef` statements). On a system with *glibc* 2.12 installed, these constants would have the values 2 and 12. However, these constants are of limited use in a program that is compiled on one system but run on another system with a different *glibc*. To handle this possibility, a program can call the *gnu_get_libc_version()* function to determine the version of *glibc* available at run time.

```
#include <gnu/libc-version.h>

const char *gnu_get_libc_version(void);
```

Returns pointer to null-terminated, statically allocated string containing GNU C library version number

The *gnu_get_libc_version()* function returns a pointer to a string, such as *2.12*.

> We can also obtain version information by using the *confstr()* function to retrieve the value of the (*glibc*-specific) `_CS_GNU_LIBC_VERSION` configuration variable. This call returns a string such as *glibc 2.12*.

3.4 Handling Errors from System Calls and Library Functions

Almost every system call and library function returns some type of status value indicating whether the call succeeded or failed. This status value should *always* be checked to see whether the call succeeded. If it did not, then appropriate action should be taken—at the very least, the program should display an error message warning that something unexpected occurred.

Although it is tempting to save typing time by excluding these checks (especially after seeing examples of UNIX and Linux programs where status values are not checked), it is a false economy. Many hours of debugging time can be wasted because a check was not made on the status return of a system call or library function that "couldn't possibly fail."

> A few system calls never fail. For example, *getpid()* always successfully returns the ID of a process, and *_exit()* always terminates a process. It is not necessary to check the return values from such system calls.

Handling system call errors

The manual page for each system call documents the possible return values of the call, showing which value(s) indicate an error. Usually, an error is indicated by a return of −1. Thus, a system call can be checked with code such as the following:

```
fd = open(pathname, flags, mode);       /* system call to open a file */
if (fd == -1) {
    /* Code to handle the error */
}
...
```

```
if (close(fd) == -1) {
    /* Code to handle the error */
}
```

When a system call fails, it sets the global integer variable *errno* to a positive value that identifies the specific error. Including the <errno.h> header file provides a declaration of *errno*, as well as a set of constants for the various error numbers. All of these symbolic names commence with E. The section headed ERRORS in each manual page provides a list of possible *errno* values that can be returned by each system call. Here is a simple example of the use of *errno* to diagnose a system call error:

```
cnt = read(fd, buf, numbytes);
if (cnt == -1) {
    if (errno == EINTR)
        fprintf(stderr, "read was interrupted by a signal\n");
    else {
        /* Some other error occurred */
    }
}
```

Successful system calls and library functions never reset *errno* to 0, so this variable may have a nonzero value as a consequence of an error from a previous call. Furthermore, SUSv3 permits a successful function call to set *errno* to a nonzero value (although few functions do this). Therefore, when checking for an error, we should always first check if the function return value indicates an error, and only then examine *errno* to determine the cause of the error.

A few system calls (e.g., *getpriority()*) can legitimately return −1 on success. To determine whether an error occurs in such calls, we set *errno* to 0 before the call, and then check it afterward. If the call returns −1 and *errno* is nonzero, an error occurred. (A similar statement also applies to a few library functions.)

A common course of action after a failed system call is to print an error message based on the *errno* value. The *perror()* and *strerror()* library functions are provided for this purpose.

The *perror()* function prints the string pointed to by its *msg* argument, followed by a message corresponding to the current value of *errno*.

```
#include <stdio.h>

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

A simple way of handling errors from system calls would be as follows:

```
fd = open(pathname, flags, mode);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

The *strerror()* function returns the error string corresponding to the error number given in its *errnum* argument.

```
#include <string.h>

char *strerror(int errnum);
                                Returns pointer to error string corresponding to errnum
```

The string returned by *strerror()* may be statically allocated, which means that it could be overwritten by subsequent calls to *strerror()*.

If *errnum* specifies an unrecognized error number, *strerror()* returns a string of the form *Unknown error nnn*. On some other implementations, *strerror()* instead returns NULL in this case.

Because *perror()* and *strerror()* functions are locale-sensitive (Section 10.4), error descriptions are displayed in the local language.

Handling errors from library functions

The various library functions return different data types and different values to indicate failure. (Check the manual page for each function.) For our purposes, library functions can be divided into the following categories:

- Some library functions return error information in exactly the same way as system calls: a −1 return value, with *errno* indicating the specific error. An example of such a function is *remove()*, which removes a file (using the *unlink()* system call) or a directory (using the *rmdir()* system call). Errors from these functions can be diagnosed in the same way as errors from system calls.
- Some library functions return a value other than −1 on error, but nevertheless set *errno* to indicate the specific error condition. For example, *fopen()* returns a NULL pointer on error, and the setting of *errno* depends on which underlying system call failed. The *perror()* and *strerror()* functions can be used to diagnose these errors.
- Other library functions don't use *errno* at all. The method for determining the existence and cause of errors depends on the particular function and is documented in the function's manual page. For these functions, it is a mistake to use *errno*, *perror()*, or *strerror()* to diagnose errors.

3.5 Notes on the Example Programs in This Book

In this section, we describe various conventions and features commonly employed by the example programs presented in this book.

3.5.1 Command-Line Options and Arguments

Many of the example programs in this book rely on command-line options and arguments to determine their behavior.

Traditional UNIX command-line options consist of an initial hyphen, a letter that identifies the option, and an optional argument. (GNU utilities provide an extended option syntax consisting of two initial hyphens, followed by a string identifying the option and an optional argument.) To parse these options, we use the standard *getopt()* library function (described in Appendix B).

