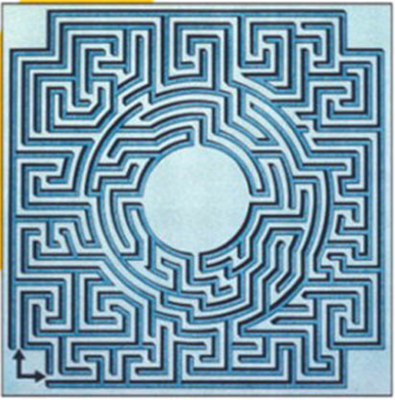
▸Updated Classic!

Advanced UNIX® Programming

SECOND EDITION



MARC J. ROCHKIND

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Advanced UNIX Programming

Advanced UNIX Programming

Second Edition

Marc J. Rochkind

Addison-Wesley
Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

ISBN 0-13-141154-3

Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.

6th Printing September 2008

For Claire and Gillian

Contents

Preface

This book updates the 1985 edition of *Advanced UNIX Programming* to cover a few changes that have occurred in the last eighteen years. Well, maybe "few" isn't the right word! And "updates" isn't right either. Indeed, aside from a sentence here and there, this book is all new. The first edition included about 70 system calls; this one includes about 300. And none of the UNIX standards and implementations discussed in this book—POSIX, Solaris, Linux, FreeBSD, and Darwin (Mac OS X)—were even around in 1985. A few sentences from the 1985 Preface, however, are among those that I can leave almost unchanged:

> The subject of this book is UNIX system calls—the interface between the UNIX kernel and the user programs that run on top of it. Those who interact only with commands, like the shell, text editors, and other application programs, may have little need to know much about system calls, but a thorough knowledge of them is essential for UNIX programmers. System calls are the *only* way to access kernel facilities such as the file system, the multitasking mechanisms, and the interprocess communication primitives.
>
> System calls define what UNIX is. Everything else—subroutines and commands—is built on this foundation. While the novelty of many of these higher-level programs has been responsible for much of UNIX's renown, they could as well have been programmed on any modern operating system. When one describes UNIX as elegant, simple, efficient, reliable, and portable, one is referring not to the commands (some of which are none of these things), but to the kernel.

That's all still true, except that, regrettably, the programming interface to the kernel is no longer elegant or simple. In fact, because UNIX development has splintered into many directions over the last couple of decades, and because the principal standards organization, The Open Group, sweeps up almost everything that's out there (1108 functions altogether), the interface is clumsy, inconsistent, redundant, error-prone, and confusing. But it's still efficient, reliably implemented, and portable, and that's why UNIX and UNIX-like systems are so successful. Indeed, the UNIX system-call interface is the only widely implemented portable one we have and are likely to have in our lifetime.

To sort things out, it's not enough to have complete documentation, just as the Yellow Pages isn't enough to find a good restaurant or hotel. You need a guide that tells you what's good and bad, not just what exists. That's the purpose of this book, and why it's different from most other UNIX programming books. I tell you not only how to use the system calls, but also which ones to stay away from because they're unnecessary, obsolete, improperly implemented, or just plain poorly designed.

Here's how I decided what to include in this book: I started with the 1108 functions defined in Version 3 of the Single UNIX Specification and eliminated about 590 Standard C and other library functions that aren't at the kernel-interface level, about 90 POSIX Threads functions (keeping a dozen of the most important), about 25 accounting and logging functions, about 50 tracing functions, about 15 obscure and obsolete functions, and about 40 functions for scheduling and other things that didn't seem to be generally useful. That left exactly 307 for this book. (See Appendix D for a list.) Not that the 307 are all good—some of them are useless, or even dangerous. But those 307 are the ones you need to know.

This book doesn't cover kernel implementation (other than some basics), writing device drivers, C programming (except indirectly), UNIX commands (shell, vi, emacs, etc.), or system administration.

There are nine chapters: Fundamental Concepts, Basic File I/O, Advanced File I/O, Terminal I/O, Processes and Threads, Basic Interprocess Communication, Advanced Interprocess Communication, Networking and Sockets, and Signals and Timers. Read all of Chapter 1, but then feel free to skip around. There are lots of cross-references to keep you from getting lost.

Like the first edition, this new book includes thousands of lines of example code, most of which are from realistic, if simplified, applications such as a shell, a full-screen menu system, a Web server, and a real-time output recorder. The examples are all in C, but I've provided interfaces in Appendices B and C so you can program in C++, Java, or Jython (a variant of Python) if you like.

The text and example code are just resources; you really learn UNIX programming by doing it. To give you something to do, I've included exercises at the end of each chapter. They range in difficulty from questions that can be answered in a few sentences to simple programming problems to semester-long projects.

I used four UNIX systems for nuts-and-bolts research and to test the examples: Solaris 8, SuSE Linux 8 (2.4 kernel), FreeBSD 4.6, and Darwin (the Mac OS X kernel) 6.8. I kept the source on the FreeBSD system, mounted on the others with NFS or Samba.[1]

I edited the code with TextPad on a Windows system and accessed the four test systems with Telnet or SSH (PuTTY) or with the X Window System (XFree86 and Cygwin). Having the text editor and the four Telnet/SSH/Xterm windows open on the same screen turned out to be incredibly convenient, because it takes only a few minutes to write some code and check it out on the four systems. In addition, I usually had one browser window open to the Single UNIX Specification and one to Google, and another window running Microsoft Word for editing the book. With the exception of Word, which is terrible for big documents like books (crashes, mixed-up styles, weak cross-referencing, flakey document-assembly), all of these tools worked great.[2] I used Perl and Python for various things like extracting code samples and maintaining the database of system calls.

All of the example code (free open source), errata, and more is on the book Web site at *www.basepath.com/aup.*

I'd like to thank those who reviewed the drafts or who otherwise provided technical assistance: Tom Cargill, Geoff Clare, Andrew Gierth, Andrew Josey, Brian Kernighan, Barry Margolin, Craig Partridge, and David Schwartz. And, special thanks to one dedicated, meticulous reviewer who asked to remain anonymous. Naturally, none of these folks is to be blamed for any mistakes you find—I get full credit for those.

I'd also like to thank my editor, Mary Franz, who suggested this project a year or so ago. Luckily, she caught me at a time when I was looking into Linux in depth and started to get excited about UNIX all over again. Reminds me a bit of 1972….

I hope you enjoy the book! If you find anything wrong, or if you port the code to any new systems, or if you just want to share your thoughts, please email me at aup@basepath.com.

Marc J. Rochkind
Boulder, Colorado
April, 2004

1. The four systems are running on various old PCs I've collected over the years and on a Mac I bought on eBay for $200. I had no particular reason for using SuSE Linux and have since switched that machine over to RedHat 9.

2. I could have used any of the systems as my base. Windows turned out to be convenient because my big LCD monitor is attached to that system and because I like TextPad (*www.textpad.com*). Information on PuTTY is at *www.chiark.greenend.org.uk/~sgtatham/putty/.* (Google "PuTTY" if that link doesn't work.)

This page intentionally left blank

1 Fundamental Concepts

1.1 A Whirlwind Tour of UNIX and Linux

This section takes you on a quick tour of the facilities provided by the UNIX and Linux kernels. I won't deal with the user programs (commands) that normally come with UNIX, such as `ls`, `vi`, and `grep`. A discussion of these is well outside the scope of this book. And I won't say much about the internals of the kernel (such as how the file system is implemented) either. (From now on, whenever I say UNIX, I mean Linux, too, unless I say otherwise.)

This tour is meant to be a refresher. I'll use terms such as *process* before defining them, because I assume you already know roughly what they mean. If too much sounds new to you, you may want to become more familiar with UNIX before proceeding. (If you don't know what a process is, you definitely need to get more familiar!) There are lots of introductory UNIX books to start with. Two good ones are *The UNIX Programming Environment* [Ker1984] and *UNIX for the Impatient* [Abr1996] (Chapter 2 is a great introduction).[1]

1.1.1 Files

There are several kinds of UNIX files: regular files, directories, symbolic links, special files, named pipes (FIFOs), and sockets. I'll introduce the first four here and the last two in Section 1.1.7.

1.1.1.1 Regular Files

Regular files contain bytes of data, organized into a linear array. Any byte or sequence of bytes may be read or written. Reads and writes start at a byte loca-

1. You'll find the References at the end of the book.

tion specified by the *file offset,* which can be set to any value (even beyond the end of the file). Regular files are stored on disk.

It isn't possible to insert bytes into the middle of a file (spreading the file apart), or to delete bytes from the middle (closing the gap). As bytes are written onto the end of a file, it gets bigger, one byte at a time. A file can be shrunk or enlarged to any length, discarding bytes or adding bytes of zeroes.

Two or more processes can read and write the same file concurrently. The results depend on the order in which the individual I/O requests occur and are in general unpredictable. To maintain order, there are file-locking facilities and *semaphores*, which are system-wide flags that processes can test and set (more on them in Section 1.1.7).

Files don't have names; they have numbers called *i-numbers.* An i-number is an index into an array of *i-nodes,* kept at the front of each region of disk that contains a UNIX file system. Each i-node contains important information about one file. Interestingly, this information doesn't include either the name or the data bytes. It does include the following: type of file (regular, directory, socket, etc.); number of links (to be explained shortly); owner's user and group ID; three sets of access permissions—for the owner, the group, and others; size in bytes; time of last access, last modification, and status change (when the i-node itself was last modified); and, of course, pointers to disk blocks containing the file's contents.

1.1.1.2 Directories and Symbolic Links

Since it's inconvenient to refer to files by i-number, *directories* are provided to allow names to be used. In practice, a directory is almost always used to access a file.

Each directory consists, conceptually, of a two-column table, with a name in one column and its corresponding i-number in the other column. A name/i-node pair is called a *link.* When the UNIX kernel is told to access a file by name, it automatically looks in a directory to find the i-number. Then it gets the corresponding i-node, which contains more information about the file (such as who can access it). If the data itself is to be accessed, the i-node tells where to find it on the disk.

Directories, which are almost like regular files, occupy an i-node and have data. Therefore, the i-node corresponding to a particular name in a directory could be the i-node of another directory. This allows users to arrange their files into the hierarchical structure that's familiar to users of UNIX. A *path* such as `memo/july/smith` instructs the kernel to get the i-node of the *current directory* to locate its data bytes, find `memo` among those data bytes, take the corresponding i-number, get that i-node to locate the `memo` directory's data bytes, find `july` among those, take the corresponding i-number, get the i-node to locate the `july` directory's data bytes, find `smith`, and, finally, take the corresponding i-node, the one associated with `memo/july/smith`.

In following a *relative path* (one that starts with the current directory), how does the kernel know where to start? It simply keeps track of the i-number of the current directory for each process. When a process changes its current directory, it must supply a path to the new directory. That path leads to an i-number, which then is saved as the i-number of the new current directory.

An *absolute path* begins with a / and starts with the *root* directory. The kernel simply reserves an i-number (2, say) for the root directory. This is established when a file system is first constructed. There is a system call to change a process's root directory (to an i-number other than *2*).

Because the two-column structure of directories is used directly by the kernel (a rare case of the kernel caring about the contents of files), and because an invalid directory could easily destroy an entire UNIX system, a program (even if run by the superuser) cannot write a directory as if it were a regular file. Instead, a program manipulates a directory by using a special set of system calls. After all, the only legal writing actions are to add or remove a link.

It is possible for two or more links, in the same or different directories, to refer to the same i-number. This means that the same file may have more than one name. There is no ambiguity when accessing a file by a given path, since only one i-number will be found. It might have been found via another path also, but that's irrelevant. When a link is removed from a directory, however, it isn't immediately clear whether the i-node and the associated data bytes can be thrown away too. That is why the i-node contains a link count. Removing a link to an i-node merely decrements the link count; when the count reaches zero, the kernel discards the file.

There is no structural reason why there can't be multiple links to directories as well as to regular files. However, this complicates the programming of commands that scan the entire file system, so most kernels outlaw it.

Multiple links to a file using i-numbers work only if the links are in the same file system, as i-numbers are unique only within a file system. To get around this, there are also *symbolic links*, which put the path of the file to be linked to in the data part of an actual file. This is more overhead than just making a second directory link somewhere, but it's more general. You don't read and write these symbolic-link files, but instead use special system calls just for symbolic links.

1.1.1.3 Special Files

A *special file* is typically some type of *device* (such as a CD-ROM drive or communications link).[2]

There are two principal kinds of device special files: block and character. *Block special files* follow a particular model: The device contains an array of fixed-size blocks (say, 4096 bytes each), and a pool of kernel *buffers* are used as a cache to speed up I/O. *Character special files* don't have to follow any rules at all. They might do I/O in very small chunks (characters) or very big chunks (disk tracks), and so they're too irregular to use the buffer cache.

The same physical device could have both block and character special files, and, in fact, this is usually true for disks. Regular files and directories are accessed by the file-system code in the kernel via a block special file, to gain the benefits of the buffer cache. Sometimes, primarily in high-performance applications, more direct access is needed. For instance, a database manager can bypass the file system entirely and use a character special file to access the disk (but not the same area that's being used by the file system). Most UNIX systems have a character special file for this purpose that can directly transfer data between a process's address space and the disk using *direct memory access* (*DMA*), which can result in orders-of-magnitude better performance. More robust error detection is another benefit, since the indirectness of the buffer cache tends to make error detection difficult to implement.

A special file has an i-node, but there aren't any data bytes on disk for the i-node to point to. Instead, that part of the i-node contains a *device number*. This is an index into a table used by the kernel to find a collection of subroutines called a *device driver*.

When a system call is executed to perform an operation on a special file, the appropriate device driver subroutine is invoked. What happens then is entirely up to the designer of the device driver; since the driver runs in the kernel, and not as

2. Sometimes named pipes are considered special files, too, but we'll consider them a category of their own.

a user process, it can access—and perhaps modify—any part of the kernel, any user process, and any registers or memory of the computer itself. It is relatively easy to add new device drivers to the kernel, so this provides a hook with which to do many things besides merely interfacing to new kinds of I/O devices. It's the most popular way to get UNIX to do something its designers never intended it to do. Think of it as the approved way to do something wild.

1.1.2 Programs, Processes, and Threads

A *program* is a collection of *instructions* and *data* that is kept in a regular file on disk. In its i-node the file is marked executable, and the file's contents are arranged according to rules established by the kernel. (Another case of the kernel caring about the contents of a file.)

