MODULE 3

UNIX Standardization and Implementations

2.1 Introduction

Much work has gone into standardizing the UNIX programming environment and the C programming language. Although applications have always been quite portable across different versions of the UNIX operating system, the proliferation of versions and differences during the 1980s led many large users, such as the U.S. government, to call for standardization.

In this chapter we first look at the various standardization efforts that have been under way over the past two and a half decades. We then discuss the effects of these UNIX programming standards on the operating system implementations that are described in this book. An important part of all the standardization efforts is the specification of various limits that each implementation must define, so we look at these limits and the various ways to determine their values.

2.2 UNIX Standardization

2.2.1 ISO C

In late 1989, ANSI Standard X3.159-1989 for the C programming language was approved. This standard was also adopted as International Standard ISO/IEC 9899:1990. ANSI is the American National Standards Institute, the U.S. member in the International Organization for Standardization (ISO). IEC stands for the International Electrotechnical Commission.

The C standard is now maintained and developed by the ISO/IEC international standardization working group for the C programming language, known as ISO/IEC JTC1/SC22/WG14, or WG14 for short. The intent of the ISO C standard is to provide portability of conforming C programs to a wide variety of operating systems, not only the UNIX System. This standard defines not only the syntax and semantics of the programming language but also a standard library [Chapter 7 of ISO 1999; Plauger 1992; Appendix B of Kernighan and Ritchie 1988]. This library is important because all contemporary UNIX systems, such as the ones described in this book, provide the library routines that are specified in the C standard.

In 1999, the ISO C standard was updated and approved as ISO/IEC 9899:1999, largely to improve support for applications that perform numerical processing. The changes don't affect the POSIX interfaces described in this book, except for the addition of the `restrict` keyword to some of the function prototypes. This keyword is used to tell the compiler which pointer references can be optimized, by indicating that the object to which the pointer refers is accessed in the function only via that pointer.

Since 1999, three technical corrigenda have been published to correct errors in the ISO C standard—one in 2001, one in 2004, and one in 2007. As with most standards, there is a delay between the standard's approval and the modification of software to conform to it. As each vendor's compilation systems evolve, they add more support for the latest version of the ISO C standard.

A summary of the current level of conformance of `gcc` to the 1999 version of the ISO C standard is available at `http://gcc.gnu.org/c99status.html`. Although the C standard was updated in 2011, we deal only with the 1999 version in this text, because the other standards haven't yet caught up with the relevant changes.

The ISO C library can be divided into 24 areas, based on the headers defined by the standard (see Figure 2.1). The POSIX.1 standard includes these headers, as well as others. As Figure 2.1 shows, all of these headers are supported by the four implementations (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10) that are described later in this chapter.

The ISO C headers depend on which version of the C compiler is used with the operating system. FreeBSD 8.0 ships with version 4.2.1 of `gcc`, Solaris 10 ships with version 3.4.3 of `gcc` (in addition to its own C compiler in Sun Studio), Ubuntu 12.04 (Linux 3.2.0) ships with version 4.6.3 of `gcc`, and Mac OS X 10.6.8 ships with both versions 4.0.1 and 4.2.1 of `gcc`.

2.2.2 IEEE POSIX

POSIX is a family of standards initially developed by the IEEE (Institute of Electrical and Electronics Engineers). POSIX stands for Portable Operating System Interface. It originally referred only to the IEEE Standard 1003.1-1988—the operating system interface—but was later extended to include many of the standards and draft standards with the 1003 designation, including the shell and utilities (1003.2).