Each of our example programs that has a nontrivial command-line syntax provides a simple help facility for the user: if invoked with the *--help* option, the program displays a usage message that indicates the syntax for command-line options and arguments.

3.5.2 Common Functions and Header Files

Most of the example programs include a header file containing commonly required definitions, and they also use a set of common functions. We discuss the header file and functions in this section.

Common header file

Listing 3-1 is the header file used by nearly every program in this book. This header file includes various other header files used by many of the example programs, defines a *Boolean* data type, and defines macros for calculating the minimum and maximum of two numeric values. Using this header file allows us to make the example programs a bit shorter.

Listing 3-1: Header file used by most example programs

```
lib/tlpi_hdr.h
#ifndef TLPI_HDR_H
#define TLPI_HDR_H      /* Prevent accidental double inclusion */

#include <sys/types.h>  /* Type definitions used by many programs */
#include <stdio.h>      /* Standard I/O functions */
#include <stdlib.h>     /* Prototypes of commonly used library functions,
                           plus EXIT_SUCCESS and EXIT_FAILURE constants */
#include <unistd.h>     /* Prototypes for many system calls */
#include <errno.h>      /* Declares errno and defines error constants */
#include <string.h>     /* Commonly used string-handling functions */

#include "get_num.h"    /* Declares our functions for handling numeric
                           arguments (getInt(), getLong()) */

#include "error_functions.h"  /* Declares our error-handling functions */

typedef enum { FALSE, TRUE } Boolean;

#define min(m,n) ((m) < (n) ? (m) : (n))
#define max(m,n) ((m) > (n) ? (m) : (n))

#endif
lib/tlpi_hdr.h
```

Error-diagnostic functions

To simplify error handling in our example programs, we use the error-diagnostic functions whose declarations are shown in Listing 3-2.

Listing 3-2: Declarations for common error-handling functions

lib/error_functions.h

```
#ifndef ERROR_FUNCTIONS_H
#define ERROR_FUNCTIONS_H

void errMsg(const char *format, ...);

#ifdef __GNUC__

/* This macro stops 'gcc -Wall' complaining that "control reaches
       end of non-void function" if we use the following functions to
       terminate main() or some other non-void function. */

#define NORETURN __attribute__ ((__noreturn__))
#else
#define NORETURN
#endif

void errExit(const char *format, ...) NORETURN ;

void err_exit(const char *format, ...) NORETURN ;

void errExitEN(int errnum, const char *format, ...) NORETURN ;

void fatal(const char *format, ...) NORETURN ;

void usageErr(const char *format, ...) NORETURN ;

void cmdLineErr(const char *format, ...) NORETURN ;

#endif
```

lib/error_functions.h

To diagnose errors from system calls and library functions, we use *errMsg()*, *errExit()*, *err_exit()*, and *errExitEN()*.

```
#include "tlpi_hdr.h"

void errMsg(const char *format, ...);
void errExit(const char *format, ...);
void err_exit(const char *format, ...);
void errExitEN(int errnum, const char *format, ...);
```

The *errMsg()* function prints a message on standard error. Its argument list is the same as for *printf()*, except that a terminating newline character is automatically appended to the output string. The *errMsg()* function prints the error text corresponding to the current value of *errno*—this consists of the error name, such as EPERM, plus the error description as returned by *strerror()*—followed by the formatted output specified in the argument list.

The *errExit()* function operates like *errMsg()*, but also terminates the program, either by calling *exit()* or, if the environment variable EF_DUMPCORE is defined with a

nonempty string value, by calling *abort()* to produce a core dump file for use with the debugger. (We explain core dump files in Section 22.1.)

The *err_exit()* function is similar to *errExit()*, but differs in two respects:

- It doesn't flush standard output before printing the error message.
- It terminates the process by calling *_exit()* instead of *exit()*. This causes the process to terminate without flushing *stdio* buffers or invoking exit handlers.

The details of these differences in the operation of *err_exit()* will become clearer in Chapter 25, where we describe the differences between *_exit()* and *exit()*, and consider the treatment of *stdio* buffers and exit handlers in a child created by *fork()*. For now, we simply note that *err_exit()* is especially useful if we write a library function that creates a child process that needs to terminate because of an error. This termination should occur without flushing the child's copy of the parent's (i.e., the calling process's) *stdio* buffers and without invoking exit handlers established by the parent.

The *errExitEN()* function is the same as *errExit()*, except that instead of printing the error text corresponding to the current value of *errno*, it prints the text corresponding to the error number (thus, the *EN* suffix) given in the argument *errnum*.

Mainly, we use *errExitEN()* in programs that employ the POSIX threads API. Unlike traditional UNIX system calls, which return −1 on error, the POSIX threads functions diagnose an error by returning an error number (i.e., a positive number of the type normally placed in *errno*) as their function result. (The POSIX threads functions return 0 on success.)

We could diagnose errors from the POSIX threads functions using code such as the following:

```
errno = pthread_create(&thread, NULL, func, &arg);
if (errno != 0)
    errExit("pthread_create");
```

However, this approach is inefficient because *errno* is defined in threaded programs as a macro that expands into a function call that returns a modifiable lvalue. Thus, each use of *errno* results in a function call. The *errExitEN()* function allows us to write a more efficient equivalent of the above code:

```
int s;

s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
    errExitEN(s, "pthread_create");
```

> In C terminology, an *lvalue* is an expression referring to a region of storage. The most common example of an lvalue is an identifier for a variable. Some operators also yield lvalues. For example, if *p* is a pointer to a storage area, then **p* is an lvalue. Under the POSIX threads API, *errno* is redefined to be a function that returns a pointer to a thread-specific storage area (see Section 31.3).