Programmers can create executable files any way they choose. As long as the contents obey the rules and the file is marked executable, the program can be run. In practice, it usually goes like this: First, the source program, in some programming language (C or C++, say), is typed into a regular file, often referred to as a *text file,* because it's arranged into text lines. Next, another regular file, called an *object file,* is created that contains the machine-language translation of the source program. This job is done by a compiler or assembler (which are themselves programs). If this object file is complete (no missing subroutines), it is marked executable and may be run as is. If not, the *linker* (sometimes called a "loader" in UNIX jargon) is used to bind this object file with others previously created, possibly taken from collections of object files called *libraries.* Unless the linker couldn't find something it was looking for, its output is complete and executable.[3]

In order to run a program, the kernel is first asked to create a new *process,* which is an environment in which a *program* executes. A process consists of three segments: *instruction segment,*[4] *user data segment,* and *system data segment.* The program is used to initialize the instructions and user data. After this initialization, the process begins to deviate from the program it is running. Although modern programmers don't normally modify instructions, the data does get modi-

3. This isn't how interpretive languages like Java, Perl, Python, and shell scripts work. For them, the executable is an interpreter, and the program, even if compiled into some intermediate code, is just data for the interpreter and isn't something the UNIX kernel ever sees or cares about. The kernel's customer is the interpreter.

4. In UNIX jargon, the instruction segment is called the "text segment," but I'll avoid that confusing term.

fied. In addition, the process may acquire resources (more memory, open files, etc.) not present in the program.

While the process is running, the kernel keeps track of its *threads,* each of which is a separate flow of control through the instructions, all potentially reading and writing the same parts of the process's data. (Each thread has its own stack, however.) When you're programming, you start with one thread, and that's all you get unless you execute a special system call to create another. So, beginners can think of a process as being single-threaded.[5]

Several concurrently running processes can be initialized from the same program. There is no functional relationship, however, between these processes. The kernel might be able to save memory by arranging for such processes to share instruction segments, but the processes involved can't detect such sharing. By contrast, there is a strong functional relationship between threads in the same process.

A process's *system data* includes attributes such as current directory, open file descriptors, accumulated CPU time, and so on. A process can't access or modify its system data directly, since it is outside of its address space. Instead, there are various system calls to access or modify attributes.

A process is created by the kernel on behalf of a currently executing process, which becomes the *parent* of the new *child* process. The child inherits most of the parent's system-data attributes. For example, if the parent has any files open, the child will have them open too. Heredity of this sort is absolutely fundamental to the operation of UNIX, as we shall see throughout this book. This is different from a thread creating a new thread. Threads in the same process are equal in most respects, and there's no inheritance. All threads have equal access to all data and resources, not copies of them.

1.1.3 Signals

The kernel can send a *signal* to a process. A signal can be originated by the kernel itself, sent from a process to itself, sent from another process, or sent on behalf of the user.

5. Not every version of UNIX supports multiple threads. They're part of an optional feature called POSIX Threads, or "pthreads," and were introduced in the mid-1990s. More on POSIX in Section 1.5 and threads in Chapter 5.

An example of a kernel-originated signal is a segmentation-violation signal, sent when a process attempts to access memory outside of its address space. An example of a signal sent by a process to itself is an abort signal, sent by the `abort` function to terminate a process with a core dump. An example of a signal sent from one process to another is a termination signal, sent when one of several related processes decides to terminate the whole family. Finally, an example of a user-originated signal is an interrupt signal, sent to all processes created by the user when he or she types Ctrl-c.

There are about 28 types of signals (some versions of UNIX have a few more or a few less). For all but 2, the kill and stop signals, a process can control what happens when it receives the signal. It can accept the default action, which usually results in termination of the process; it can ignore the signal; or it can catch the signal and execute a function, called a signal handler, when the signal arrives. The signal type (`SIGALRM`, say) is passed as an argument to the handler. There isn't any direct way for the handler to determine who sent the signal, however.[6] When the signal handler returns, the process resumes executing at the point of interruption. Two signals are left unused by the kernel. These may be used by an application for its own purposes.

1.1.4 Process-IDs, Process Groups, and Sessions

Every process has a *process-ID,* which is a positive integer. At any instant, these are guaranteed to be unique. Every process but one has a parent.

A process's system data also records its *parent-process-ID,* the process-ID of its parent. If a process is orphaned because its parent terminated before it did, its parent-process-ID is changed to 1 (or some other fixed number). This is the process-ID of the initialization process (init), created at boot time, which is the ancestor of all other processes. In other words, the initialization process adopts all orphans.

Sometimes programmers choose to implement a subsystem as a group of related processes instead of as a single process. The UNIX kernel allows these related processes to be organized into a *process group*. Process groups are further organized into *sessions*.

6. That's true of the basic 28 signals. The Realtime Signals option adds some signals for which this information, and more, is available. More in Chapter 9.

One of the session members is the *session leader,* and one member of each process group is the *process-group leader*. For each process, UNIX keeps track of the process IDs of its process-group leader and session leader so that a process can locate itself in the hierarchy.

The kernel provides a system call to send a signal to each member of a process group. Typically, this would be used to terminate the entire group, but any signal can be broadcast in this way.

UNIX shells generally create one session per login. They usually form a separate process group for the processes in a pipeline; in this context a process group is also called a *job*. But that's just the way shells like to do things; the kernel allows more flexibility so that any number of sessions can be formed within a single login, and each session can have its process groups formed any way it likes, as long as the hierarchy is maintained.

It works this way: Any process can resign from its session and then become a leader of its own session (of one process group containing only one process). It can then create child processes to round out the new process group. Additional process groups within the session can be formed and processes assigned to the various groups. (The assignments can be changed, too.) Hence, a single user could be running, say, a session of 10 processes formed into, say, three process groups.

A session, and thus the processes in it, can have a *controlling terminal,* which is the first terminal device opened by the session leader; the session leader then becomes the *controlling process*. Normally, the controlling terminal for a user's processes is the terminal from which the user logged in. When a new session is formed, the processes in the new session no longer have a controlling terminal, until one is opened. Not having a controlling terminal is typical of so-called *daemons*[7] (Web servers, the `cron` facility, etc.) which, once started, run divorced from whatever terminal was used to start them.

It's possible to organize things so that only one process group in a session—the foreground job—has access to the terminal and to move process groups back and forth between foreground and background. This is called *job control*. A background process group that attempts to access the terminal is suspended until

7. "Demon" and "daemon" are two spellings of the same word, but the former connotes an evil spirit, whereas the latter connotes a supernatural being somewhere between god and man. Misbehaving daemons may properly be referred to as demons.

moved to the foreground. This prevents it from stealing input from the foreground job or scribbling on the foreground process group's output.

If job control isn't being used (generally, only shells use it), all the processes in a session are effectively foreground processes and are equally free to access the terminal, even if havoc ensues.

The terminal device driver sends interrupt, quit, and hangup signals coming from a terminal to every process in the foreground process group for which that terminal is the controlling terminal. For example, unless precautions are taken, hanging up a terminal (e.g., closing a `telnet` connection) will terminate all of the user's processes in that group. To prevent this, a process can arrange to ignore hang-ups.

Additionally, when a session leader (controlling process) terminates for *any* reason, all processes in the foreground process group are sent a hangup signal. So simply logging out usually terminates all of a user's processes unless specific steps have been taken to either make them background processes or otherwise make them immune to the hangup signal.

In summary, there are four process-IDs associated with each process:

- process-ID: Positive integer that uniquely identifies this process.
- parent-process-ID: Process-ID of this process's parent.
- process-group-ID: Process-ID of the process-group leader. If equal to the process-ID, this process is the group leader.
- session-leader-ID: Process-ID of the session leader.

1.1.5 Permissions

A *user-ID* is a positive integer that is associated with a user's *login name* in the *password file* (/etc/passwd). When a user logs in, the `login` command makes this ID the user-ID of the first process created, the login shell. Processes descended from the shell inherit this user-ID.

Users are also organized into *groups* (not to be confused with process groups), which have IDs too, called *group-IDs*. A user's login group-ID is made the group-ID of his or her login shell.

Groups are defined in the *group file* (/etc/group). While logged in, a user can change to another group of which he or she is a member. This changes the group-ID of the process that handles the request (normally the shell, via the `newgrp` command), which then is inherited by all descendent processes. As a user may be a member of more than one group, a process also has a list of *supplementary group-IDs*. For most purposes these are also checked when a process's group permissions are at issue, so a user doesn't have to be constantly switching groups manually.

These two user and group login IDs are called the *real user-ID* and the *real group-ID* because they are representative of the real user, the person who is logged in. Two other IDs are also associated with each process: the *effective user-ID* and the *effective group-ID.* These IDs are normally the same as the corresponding real IDs, but they can be different, as we shall see shortly.

The effective ID is always used to determine permissions. The real ID is used for accounting and user-to-user communication. One indicates the user's permissions; the other indicates the user's identity.

Each file (regular, directory, socket, etc.) has, in its i-node, an *owner user-ID* (*owner* for short) and an *owner group-ID* (*group* for short). The i-node also contains three sets of three permission bits (nine bits in all). Each set has one bit for *read permission,* one bit for *write permission,* and one bit for *execute permission.* A bit is 1 if the permission is granted and 0 if not. There is a set for the owner, for the group, and for others (not in either of the first two categories). Table 1.1 shows the bit assignments (bit 0 is the rightmost bit).

Table 1.1 Permission Bits

Bit	Meaning
8	owner read
7	owner write
6	owner execute
5	group read
4	group write
3	group execute
2	others read
1	others write
0	others execute

Permission bits are frequently specified using an octal (base 8) number. For example, octal 775 would mean read, write, and execute permission for the owner and the group, and only read and execute permission for others. The `ls` command would show this combination of permissions as `rwxrwxr-x`; in binary it would be 111111101, which translates directly to octal 775. (And in decimal it would be 509, which is useless, because the numbers don't line up with the permission bits.)

The permission system determines whether a given process can perform a desired action (read, write, or execute) on a given file. For regular files, the meaning of the actions is obvious. For directories, the meaning of read is obvious. "Write" permission on a directory means the ability to issue a system call that would modify the directory (add or remove a link). "Execute" permission means the ability to use the directory in a path (sometimes called "search" permission). For special files, read and write permissions mean the ability to execute the read and write system calls. What, if anything, that implies is up to the designer of the device driver. Execute permission on a special file is meaningless.

The permission system determines whether permission will be granted using this algorithm:

1. If the effective user-ID is zero, permission is instantly granted (the effective user is the *superuser*).
2. If the process's effective user-ID matches the file's user-ID, then the owner set of bits is used to see if the action will be allowed.
3. If the process's effective group-ID or one of the supplementary group-IDs matches the file's group-ID, then the group set of bits is used.
4. If neither the user-IDs nor group-IDs match, then the process is an "other" and the third set of bits is used.

The steps go in order, so if in step 3 access is denied because, say, write permission is denied for the group, then the process cannot write, even though the "other" permission (step 4) might allow writing. It might be unusual for the group to be more restricted than others, but that's the way it works. (Imagine an invitation to a surprise party for a team of employees. Everyone *except* the team should be able to read it.)

There are other actions, which might be called "change i-node," that only the owner or the superuser can do. These include changing the user-ID or group-ID of a file, changing a file's permissions, and changing a file's access or modification times. As a special case, write permission on a file allows setting of its access and modification times to the current time.

Occasionally, we want a user to temporarily take on the privileges of another user. For example, when we execute the `passwd` command to change our password, we would like the effective user-ID to be that of the superuser, because only root can write into the password file. This is done by making root (the superuser's login name) the owner of the `passwd` command (i.e., the regular file containing the `passwd` program) and then turning on another permission bit in the `passwd` command's i-node, called the *set-user-ID* bit. Executing a program with this bit on changes the effective user-ID of the process to the owner of the file containing the program. Since it's the effective user-ID, rather than the real user-ID, that determines permissions, this allows a user to temporarily take on the permissions of someone else. The *set-group-ID* bit is used in a similar way.

Since both user-IDs (real and effective) are inherited from parent process to child process, it is possible to use the set-user-ID feature to run with an effective user-ID for a very long time. That's what the `su` command does.

There is a potential loophole. Suppose you do the following: Copy the `sh` command to your own directory (you will be the owner of the copy). Then use `chmod` to turn on the set-user-ID bit and `chown` to change the file's owner to root. Now execute your copy of `sh` and take on the privileges of root! Fortunately, this loophole was closed a long time ago. If you're not the superuser, changing a file's owner automatically clears the set-user-ID and set-group-ID bits.

1.1.6 Other Process Attributes

A few other interesting attributes are recorded in a process's system data segment.

There is one open *file descriptor* (an integer from 0 through 1000 or so) for each file (regular, special, socket, or named pipe) that the process has opened, and two for each unnamed pipe (see Section 1.1.7) that the process has created. A child doesn't inherit open file descriptors from its parent, but rather duplicates of them. Nonetheless, they are indices into the same system-wide open file table, which among other things means that parent and child share the same file offset (the byte where the next read or write takes place).

A process's *priority* is used by the kernel scheduler. Any process can lower its priority via the system call `nice`; a superuser process can raise its priority (i.e., be not nice) via the same system call. Technically speaking, `nice` sets an attribute called the *nice value,* which is only one factor in computing the actual priority.

A process's *file size limit* can be (and usually is) less than the system-wide limit; this is to prevent confused or uncivilized users from writing runaway files. A superuser process can raise its limit.

There are many more process attributes that are discussed throughout this book as the relevant system calls are described.

1.1.7 Interprocess Communication

In the oldest (pre-1980) UNIX systems, processes could communicate with one another via shared file offsets, signals, *process tracing,* files, and *pipes*. Then *named pipes* (FIFOs) were added, followed by *semaphores, file locks, messages, shared memory,* and networking *sockets*. As we shall see throughout this book, none of these eleven mechanisms is entirely satisfactory. That's why there are eleven! There are even more because there are two versions each of semaphores, messages, and shared memory. There's one very old collection from AT&T's System V UNIX, called "System V IPC," and one much newer, from a real-time standards group that convened in the early 1990s, called "POSIX IPC." Almost every version of UNIX (including FreeBSD and Linux) has the oldest eleven mechanisms, and the main commercial versions of UNIX (e.g., Sun's Solaris or HP's HP/UX) have all fourteen.

Shared file offsets are rarely used for interprocess communication. In theory, one process could position the file offset to some fictitious location in a file, and a second process could then find out where it points. The location (a number between, say, 0 and 100) would be the communicated data. Since the processes must be related to share a file offset, they might as well just use pipes.

Signals are sometimes used when a process just needs to poke another. For example, a print spooler could signal the actual printing process whenever a print file is spooled. But signals don't pass enough information to be helpful in most applications. Also, a signal interrupts the receiving process, making the programming more complex than if the receiver could get the communication when it was ready. Signals are mainly used just to terminate processes or to indicate very unusual events.

With *process tracing,* a parent can control the execution of its child. Since the parent can read and write the child's data, the two can communicate freely. Process tracing is used only by debuggers, since it is far too complicated and unsafe for general use.

Files are the most common form of interprocess communication. For example, one might write a file with a process running `vi` and then execute it with a process running `python`. However, files are inconvenient if the two processes are running concurrently, for two reasons: First, the reader might outrace the writer, see an end-of-file, and think that the communication is over. (This can be handled through some tricky programming.) Second, the longer the two processes communicate, the bigger the file gets. Sometimes processes communicate for days or weeks, passing billions of bytes of data. This would quickly exhaust the file system.

Using an empty file for a semaphore is also a traditional UNIX technique. This takes advantage of some peculiarities in the way UNIX creates files. More details are given in Section 2.4.3.