Of specific interest to this book is the 1003.1 operating system interface standard, whose goal is to promote the portability of applications among various UNIX System environments. This standard defines the services that an operating system must

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
`<assert.h>`	•	•	•	•	verify program assertion
`<complex.h>`	•	•	•	•	complex arithmetic support
`<ctype.h>`	•	•	•	•	character classification and mapping support
`<errno.h>`	•	•	•	•	error codes (Section 1.7)
`<fenv.h>`	•	•	•	•	floating-point environment
`<float.h>`	•	•	•	•	floating-point constants and characteristics
`<inttypes.h>`	•	•	•	•	integer type format conversion
`<iso646.h>`	•	•	•	•	macros for assignment, relational, and unary operators
`<limits.h>`	•	•	•	•	implementation constants (Section 2.5)
`<locale.h>`	•	•	•	•	locale categories and related definitions
`<math.h>`	•	•	•	•	mathematical function and type declarations and constants
`<setjmp.h>`	•	•	•	•	nonlocal goto (Section 7.10)
`<signal.h>`	•	•	•	•	signals (Chapter 10)
`<stdarg.h>`	•	•	•	•	variable argument lists
`<stdbool.h>`	•	•	•	•	Boolean type and values
`<stddef.h>`	•	•	•	•	standard definitions
`<stdint.h>`	•	•	•	•	integer types
`<stdio.h>`	•	•	•	•	standard I/O library (Chapter 5)
`<stdlib.h>`	•	•	•	•	utility functions
`<string.h>`	•	•	•	•	string operations
`<tgmath.h>`	•	•	•	•	type-generic math macros
`<time.h>`	•	•	•	•	time and date (Section 6.10)
`<wchar.h>`	•	•	•	•	extended multibyte and wide character support
`<wctype.h>`	•	•	•	•	wide character classification and mapping support

Figure 2.1 Headers defined by the ISO C standard

provide if it is to be "POSIX compliant," and has been adopted by most computer vendors. Although the 1003.1 standard is based on the UNIX operating system, the standard is not restricted to UNIX and UNIX-like systems. Indeed, some vendors supplying proprietary operating systems claim that these systems have been made POSIX compliant, while still leaving all their proprietary features in place.

Because the 1003.1 standard specifies an *interface* and not an *implementation*, no distinction is made between system calls and library functions. All the routines in the standard are called *functions*.

Standards are continually evolving, and the 1003.1 standard is no exception. The 1988 version, IEEE Standard 1003.1-1988, was modified and submitted to the International Organization for Standardization. No new interfaces or features were added, but the text was revised. The resulting document was published as IEEE Standard 1003.1-1990 [IEEE 1990]. This is also International Standard ISO/IEC 9945-1:1990. This standard was commonly referred to as *POSIX.1*, a term which we'll use in this text to refer to the different versions of the standard.

The IEEE 1003.1 working group continued to make changes to the standard. In 1996, a revised version of the IEEE 1003.1 standard was published. It included the 1003.1-1990 standard, the 1003.1b-1993 real-time extensions standard, and the interfaces for multithreaded programming, called *pthreads* for POSIX threads. This version of the standard was also published as International Standard ISO/IEC 9945-1:1996. More real-time interfaces were added in 1999 with the publication of IEEE Standard 1003.1d-1999. A year later, IEEE Standard 1003.1j-2000 was published, including even more real-time interfaces, and IEEE Standard 1003.1q-2000 was published, adding event-tracing extensions to the standard.

The 2001 version of 1003.1 departed from the prior versions in that it combined several 1003.1 amendments, the 1003.2 standard, and portions of the Single UNIX Specification (SUS), Version 2 (more on this later). The resulting standard, IEEE Standard 1003.1-2001, included the following other standards:

- ISO/IEC 9945-1 (IEEE Standard 1003.1-1996), which includes
    - IEEE Standard 1003.1-1990
    - IEEE Standard 1003.1b-1993 (real-time extensions)
    - IEEE Standard 1003.1c-1995 (pthreads)
    - IEEE Standard 1003.1i-1995 (real-time technical corrigenda)
- IEEE P1003.1a draft standard (system interface amendment)
- IEEE Standard 1003.1d-1999 (advanced real-time extensions)
- IEEE Standard 1003.1j-2000 (more advanced real-time extensions)
- IEEE Standard 1003.1q-2000 (tracing)
- Parts of IEEE Standard 1003.1g-2000 (protocol-independent interfaces)
- ISO/IEC 9945-2 (IEEE Standard 1003.2-1993)
- IEEE P1003.2b draft standard (shell and utilities amendment)
- IEEE Standard 1003.2d-1994 (batch extensions)
- The Base Specifications of the Single UNIX Specification, version 2, which include
    - System Interface Definitions, Issue 5
    - Commands and Utilities, Issue 5
    - System Interfaces and Headers, Issue 5
- Open Group Technical Standard, Networking Services, Issue 5.2
- ISO/IEC 9899:1999, Programming Languages–C

In 2004, the POSIX.1 specification was updated with technical corrections; more comprehensive changes were made in 2008 and released as Issue 7 of the Base Specifications. ISO approved this version at the end of 2008 and published it in 2009 as International Standard ISO/IEC 9945:2009. It is based on several other standards:

- IEEE Standard 1003.1, 2004 Edition
- Open Group Technical Standard, 2006, Extended API Set, Parts 1−4
- ISO/IEC 9899:1999, including corrigenda

Figure 2.2, Figure 2.3, and Figure 2.4 summarize the required and optional headers as specified by POSIX.1. Because POSIX.1 includes the ISO C standard library functions, it also requires the headers listed in Figure 2.1. All four figures summarize which headers are included in the implementations discussed in this book.

In this text we describe the 2008 edition of POSIX.1. Its interfaces are divided into required ones and optional ones. The optional interfaces are further divided into 40 sections, based on functionality. The sections containing nonobsolete programming interfaces are summarized in Figure 2.5 with their respective option codes. Option codes are two- to three-character abbreviations that identify the interfaces that belong to

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
`<aio.h>`	•	•	•	•	asynchronous I/O
`<cpio.h>`	•	•	•	•	`cpio` archive values
`<dirent.h>`	•	•	•	•	directory entries (Section 4.22)
`<dlfcn.h>`	•	•	•	•	dynamic linking
`<fcntl.h>`	•	•	•	•	file control (Section 3.14)
`<fnmatch.h>`	•	•	•	•	filename-matching types
`<glob.h>`	•	•	•	•	pathname pattern-matching and generation
`<grp.h>`	•	•	•	•	group file (Section 6.4)
`<iconv.h>`	•	•	•	•	codeset conversion utility
`<langinfo.h>`	•	•	•	•	language information constants
`<monetary.h>`	•	•	•	•	monetary types and functions
`<netdb.h>`	•	•	•	•	network database operations
`<nl_types.h>`	•	•	•	•	message catalogs
`<poll.h>`	•	•	•	•	`poll` function (Section 14.4.2)
`<pthread.h>`	•	•	•	•	threads (Chapters 11 and 12)
`<pwd.h>`	•	•	•	•	password file (Section 6.2)
`<regex.h>`	•	•	•	•	regular expressions
`<sched.h>`	•	•	•	•	execution scheduling
`<semaphore.h>`	•	•	•	•	semaphores
`<strings.h>`	•	•	•	•	string operations
`<tar.h>`	•	•	•	•	`tar` archive values
`<termios.h>`	•	•	•	•	terminal I/O (Chapter 18)
`<unistd.h>`	•	•	•	•	symbolic constants
`<wordexp.h>`	•	•	•	•	word-expansion definitions
`<arpa/inet.h>`	•	•	•	•	Internet definitions (Chapter 16)
`<net/if.h>`	•	•	•	•	socket local interfaces (Chapter 16)
`<netinet/in.h>`	•	•	•	•	Internet address family (Section 16.3)
`<netinet/tcp.h>`	•	•	•	•	Transmission Control Protocol definitions
`<sys/mman.h>`	•	•	•	•	memory management declarations
`<sys/select.h>`	•	•	•	•	`select` function (Section 14.4.1)
`<sys/socket.h>`	•	•	•	•	sockets interface (Chapter 16)
`<sys/stat.h>`	•	•	•	•	file status (Chapter 4)
`<sys/statvfs.h>`	•	•	•	•	file system information
`<sys/times.h>`	•	•	•	•	process times (Section 8.17)
`<sys/types.h>`	•	•	•	•	primitive system data types (Section 2.8)
`<sys/un.h>`	•	•	•	•	UNIX domain socket definitions (Section 17.2)
`<sys/utsname.h>`	•	•	•	•	system name (Section 6.9)
`<sys/wait.h>`	•	•	•	•	process control (Section 8.6)