To diagnose other types of errors, we use *fatal()*, *usageErr()*, and *cmdLineErr()*.

```
#include "tlpi_hdr.h"

void fatal(const char *format, ...);
void usageErr(const char *format, ...);
void cmdLineErr(const char *format, ...);
```

The *fatal()* function is used to diagnose general errors, including errors from library functions that don't set *errno*. Its argument list is the same as for *printf()*, except that a terminating newline character is automatically appended to the output string. It prints the formatted output on standard error and then terminates the program as with *errExit()*.

The *usageErr()* function is used to diagnose errors in command-line argument usage. It takes an argument list in the style of *printf()* and prints the string *Usage:* followed by the formatted output on standard error, and then terminates the program by calling *exit()*. (Some of the example programs in this book provide their own extended version of the *usageErr()* function, under the name *usageError()*.)

The *cmdLineErr()* function is similar to *usageErr()*, but is intended for diagnosing errors in the command-line arguments specified to a program.

The implementations of our error-diagnostic functions are shown in Listing 3-3.

Listing 3-3: Error-handling functions used by all programs

lib/error_functions.c

```
#include <stdarg.h>
#include "error_functions.h"
#include "tlpi_hdr.h"
#include "ename.c.inc"          /* Defines ename and MAX_ENAME */

#ifdef __GNUC__
__attribute__ ((__noreturn__))
#endif
static void
terminate(Boolean useExit3)
{
    char *s;

    /* Dump core if EF_DUMPCORE environment variable is defined and
       is a nonempty string; otherwise call exit(3) or _exit(2),
       depending on the value of 'useExit3'. */

    s = getenv("EF_DUMPCORE");

    if (s != NULL && *s != '\0')
        abort();
    else if (useExit3)
        exit(EXIT_FAILURE);
    else
        _exit(EXIT_FAILURE);
}
```

```
static void
outputError(Boolean useErr, int err, Boolean flushStdout,
        const char *format, va_list ap)
{
#define BUF_SIZE 500
    char buf[BUF_SIZE], userMsg[BUF_SIZE], errText[BUF_SIZE];

    vsnprintf(userMsg, BUF_SIZE, format, ap);

    if (useErr)
        snprintf(errText, BUF_SIZE, " [%s %s]",
                (err > 0 && err <= MAX_ENAME) ?
                ename[err] : "?UNKNOWN?", strerror(err));
    else
        snprintf(errText, BUF_SIZE, ":");

    snprintf(buf, BUF_SIZE, "ERROR%s %s\n", errText, userMsg);

    if (flushStdout)
        fflush(stdout);       /* Flush any pending stdout */
    fputs(buf, stderr);
    fflush(stderr);           /* In case stderr is not line-buffered */
}

void
errMsg(const char *format, ...)
{
    va_list argList;
    int savedErrno;

    savedErrno = errno;       /* In case we change it here */

    va_start(argList, format);
    outputError(TRUE, errno, TRUE, format, argList);
    va_end(argList);

    errno = savedErrno;
}

void
errExit(const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(TRUE, errno, TRUE, format, argList);
    va_end(argList);

    terminate(TRUE);
}
```

```
void
err_exit(const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(TRUE, errno, FALSE, format, argList);
    va_end(argList);

    terminate(FALSE);
}

void
errExitEN(int errnum, const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(TRUE, errnum, TRUE, format, argList);
    va_end(argList);

    terminate(TRUE);
}

void
fatal(const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(FALSE, 0, TRUE, format, argList);
    va_end(argList);

    terminate(TRUE);
}

void
usageErr(const char *format, ...)
{
    va_list argList;

    fflush(stdout);           /* Flush any pending stdout */

    fprintf(stderr, "Usage: ");
    va_start(argList, format);
    vfprintf(stderr, format, argList);
    va_end(argList);

    fflush(stderr);           /* In case stderr is not line-buffered */
    exit(EXIT_FAILURE);
}
```

```
void
cmdLineErr(const char *format, ...)
{
    va_list argList;

    fflush(stdout);           /* Flush any pending stdout */

    fprintf(stderr, "Command-line usage error: ");
    va_start(argList, format);
    vfprintf(stderr, format, argList);
    va_end(argList);

    fflush(stderr);           /* In case stderr is not line-buffered */
    exit(EXIT_FAILURE);
}
```

lib/error_functions.c

The file `enames.c.inc` included by Listing 3-3 is shown in Listing 3-4. This file defines an array of strings, *ename*, that are the symbolic names corresponding to each of the possible *errno* values. Our error-handling functions use this array to print out the symbolic name corresponding to a particular error number. This is a workaround to deal with the facts that, on the one hand, the string returned by *strerror()* doesn't identify the symbolic constant corresponding to its error message, while, on the other hand, the manual pages describe errors using their symbolic names. Printing out the symbolic name gives us an easy way of looking up the cause of an error in the manual pages.

> The content of the `ename.c.inc` file is architecture-specific, because *errno* values vary somewhat from one Linux hardware architecture to another. The version shown in Listing 3-4 is for a Linux 2.6/x86-32 system. This file was built using a script (`lib/Build_ename.sh`) included in the source code distribution for this book. This script can be used to build a version of `ename.c.inc` that should be suitable for a specific hardware platform and kernel version.