Finally, something I can actually recommend: *Pipes* solve the synchronization problems of files. A pipe is not a type of regular file; although it has an i-node, there are no links to it. Reading and writing a pipe is somewhat like reading and writing a file, but with some significant differences: If the reader gets ahead of the writer, the reader *blocks* (stops running for a while) until there's more data. If the writer gets too far ahead of the reader, it blocks until the reader has a chance to catch up, so the kernel doesn't have too much data queued. Finally, once a byte is read, it is gone forever, so long-running processes connected via pipes don't fill up the file system.

Pipes are well known to shell users, who can enter command lines like

```
ls | wc
```

to see how many files they have. The kernel facility, however, is far more general than what the shell provides, as we shall see in Chapter 6.

Pipes, however, have three major disadvantages: First, the processes communicating over a pipe must be related, typically parent and child or two siblings. This is too constraining for many applications, such as when one process is a database manager and the other is an application that needs to access the database. The second disadvantage is that writes of more than a locally set maximum (4096 bytes, say) are not guaranteed to be atomic, prohibiting the use of pipes when there are multiple writers—their data might get intermingled. The third disadvantage is that pipes might be too slow. The data has to be copied from the writing user process to the kernel and back again to the reader. No actual disk I/O is performed, but the copying alone can take too long for some critical applications. It's because of these disadvantages that fancier schemes have evolved.

Named pipes, also called *FIFOs,* were added to solve the first disadvantage of pipes. (FIFO stands for "first-in-first-out.") A named pipe exists as a special file, and any process with permission can open it for reading or writing. Named pipes are easy to program with, too, as we shall demonstrate in Chapter 7.

What's wrong with named pipes? They don't eliminate the second and third disadvantage of pipes: Interleaving can occur for big writes, and they are sometimes too slow. For the most critical applications, the newer interprocess communication features (e.g., messages or shared memory) can be used, but not nearly as easily.

A *semaphore* (in the computer business) is a counter that prevents two or more processes from accessing the same resource at the same time. As I've mentioned, files can be used for semaphores too, but the overhead is far too great for many applications. UNIX has two completely different semaphore mechanisms, one part of System V IPC, and the other part of POSIX IPC. (We'll get to POSIX in Section 1.5.)

A *file lock*, in effect a special-purpose semaphore, prevents two or more processes from accessing the same segment of a file. It's usually only effective if the processes bother to check; that weak form is called *advisory* locking. The stronger form, *mandatory* locking, is effective whether a process checks or not, but it's nonstandard and available only in some UNIX systems.

A *message* is a small amount of data (500 bytes, say) that can be sent to a *message queue.* Messages can be of different types. Any process with appropriate permissions can receive messages from a queue. It has lots of choices: either the first message, or the first message of a given type, or the first message of a group of types. As with semaphores, there are System V IPC messaging system calls, and completely different POSIX IPC system calls.

Shared memory potentially provides the fastest interprocess communication of all. The same memory is mapped into the address spaces of two or more processes.

As soon as data is written to the shared memory, it is instantly available to the readers. A semaphore or a message is used to synchronize the reader and writer. Sure enough, there are two versions of shared memory, and attentive readers can guess what they are.

I've saved the best for last: networking interprocess communication, using a group of system calls called *sockets*. (Other system calls in this group have names like `bind`, `connect`, and `accept`.) Unlike all of the other mechanisms I've talked about, the process you're communicating with via a socket doesn't have to be on the same machine. It can be on another machine, or anywhere on a local network or the Internet. That other machine doesn't even have to be running UNIX. It can be running Windows or whatever. It could be a networked printer or radio, for that matter.

Choosing the IPC mechanism for a particular application isn't easy, so I'll spend a lot of time comparing them in Chapters 7 and 8.

1.2 Versions of UNIX

Ken Thompson and Dennis Ritchie began UNIX as a research project at AT&T Bell Laboratories in 1969, and shortly thereafter it started to be widely used within AT&T for internal systems (e.g., to automate telephone-repair call centers). AT&T wasn't then in the business of producing commercial computers or of selling commercial software, but by the early 1970s it did make UNIX available to universities for educational purposes, provided the source code was disclosed only to other universities who had their own license. By the late 1970s AT&T was licensing source code to commercial vendors, too.

Many, if not most, of the licensees made their own changes to UNIX, to port it to new hardware, to add device drivers, or just to tune it. Perhaps the two most significant changes were made at the University of California at Berkeley: networking system-calls ("sockets") and support for virtual memory. As a result, universities and research labs throughout the world used the Berkeley system (called BSD, for Berkeley Software Distribution), even though they still got their licenses from AT&T. Some commercial vendors, notably Sun Microsystems, started with BSD UNIX as well. Bill Joy, a founder of Sun, was a principal BSD developer.

Development within Bell Labs continued as well, and by the mid-1980s the two systems had diverged widely. (AT&T's was by that time called System V.) Both supported virtual memory, but their networking system calls were completely different. System V had interprocess-communication facilities (messages, shared memory, and semaphores) that BSD provided for in different ways, and BSD had changed almost all of the commands and added lots more, notably `vi`.

There were lots of other UNIX variants, too, including a few UNIX clones that used no AT&T-licensed source code. But mostly the UNIX world divided into the BSD side, which included essentially all of academia and some important workstation makers, and the System V side, which included AT&T itself and some commercial vendors. Nearly all computer science students learned on a BSD system. (And when they came to Bell Labs they brought their favorite commands, and all their UNIX knowledge, with them.) Ironically, neither AT&T nor the University of California really wanted to be in the commercial software business! (AT&T said it did, but it was wrong.)

The Institute of Electrical and Electronics Engineers (IEEE) started an effort to standardize both UNIX system calls and commands in the mid-1980s, picking up where an industry group called /usr/group had left off. IEEE issued its first standard for the system calls in 1988. The official name was IEEE Std 1003.1-1988, which I'll call POSIX1988 for short.[8] The first standard for commands (e.g., `vi` and `grep`) came out in 1992 (IEEE Std 1003.2-1992). POSIX1988 was adopted as an international standard by the International Organization for Standardization (ISO) with minor changes, and became POSIX1990.

Standards only standardize the syntax and semantics of application program interfaces (APIs), not the underlying implementation. So, regardless of its origins or internal architecture, any system with enough functionality can conform to POSIX1990 or its successors, even systems like Microsoft Windows or Digital (now HP) VMS. If the application program conforms to the standard for operating-system APIs and for the language it's written in (e.g., C++), and the target system conforms, then the application's source code can be ported, and it will compile and run with no changes. In practice, applications, compilers, and operating systems have bugs, and probably every serious application has to use some nonstandard APIs, so porting does take some work, but the standards reduce it to a manageable level, way more manageable than it was before the standards came along.

8. POSIX stands for Portable Operating System Interface and is pronounced *pahz-icks*.

POSIX1990 was a tremendous achievement, but it wasn't enough to unify the System V and BSD camps because it didn't include important APIs, such as BSD sockets and System V messages, semaphores, and shared memory. Also, even newer APIs were being introduced for new applications of UNIX, principally real-time (applications that interface to the real world, such as engine-management for a car) and threads. Fortunately, IEEE had standards groups working on real-time and threads, too. (Section 1.5 has more on where standards went after POSIX1990.)

In 1987 AT&T and Sun, which used a BSD-derived UNIX, thought they could unify the two main versions of UNIX by working together to produce a single system, but this scared the rest of the industry into forming the Open Software Foundation, which ultimately produced its own version of UNIX (still containing code licensed from AT&T, however). So, in the early 1990s we had BSD, AT&T/Sun UNIX, the vendors' pre-OSF UNIX versions, and an OSF version.

It looked like UNIX might implode, with Microsoft Windows taking over the world, and this prompted the industry to get behind the X/Open Company, which was much more aggressive than IEEE about getting the important APIs into a standard. The newest standard defines 1108 function interfaces!

By the mid-1990s everything was actually starting to settle down a bit, and then Linux came along. It started quietly in 1991 with a Usenet newsgroup post titled "Free minix-like[9] kernel sources for 386-AT" from a grad student named Linus Torvalds, which started like this:

> Do you pine for the nice days of minix-1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on a OS you can try to modify for your needs? Are you finding it frustrating when everything works on minix? No more all-nighters to get a nifty program working? Then this post might be just for you :-)
>
> As I mentioned a month(?) ago, I'm working on a free version of a minix-lookalike for AT-386 computers. It has finally reached the stage where it's even usable (though may not be depending on what you want), and I am willing to put out the sources for wider distribution. It is just version 0.02 (+1 (very small) patch already), but I've successfully run bash/gcc/gnu-make/gnu-sed/compress etc under it.[10]

By the end of the 1990s, Linux had developed into a serious OS, largely through the efforts of hundreds of developers who work on its free and open source code.

9. Minix is a small instructional OS developed by Andrew Tanenbaum of the Free University of Amsterdam, first released in 1987. It used no AT&T code.

10. The original message and the other 700 million Usenet messages since 1981 are archived at Google Groups (*www.google.com/grphp*).

There are commercial vendors of Linux binary distributions for Intel-based PCs (e.g., Red Hat, SuSE, Mandrake) and hardware manufacturers who ship Linux computers (e.g., Dell, IBM, and even Sun). All of them make the source available, as required by the Linux license, which is the complete opposite of the AT&T license!

Meanwhile, although they had been making extensive changes to the kernel for 15 years, the programmers at Berkeley still had a system with AT&T licensed code in it. They began removing the AT&T code but ran out of funding before they could finish. They released an incomplete, but unencumbered, system as 4.4BSD-Lite in the early 1990s. (Had they done it a bit earlier, Torvalds would have started with that, instead of with Minix.) Several groups immediately set to work to flesh out 4.4BSD-Lite, and those efforts are known today as FreeBSD (for Intel CPUs and a few others), NetBSD/OpenBSD (portable), WindRiver's BSD/OS (commercially supported), and Darwin (the UNIX inside Mac OS X). Alas, the Berkeley group is no longer around.

Today, there are three principal strains of UNIX systems:

- Commercial, closed systems historically based on AT&T System V or BSD (Solaris, HP/UX, AIX, etc.)
- BSD-based systems, of which FreeBSD is the best-known, with Darwin coming on strong
- Linux

Which of these are "UNIX"? AT&T's UNIX-development organization, which owned the UNIX trademark, was sold to Novell in 1993. Novell promptly turned the UNIX trademark over to X/Open. In 1996, OSF and X/Open merged to form the Open Group, which today develops a comprehensive standard called the Single UNIX Specification [SUS2002]. "UNIX" is now a brand that can legally be applied to any system that meets the Open Group's requirements. The major hardware vendors all have UNIX-branded systems—even IBM's mainframe MVS (now called OS/390) carries the UNIX brand. But the open-source systems Linux and FreeBSD do not, even though they do claim to conform to one or another of the POSIX standards. It's convenient to refer to all of these systems as "UNIX-like."

1.3 Using System Calls

This section explains how system calls are used with C and other languages and presents some guidelines for using them correctly.

1.3.1 C (and C++) Bindings

How does a C or C++ programmer actually issue a system call? Just like any other function call. For example, `read` might be called like this:

```
amt = read(fd, buf, numbyte);
```

The implementation of the function `read` varies with the UNIX implementation. Usually it's a small function that executes some special code that transfers control from the user process to the kernel and then returns the results. The real work is done inside the kernel, which is why it's a system call and not just a library routine that can do its work entirely in user space, such as `qsort` or `strlen`.

Remember, though, that since a system call involves two context switches (from user to kernel and back), it takes much longer than a simple function call within a process's own address space. So it's a good idea to avoid excessive system calls. This point will be emphasized in Section 2.12 when we look into buffered I/O.

Every system call is defined in a header file, and you have to ensure that you've included the correct headers (sometimes more than one is needed) before you make the call. For instance, `read` requires that we include the header unistd.h, like this:

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

A single header typically defines more than one call (unistd.h defines about 80), so a typical program needs only a handful of includes. Even with a bunch more for Standard C[11] functions (e.g., string.h), it's still manageable. There's no harm in including a header you don't actually need, so it's easiest to collect the most common headers into a master header and just include that. For this book, the master header is defs.h, which I present in Section 1.6. I won't show the include for it in the example code in this book, but you should assume it's included in every C file I write.

Unfortunately, there are header-file conflicts caused by ambiguities in the various standards or by misunderstandings on the part of implementors, so you'll sometimes find that you need to jigger the order of includes or play some tricks with the C preprocessor to make everything come out OK. You can see a bit of this in defs.h, but at least it hides the funny business from the code that includes it.

11. By Standard C, I mean the original 1989 standard, the 1995 update, or the latest, known as C99. When I mean C99 specifically, I'll say so.

No standard requires that a given facility be implemented as a system call, rather than a library function, and I won't make much of the distinction between the two when I introduce one or when I show one being used in an example program. Don't take my use of the term "system call" too literally, either; I just mean that the facility is usually implemented that way.

1.3.2 Other Language Bindings

While the main UNIX standards documents, POSIX and SUS, describe the interfaces in C, they recognize that C is only one of many languages, and so they've defined bindings for other standardized languages as well, such as Fortran and Ada. No problem with easy functions like `read`, as long as the language has some way of passing in integers and addresses of buffers and getting an integer back, which most do (Fortran was a challenge). It's tougher to handle structures (used by `stat`, for instance) and linked lists (used by `getaddrinfo`) . Still, language experts usually find a way.[12]

Lots of languages, like Java, Perl, and Python, have standard facilities that support some POSIX facilities. For instance, here's a Python program that uses the UNIX system called `fork` to create a child process. (I'll get to `fork` in Chapter 5; for now all you need to know is that it creates a child process and then returns in both the child *and* the parent, with different values, so both the `if` and `else` are true, in the parent and child, respectively.)

```
import os
pid = os.fork()
if pid == 0:
    print 'Parent says, "HELLO!"'
else:
    print 'Child says, "hello!"'
```

This is the output:

```
Parent says, "HELLO!"
Child says, "hello!"
```

There won't be any further examples using languages other than C and C++ in this book other than in Appendix C, which talks about Java and Jython (a version of Python that runs in a Java environment).

12. For example, Jtux, described in Appendix C, makes extensive use of the Java Native Interface (JNI).

1.3.3 Guidelines for Calling Library Functions

Here are some general guidelines for calling library functions (from C or C++), which apply to system calls or to any other functions:

- Include the necessary **headers** (as I discussed previously).
- Make sure you know how the function indicates an **error**, and check for it unless you have a good reason not to (Section 1.4). If you're not going to check, document your decision by casting the return value to `void`, like this:

  ```
  (void)close(fd);
  ```

  We violate this guideline consistently for `printf`, but with few other exceptions I follow it in our example programs.