Figure 2.2 Required headers defined by the POSIX standard

each functional area and highlight text describing aspects of the standard that depend on the support of a particular option. Many options deal with real-time extensions.

POSIX.1 does not include the notion of a superuser. Instead, certain operations require "appropriate privileges," although POSIX.1 leaves the definition of this term up to the implementation. UNIX systems that conform to the Department of Defense's security guidelines have many levels of security. In this text, however, we use the traditional terminology and refer to operations that require superuser privilege.

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
`<fmtmsg.h>`	•	•	•	•	message display structures
`<ftw.h>`	•	•	•	•	file tree walking (Section 4.22)
`<libgen.h>`	•	•	•	•	pathname management functions
`<ndbm.h>`	•		•	•	database operations
`<search.h>`	•	•	•	•	search tables
`<syslog.h>`	•	•	•	•	system error logging (Section 13.4)
`<utmpx.h>`		•	•	•	user accounting database
`<sys/ipc.h>`	•	•	•	•	IPC (Section 15.6)
`<sys/msg.h>`	•	•	•	•	XSI message queues (Section 15.7)
`<sys/resource.h>`	•	•	•	•	resource operations (Section 7.11)
`<sys/sem.h>`	•	•	•	•	XSI semaphores (Section 15.8)
`<sys/shm.h>`	•	•	•	•	XSI shared memory (Section 15.9)
`<sys/time.h>`	•	•	•	•	time types
`<sys/uio.h>`	•	•	•	•	vector I/O operations (Section 14.6)

Figure 2.3 XSI option headers defined by the POSIX standard

After more than twenty years of work, the standards are mature and stable. The POSIX.1 standard is maintained by an open working group known as the Austin Group (`http://www.opengroup.org/austin`). To ensure that they are still relevant, the standards need to be either updated or reaffirmed every so often.

2.2.3 The Single UNIX Specification

The Single UNIX Specification, a superset of the POSIX.1 standard, specifies additional interfaces that extend the functionality provided by the POSIX.1 specification. POSIX.1 is equivalent to the Base Specifications portion of the Single UNIX Specification.

The *X/Open System Interfaces* (XSI) option in POSIX.1 describes optional interfaces and defines which optional portions of POSIX.1 must be supported for an implementation to be deemed *XSI conforming*. These include file synchronization, thread stack address and size attributes, thread process-shared synchronization, and the `_XOPEN_UNIX` symbolic constant (marked "SUS mandatory" in Figure 2.5). Only XSI-conforming implementations can be called UNIX systems.

The Open Group owns the UNIX trademark and uses the Single UNIX Specification to define the interfaces an implementation must support to call itself a UNIX system. Vendors must file conformance statements, pass test suites to verify conformance, and license the right to use the UNIX trademark.