Note that some of the strings in the *ename* array are empty. These correspond to unused error values. Furthermore, some of the strings in *ename* consist of two error names separate by a slash. These strings correspond to cases where two symbolic error names have the same numeric value.

> From the `ename.c.inc` file, we can see that the `EAGAIN` and `EWOULDBLOCK` errors have the same value. (SUSv3 explicitly permits this, and the values of these constants are the same on most, but not all, other UNIX systems.) These errors are returned by a system call in circumstances in which it would normally block (i.e., be forced to wait before completing), but the caller requested that the system call return an error instead of blocking. `EAGAIN` originated on System V, and it was the error returned for system calls performing I/O, semaphore operations, message queue operations, and file locking (*fcntl()*). `EWOULDBLOCK` originated on BSD, and it was returned by file locking (*flock()*) and socket-related system calls.
>
> Within SUSv3, `EWOULDBLOCK` is mentioned only in the specifications of various interfaces related to sockets. For these interfaces, SUSv3 permits either `EAGAIN` or `EWOULDBLOCK` to be returned by nonblocking calls. For all other nonblocking calls, only the error `EAGAIN` is specified in SUSv3.

Listing 3-4: Linux error names (x86-32 version)

lib/ename.c.inc

```
static char *ename[] = {
    /*   0 */ "",
    /*   1 */ "EPERM", "ENOENT", "ESRCH", "EINTR", "EIO", "ENXIO", "E2BIG",
    /*   8 */ "ENOEXEC", "EBADF", "ECHILD", "EAGAIN/EWOULDBLOCK", "ENOMEM",
    /*  13 */ "EACCES", "EFAULT", "ENOTBLK", "EBUSY", "EEXIST", "EXDEV",
    /*  19 */ "ENODEV", "ENOTDIR", "EISDIR", "EINVAL", "ENFILE", "EMFILE",
    /*  25 */ "ENOTTY", "ETXTBSY", "EFBIG", "ENOSPC", "ESPIPE", "EROFS",
    /*  31 */ "EMLINK", "EPIPE", "EDOM", "ERANGE", "EDEADLK/EDEADLOCK",
    /*  36 */ "ENAMETOOLONG", "ENOLCK", "ENOSYS", "ENOTEMPTY", "ELOOP", "",
    /*  42 */ "ENOMSG", "EIDRM", "ECHRNG", "EL2NSYNC", "EL3HLT", "EL3RST",
    /*  48 */ "ELNRNG", "EUNATCH", "ENOCSI", "EL2HLT", "EBADE", "EBADR",
    /*  54 */ "EXFULL", "ENOANO", "EBADRQC", "EBADSLT", "", "EBFONT", "ENOSTR",
    /*  61 */ "ENODATA", "ETIME", "ENOSR", "ENONET", "ENOPKG", "EREMOTE",
    /*  67 */ "ENOLINK", "EADV", "ESRMNT", "ECOMM", "EPROTO", "EMULTIHOP",
    /*  73 */ "EDOTDOT", "EBADMSG", "EOVERFLOW", "ENOTUNIQ", "EBADFD",
    /*  78 */ "EREMCHG", "ELIBACC", "ELIBBAD", "ELIBSCN", "ELIBMAX",
    /*  83 */ "ELIBEXEC", "EILSEQ", "ERESTART", "ESTRPIPE", "EUSERS",
    /*  88 */ "ENOTSOCK", "EDESTADDRREQ", "EMSGSIZE", "EPROTOTYPE",
    /*  92 */ "ENOPROTOOPT", "EPROTONOSUPPORT", "ESOCKTNOSUPPORT",
    /*  95 */ "EOPNOTSUPP/ENOTSUP", "EPFNOSUPPORT", "EAFNOSUPPORT",
    /*  98 */ "EADDRINUSE", "EADDRNOTAVAIL", "ENETDOWN", "ENETUNREACH",
    /* 102 */ "ENETRESET", "ECONNABORTED", "ECONNRESET", "ENOBUFS", "EISCONN",
    /* 107 */ "ENOTCONN", "ESHUTDOWN", "ETOOMANYREFS", "ETIMEDOUT",
    /* 111 */ "ECONNREFUSED", "EHOSTDOWN", "EHOSTUNREACH", "EALREADY",
    /* 115 */ "EINPROGRESS", "ESTALE", "EUCLEAN", "ENOTNAM", "ENAVAIL",
    /* 120 */ "EISNAM", "EREMOTEIO", "EDQUOT", "ENOMEDIUM", "EMEDIUMTYPE",
    /* 125 */ "ECANCELED", "ENOKEY", "EKEYEXPIRED", "EKEYREVOKED",
    /* 129 */ "EKEYREJECTED", "EOWNERDEAD", "ENOTRECOVERABLE", "ERFKILL"
};

#define MAX_ENAME 132
```

lib/ename.c.inc

Functions for parsing numeric command-line arguments

The header file in Listing 3-5 provides the declaration of two functions that we frequently use for parsing integer command-line arguments: *getInt()* and *getLong()*. The primary advantage of using these functions instead of *atoi()*, *atol()*, and *strtol()* is that they provide some basic validity checking of numeric arguments.

```
#include "tlpi_hdr.h"

int getInt(const char *arg, int flags, const char *name);
long getLong(const char *arg, int flags, const char *name);
```

Both return *arg* converted to numeric form

The *getInt()* and *getLong()* functions convert the string pointed to by *arg* to an *int* or a *long*, respectively. If *arg* doesn't contain a valid integer string (i.e., only digits and the characters + and -), then these functions print an error message and terminate the program.