- Don't use **casts** unless absolutely necessary, because they might hide a mistake. Here's what you shouldn't do:

  ```
  int n;
  struct xyz *p;
  ...
  free((void *)p); /* gratuitous cast */
  ```

  The problem is that if you mistakenly write `n` instead of `p`, the cast will suppress a compiler warning. You don't need the cast when the function's prototype specifies `void *`. If you're not sure whether the type you want to use is OK, leave the cast off so the compiler can tell you. This would apply to the case when the function calls, say, for an `int` and you want to give it a `pid_t` (the type for a process ID). Better to get the warning if there will be one than to shut the compiler up before it has a chance to speak. If you do get a warning, and you've determined that a cast is the fix, you can put the cast in then.
- If you've got more than one **thread**, know whether the function you want to call is thread safe—that is, whether it handles global variables, mutexes (semaphores), signals, etc., correctly for a multithreaded program. Many functions don't.
- Try to write to the **standard interfaces** rather than to those on your particular system. This has a lot of advantages: Your code will be more portable, the standardized functionality is probably more thoroughly tested, it will be easier for another programmer to understand what you're doing, and your code is more likely to be compatible with the next version of the operating system. In practice, what you can do is keep a browser open to the Open

Group Web site (Section 1.9 and [SUS2002]), where there's a terrific alphabetized reference to all standard functions (including the Standard C library) and all header files. This is a much better approach than, say, using the `man` pages on your local UNIX system. Sure, sometimes you do need to look up particulars for your system, and occasionally you need to use something nonstandard. But make that the exception rather than the rule, and document that it's nonstandard with a comment. (Lots more about what I mean by "standard" in Section 1.5.)

1.3.4 Function Synopses

I'll formally introduce each system call or function that I discuss in detail with a brief synopsis of it that documents the necessary headers, the arguments, and how errors are reported. Here's one for `atexit`, a Standard C function, that we're going to need in Section 1.4.2:

atexit—register function to be called when process exits

```
#include <stdlib.h>

int atexit(
    void (*fcn)(void)      /* function to be called */
);
/* Returns 0 on success, non-zero on error (errno not defined) */
```

(If you're not already familiar with `errno`, it's explained in the next section.)

The way `atexit` works is that you declare a function like this:

```
static void fcn(void)
{
    /* work to be done at exit goes here */
}
```

and then you register it like this:

```
if (atexit(fcn) != 0) {
    /* handle error */
}
```

Now when the process exits, your function is automatically called. You can register more than 1 function (up to 32), and they're all called in the reverse order of their registration. Note that the test in the `if` statement was for non-zero, since that's exactly what the wording in the synopsis said. It wasn't a test for 1, or greater than zero, or –1, or `false`, or `NULL`, or anything else. That's the only safe way to do it. Also, it makes it easier later if you're looking for a bug and are trying to compare the code to the documentation. You don't have to solve a little puzzle in your head to compare the two.

1.4 Error Handling

Testing error returns from system calls is tricky, and handling an error once you discover it is even tricker. This section explains the problem and gives some practical solutions.

1.4.1 Checking for Errors

Most system calls return a value. In the `read` example (Section 1.3.1), the number of bytes read is returned. To indicate an error, a system call usually returns a value that can't be mistaken for valid data, typically –1. Therefore, my example should have been coded something like this:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed!\n");
    exit(EXIT_FAILURE);
}
```

Note that `exit` is a system call too, but it can't return an error because it doesn't return. The symbol `EXIT_FAILURE` is in Standard C.

Of the system calls covered in this book, about 60% return –1 on an error, about 20% return something else, such as `NULL`, zero, or a special symbol like `SIG_ERR`, and about 20% don't report errors at all. So, you just can't assume that they all behave the same way—you have to read the documentation for each call. I'll provide the information when I introduce each system call.

There are lots of reasons why a system call that returns an error indication might have failed. For 80% of them, the integer symbol `errno` contains a code that indicates the reason. To get at `errno` you include the header errno.h. You can use `errno` like an integer, although it's not necessarily an integer variable. If you're using threads, `errno` is likely to involve a function call, because the different threads can't reliably all use the same global variable. So don't declare `errno` yourself (which used to be exactly what you were supposed to do), but use the definition in the header, like this (other headers not shown):

```
#include <errno.h>

if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

If, say, the file descriptor is bad, the output would be:

```
Read failed! errno = 9
```

Almost always, you can use the value of `errno` only if you've first checked for an error; you can't just check `errno` to see if an error occurred, because its value is set only when a function that is specified to set it returns an error. So, this code would be wrong:

```
amt = read(fd, buf, numbyte);
if (errno != 0) { /* wrong! */
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

Setting `errno` to zero first works:

```
errno = 0;
amt = read(fd, buf, numbyte);
if (errno != 0) { /* bad! */
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

But it's still a bad idea because:

- If you modify the code later to insert another system call, or any function that eventually executes a system call, before the call to `read`, the value of `errno` may be set by *that* call.
- Not all system calls set the value of `errno`, and you should get into the habit of checking for an error that conforms exactly to the function's specification.

Thus, for almost all system calls, check `errno` only after you've established that an error occurred.

Now, having warned you about using `errno` alone to check for an error, this being UNIX, I have to say that there are a few exceptions (e.g., `sysconf` and `readdir`) that do rely on a changed `errno` value to indicate an error, but even they return a specific value that tells you to check `errno`. Therefore, the rule about not checking `errno` before first checking the return value is a good one, and it applies to most of the exceptions, too.

The `errno` value 9 that was displayed a few paragraphs up doesn't mean much, and isn't standardized, so it's better to use the Standard C function `perror`, like this:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    perror("Read failed!");
    exit(EXIT_FAILURE);
}
```

Now the output is:

```
Read failed!: Bad file number
```

Another useful Standard C function, `strerror`, just provides the message as a string, without also displaying it like `perror` does.

But the message "Bad file number," while clear enough, isn't standardized either, so there's still a problem: The official documentation for system calls and other functions that use `errno` refer to the various errors with symbols like `EBADF`, not by the text messages. For example, here's an excerpt from the SUS entry for `read`:

> [EAGAIN]
>
> The O_NONBLOCK flag is set for the file descriptor and the process would be delayed.
>
> [EBADF]
>
> The fildes argument is not a valid file descriptor open for reading.
>
> [EBADMSG]
>
> The file is a STREAM file that is set to control-normal mode and the message waiting to be read includes a control part.

It's straightforward to match "Bad file number" to `EBADF`, even though those exact words don't appear in the text, but not for the more obscure errors. What you really want along with the text message is the actual symbol, and there's no Standard C or SUS function that gives it to you. So, we can write our own function that translates the number to the symbol. We built the list of symbols in the code that follows from the errno.h files on Linux, Solaris, and BSD, since many symbols are system specific. You'll probably have to adjust this code for your own system. For brevity, not all the symbols are shown, but the complete code is on the AUP Web site (Section 1.8 and [AUP2003]).

```
static struct {
    int code;
    char *str;
} errcodes[] =
{
    { EPERM, "EPERM" },
    { ENOENT, "ENOENT" },
      …
    { EINPROGRESS, "EINPROGRESS" },
    { ESTALE, "ESTALE" },
#ifndef BSD
    { ECHRNG, "ECHRNG" },
    { EL2NSYNC, "EL2NSYNC" },
      …
    { ESTRPIPE, "ESTRPIPE" },
    { EDQUOT, "EDQUOT" },
#ifndef SOLARIS
    { EDOTDOT, "EDOTDOT" },
    { EUCLEAN, "EUCLEAN" },
      …
    { ENOMEDIUM, "ENOMEDIUM" },
    { EMEDIUMTYPE, "EMEDIUMTYPE" },
#endif
#endif
    { 0, NULL}
};

const char *errsymbol(int errno_arg)
{
    int i;

    for (i = 0; errcodes[i].str != NULL; i++)
        if (errcodes[i].code == errno_arg)
            return errcodes[i].str;
    return "[UnknownSymbol]";
}
```

Here's the error checking for `read` with the new function:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed!: %s (errno = %d; %s)\n",
      strerror(errno), errno, errsymbol(errno));
    exit(EXIT_FAILURE);
}
```

Now the output is complete:

```
Read failed!: Bad file descriptor (errno = 9; EBADF)
```

It's convenient to write one more utility function to format the error information, so we can use it in some code we're going to write in the next section:

```
char *syserrmsg(char *buf, size_t buf_max, const char *msg, int errno_arg)
{
    char *errmsg;

    if (msg == NULL)
        msg = "???";
    if (errno_arg == 0)
        snprintf(buf, buf_max, "%s", msg);
    else {
        errmsg = strerror(errno_arg);
        snprintf(buf, buf_max, "%s\n\t\t*** %s (%d: \"%s\") ***", msg,
          errsymbol(errno_arg), errno_arg,
          errmsg != NULL ? errmsg : "no message string");
    }
    return buf;
}
```

We would use `syserrmsg` like this:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
      "Call to read function failed", errno));
    exit(EXIT_FAILURE);
}
```

with output like this:

```
Call to read function failed
   *** EBADF (9: "Bad file descriptor") ***
```

What about the other 20% of the calls that report an error but don't set `errno`? Well, around 20 of them report the error another way, typically by simply returning the error code (that is, a non-zero return indicates that an error occurred and also what the code is), and the rest don't provide a specific reason at all. I'll provide the details for every function (all 300 or so) in this book. Here's one that returns the error code directly (what this code is supposed to do isn't important right now):

```
struct addrinfo *infop;

if ((r = getaddrinfo("localhost", "80", NULL, &infop)) != 0) {
    fprintf(stderr, "Got error code %d from getaddrinfo\n", r);
    exit(EXIT_FAILURE);
}
```

The function `getaddrinfo` is one of those that doesn't set `errno`, and you can't pass the error code it returns into `strerror`, because that function works only with `errno` values. The various error codes returned by the non-`errno` functions are defined in [SUS2002] or in your system's manual, and you certainly could write a version of `errsymbol` (shown earlier) for those functions. But what makes this difficult is that the symbols for one function (e.g., `EAI_BADFLAGS` for `getaddrinfo`) aren't specified to have values distinct from the symbols for another function. This means that you can't write a function that takes an error code alone and looks it up, like `errsymbol` did. You have to pass the name of the function in as well. (If you do, you could take advantage of `gai_strerror`, which is a specially tailored version of `strerror` just for `getaddrinfo`.)

There are about two dozen functions in this book for which the standard [SUS2002] doesn't define any `errno` values or even say that `errno` is set, but for which your implementation may set `errno`. The phrase "errno not defined" appears in the function synopses for these.

Starting to get a headache? UNIX error handling is a real mess. This is unfortunate because it's hard to construct test cases to make system calls misbehave so the error handling you've coded can be checked, and the inconsistencies make it hard to get it right every single time. But the chances of it getting any better soon are zero (it's frozen by the standards), so you'll have to live with it. Just be careful!

1.4.2 Error-Checking Convenience Macros for C

It's tedious to put every system call in an `if` statement and follow it with code to display the error message and exit or return. When there's cleanup to do, things get even worse, as in this example:

```
if ((p = malloc(sizeof(buf))) == NULL) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
      "malloc failed", errno));
    return false;
}
if ((fdin = open(filein, O_RDONLY)) == -1) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
      "open (input) failed", errno));
```

```
        free(p);
        return false;
    }
    if ((fdout = open(fileout, O_WRONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "open (output) failed", errno));
        (void)close(fdin);
        free(p);
        return false;
    }
```

The cleanup code gets longer and longer as we go. It's hard to write, hard to read, and hard to maintain. Many programmers will just use a `goto` so the cleanup code can be written just once. Ordinarily, `goto`s are to be avoided, but here they seem worthwhile. Note that we have to carefully initialize the variables that are involved in cleanup and then test the file-descriptor values so that the cleanup code can execute correctly no matter what the incomplete state.

```
    char *p = NULL;
    int fdin = -1, fdout = -1;

    if ((p = malloc(sizeof(buf))) == NULL) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "malloc failed", errno));
        goto cleanup;
    }
    if ((fdin = open(filein, O_RDONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "open (input) failed", errno));
        goto cleanup;
    }
    if ((fdout = open(fileout, O_WRONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "open (output) failed", errno));
        goto cleanup;
    }
    return true;

cleanup:
    free(p);
    if (fdin != -1)
        (void)close(fdin);
    if (fdout != -1)
        (void)close(fdout);
    return false;
```

Still, coding all those `if`s, `fprintf`s, and `goto`s is a pain. The system calls themselves are almost buried!

We can streamline the jobs of checking for the error, displaying the error information, and getting out of the function with some macros. I'll first show how they're used and then how they're implemented. (This stuff is just Standard C coding, not especially connected to system calls or even to UNIX, but I've included it because it'll be used in all the subsequent examples in this book.)

Here's the previous example rewritten to use these error-checking ("ec") macros. The context isn't shown, but this code is inside of a function named `fcn`:

```
    char *p = NULL;
    int fdin = -1, fdout = -1;

    ec_null( p = malloc(sizeof(buf)) )
    ec_neg1( fdin = open(filein, O_RDONLY) )
    ec_neg1( fdout = open(fileout, O_WRONLY) )

    return true;

EC_CLEANUP_BGN
    free(p);
    if (fdin != -1)
        (void)close(fdin);
    if (fdout != -1)
        (void)close(fdout);
    return false;
EC_CLEANUP_END
```

Here's the call to the function. Because it's in the `main` function, it makes sense to exit on an error.