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
`<mqueue.h>`	•	•		•	message queues
`<spawn.h>`	•	•	•	•	real-time spawn interface

Figure 2.4 Optional headers defined by the POSIX standard

Code	SUS mandatory	Symbolic constant	Description
ADV		`_POSIX_ADVISORY_INFO`	advisory information (real-time)
CPT		`_POSIX_CPUTIME`	process CPU time clocks (real-time)
FSC	•	`_POSIX_FSYNC`	file synchronization
IP6		`_POSIX_IPV6`	IPv6 interfaces
ML		`_POSIX_MEMLOCK`	process memory locking (real-time)
MLR		`_POSIX_MEMLOCK_RANGE`	memory range locking (real-time)
MON		`_POSIX_MONOTONIC_CLOCK`	monotonic clock (real-time)
MSG		`_POSIX_MESSAGE_PASSING`	message passing (real-time)
MX		`__STDC_IEC_559__`	IEC 60559 floating-point option
PIO		`_POSIX_PRIORITIZED_IO`	prioritized input and output
PS		`_POSIX_PRIORITY_SCHEDULING`	process scheduling (real-time)
RPI		`_POSIX_THREAD_ROBUST_PRIO_INHERIT`	robust mutex priority inheritance (real-time)
RPP		`_POSIX_THREAD_ROBUST_PRIO_PROTECT`	robust mutex priority protection (real-time)
RS		`_POSIX_RAW_SOCKETS`	raw sockets
SHM		`_POSIX_SHARED_MEMORY_OBJECTS`	shared memory objects (real-time)
SIO		`_POSIX_SYNCHRONIZED_IO`	synchronized input and output (real-time)
SPN		`_POSIX_SPAWN`	spawn (real-time)
SS		`_POSIX_SPORADIC_SERVER`	process sporadic server (real-time)
TCT		`_POSIX_THREAD_CPUTIME`	thread CPU time clocks (real-time)
TPI		`_POSIX_THREAD_PRIO_INHERIT`	nonrobust mutex priority inheritance (real-time)
TPP		`_POSIX_THREAD_PRIO_PROTECT`	nonrobust mutex priority protection (real-time)
TPS		`_POSIX_THREAD_PRIORITY_SCHEDULING`	thread execution scheduling (real-time)
TSA	•	`_POSIX_THREAD_ATTR_STACKADDR`	thread stack address attribute
TSH	•	`_POSIX_THREAD_PROCESS_SHARED`	thread process-shared synchronization
TSP		`_POSIX_THREAD_SPORADIC_SERVER`	thread sporadic server (real-time)
TSS	•	`_POSIX_THREAD_ATTR_STACKSIZE`	thread stack size address
TYM		`_POSIX_TYPED_MEMORY_OBJECTS`	typed memory objects (real-time)
XSI	•	`_XOPEN_UNIX`	X/Open interfaces

Figure 2.5 POSIX.1 optional interface groups and codes

Several of the interfaces that are optional for XSI-conforming systems are divided into *option groups* based on common functionality, as follows:

- Encryption: denoted by the `_XOPEN_CRYPT` symbolic constant
- Real-time: denoted by the `_XOPEN_REALTIME` symbolic constant
- Advanced real-time
- Real-time threads: denoted by `_XOPEN_REALTIME_THREADS`
- Advanced real-time threads

The Single UNIX Specification is a publication of The Open Group, which was formed in 1996 as a merger of X/Open and the Open Software Foundation (OSF), both industry consortia. X/Open used to publish the *X/Open Portability Guide*, which adopted specific standards and filled in the gaps where functionality was missing. The goal of these guides was to improve application portability beyond what was possible by merely conforming to published standards.

The first version of the Single UNIX Specification was published by X/Open in 1994. It was also known as "Spec 1170," because it contained roughly 1,170 interfaces. It grew out of the Common Open Software Environment (COSE) initiative, whose goal was to improve application portability across all implementations of the UNIX operating system. The COSE group—Sun, IBM, HP, Novell/USL, and OSF—went further than endorsing standards by including interfaces used by common commercial applications. The resulting 1,170 interfaces were selected from these applications, and also included the X/Open Common Application Environment (CAE), Issue 4 (known as "XPG4" as a historical reference to its predecessor, the X/Open Portability Guide), the System V Interface Definition (SVID), Edition 3, Level 1 interfaces, and the OSF Application Environment Specification (AES) Full Use interfaces.