If the *name* argument is non-NULL, it should contain a string identifying the argument in *arg*. This string is included as part of any error message displayed by these functions.

The *flags* argument provides some control over the operation of the *getInt()* and *getLong()* functions. By default, these functions expect strings containing signed decimal integers. By ORing (|) one or more of the GN_* constants defined in Listing 3-5 into *flags*, we can select alternative bases for conversion and restrict the range of the number to being nonnegative or greater than 0.

The implementations of the *getInt()* and *getLong()* functions are provided in Listing 3-6.

> Although the *flags* argument allows us to enforce the range checks described in the main text, in some cases, we don't request such checks in our example programs, even though it might seem logical to do so. For example, in Listing 47-1, we don't check the *init-value* argument. This means that the user could specify a negative number as the initial value for a semaphore, which would result in an error (ERANGE) in the subsequent *semctl()* system call, because a semaphore can't have a negative value. Omitting range checks in such cases allows us to experiment not just with the correct use of system calls and library functions, but also to see what happens when invalid arguments are supplied. Real-world applications would usually impose stronger checks on their command-line arguments.

Listing 3-5: Header file for get_num.c

lib/get_num.h

```
#ifndef GET_NUM_H
#define GET_NUM_H

#define GN_NONNEG       01      /* Value must be >= 0 */
#define GN_GT_0         02      /* Value must be > 0 */

                                /* By default, integers are decimal */
#define GN_ANY_BASE   0100      /* Can use any base - like strtol(3) */
#define GN_BASE_8     0200      /* Value is expressed in octal */
#define GN_BASE_16    0400      /* Value is expressed in hexadecimal */

long getLong(const char *arg, int flags, const char *name);

int getInt(const char *arg, int flags, const char *name);

#endif
```

lib/get_num.h

Listing 3-6: Functions for parsing numeric command-line arguments

lib/get_num.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include "get_num.h"
```

```
static void
gnFail(const char *fname, const char *msg, const char *arg, const char *name)
{
    fprintf(stderr, "%s error", fname);
    if (name != NULL)
        fprintf(stderr, " (in %s)", name);
    fprintf(stderr, ": %s\n", msg);
    if (arg != NULL && *arg != '\0')
        fprintf(stderr, "        offending text: %s\n", arg);

    exit(EXIT_FAILURE);
}

static long
getNum(const char *fname, const char *arg, int flags, const char *name)
{
    long res;
    char *endptr;
    int base;

    if (arg == NULL || *arg == '\0')
        gnFail(fname, "null or empty string", arg, name);

    base = (flags & GN_ANY_BASE) ? 0 : (flags & GN_BASE_8) ? 8 :
                        (flags & GN_BASE_16) ? 16 : 10;

    errno = 0;
    res = strtol(arg, &endptr, base);
    if (errno != 0)
        gnFail(fname, "strtol() failed", arg, name);

    if (*endptr != '\0')
        gnFail(fname, "nonnumeric characters", arg, name);

    if ((flags & GN_NONNEG) && res < 0)
        gnFail(fname, "negative value not allowed", arg, name);

    if ((flags & GN_GT_0) && res <= 0)
        gnFail(fname, "value must be > 0", arg, name);

    return res;
}

long
getLong(const char *arg, int flags, const char *name)
{
    return getNum("getLong", arg, flags, name);
}

int
getInt(const char *arg, int flags, const char *name)
{
    long res;

    res = getNum("getInt", arg, flags, name);
```

```
    if (res > INT_MAX || res < INT_MIN)
        gnFail("getInt", "integer out of range", arg, name);

    return (int) res;
}
```
lib/get_num.c

3.6 Portability Issues

In this section, we consider the topic of writing portable system programs. We introduce feature test macros and the standard system data types defined by SUSv3, and then look at some other portability issues.

3.6.1 Feature Test Macros

Various standards govern the behavior of the system call and library function APIs (see Section 1.3). Some of these standards are defined by standards bodies such The Open Group (Single UNIX Specification), while others are defined by the two historically important UNIX implementations: BSD and System V Release 4 (and the associated System V Interface Definition).

Sometimes, when writing a portable application, we may want the various header files to expose only the definitions (constants, function prototypes, and so on) that follow a particular standard. To do this, we define one or more of the *feature test macros* listed below when compiling a program. One way that we can do this is by defining the macro in the program source code before including any header files:

```
#define _BSD_SOURCE 1
```

Alternatively, we can use the *–D* option to the C compiler:

```
$ cc -D_BSD_SOURCE prog.c
```

> The term *feature test macro* may seem confusing, but it makes sense if we look at things from the perspective of the implementation. The implementation decides which of the *features* available in each header it should make visible, by *testing* (with `#if`) which values the application has defined for these *macros*.

The following feature test macros are specified by the relevant standards, and consequently their usage is portable to all systems that support these standards:

`_POSIX_SOURCE`
: If defined (with any value), expose definitions conforming to POSIX.1-1990 and ISO C (1990). This macro is superseded by `_POSIX_C_SOURCE`.

`_POSIX_C_SOURCE`
: If defined with the value 1, this has the same effect as `_POSIX_SOURCE`. If defined with a value greater than or equal to 199309, also expose definitions for POSIX.1b (realtime). If defined with a value greater than or equal to 199506, also expose definitions for POSIX.1c (threads). If defined with the value 200112, also expose definitions for the POSIX.1-2001 base specification (i.e., the XSI extension is excluded). (Prior to version 2.3.3, the

glibc headers don't interpret the value 200112 for _POSIX_C_SOURCE.) If defined with the value 200809, also expose definitions for the POSIX.1-2008 base specification. (Prior to version 2.10, the *glibc* headers don't interpret the value 200809 for _POSIX_C_SOURCE.)

_XOPEN_SOURCE
: If defined (with any value), expose POSIX.1, POSIX.2, and X/Open (XPG4) definitions. If defined with the value 500 or greater, also expose SUSv2 (UNIX 98 and XPG5) extensions. Setting to 600 or greater additionally exposes SUSv3 XSI (UNIX 03) extensions and C99 extensions. (Prior to version 2.2, the *glibc* headers don't interpret the value 600 for _XOPEN_SOURCE.) Setting to 700 or greater also exposes SUSv4 XSI extensions. (Prior to version 2.10, the *glibc* headers don't interpret the value 700 for _XOPEN_SOURCE.) The values 500, 600, and 700 for _XOPEN_SOURCE were chosen because SUSv2, SUSv3, and SUSv4 are Issues 5, 6, and 7, respectively, of the X/Open specifications.

The following feature test macros listed are *glibc*-specific:

_BSD_SOURCE
: If defined (with any value), expose BSD definitions. Defining this macro also defines _POSIX_C_SOURCE with the value 199506. Explicitly setting just this macro causes BSD definitions to be favored in a few cases where standards conflict.

_SVID_SOURCE
: If defined (with any value), expose System V Interface Definition (SVID) definitions.

_GNU_SOURCE
: If defined (with any value), expose all of the definitions provided by setting all of the preceding macros, as well as various GNU extensions.

When the GNU C compiler is invoked without special options, _POSIX_SOURCE, _POSIX_C_SOURCE=200809 (200112 with *glibc* versions 2.5 to 2.9, or 199506 with *glibc* versions earlier than 2.4), _BSD_SOURCE, and _SVID_SOURCE are defined by default.

If individual macros are defined, or the compiler is invoked in one of its standard modes (e.g., *cc −ansi* or *cc −std=c99*), then only the requested definitions are supplied. There is one exception: if _POSIX_C_SOURCE is not otherwise defined, and the compiler is not invoked in one of its standard modes, then _POSIX_C_SOURCE is defined with the value 200809 (200112 with *glibc* versions 2.4 to 2.9, or 199506 with *glibc* versions earlier than 2.4).

Defining multiple macros is additive, so that we could, for example, use the following *cc* command to explicitly select the same macro settings as are provided by default:

```
$ cc -D_POSIX_SOURCE -D_POSIX_C_SOURCE=199506 \
                -D_BSD_SOURCE -D_SVID_SOURCE prog.c
```

The <features.h> header file and the *feature_test_macros(7)* manual page provide further information on precisely what values are assigned to each of the feature test macros.

_POSIX_C_SOURCE, _XOPEN_SOURCE, and POSIX.1/SUS

Only the _POSIX_C_SOURCE and _XOPEN_SOURCE feature test macros are specified in POSIX.1-2001/SUSv3, which requires that these macros be defined with the values 200112 and 600, respectively, in conforming applications. Defining _POSIX_C_SOURCE as 200112 provides conformance to the POSIX.1-2001 base specification (i.e., *POSIX conformance*, excluding the XSI extension). Defining _XOPEN_SOURCE as 600 provides conformance to SUSv3 (i.e., *XSI conformance*, the base specification plus the XSI extension). Analogous statements apply for POSIX.1-2008/SUSv4, which require that the two macros be defined with the values 200809 and 700.

SUSv3 specifies that setting _XOPEN_SOURCE to 600 should supply all of the features that are enabled if _POSIX_C_SOURCE is set to 200112. Thus, an application needs to define only _XOPEN_SOURCE for SUSv3 (i.e., XSI) conformance. SUSv4 makes an analogous specification that setting _XOPEN_SOURCE to 700 should supply all of the features that are enabled if _POSIX_C_SOURCE is set to 200809.

Feature test macros in function prototypes and source code examples

The manual pages describe which feature test macro(s) must be defined in order to make a particular constant definition or function declaration visible from a header file.

All of the source code examples in this book are written so that they will compile using either the default GNU C compiler options or the following options:

```
$ cc -std=c99 -D_XOPEN_SOURCE=600
```

The prototype of each function shown in this book indicates any feature test macro(s) that must be defined in order to employ that function in a program compiled with either the default compiler options or the options in the *cc* just shown. The manual pages provide more precise descriptions of the feature test macro(s) required to expose the declaration of each function.

3.6.2 System Data Types

Various implementation data types are represented using standard C types, for example, process IDs, user IDs, and file offsets. Although it would be possible to use the C fundamental types such as *int* and *long* to declare variables storing such information, this reduces portability across UNIX systems, for the following reasons:

- The sizes of these fundamental types vary across UNIX implementations (e.g., a *long* may be 4 bytes on one system and 8 bytes on another), or sometimes even in different compilation environments on the same implementation. Furthermore, different implementations may use different types to represent the same information. For example, a process ID might be an *int* on one system but a *long* on another.
- Even on a single UNIX implementation, the types used to represent information may differ between releases of the implementation. Notable examples on Linux are user and group IDs. On Linux 2.2 and earlier, these values were represented in 16 bits. On Linux 2.4 and later, they are 32-bit values.

To avoid such portability problems, SUSv3 specifies various standard system data types, and requires an implementation to define and use these types appropriately.

Each of these types is defined using the C `typedef` feature. For example, the *pid_t* data type is intended for representing process IDs, and on Linux/x86-32 this type is defined as follows:

```
typedef int pid_t;
```

Most of the standard system data types have names ending in *_t*. Many of them are declared in the header file `<sys/types.h>`, although a few are defined in other header files.

An application should employ these type definitions to portably declare the variables it uses. For example, the following declaration would allow an application to correctly represent process IDs on any SUSv3-conformant system:

```
pid_t mypid;
```

Table 3-1 lists some of the system data types we'll encounter in this book. For certain types in this table, SUSv3 requires that the type be implemented as an *arithmetic type*. This means that the implementation may choose the underlying type as either an integer or a floating-point (real or complex) type.

Table 3-1: Selected system data types

Data type	SUSv3 type requirement	Description
blkcnt_t	signed integer	File block count (Section 15.1)
blksize_t	signed integer	File block size (Section 15.1)
cc_t	unsigned integer	Terminal special character (Section 62.4)
clock_t	integer or real-floating	System time in clock ticks (Section 10.7)
clockid_t	an arithmetic type	Clock identifier for POSIX.1b clock and timer functions (Section 23.6)
comp_t	not in SUSv3	Compressed clock ticks (Section 28.1)
dev_t	an arithmetic type	Device number, consisting of major and minor numbers (Section 15.1)
DIR	no type requirement	Directory stream (Section 18.8)
fd_set	structure type	File descriptor set for *select()* (Section 63.2.1)
fsblkcnt_t	unsigned integer	File-system block count (Section 14.11)
fsfilcnt_t	unsigned integer	File count (Section 14.11)
gid_t	integer	Numeric group identifier (Section 8.3)
id_t	integer	A generic type for holding identifiers; large enough to hold at least *pid_t*, *uid_t*, and *gid_t*
in_addr_t	32-bit unsigned integer	IPv4 address (Section 59.4)
in_port_t	16-bit unsigned integer	IP port number (Section 59.4)
ino_t	unsigned integer	File i-node number (Section 15.1)
key_t	an arithmetic type	System V IPC key (Section 45.2)
mode_t	integer	File permissions and type (Section 15.1)
mqd_t	no type requirement, but shall not be an array type	POSIX message queue descriptor
msglen_t	unsigned integer	Number of bytes allowed in System V message queue (Section 46.4)

Table 3-1: Selected system data types (continued)

Data type	SUSv3 type requirement	Description
msgqnum_t	unsigned integer	Counts of messages in System V message queue (Section 46.4)
nfds_t	unsigned integer	Number of file descriptors for *poll()* (Section 63.2.2)
nlink_t	integer	Count of (hard) links to a file (Section 15.1)
off_t	signed integer	File offset or size (Sections 4.7 and 15.1)
pid_t	signed integer	Process ID, process group ID, or session ID (Sections 6.2, 34.2, and 34.3)
ptrdiff_t	signed integer	Difference between two pointer values, as a signed integer
rlim_t	unsigned integer	Resource limit (Section 36.2)
sa_family_t	unsigned integer	Socket address family (Section 56.4)
shmatt_t	unsigned integer	Count of attached processes for a System V shared memory segment (Section 48.8)
sig_atomic_t	integer	Data type that can be atomically accessed (Section 21.1.3)
siginfo_t	structure type	Information about the origin of a signal (Section 21.4)
sigset_t	integer or structure type	Signal set (Section 20.9)
size_t	unsigned integer	Size of an object in bytes
socklen_t	integer type of at least 32 bits	Size of a socket address structure in bytes (Section 56.3)
speed_t	unsigned integer	Terminal line speed (Section 62.7)
ssize_t	signed integer	Byte count or (negative) error indication
stack_t	structure type	Description of an alternate signal stack (Section 21.3)
suseconds_t	signed integer allowing range [−1, 1000000]	Microsecond time interval (Section 10.1)
tcflag_t	unsigned integer	Terminal mode flag bit mask (Section 62.2)
time_t	integer or real-floating	Calendar time in seconds since the Epoch (Section 10.1)
timer_t	an arithmetic type	Timer identifier for POSIX.1b interval timer functions (Section 23.6)
uid_t	integer	Numeric user identifier (Section 8.1)

When discussing the data types in Table 3-1 in later chapters, we'll often make statements that some type "is an integer type [specified by SUSv3]." This means that SUSv3 requires the type to be defined as an integer, but doesn't require that a particular native integer type (e.g., *short*, *int*, or *long*) be used. (Often, we won't say which particular native data type is actually used to represent each of the system data types in Linux, because a portable application should be written so that it doesn't care which data type is used.)

Printing system data type values

When printing values of one of the numeric system data types shown in Table 3-1 (e.g., *pid_t* and *uid_t*), we must be careful not to include a representation dependency in the *printf()* call. A representation dependency can occur because C's argument promotion rules convert values of type *short* to *int*, but leave values of type *int* and *long* unchanged. This means that, depending on the definition of the system data type, either an *int* or a *long* is passed in the *printf()* call. However, because *printf()* has no way to determine the types of its arguments at run time, the caller must explicitly provide this information using the `%d` or `%ld` format specifier. The problem is that simply coding one of these specifiers within the *printf()* call creates an implementation dependency. The usual solution is to use the `%ld` specifier and always cast the corresponding value to *long*, like so:

```
pid_t mypid;

mypid = getpid();           /* Returns process ID of calling process */
printf("My PID is %ld\n", (long) mypid);
```

We make one exception to the above technique. Because the *off_t* data type is the size of *long long* in some compilation environments, we cast *off_t* values to this type and use the `%lld` specifier, as described in Section 5.10.

> The C99 standard defines the `z` length modifier for *printf()*, to indicate that the following integer conversion corresponds to a *size_t* or *ssize_t* type. Thus, we could write `%zd` instead of using `%ld` plus a cast for these types. Although this specifier is available in *glibc*, we avoid it because it is not available on all UNIX implementations.
>
> The C99 standard also defines the `j` length modifier, which specifies that the corresponding argument is of type *intmax_t* (or *uintmax_t*), an integer type that is guaranteed to be large enough to be able to represent an integer of any type. Ultimately, the use of an *(intmax_t)* cast plus the `%jd` specifier should replace the *(long)* cast plus the `%ld` specifier as the best way of printing numeric system data type values, since the former approach also handles *long long* values and any extended integer types such as *int128_t*. However, (again) we avoid this technique since it is not possible on all UNIX implementations.

3.6.3 Miscellaneous Portability Issues

In this section, we consider a few other portability issues that we may encounter when writing system programs.

Initializing and using structures

Each UNIX implementation specifies a range of standard structures that are used in various system calls and library functions. As an example, consider the *sembuf* structure, which is used to represent a semaphore operation to be performed by the *semop()* system call:

```
struct sembuf {
    unsigned short sem_num;         /* Semaphore number */
    short          sem_op;          /* Operation to be performed */
    short          sem_flg;         /* Operation flags */
};
```

Although SUSv3 specifies structures such as *sembuf*, it is important to realize the following:

- In general, the order of field definitions within such structures is not specified.
- In some cases, extra implementation-specific fields may be included in such structures.

Consequently, it is not portable to use a structure initializer such as the following:

```
struct sembuf s = { 3, -1, SEM_UNDO };
```

Although this initializer will work on Linux, it won't work on another implementation where the fields in the *sembuf* structure are defined in a different order. To portably initialize such structures, we must use explicit assignment statements, as in the following:

```
struct sembuf s;

s.sem_num = 3;
s.sem_op  = -1;
s.sem_flg = SEM_UNDO;
```

If we are using C99, then we can employ that language's new syntax for structure initializers to write an equivalent initialization:

```
struct sembuf s = { .sem_num = 3, .sem_op = -1, .sem_flg = SEM_UNDO };
```

Considerations about the order of the members of standard structures also apply if we want to write the contents of a standard structure to a file. To do this portably, we can't simply do a binary write of the structure. Instead, the structure fields must be written individually (probably in text form) in a specified order.

Using macros that may not be present on all implementations

In some cases, a macro may be not be defined on all UNIX implementations. For example, the `WCOREDUMP()` macro (which checks whether a child process produced a core dump file) is widely available, but it is not specified in SUSv3. Therefore, this macro might not be present on some UNIX implementations. In order to portably handle such possibilities, we can use the C preprocessor `#ifdef` directive, as in the following example:

```
#ifdef WCOREDUMP
    /* Use WCOREDUMP() macro */
#endif
```

Variation in required header files across implementations

In some cases, the header files required to prototype various system calls and library functions vary across UNIX implementations. In this book, we show the requirements on Linux and note any variations from SUSv3.

Some of the function synopses in this book show a particular header file with the accompanying comment */* For portability */*. This indicates that the header file

is not required on Linux or by SUSv3, but because some other (especially older) implementations may require it, we should include it in portable programs.

> For many of the functions that it specified, POSIX.1-1990 required that the header <sys/types.h> be included before any other headers associated with the function. However, this requirement was redundant, because most contemporary UNIX implementations did not require applications to include this header for these functions. Consequently, SUSv1 removed this requirement. Nevertheless, when writing portable programs, it is wise to make this one of the first header files included. (However, we omit this header from our example programs because it is not required on Linux and omitting it allows us to make the example programs one line shorter.)

3.7 Summary

System calls allow processes to request services from the kernel. Even the simplest system calls have a significant overhead by comparison with a user-space function call, since the system must temporarily switch to kernel mode to execute the system call, and the kernel must verify system call arguments and transfer data between user memory and kernel memory.

The standard C library provides a multitude of library functions that perform a wide range of tasks. Some library functions employ system calls to do their work; others perform tasks entirely within user space. On Linux, the usual standard C library implementation that is used is *glibc*.

Most system calls and library functions return a status indicating whether a call has succeeded or failed. Such status returns should always be checked.

We introduced a number of functions that we have implemented for use in the example programs in this book. The tasks performed by these functions include diagnosing errors and parsing command-line arguments.

We discussed various guidelines and techniques that can help us write portable system programs that run on any standards-conformant system.

When compiling an application, we can define various feature test macros that control the definitions exposed by header files. This is useful if we want to ensure that a program conforms to some formal or implementation-defined standard(s).

We can improve the portability of system programs by using the system data types defined in various standards, rather than native C types. SUSv3 specifies a wide range of system data types that an implementation should support and that an application should employ.

3.8 Exercise

3-1. When using the Linux-specific *reboot()* system call to reboot the system, the second argument, *magic2*, must be specified as one of a set of magic numbers (e.g., `LINUX_REBOOT_MAGIC2`). What is the significance of these numbers? (Converting them to hexadecimal provides a clue.)