```
    ec_false( fcn() )

    /* other stuff here */

    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
```

Here's what's going on: The macros `ec_null`, `ec_neg1`, and `ec_false` check their argument expression against `NULL`, –1, and `false`, respectively, store away the error information, and go to a label that was placed by the `EC_CLEANUP_BGN` macro. Then the same cleanup code as before is executed. In `main`, the test of the return value of `fcn` also causes a jump to the same label in `main` and the program exits. A function installed with `atexit` (introduced in Section 1.3.4) displays all the accumulated error information:

```
ERROR:  0: main [/aup/c1/errorhandling.c:41] fcn()
        1: fcn [/aup/c1/errorhandling.c:15] fdin = open(filein, 0x0000)
                *** ENOENT (2: "No such file or directory") ***
```

What we see is the `errno` symbol, value, and descriptive text on the last line. It's preceded by a reverse trace of the error returns. Each trace line shows the level, the name of the function, the file name, the line number, and the code that returned the error indication. This isn't the sort of information you want your end users to see, but during development it's terrific. Later, you can change the macros (we'll see how shortly) to put these details in a log file, and your users can see something more meaningful to them.

We accumulated the error information rather than printing it as we went along because that gives an application developer the most freedom to handle errors as he or she sees fit. It really doesn't do for a function in a library to just write error messages to `stderr`. That may not be the right place, and the wording may not be appropriate for the application's end users. In the end we did print it, true, but that decision can be easily changed if you use these macros in a real application.

So, what these macros give us is:

- Easy and readable error checking
- An automatic jump to cleanup code
- Complete error information along with a back trace

The downside is that the macros have a somewhat strange syntax (no semicolon at the end) and a buried jump in control flow, which some programmers think is a very bad idea. If you think the benefits outweigh the costs, use the macros (as I will in this book). If not, work out your own set (maybe with an explicit `goto` instead of a buried one), or skip them entirely.

Here's most of the header file (ec.h) that implements the error-checking macros (function declarations and a few other minor details are omitted):

```
extern const bool ec_in_cleanup;

typedef enum {EC_ERRNO, EC_EAI} EC_ERRTYPE;

#define EC_CLEANUP_BGN\
    ec_warn();\
    ec_cleanup_bgn:\
    {\
        bool ec_in_cleanup;\
        ec_in_cleanup = true;
#define EC_CLEANUP_END\
    }

#define ec_cmp(var, errrtn)\
    {\
        assert(!ec_in_cleanup);\
        if ((intptr_t)(var) == (intptr_t)(errrtn)) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errno, EC_ERRNO);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_rv(var)\
    {\
        int errrtn;\
        assert(!ec_in_cleanup);\
        if ((errrtn = (var)) != 0) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errrtn, EC_ERRNO);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_ai(var)\
    {\
        int errrtn;\
        assert(!ec_in_cleanup);\
        if ((errrtn = (var)) != 0) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errrtn, EC_EAI);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_neg1(x) ec_cmp(x, -1)
#define ec_null(x) ec_cmp(x, NULL)
#define ec_false(x) ec_cmp(x, false)
#define ec_eof(x) ec_cmp(x, EOF)
#define ec_nzero(x)\
    {\
        if ((x) != 0)\
            EC_FAIL\
    }

#define EC_FAIL ec_cmp(0, 0)

#define EC_CLEANUP goto ec_cleanup_bgn;
```

```
#define EC_FLUSH(str)\
    {\
        ec_print();\
        ec_reinit();\
    }
```

Before I explain the macros, I have to bring up a problem and talk about its solution. The problem is that if you call one of the error-checking macros (e.g., `ec_neg1`) inside the cleanup code and an error occurs, there will most likely be an infinite loop, since the macro will jump to the cleanup code! Here's what I'm worried about:

```
EC_CLEANUP_BGN
    free(p);
    if (fdin != -1)
        ec_neg1( close(fdin) )
    if (fdout != -1)
        ec_neg1( close(fdout) )
    return false;
EC_CLEANUP_END
```

It looks like the programmer is being very careful to check the error return from `close`, but it's a disaster in the making. What's really bad about this is that the loop would occur only when there was an error cleaning up after an error, a rare situation that's unlikely to be caught during testing. We want to guard against this—the error-checking macros should increase reliability, not reduce it!

Our solution is to set a local variable `ec_in_cleanup` to `true` in the cleanup code, which you can see in the definition for the macro `EC_CLEANUP_BGN`. The test against it is in the macro `ec_cmp`—if it's set, the `assert` will fire and we'll know right away that we goofed.

(The type `bool` and its values, `true` and `false`, are new in C99. If you don't have them, you can just stick the code

```
typedef int bool;
#define true 1
#define false 0
```

in one of your header files.)

To prevent the assertion from firing when `ec_cmp` is called outside of the cleanup code (i.e., a normal call), we have a global variable, also named `ec_in_cleanup`, that's permanently set to `false`. This is a rare case when it's OK (essential, really) to hide a global variable with a local one.

Why have the local variable at all? Why not just set the global to `true` at the start of the cleanup code, and back to `false` at the end? That won't work if you call a function from within the cleanup code that happens to use the `ec_cmp` macro legitimately. It will find the global set to true and think it's in its own cleanup code, which it isn't. So, each function (that is, each unique cleanup-code section) needs a private guard variable.

Now I'll explain the macros one-by-one:

- `EC_CLEANUP_BGN` includes the label for the cleanup code (`ec_cleanup_bgn`), preceded by a function call that just outputs a warning that control flowed into the label. This guards against the common mistake of forgetting to put a `return` statement before the label and flowing into the cleanup code even when there was no error. (I put this in after I wasted an hour looking for an error that wasn't there.) Then there's the local `ec_in_cleanup`, which I already explained.
- `EC_CLEANUP_END` just supplies the closing brace. We needed the braces to create the local context.
- `ec_cmp` does most of the work: Ensuring we're not in cleanup code, checking the error, calling `ec_push` (which I'll get to shortly) to push the location information (`__FILE__`, etc.) onto a stack, and jumping to the cleanup code. The type `intptr_t` is new in C99: It's an integer type guaranteed to be large enough to hold a pointer. If you don't have it yet, `typedef` it to be a `long` and you'll probably be OK. Just to be extra safe, stick some code in your program somewhere to test that `sizeof(void *)` is equal to `sizeof(long)`. (If you're not familiar with the notation `#var`, read up on your C—it makes whatever `var` expands to into a string.)
- `ec_rv` is similar to `ec_cmp`, but it's for functions that return a non-zero error code to indicate an error and which don't use `errno` itself. However, the codes it returns are `errno` values, so they can be passed directly to `ec_push`.
- `ec_ai` is similar to `ec_rv`, but the error codes it deals with aren't `errno` values. The last argument to `ec_push` becomes `EC_EAI` to indicate this. (Only a couple of functions, both in Chapter 8, use this approach.)
- The macros `ec_neg1`, `ec_null`, `ec_false`, and `ec_eof` call `ec_cmp` with the appropriate arguments, and `ec_nzero` does its own checking. They cover the most common cases, and we can just use `ec_cmp` directly for the others.

- `EC_FAIL` is used when an error condition arises from a test that doesn't use the macros in the previous paragraph.
- `EC_CLEANUP` is used when we just want to jump to the cleanup code.
- `EC_FLUSH` is used when we just want to display the error information, without waiting to exit. It's handy in interactive programs that need to keep going. (The argument isn't used.)

The various service functions called from the macros won't be shown here, since they don't illustrate much about UNIX system calls (they just use Standard C), but you can go to the AUP Web site [AUP2003] to see them along with an explanation of how they work. Here's a summary:

- `ec_push` pushes the error and context information passed to it (by the `ec_cmp` macro, say) onto a stack.
- There's a function registered with `atexit` that prints the information on the stack when the program exits:

  ```
  static void ec_atexit_fcn(void)
  {
      ec_print();
  }
  ```

- `ec_print` walks down the stack to print the trace and error information.
- `ec_reinit` erases what's on the stack, so error-checking can start with a fresh trace.
- `ec_warn` is called from the `EC_CLEANUP_BGN` code if we accidentally fall into it.

All the functions are thread-safe so they can be used from within multithreaded programs. More on what this means in Section 5.17.

1.4.3 Using C++ Exceptions

Before you spend too much time and energy deciding whether you like the "ec" macros in the previous section and coming up with some improvements, you might ask yourself whether you'll even be programming in C. These days it's much more likely that you'll use C++. Just about everything in this book works fine in a C++ program, after all. C is still often preferred for embedded systems, operating systems (e.g., Linux), compilers, and other relatively low-level software, but those kinds of systems are likely to have their own, highly specialized, error-handling mechanisms.

C++ provides an opportunity to handle errors with exceptions, built into the C++ language, rather than with the combination of `goto`s and `return` statements that we used in C. Exceptions have their own pitfalls, but if used carefully they're easier to use and more reliable than the "ec" macros, which don't protect you from, say, using `ec_null` when you meant to use `ec_neg1`.

As the library that contains the system-call wrapper code is usually set up just for C, it won't throw exceptions unless someone's made a special version for C++. So, to use exceptions you need another layer of wrapping, something like this for the `close` system call:

```
class syscall_ex {
public:
    int se_errno;

    syscall_ex(int n)
        : se_errno(n)
        { }
    void print(void)
        {
            fprintf(stderr, "ERROR: %s\n", strerror(se_errno));
        }

};

class syscall {
public:
    static int close(int fd)
        {
            int r;
            if ((r = ::close(fd)) == -1)
                throw(syscall_ex(errno));
            return r;
        }
};
```

Then you just call `syscall::close` instead of plain `close`, and it throws an exception on an error. You probably don't want to type in the code for the other 1100 or so UNIX functions, but perhaps just the ones your application uses.

If you want the exception information to include location information such as file and line, you need to define *another* wrapper, this time a macro, to capture the preprocessor data (e.g., via `__LINE__`),[13] so here's an even fancier couple of classes:

13. We need the macro because if you just put `__LINE__` and the others as direct arguments to the `syscall_ex` constructor, you get the location in the definition of `class syscall`, which is the wrong place.

```
class syscall_ex {
public:
    int se_errno;
    const char *se_file;
    int se_line;
    const char *se_func;

    syscall_ex(int n, const char *file, int line, const char *func)
        : se_errno(n), se_file(file), se_line(line), se_func(func)
        { }
    void print(void)
        {
            fprintf(stderr, "ERROR: %s [%s:%d %s()]\n",
              strerror(se_errno), se_file, se_line, se_func);
        }

};

class syscall {
public:
    static int close(int fd, const char *file, int line, const char *func)
        {
            int r;
            if ((r = ::close(fd)) == -1)
                throw(syscall_ex(errno, file, line, func));
            return r;
        }
};

#define Close(fd) (syscall::close(fd, __FILE__, __LINE__, __func__))
```

This time you call `Close` instead of `close`.

You can goose this up with a call-by-call trace, as we did for the "ec" macros if you like, and probably go even further than that.

There's an example C++ wrapper, Ux, for all the system calls in this book that's described in Appendix B.

1.5 UNIX Standards

Practically speaking, what you'll mostly find in the real world is that the commercial UNIX vendors follow the Open Group branding (Section 1.2), and the open-source distributors claim only conformance to POSIX1990, although, with few exceptions, they don't bother to actually run the certification tests. (The tests have now been made freely available, so future certification is now more likely.)

1.5.1 Evolution of the API Standards

Enumerating all the various POSIX and Open Group standards, guides, and specifications would be enormously complicated and confusing. For most purposes, the relevant developments can be simplified by considering them as a progression of standards relevant to the UNIX API, of which only the eight shown in Table 1.2 are important for the purposes of this book.

Table 1.2 POSIX and Open Group API Standards

Name	Standard	Comments
POSIX1988	IEEE Std 1003.1-1988 (198808L*)	First standard
POSIX1990	IEEE Std 1003.1-1990/ ISO 9945-1:1990 (199009L)	Minor update of POSIX1988
POSIX1993	IEEE Std 1003.1b-1993 (199309L)	POSIX1990 + real-time
POSIX1996	IEEE Std 1003.1-1996/ISO 9945-1:1996 (199506L)	POSIX1993 + threads + fixes to real-time
XPG3	X/Open Portability Guide	First widely distributed X/Open guide
SUS1	Single UNIX Specification, Version 1	POSIX1990 + all commonly used APIs from BSD, AT&T System V, and OSF; also known as Spec 1170; certified systems branded UNIX 95†
SUS2	Single UNIX Specification, Version 2	SUS1 updated to POSIX1996 + 64-bit, large file, enhanced multibyte, and Y2K; UNIX 98 brand
SUS3	Single UNIX Specification, Version 3 (200112L)	Update to SUS2; API part identical to IEEE Std 1003.1-2001 (POSIX fully merged with Open Group); UNIX 03 brand

* These huge numbers are the year and month of approval by IEEE. They're used in feature testing as explained in the next section.
† Spec 1170 gets its name from the total number of APIs, headers, and commands.

Actually, there is also a POSIX2001 (IEEE Std POSIX.1-2001), but as it's contained in SUS3, we don't need a separate row for it.

Even with this over-simplified list, we can already see that the loose statement that an OS "conforms to POSIX" is meaningless and probably indicates that whoever is making it either doesn't really know POSIX (or SUS) or is assuming that the listener doesn't. What POSIX? And, since newer features like real time and threads are optional, what options? Generally, here's how to interpret what you're hearing:

- If all you hear is POSIX, it probably means POSIX1990, which was the first and last OS standard that most people bothered to become aware of.
- Unless you know that the OS has passed certification tests established by standards organizations, all you can be sure of is that the OS developers took the POSIX standard into account somehow.
- You have to investigate the status of optional features separately (more on that shortly).

1.5.2 Telling the System What You Want

In this and the next section I'll describe what you're *supposed* to do according to the standards to ensure that your application conforms and to check what version of the standard the OS conforms to. Then you can decide how much of this you want to do and how much is just mumbo-jumbo that you can skip.

First of all, your application can state what version of the standard it expects by defining a preprocessor symbol before including any POSIX headers, such as unistd.h. This limits the standard headers (e.g., stdio.h, unistd.h, errno.h) to only those symbols that are defined in the standard. To limit yourself to classic POSIX1990, you do this:

```
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>
```

For a newer POSIX standard you have to be more specific, with another symbol, like this:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#include <sys/types.h>
#include <unistd.h>
```

Note that unlike `_POSIX_SOURCE`, `_POSIX_C_SOURCE` has a value which is normally a long integer formed from the year and month when the standard was approved (June 1995 in the example). The other possible long integer is `200112L`. But for POSIX1990, you set it to `1`, not to `199009L`.[14]

If you're interested in going beyond POSIX into SUS, which you will be if you're using a commercial UNIX system, there's a separate symbol for that, `_XOPEN_SOURCE`, and two special numbers for it: 500 for SUS2, and 600 for SUS3. So, if you want to use what's in a SUS2, UNIX 98-branded system, you do this:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#define _XOPEN_SOURCE 500
#include <sys/types.h>
#include <unistd.h>
```

For SUS1, you define `_XOPEN_SOURCE` without a value, and also define `_XOPEN_SOURCE_EXTENDED` as 1:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 2
#define _XOPEN_SOURCE
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

Technically, you don't need to define the first two POSIX symbols if you really have a SUS2 system, since they will be defined automatically. But, the source you're writing may be used on an older system, and you need to tell it that you want POSIX in terms it will understand. `_POSIX_C_SOURCE` should use 2 when `_XOPEN_SOURCE` is defined with no value, `199506L` when it's 500, and `200112L` when it's 600.

Are you lost? Too bad, because it gets much worse. To help a little, here's a summary of what to do:

- Decide what level of standardization your program is written to expect.
- If it's only POSIX1990, define only `_POSIX_SOURCE`.
- If it's POSIX1993 or POSIX1996, define both `_POSIX_SOURCE` and `_POSIX_C_SOURCE`, and use the appropriate number from Table 1.2 in Section 1.5.1.

14. Or you can set it to 2 if you also want the C bindings from 1003.2-1992.

- If it's SUS1, SUS2, or SUS3, define `_XOPEN_SOURCE` (to nothing, to 500, or to 600) and also use the two POSIX symbols with the appropriate numbers.
- For SUS1, also define `_XOPEN_SOURCE_EXTENDED` as 1.
- Forget about XPG3 (and XPG4, which I haven't even mentioned)—you have enough to worry about already.

It's easy to write a header file that sets the various request symbols based on a single definition that you set before including it. Here's the header suvreq.h;[15] I'll show how it's used in the next section:

```
/*
    Header to request specific standard support. Before including it, one
    of the following symbols must be defined (1003.1-1988 isn't supported):

        SUV_POSIX1990    for 1003.1-1990
        SUV_POSIX1993    for 1003.1b-1993 - real-time
        SUV_POSIX1996    for 1003.1-1996
        SUV_SUS1         for Single UNIX Specification, v. 1 (UNIX 95)
        SUV_SUS2         for Single UNIX Specification, v. 2 (UNIX 98)
        SUV_SUS3         for Single UNIX Specification, v. 3 (UNIX 03)
*/
#if defined(SUV_POSIX1990)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 1

#elif defined(SUV_POSIX1993)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L

#elif defined(SUV_POSIX1996)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L

#elif defined(SUV_SUS1)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 2
#define _XOPEN_SOURCE
#define _XOPEN_SOURCE_EXTENDED 1

#elif defined(SUV_SUS2)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#define _XOPEN_SOURCE 500
#define _XOPEN_SOURCE_EXTENDED 1
```

15. It's pronounced "S-U-V wreck." SUV is for Standard UNIX Version. What did you think it was?