The second version of the Single UNIX Specification was published by The Open Group in 1997. The new version added support for threads, real-time interfaces, 64-bit processing, large files, and enhanced multibyte character processing.

The third version of the Single UNIX Specification (SUSv3) was published by The Open Group in 2001. The Base Specifications of SUSv3 are the same as IEEE Standard 1003.1-2001 and are divided into four sections: Base Definitions, System Interfaces, Shell and Utilities, and Rationale. SUSv3 also includes X/Open Curses Issue 4, Version 2, but this specification is not part of POSIX.1.

In 2002, ISO approved the IEEE Standard 1003.1-2001 as International Standard ISO/IEC 9945:2002. The Open Group updated the 1003.1 standard again in 2003 to include technical corrections, and ISO approved this as International Standard ISO/IEC 9945:2003. In April 2004, The Open Group published the Single UNIX Specification, Version 3, 2004 Edition. It merged more technical corrections into the main text of the standard.

In 2008, the Single UNIX Specification was updated, including corrections and new interfaces, removing obsolete interfaces, and marking other interfaces as being obsolescent in preparation for future removal. Additionally, some previously optional interfaces were promoted to nonoptional status, including asynchronous I/O, barriers, clock selection, memory-mapped files, memory protection, reader–writer locks, real-time signals, POSIX semaphores, spin locks, thread-safe functions, threads, timeouts, and timers. The resulting standard is known as Issue 7 of the Base Specifications, and is the same as POSIX.1-2008. The Open Group bundled this version with an updated X/Open Curses specification and released them as version 4 of the Single UNIX Specification in 2010. We'll refer to this as SUSv4.

2.2.4 FIPS

FIPS stands for Federal Information Processing Standard. It was published by the U.S. government, which used it for the procurement of computer systems. FIPS 151-1 (April 1989) was based on the IEEE Standard 1003.1-1988 and a draft of the ANSI C standard. This was followed by FIPS 151-2 (May 1993), which was based on the IEEE Standard 1003.1-1990. FIPS 151-2 required some features that POSIX.1 listed as optional. All these options were included as mandatory in POSIX.1-2001.

The effect of the POSIX.1 FIPS was to require any vendor that wished to sell POSIX.1-compliant computer systems to the U.S. government to support some of the optional features of POSIX.1. The POSIX.1 FIPS has since been withdrawn, so we won't consider it further in this text.

2.3 UNIX System Implementations

The previous section described ISO C, IEEE POSIX, and the Single UNIX Specification—three standards originally created by independent organizations. Standards, however, are interface specifications. How do these standards relate to the real world? These standards are taken by vendors and turned into actual implementations. In this book, we are interested in both these standards and their implementation.

Section 1.1 of McKusick et al. [1996] gives a detailed history (and a nice picture) of the UNIX System family tree. Everything starts from the Sixth Edition (1976) and Seventh Edition (1979) of the UNIX Time-Sharing System on the PDP-11 (usually called Version 6 and Version 7, respectively). These were the first releases widely distributed outside of Bell Laboratories. Three branches of the tree evolved.

1. One at AT&T that led to System III and System V, the so-called commercial versions of the UNIX System.
2. One at the University of California at Berkeley that led to the 4.xBSD implementations.
3. The research version of the UNIX System, developed at the Computing Science Research Center of AT&T Bell Laboratories, that led to the UNIX Time-Sharing System 8th Edition, 9th Edition, and ended with the 10th Edition in 1990.