```
#elif defined(SUV_SUS3)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 200112L
#define _XOPEN_SOURCE 600
#define _XOPEN_SOURCE_EXTENDED 1
#endif
```

Limiting your program to only what's in a standard is great if you're trying to write a highly portable application, because it helps ensure that you don't accidentally use anything nonstandard. But, for other applications, that's too limiting. For example, FreeBSD claims only to conform to POSIX1990, yet it does implement System V messages, which are at the SUS1 level. If you need messages, or any other post-POSIX1990 stuff that FreeBSD offers, you don't want to define `_POSIX_SOURCE`. So, don't—it's entirely optional. That's what I had to do to get the example code in this book to run on FreeBSD.[16]

Hmmm. We fried your brain explaining these symbols and all the funny numbers and then told you that you probably don't want to use them. Welcome to the world of standards!

1.5.3 Asking the System What It Has

Just because you've asked for a certain level with the `_POSIX_SOURCE`, `_POSIX_C_SOURCE`, and `_XOPEN_SOURCE` symbols, that doesn't mean that that's what you're going to get. If the system is only POSIX1993, that's all it can be. If that's all you need, then fine. If not, you need to refuse to compile (with an `#error` directive), disable an optional feature of your application, or enable an alternative implementation.

The way you find out what the OS has to offer, after you've included unistd.h, is to check the `_POSIX_VERSION` symbol and, if you're on a SUS system, the `_XOPEN_UNIX` and `_XOPEN_VERSION` symbols. The four "SOURCE" symbols in the previous section were you talking to the system, and now the system is talking back to you.

16. Also, most compilers and operating systems have additional symbols to bring in useful nonstandard features, such as `_GNU_SOURCE` for gcc or `__EXTENSIONS__` for Solaris. Check your system's documentation.

Of course, these symbols may not even be defined, although that would be strange for `_POSIX_VERSION`, since you've already successfully included unistd.h. So, you have to test carefully, and the way to do it is with a preprocessor command like this:

```
#if _POSIX_VERSION >= 199009L
    /* we have some level of POSIX
#else
#error "Can be compiled only on a POSIX system!"
#endif
```

If `_POSIX_VERSION` is undefined, it will be replaced by 0.

`_POSIX_VERSION` takes on the numbers (e.g., 199009L) from the table in the previous section. If you have at least a SUS1 system, `_XOPEN_UNIX` will be defined (with no value), but it's better to check against `_XOPEN_VERSION`, which will be equal to 4, 500, or 600, or perhaps someday an even larger number.

Let's write a little program that prints what the headers are telling us after we've requested SUS2 compatibility (the header suvreq.h is in the previous section):

```
#define SUV_SUS2

#include "suvreq.h"
#include <unistd.h>
#include <stdio.h>

int main(void)
{
    printf("Request:\n");
#ifdef _POSIX_SOURCE
    printf("\t_POSIX_SOURCE defined\n");
    printf("\t_POSIX_C_SOURCE = %ld\n", (long)_POSIX_C_SOURCE);
#else
    printf("\t_POSIX_SOURCE undefined\n");
#endif

#ifdef _XOPEN_SOURCE
    #if _XOPEN_SOURCE +0 == 0
        printf("\t_XOPEN_SOURCE defined (no value)\n");
    #else
        printf("\t_XOPEN_SOURCE = %d\n", _XOPEN_SOURCE);
    #endif
#else
    printf("\t_XOPEN_SOURCE undefined\n");
#endif
#ifdef _XOPEN_SOURCE_EXTENDED
    printf("\t_XOPEN_SOURCE_EXTENDED defined\n");
#else
    printf("\t_XOPEN_SOURCE_EXTENDED undefined\n");
#endif

    printf("Claims:\n");
#ifdef _POSIX_VERSION
    printf("\t_POSIX_VERSION = %ld\n", _POSIX_VERSION);
#else
    printf("\tNot POSIX\n");
#endif

#ifdef _XOPEN_UNIX
    printf("\tX/Open\n");
    #ifdef _XOPEN_VERSION
        printf("\t_XOPEN_VERSION = %d\n", _XOPEN_VERSION);
    #else
        printf("\tError: _XOPEN_UNIX defined, but not "
          "_XOPEN_VERSION\n");
    #endif
#else
    printf("\tNot X/Open\n");
#endif
    return 0;
}
```

When we ran this on a Solaris 8 system, this was the output:

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 199506
        _XOPEN_SOURCE = 500
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 500
```

So, Solaris is willing to be SUS2. And, if we change the first line to define `SUV_SUS1` instead, the output is now

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 2
        _XOPEN_SOURCE defined (no value)
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 4
```

which means that Solaris is also set up to behave like a SUS1 system, hiding any features newer than SUS1.

Running the program on FreeBSD 4.6 produced this output:

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 2
        _XOPEN_SOURCE defined (no value)
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 199009
        Not X/Open
```

This means that FreeBSD is claiming only to be POSIX1990, despite what we'd like it to be.

As I said, you probably don't want to make this request of FreeBSD because that just cuts out too much that they've included, some of which probably conforms to standards later than POSIX1990. When we reran our program without the suvreq.h header at all, this is what we got:

```
Request:
        _POSIX_SOURCE undefined
        _XOPEN_SOURCE undefined
        _XOPEN_SOURCE_EXTENDED undefined
Claims:
        _POSIX_VERSION = 199009
        Not X/Open
```

Still claiming to be POSIX1990, of course. Interestingly, when we do the same thing on Solaris, it decides it's going to be XPG3, so we might call that its default level. Why the default isn't SUS2, especially since Solaris is branded as UNIX 98, is a mystery. Here's the Solaris output:

```
Request:
        _POSIX_SOURCE undefined
        _XOPEN_SOURCE undefined
        _XOPEN_SOURCE_EXTENDED undefined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 3
```

On SuSE Linux 8.0 (Linux version 2.4, glibc 2.95.3), when we defined `SUV_SUS2` we got this:

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 199506
        _XOPEN_SOURCE = 500
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 500
```

So Linux is willing to be a SUS2 system. But its default is SUS1, since we got this when we didn't include suvreq.h at all:

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 199506
        _XOPEN_SOURCE undefined
        _XOPEN_SOURCE_EXTENDED undefined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 4
```

Compare this output to the output from Solaris when we didn't include the header—this time Linux went ahead and defined `_POSIX_SOURCE` for us when we included unistd.h. That probably makes some sense, since defining it has always been a POSIX requirement.

Darwin is really modest—it claims only POSIX1988 (with suvreq.h included):

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 199506
        _XOPEN_SOURCE = 500
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 198808
        Not X/Open
```

Anyway, for all the example code in this book we define `SUV_SUS2` before including suvreq.h, except for FreeBSD and Darwin, where we don't make any request at all. Which leads to the question, how do we know we're on FreeBSD or Darwin? (Answer in Section 1.5.8.)

1.5.4 Checking for Options

I said that POSIX1996 (IEEE Std 1003.1-1996) included real-time and threads, but that doesn't mean that systems that conform to it are required to support real-time and threads, because those are optional features. It means that if they don't, they have to say so in a standard way, as I'm about to explain.

It turns out that as complex as the symbology has been already, it's much too coarse to tell us about options. For that, a whole bunch of additional symbols have been defined for option groups (e.g., threads, POSIX messages) and, for some of them, suboptions. I won't explain each one here, but just indicate how to check, in general, whether an option is supported. My explanation is somewhat simplified, as the option-checking rules have evolved from POSIX1990 to SUS1 to SUS3 into something pretty complicated. (See the Web page [Dre2003] for an in-depth discussion of the options, cross-referenced with the affected functions.)

In practice, I've encountered this unpleasant state of affairs:

- The principal commercial systems, such as Solaris, support all the options. They also handle the option-checking symbols correctly, although, since the options are supported, not checking would also work.
- The open-source systems, Linux, FreeBSD, and Darwin, tend not to support many options, so it's important to do the option checking. However, they don't handle the option checking correctly, so the checking code you put in often doesn't work.

Clearly, no amount of work by the standards groups will improve the situation if the OS developers don't follow the current standards.

Anyway, each option has a preprocessor symbol defined in unistd.h that can be tested to see if the option is supported, such as `_POSIX_ASYNCHRONOUS_IO` for the Asynchronous Input and Output option, which includes the system calls `aio_read`, `aio_write`, and a few others (more on them in Chapter 3). If your program uses this option, you first need to test the symbol, like this:

```
#if _POSIX_ASYNCHRONOUS_IO <= 0
    /* substitute code, or message, or compile failure with #error */
#else
    /* code that uses the feature */
#endif
```

What this means is that you can't use those system calls and you can't include the associated header (aio.h) if the symbol is undefined or defined with a value of 0 or less. If the symbol is defined with a value greater than 0, you can assume the feature is always supported.[17]

Some options depend on what file you're using, and there the rules are different. For example, if `_POSIX_ASYNCHRONOUS_IO` is supported, you then have to check the suboption `_POSIX_ASYNC_IO` to see if asynchronous I/O is supported on the file you want to use it on. The rules for suboptions vary a bit from option to option, but in most cases if the option is file related, you first check to see whether it's defined at all. If it is, a value of –1 means it isn't supported for any file, and any other value means it's supported for all files. If it's undefined, you have to call `pathconf` or `fpathconf` to test it (Section 1.5.6).

The file-dependent option rules I just explained were for SUS2. For pre-SUS2—POSIX1993 and POSIX1996—the rules were different: You only had to check the file if the top-level option (`_POSIX_ASYNCHRONOUS_IO` in our example) was undefined; if it was defined with a value other than –1, you could assume support on all files.

Linux is a little confused: It claims to be SUS2, yet it follows the pre-SUS2 rules.

One way to deal with this option mess is by encapsulating the option-checking into a function for each group of optional system calls that you use. For example, here's one to test to see if the asynchronous I/O functions are supported, complete with the adjustment for Linux (Section 1.5.6 explains `pathconf`):

```
typedef enum {OPT_NO = 0, OPT_YES = 1, OPT_ERROR = -1} OPT_RETURN;

OPT_RETURN option_async_io(const char *path)
{
#if _POSIX_ASYNCHRONOUS_IO <= 0
    return OPT_NO;
#elif _XOPEN_VERSION >= 500 && !defined(LINUX)
    #if !defined(_POSIX_ASYNC_IO)
        errno = 0;
        if (pathconf(path, _PC_ASYNC_IO) == -1)
            if (errno == 0)
                return OPT_NO;
            else
                EC_FAIL
        else
            return OPT_YES;
```

17. The test as I've coded it is oversimplified because it doesn't treat the undefined or 0 case as special. For some symbols and some versions of SUS, you get headers and function stubs so you can test at run-time with `sysconf`. But some systems, notably Linux, omit the stubs in the undefined case. So the test as we've shown it fails to account for systems where the symbol is zero, but `sysconf` will report that the feature is supported.

```
    EC_CLEANUP_BGN
        return OPT_ERROR;
    EC_CLEANUP_END
    #elif _POSIX_ASYNC_IO == -1
        return OPT_NO;
    #else
        return OPT_YES;
    #endif /* _POSIX_ASYNC_IO */
#elif _POSIX_VERSION >= 199309L
    return OPT_YES;
#else
    errno = EINVAL;
    return OPT_ERROR;
#endif /* _POSIX_ASYNCHRONOUS_IO */
}
```

Note that you can't use the function to find out whether code that uses an option will compile. For that you have to use the preprocessor, and hence the option symbol (e.g., `_POSIX_ASYNCHRONOUS_IO`) directly. So you still have to do something like this:

```
#if _POSIX_ASYNCHRONOUS_IO <= 0
    /* substitute code, or message, or compile failure with #error */
#else
    /*
        code that calls option_async_io and uses the
        option if it returns OPT_YES
    */
#endif
```

Just to recap in case you got lost: The option `_POSIX_ASYNCHRONOUS_IO` is for testing whether the system supports asynchronous I/O at all; on a SUS2 system, to find out whether it's supported for the particular file we want to use it on, we have to then check `_POSIX_ASYNC_IO`, possibly at run-time with a call to `pathconf` or `fpathconf`.

Don't even try to learn the symbols for all the options—there are too many of them, most of which you'll never use, and there are too many irregularities in how they work. Instead, for most optional system calls that I cover in this book, I'll indicate what symbols have to be checked.[18] But I usually won't show the actual option-checking in my example programs.

18. Don't confuse optional, which is still standardized, with nonstandard. I cover only a handful of nonstandard calls in this book.

1.5.5 `sysconf` System Call

You use the `sysconf` system call not only to check at run-time for support of optional features (Section 1.5.4), but also to see what various implementation-dependent limits are, such as the number of files you can have open at once. You can see a list of things it can query from the man page or from [SUS2002].

sysconf—get system option or limit

```
#include <unistd.h>

long sysconf(
    int name           /* option or limit name */
);
/* Returns option/limit value or -1 (sets errno on error) */
```

The only error you can make with `sysconf`, assuming your code compiles, is to use a symbol it doesn't know about, in which case it returns –1 and sets `errno` to `EINVAL`. Otherwise, if `errno` wasn't set, a –1 return just means that the option isn't supported or, if you're testing a limit, that there is no limit. So you have to set `errno` to zero before calling `sysconf`. Also, don't use the `ec_neg1` macro, because it thinks all –1 returns are errors. In the following example that checks a limit, we do the checking ourselves and use `EC_FAIL` to trigger the error. (The "ec" stuff was explained in Section 1.4.2.)

```
    long value;

#if defined(_SC_ATEXIT_MAX)
    errno = 0;
    if ((value = sysconf(_SC_ATEXIT_MAX)) == -1)
        if (errno == 0)
            printf("max atexit registrations: unlimited\n");
        else
            EC_FAIL
    else
        printf("max atexit registrations: %ld\n", value);
#else
    printf("_SC_ATEXIT_MAX undefined\n");
#endif
```

On Linux we got:

```
max atexit registrations: 2147483647
```

which is the biggest value a `long` can have (the same as the Standard C symbol `LONG_MAX`). It probably just means that Linux keeps the registrations in a linked list rather than a fixed-sized array. But on FreeBSD we got:

```
_SC_ATEXIT_MAX undefined
```

which is also OK, as the symbol isn't part of POSIX1990, which is all FreeBSD claims to conform to. It just illustrates the importance of checking to see whether the symbol is defined. (Since Standard C says that the minimum value for `atexit` is 32, we can assume that the number for FreeBSD is 32, although we got that from the C Standard, not from `sysconf`.)

1.5.6 `pathconf` and `fpathconf` System Calls

`sysconf` is only for checking system-wide options and limits. Others depend on what file you're talking about, and for that there are two very similar functions named `fpathconf` and `pathconf`:

pathconf—get system option or limit by path

```
#include <unistd.h>

long pathconf(
      const char *path,      /* pathname */
      int name               /* option or limit name */
);
/* Returns option/limit value or -1 (sets errno on error) */
```

fpathconf—get system option or limit by file descriptor

```
#include <unistd.h>

long fpathconf(
      int fd,                /* file descriptor */
      int name               /* option or limit name */
);
/* Returns option/limit value or -1 (sets errno on error) */
```

These two functions take the same second argument and return the same result; you use `fpathconf` when you have an open file, and `pathconf` if you have a path name to a file that may or may not be open. The valid values for `name` are in [SUS2002] or on your system's `man` page.

The interpretation of the return value is the same as for `sysconf` (previous section). In particular, a –1 means an error only if `errno` changed, so you have to set it to zero before making the call. We showed code using `pathconf` in Section 1.5.4.

1.5.7 `confstr` System Call

The call `confstr` is used like `sysconf`, but it's for string values, rather than numeric ones:

confstr—get configuration string

```
#include <unistd.h>

size_t confstr(
      int name,          /* option or limit name */
      char *buf,         /* returned string value */
      size_t len         /* size of buf */
);
/* Returns size of value or 0 on error (sets errno on error) */
```

When you make the call, you set `len` to the size of the buffer you pass in as the `buf` argument; on output it fills the buffer with a NUL-terminated string, truncating it if there isn't room for the whole thing. The size that would be needed for the whole string is returned as the value. If 0 is returned, `errno` indicates an error if it was changed. If `errno` wasn't changed, it means that name was invalid. So, as with the previous few functions, you have to set `errno` to zero before the call.

1.5.8 Checking for a Specific OS

Checking for features the POSIX/SUS way is best, but sometimes you really do need to know whether you're on Solaris, HP/UX, AIX, FreeBSD, Darwin, Linux, or whatever. Sometimes you even need to know the major and minor version numbers. We already saw one case where this is legitimate: We don't want to define `_POSIX_SOURCE` on a FreeBSD or Darwin system. Other cases are where an OS has bugs, or has a feature (e.g., System V messages) that goes beyond the standard they're claiming to conform to (see next section).

There's no portable way to check for the OS and, on some systems (e.g., Solaris), no nonportable way, either. On most systems a symbol may be set once you include a particular header, but we want the indication very early, before we've even included unistd.h, which should usually be the first POSIX or Standard C header to be included. So, we set a symbol at compile time, making sure it's different for each system, like this:

```
$ gcc -DSOLARIS -c xyz.c
```

Actually, our command lines are more complicated than that, and they're inside makefiles, but you get the idea.

1.5.9 Bonus Features

Systems like FreeBSD and Darwin are pretty conservative in their claims of POSIX conformance, yet they're much more complete than a pure POSIX 1988 or POSIX1990 system, which included neither socket-style networking nor System V IPC. FreeBSD has those features, which mostly work just as they're supposed to, and lots more.[19]

It isn't possible for an OS to set a POSIX level and also an option symbol to indicate that it has bonus features, because (1) some features, like those I listed, were never in any POSIX standard prior to SUS3 (POSIX2001, if you like) and (2) in SUS3, they are not optional, so there's no symbol for them. FreeBSD can't solve the problem by claiming to conform to, say, SUS1, because it doesn't.

At the same time, we certainly don't want to put tests for the FREEBSD symbol all over our code, because that's not what we mean: We mean "sockets" or "System V IPC." Gratuitous OS dependencies make porting the code, especially by a different programmer, difficult because the porter has no idea what point you're making with the OS-dependent test. Somebody porting to, say, Linux may be an expert on Linux but be clueless about the peculiarities of FreeBSD.

So, we need to make up even more symbols for what we call bonus features: those that are standard someplace, but not in the standard to which the OS claims to conform. I'll introduce these as we go along and you'll sometimes see them in the example programs, with code something like this:

```
#if (defined(_XOPEN_SOURCE) && _XOPEN_SOURCE >= 4) || defined(BSD_DERIVED)
#define HAVE_SYSVMSG
#endif
```

Our common header (see next section) defines `BSD_DERIVED` if `FREEBSD` or `DARWIN` is defined, as explained in Section 1.5.8.

19. Darwin 6.6, the version in Mac OS X 10.2.6, is missing System V messages, although it has the other System V IPC features.

You might think of what we're doing as formalizing the idea of "partial conformance," an idea that makes the standards people wince, but one that OS implementors and application writers find essential.

1.6 Common Header File

All of the examples in this book include the common header file defs.h that contains our request for SUV_SUS2 and the include for suvreq.h, which I talked about in Section 1.5.3. It also has many of the standard includes that we often need; we don't worry about including any extras if a program doesn't need them all. Other, less common, headers we include as we need them. I won't show it here, but defs.h also includes the prototypes for utility functions like `syserrmsg` (Section 1.4.1). Here's most of defs.h, or at least what it looked like when this was being written; the latest version is on the Web site [AUP2003]:

```
#if defined(FREEBSD) || defined(DARWIN)
#define BSD_DERIVED
#endif

#if !defined(BSD_DERIVED) /* _POSIX_SOURCE too restrictive */
#define SUV_SUS2
#include "suvreq.h"
#endif

#ifdef __GNUC__
#define _GNU_SOURCE /* bring GNU as close to C99 as possible */
#endif

#include <unistd.h>

#ifndef __cplusplus
#include <stdbool.h> /* C99 only */
#endif
#include <sys/types.h>
#include <time.h>
#include <limits.h>
#ifdef SOLARIS
#define _VA_LIST /* can't define it in stdio.h */
#endif
#include <stdio.h>
#ifdef SOLARIS
#undef _VA_LIST
#endif
#include <stdarg.h> /* this is the place to define _VA_LIST */
```

```
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <assert.h>
#include "ec.h"
```

The `#ifdef SOLARIS` stuff about two-thirds of the way down is a perfect example of why you sometimes need OS-dependent code. There's disagreement about whether the "va" macros (used for variable argument lists in C) go in stdio.h or stdarg.h, and gcc on Solaris does it differently from the other systems. So, I used a little trick to effectively disable the definitions in stdio.h. Not pretty, and worth some effort to avoid, but sometimes there's just no other way, and you need to get on with your work.

The order of the includes is a little strange, and probably could be partially alphabetized, but in a few cases I came upon order dependencies and had to juggle things around a bit to get a clean compile on all the systems I tried. What you see is where things ended up. There aren't supposed to be order dependencies because a header should include other headers it depends on, but there are.

1.7 Dates and Times

Two kinds of time are commonly used in UNIX: *calendar time* and *execution time*. This section includes system calls and functions for getting the time and manipulating it. There are also interval timers that you can set; they're discussed in Section 9.7.

1.7.1 Calendar Time

Calendar time is used for the access, modification, and status-change times of files, for recording when a user logged in, for displaying the current date and time, and so on.

Calendar time is usually in one of four forms:

- An arithmetic type, `time_t`, which is a count of the number of seconds since the *epoch,* which is traditionally set at midnight, January 1, 1970

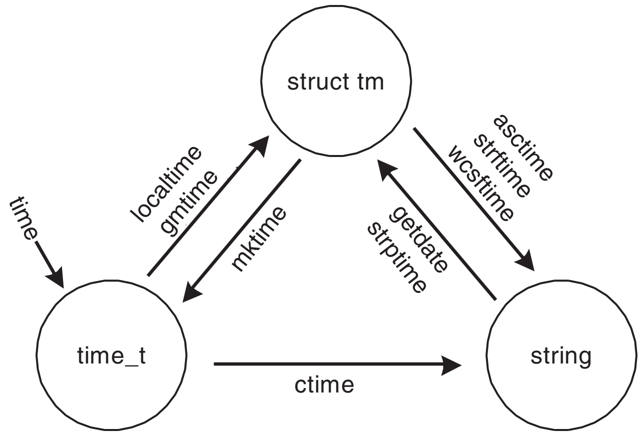


Figure 1.1 Time-conversion functions.

UTC.[20] It's a good way to store time in files and to pass it around among functions, because it's compact and time-zone-independent.

- A structure type, `struct timeval`, which holds a time in seconds and microseconds. (Used for both calendar and execution times.)
- A structure type, `struct tm`, which has time broken down into year, month, day, hour, minute, second, and a few other things.
- A string, such as `Tue Jul 23 09:44:17 2002`.

There's a full set of library functions, most from Standard C, to convert between the three forms. The function names are a crazy collection that make no sense at all (nothing simple like `cvt_tm_to_time`); Figure 1.1 provides a map that shows what the nine conversion functions do. The tenth function, `time`, is for getting the current time.

Here are the `tm` and `timeval` structures[21] followed by the synopses of the primary calendar-time functions:[22]

20. Universal Coordinated Time. Formerly known as GMT, or Greenwich Mean Time.

21. There's one obsolete structure, `timeb`, which holds the time in seconds and milliseconds, and an obsolete function, `ftime`, to fill it. Use `gettimeofday` instead.

22. Some of the Standard C functions also have a re-entrant form (e.g., `ctime_r`) that doesn't use a static buffer, but these aren't covered in this book.

struct tm—structure for broken-down time

```
struct tm {
    int tm_sec;            /* second [0,61] (up to 2 leap seconds) */
    int tm_min;            /* minute [0,59] */
    int tm_hour;           /* hour [0,23] */
    int tm_mday;           /* day of month [1,31] */
    int tm_mon;            /* month  [0,11] */
    int tm_year;           /* years since 1900 */
    int tm_wday;           /* day of week [0,6] (0 = Sunday) */
    int tm_yday;           /* day of year [0,365] */
    int tm_isdst;          /* daylight-savings flag */
};
```

struct timeval—structure for gettimeofday

```
struct timeval {
    time_t tv_sec;         /* seconds */
    suseconds_t tv_usec;   /* microseconds */
};
```

time—get current date and time as time_t

```
#include <time.h>

time_t time(
    time_t *t              /* NULL or returned time */
);
/* Returns time or -1 on error (errno not defined) */
```

gettimeofday—get current date and time as timeval

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

int gettimeofday(
    struct timeval *tvalbuf,      /* returned time */
    void *dummy                   /* always NULL */
);
/* Returns 0 on success or -1 (maybe) on error (may set errno) */
```

localtime—convert time_t to local broken-down time

```
#include <time.h>

struct tm *localtime(
    const time_t *t        /* time */
);
/* Returns broken-down time or NULL on error (sets errno) */
```

gmtime—convert time_t to UTC broken-down time

```
#include <time.h>

struct tm *gmtime(
    const time_t *t        /* time */
);
/* Returns broken-down time or NULL on error (sets errno) */
```

mktime—convert local broken-down time to time_t

```
#include <time.h>

time_t mktime(
    struct tm *tmbuf            /* broken-down time */
);
/* Returns time or -1 on error (errno not defined) */
```

ctime—convert time_t to local-time string

```
#include <time.h>

char *ctime(
    const time_t *t             /* time */
);
/* Returns string or NULL on error (errno not defined) */
```

asctime—convert broken-down time to local-time string[23]

```
#include <time.h>

char *asctime(
    const struct tm *tmbuf      /* broken-down time */
);
/* Returns string or NULL on error (errno not defined) */
```

strftime—convert broken-down time to string with format

```
#include <time.h>

size_t strftime(
    char *buf,                  /* output buffer */
    size_t bufsize,             /* size of buffer */
    const char *format,         /* format */
    const struct tm *tmbuf      /* broken-down time */
);
/* Returns byte count or 0 on error (errno not defined) */
```

wcsftime—convert broken-down time to wide-character string with format

```
#include <wchar.h>
size_t wcsftime(
    wchar_t *buf,               /* output buffer */
    size_t bufsize,             /* size of buffer */
    const wchar_t *format,      /* format */
    const struct tm *tmbuf      /* broken-down time */
);
/* Returns wchar_t count or 0 on error (errno not defined) */
```

23. The SUS doesn't say that `asctime` returns NULL on an error, but it's a good idea to check anyway.

getdate—convert string to broken-down time with rules

```
#include <time.h>

struct tm *getdate(
     const char *s               /* string to convert */
);
/* Returns broken-down time or NULL on error (sets getdate_err) */
```

strptime—convert string to broken-down time with format

```
#include <time.h>

char *strptime(
     const char *s,              /* string to convert */
     const char *format,         /* format */
     struct tm *tmbuf            /* broken-down time (output) */
);
/* Returns pointer to first unparsed char or NULL on error (errno not
defined) */
```

According to the standard, none of these functions set `errno`, and most don't even return an error indication, although it's a good idea to test any returned pointers against `NULL` anyway. `getdate` is really weird: On an error it sets the variable or macro `getdate_err`, used nowhere else, to a number from 1 to 8 to indicate the nature of the problem; see [SUS2002] for details.

Most UNIX systems implement `time_t` as a `long` integer, which is often only 32 bits (signed); that's enough seconds to take us to 19-Jan-2038.[24] At some point it needs more bits, and it will be changed to something that's guaranteed to be 64 bits.[25] To prepare, always use a `time_t` (rather than, say, a `long`) and don't make any assumptions about what it is. A `time_t` is already inadequate for historical times, as on many systems it goes (with a negative number) only back to 1901.

Without making any assumptions about what a `time_t` is, it's difficult to subtract two of them, which is a common need, so there's a function just for that:

difftime—subtract two time_t values

```
#include <time.h>

double difftime(
     time_t time1,               /* time */
     time_t time0                /* time */
);
/* Returns time1 - time0 in seconds (no error return) */
```

24. If we keep to our aggressive 19-year revision cycle, this should be about when the fourth edition of this book will appear.
25. Clearly that's overkill, but with computers the number after 32 is often 64.

`gettimeofday` is a higher-resolution alternative to `time`, but none of the other functions take a `struct timeval` as an argument. They'll take the `tv_sec` field by itself, however, since it's a `time_t`, and you can deal with the microseconds separately. All you can safely assume about the type `tv_usec` is that it's a signed integer of some size. However, there is no reason for it to be any wider than 32 bits, as there are only 1,000,000 microseconds in a second, so casting the `tv_usec` field to a `long` will work fine. You can time an execution interval by bracketing it with two calls to `gettimeofday` and then use `difftime` on the `tv_sec` fields and naked subtraction on the `tv_usec` fields.

Some implementations (FreeBSD, Darwin, and Linux, but not Solaris) use the second argument to `gettimeofday` to return time-zone information, but that's nonstandard. Most implementations indicate an error with a –1 return, and even set `errno`. Those that don't, return zero always, so you can safely check the error return in all cases.

A `time_t` is always in terms of UTC, but broken-down time (`struct tm`) and strings can be in UTC or in local time. It's convenient to use local time when times are entered or printed out, and this complicates things. There must be some way for the computer to know what time zone the user is in and whether it is currently standard or daylight time. Worse, when printing past times, the computer must know whether it was standard or daylight time back *then*. The various functions concerned with local time deal with this problem fairly well.

The user's time zone is recorded in the environment variable `TZ` or as a system default if `TZ` isn't set. To provide portability, there's one function and three global variables to deal with time-zone and daylight-time settings:

tzset—set time zone information

```
#include <time.h>

extern int daylight;        /* DST? (not in FreeBSD/Darwin) */
extern long timezone;       /* timezone in secs (not in FreeBSD/Darwin) */
extern char *tzname[2];     /* timezone strings (not in FreeBSD/Darwin) */

void tzset(void);
```

An application can call `tzset` to somehow get the time zone (from `TZ` or wherever) and set the three globals, as follows:

- `daylight` is set to zero (`false`) if Daylight Savings Time should never be applied to the time zone, and non-zero (`true`) if it should. The switchover periods are built into the library functions in an implementation-defined way.
- `timezone` is set to the difference in seconds between UTC and local standard time. (Divide it by 3,600 to get the difference in hours.)
- `tzname` is an array of two strings, `tzname[0]` to be used to designate the local time zone during standard time, and `tzname[1]` for Daylight Savings Time.

The three globals were not in POSIX earlier than POSIX2002 and are not defined on FreeBSD or Darwin. But usually you don't need to call `tzset` or interrogate the globals directly. The standard functions do that as needed.

Here's an example anyway:

```
tzset();
printf("daylight = %d; timezone = %ld hrs.; tzname = %s/%s\n",
  daylight, timezone / 3600, tzname[0], tzname[1]);
```

with this output:

```
daylight = 1; timezone = 7 hrs.; tzname = MST/MDT
```

For all the details of how all the calendar-time functions are used, especially the ones that involve format strings, see [SUS2002] or a good book on Standard C, such as [Har2002]. Here are some miscellaneous points to keep in mind:

- Ignoring the possibility of an error return, `ctime(t)` is defined as `asctime(localtime(t))`.
- The `tm_sec` field of the `tm` structure can go to 61 (instead of just 59), to allow for occasional leap seconds.
- The `tm_year` field of the `tm` structure is an offset from 1900. It's still perfectly Y2K compatible, just strange (2002 is represented as 102).
- Just as strangely, `ctime` and `asctime` put a newline at the end of the output string.
- Several functions take a pointer to a `time_t`, when a call-by-value `time_t` would work as well. This is a holdover from the earliest days of UNIX when the C language didn't have a `long` data type and you had to pass in an array of two 16-bit `int`s.

- The time_t pointer argument to time is totally unnecessary and should just be NULL. It looks like it supplies a buffer for time to use, but no buffer is needed, as the returned time_t is an arithmetic type.
- All of the functions presented here that return a pointer to a tm structure use a static buffer internally on most implementations, so use the buffer or copy it before you call another such function. There are re-entrant forms with an _r suffix that you can use in an multithreaded program if you need to.
- The functions that use formats, strftime, wcsftime, getdate, and strptime are very complicated, but very powerful, and are well worth learning. The first two are in Standard C; the last two are in the SUS. getdate is not in FreeBSD or Darwin, although there is a version of it in the GNU C library that you can install if you want.
- Use getdate to get dates and times that users type in, and strptime or getdate when reading data files containing dates. The reason is that strptime takes only a single format, which makes it too difficult for users to format the date and time correctly. getdate uses a whole list of formats and tries them one-by-one looking for a match.

1.7.2 Execution Time

Execution time is used to measure timing intervals and process-execution times, and for accounting records. The kernel automatically records each process's execution time. Interval times come in various subsecond units ranging from nanoseconds to hundredths of a second.

The execution-time types and functions are just as confusing as those for calendar time. First, here are the principal types:

- clock_t, in Standard C, an arithmetic type (typically a long, but don't assume so) which is a time interval in units of CLOCKS_PER_SEC or clock ticks
- struct timeval, which holds an interval in seconds and microseconds (also used for calendar time, and explained in detail in the previous section)
- struct timespec, which holds an interval in seconds and nanoseconds

Here's the structure for timespec (we already showed timeval), which is defined in the header <time.h>:

struct timespec—structure for time in seconds and nanoseconds

```
struct timespec {
    time_t tv_sec;          /* seconds */
    long tv_nsec;           /* nanoseconds */
};
```

There are three primary functions for measuring a time interval, `gettimeofday`, detailed in the previous section, `clock`, and `times`. The latter two and the structure used by `times` are:

clock—get execution time

```
#include <time.h>

clock_t clock(void);
/* Returns time in CLOCKS_PER_SEC or -1 on error (errno not defined) */
```

times—get process and child-process execution times

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

clock_t times(
    struct tms *buffer      /* returned times */
);
/* Returns time in clock ticks or -1 on error (sets errno) */
```

struct tms—structure for times system call

```
struct tms {
    clock_t tms_utime;      /* user time */
    clock_t tms_cutime;     /* user time of terminated children */
    clock_t tms_stime;      /* sys time */
    clock_t tms_cstime;     /* sys time of terminated children */
};
```

`times` returns the time elapsed since some point in the past (usually since the system was booted), sometimes called the *wall-clock* or *real* time. This is different from the time returned from the functions in the previous section, which returned the time since the epoch (normally 1-Jan-1970).

`times` uses units of clock ticks; you get the number of clock ticks per second with `sysconf` (Section 1.5.5) like this:

```
clock_ticks = sysconf(_SC_CLK_TCK);
```

Additionally, `times` loads the `tms` structure with more specific information:

- tms_utime (user time) is the time spent executing instructions from the process's user code. It's CPU time only and doesn't include time spent waiting to run.
- tms_stime (system time) is the CPU time spent executing system calls on behalf of the process.
- tms_cutime (child user time) is the total of user CPU times for all the process's child processes that have terminated and for which the parent has issued a wait system call (explained fully in Chapter 5).
- tms_cstime (child system time) is the total of system CPU times for terminated and waited-for child processes.

Although clock returns the same type as times, the value is CPU time *used* since the process started, not real time, and the units are given by the macro CLOCKS_PER_SEC, not clock ticks. It's equivalent (except for the different units) to tms_utime + tms_stime.

CLOCKS_PER_SEC is defined in the SUS as 1,000,000 (i.e., microseconds), but non-SUS systems may use a different value, so always use the macro. (FreeBSD defines CLOCKS_PER_SEC as 128; Darwin as 100.)

If clock_t is a 32-bit signed integer, which it often is, and uses units of microseconds, it wraps after about 36 minutes, but at least clock starts timing when the process starts. times, operating in clock ticks, can run longer, but it also starts earlier—much earlier if the UNIX system has been running for weeks or months, which is very common. Still, a 32-bit signed clock_t operating in units of ticks won't wrap for about 250 days, so a semiannual reboot, if you can arrange for one, solves the problem.

However, you can't assume that clock_t is a 32-bit signed integer, or that it's signed, or even that it's an integer. It could be an unsigned long, or even a double.

We'll use times throughout this book to time various programs because it gives us the real time and the user and system CPU times all at once. Here are two handy functions that we'll use—one to start timing, and one to stop and print the results:

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

static struct tms tbuf1;
static clock_t real1;
static long clock_ticks;
```

```
void timestart(void)
{
    ec_neg1( real1 = times(&tbuf1) )
    /* treat all -1 returns as errors */
    ec_neg1( clock_ticks = sysconf(_SC_CLK_TCK) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("timestart");
EC_CLEANUP_END
}

void timestop(char *msg)
{
    struct tms tbuf2;
    clock_t real2;

    ec_neg1( real2 = times(&tbuf2) )
    fprintf(stderr,"%s:\n\t\"Total (user/sys/real)\", %.2f, %.2f, %.2f\n"
      "\t\"Child (user/sys)\", %.2f, %.2f\n", msg,
      ((double)(tbuf2.tms_utime + tbuf2.tms_cutime) -
      (tbuf1.tms_utime + tbuf1.tms_cutime)) / clock_ticks,
      ((double)(tbuf2.tms_stime + tbuf2.tms_cstime) -
      (tbuf1.tms_stime + tbuf1.tms_cstime)) / clock_ticks,
      (double)(real2 - real1) / clock_ticks,
      (double)(tbuf2.tms_cutime - tbuf1.tms_cutime) / clock_ticks,
      (double)(tbuf2.tms_cstime - tbuf1.tms_cstime) / clock_ticks);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("timestop");
EC_CLEANUP_END
}
```

When we want to time something, we just call `timestart` at the beginning of the interval and `timestop` at the end, as in this example, which also includes calls to `gettimeofday` and `clock` for a comparison:

```
#define REPS 1000000
#define TV_SUBTRACT(t2, t1)\
    (double)(t2).tv_sec + (t2).tv_usec / 1000000.0 -\
    ((double)(t1).tv_sec + (t1).tv_usec / 1000000.0)

int main(void)
{
    int i;
    char msg[100];
    clock_t c1, c2;
    struct timeval tv1, tv2;
    snprintf(msg, sizeof(msg), "%d getpids", REPS);
    ec_neg1( c1 = clock() )
    gettimeofday(&tv1, NULL);
    timestart();
    for (i = 0; i < REPS; i++)
        (void)getpid();
    (void)sleep(2);
    timestop(msg);
    gettimeofday(&tv2, NULL);
    ec_neg1( c2 = clock() )
    printf("clock(): %.2f\n", (double)(c2 - c1) / CLOCKS_PER_SEC);
    printf("gettimeofday(): %.2f\n", TV_SUBTRACT(tv2, tv1));
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

This was the output on Linux (Solaris, FreeBSD, and Darwin gave similar results):

```
1000000 getpids:
        "Total (user/sys/real)", 0.72, 0.44, 3.19
        "Child (user/sys)", 0.00, 0.00
clock(): 1.16
gettimeofday(): 3.19
```

So `clock` measured 1.16,[26] exactly the sum of .72 and .44, and `gettimeofday` measured the same elapsed time as `times`. That's good, or we would have more explaining to do.

If you want more resolution than `times` can give you, use `getrusage` instead (Section 5.16).

1.8 About the Example Code

There's lots of example code in this book, and you're free to use it for your own educational purposes or even in commercial products, as long as you give credit where you normally would, such as in an About dialog box or the credits page of a manual or book. For details, see the Web page *www.basepath.com/aup/copyright.htm*.

26. In the 1985 edition of this book, we got a time of 1.43 seconds for only 1000 `getpids`!

Sometimes the examples in this book are abridged to save space (as in the error-handling code earlier in this chapter and the code in defs.h, but the complete files are on the Web at *www.basepath.com/aup* [AUP2003]. (There's lots of other information there, too, such as errata.) The code on the Web will be updated, as bugs are found after this book's publication, so if you're confused by something you think is wrong in the book, check the newer code first. Then, if you still think there's a bug, please email me at aup@basepath.com and let me know. If you have a different view of something I've said or a better way to do things, or even if you just want to say that this is the best book you've ever read, I'd love to get those emails, too.

The examples are written in C99, but the new stuff (e.g., `intptr_t`, `bool`) shouldn't present a problem if your compiler isn't at that level yet, as you can define your own macros and `typedef`s. Most of the code has been tested with the gcc compiler on SuSE Linux 8.0 (based on Linux 2.4), FreeBSD 4.6, and Solaris 8, all running on Intel CPUs, and Darwin 6.6 (the UNIX part of Mac OS X 10.2.6) on a PowerPC-based iMac. As time goes on, more testing on more systems will be done, and if there are any changes they'll be posted on the Web site.

1.9 Essential Resources

Here's a list of the essential resources (besides this book!) you'll want to have handy while you're programming:

- A good **reference book for your language**. For C, the best is *C: A Reference Manual*, by Harbison and Steele [Har2002]; for C++, perhaps *The C++ Programming Language*, by Bjarne Stroustrup [Str2000]. The idea is to have one book with all the answers, even if finding them is a bit of work. After you know the book, it's no longer any work, and you save lots of time and desk space with only one book instead of several.
- The **Open Group SUS** (Single UNIX Specification) at:

  www.unix.org/version3

  This is a fantastic site. You click on a function name in the left frame, and the spec shows in the right frame. It's also available on CD and on paper, but the Web version is free.
- The Usenet newsgroup **comp.unix.programmer**. Post almost any UNIX question, no matter how difficult, and you'll get an answer. It may not be

correct, but usually it will at least point you in the right direction. (To get to newsgroups, you need a mail client that can handle them and a news feed. Your ISP may provide one or, if not, you can pay a fee to *www.supernews.com* or *www.giganews.com*. Or, access them through Google Groups at *http://groups.google.com/grphp.*) There are some Web forums, too, such as *www.unix.com*, but they're not nearly as good as comp.unix.programmer. Stick to UNIX (or Linux or FreeBSD) questions—you'll get treated abruptly if you post a C or C++ question. There are separate newsgroups for those subjects (tens of thousands of newsgroups altogether). Google Groups also allows you to search through the Usenet archives to see if maybe someone in your parents' generation asked the same question 20 years ago.

- **Google**, the Web search site. Suppose the `sigwait` system call seems to be misbehaving on FreeBSD, or you're not sure whether Linux implements asynchronous I/O. You can get the answers the methodical way (from the FreeBSD or Linux sites, for example), or you can google around for the answer. About half the time, I get what I'm looking for not from the official source, but from some obscure forum post or private Web site that Google just happened to index.
- **GNU C Library**, a great source for coding examples, available for download at:

  `www.gnu.org/software/libc`

- The **`man` pages** for your system. Somewhat inconvenient to navigate through (the `apropos` command helps a little), but usually excellent once you've found the right page. It's a good idea to spend some time learning your way around so you'll be able to use them efficiently when the time comes.

Exercises

1.1. Write a program in C that displays "Hello Word" using the include file defs.h and at least one of the error-checking macros (`ec_neg1`, say). The purpose of this exercise is to get you to install all of the code for this book on your system, to make sure your C compiler is running OK, and to ensure the tools you'll need, such as a text editor and `make`, are in place. If you don't have access to a UNIX or Linux system, dealing with that problem is part of this exercise, too.

1.2. Write a program that displays what version of POSIX and the SUS your system conforms to. Do it for various requests (e.g., `SUV_SUS2`).

1.3. Write a program that displays all the `sysconf` values for your system. Figure out how to produce all the lines containing `sysconf` calls without having to type every symbol one-by-one.

1.4. Same as 1.3, but for `pathconf`. Design the program so that you can enter a path as an argument.

1.5. Same as 1.3, but for `confstr`.

1.6. Write a version of the `date` command, with no command-line options. For extra credit, handle as many options as you can. Base it on the SUS or one of the POSIX standards. No fair peeking at Gnu source or any other similar examples.