2.3.1 UNIX System V Release 4

UNIX System V Release 4 (SVR4) was a product of AT&T's UNIX System Laboratories (USL, formerly AT&T's UNIX Software Operation). SVR4 merged functionality from AT&T UNIX System V Release 3.2 (SVR3.2), the SunOS operating system from Sun Microsystems, the 4.3BSD release from the University of California, and the Xenix system from Microsoft into one coherent operating system. (Xenix was originally developed from Version 7, with many features later taken from System V.) The SVR4 source code was released in late 1989, with the first end-user copies becoming available during 1990. SVR4 conformed to both the POSIX 1003.1 standard and the X/Open Portability Guide, Issue 3 (XPG3).

AT&T also published the System V Interface Definition (SVID) [AT&T 1989]. Issue 3 of the SVID specified the functionality that an operating system must offer to qualify as a conforming implementation of UNIX System V Release 4. As with POSIX.1, the SVID specified an interface, not an implementation. No distinction was made in the SVID between system calls and library functions. The reference manual for an actual implementation of SVR4 must be consulted to see this distinction [AT&T 1990e].

2.3.2 4.4BSD

The Berkeley Software Distribution (BSD) releases were produced and distributed by the Computer Systems Research Group (CSRG) at the University of California at Berkeley; 4.2BSD was released in 1983 and 4.3BSD in 1986. Both of these releases ran on the VAX minicomputer. The next release, 4.3BSD Tahoe in 1988, also ran on a particular minicomputer called the Tahoe. (The book by Leffler et al. [1989] describes the 4.3BSD Tahoe release.) This was followed in 1990 with the 4.3BSD Reno release; 4.3BSD Reno supported many of the POSIX.1 features.

The original BSD systems contained proprietary AT&T source code and were covered by AT&T licenses. To obtain the source code to the BSD system you had to have a UNIX source license from AT&T. This changed as more and more of the AT&T source code was replaced over the years with non-AT&T source code and as many of the new features added to the Berkeley system were derived from non-AT&T sources.

In 1989, Berkeley identified much of the non-AT&T source code in the 4.3BSD Tahoe release and made it publicly available as the BSD Networking Software, Release 1.0. Release 2.0 of the BSD Networking Software followed in 1991, which was derived from the 4.3BSD Reno release. The intent was that most, if not all, of the 4.4BSD system would be free of AT&T license restrictions, thus making the source code available to all.

4.4BSD-Lite was intended to be the final release from the CSRG. Its introduction was delayed, however, because of legal battles with USL. Once the legal differences were resolved, 4.4BSD-Lite was released in 1994, fully unencumbered, so no UNIX source license was needed to receive it. The CSRG followed this with a bug-fix release in 1995. This release, 4.4BSD-Lite, release 2, was the final version of BSD from the CSRG. (This version of BSD is described in the book by McKusick et al. [1996].)

The UNIX system development done at Berkeley started with PDP-11s, then moved to the VAX minicomputer, and then to other so-called workstations. During the early 1990s, support was provided to Berkeley for the popular 80386-based personal computers, leading to what is called 386BSD. This support was provided by Bill Jolitz and was documented in a series of monthly articles in *Dr. Dobb's Journal* throughout 1991. Much of this code appeared in the BSD Networking Software, Release 2.0.

2.3.3 FreeBSD

FreeBSD is based on the 4.4BSD-Lite operating system. The FreeBSD project was formed to carry on the BSD line after the Computing Science Research Group at the University of California at Berkeley decided to end its work on the BSD versions of the UNIX operating system, and the 386BSD project seemed to be neglected for too long.

All software produced by the FreeBSD project is freely available in both binary and source forms. The FreeBSD 8.0 operating system was one of the four operating systems used to test the examples in this book.

Several other BSD-based free operating systems are available. The NetBSD project (`http://www.netbsd.org`) is similar to the FreeBSD project, but emphasizes portability between hardware platforms. The OpenBSD project (`http://www.openbsd.org`) is similar to FreeBSD but places a greater emphasis on security.

2.3.4 Linux

Linux is an operating system that provides a rich programming environment similar to that of a UNIX System; it is freely available under the GNU Public License. The popularity of Linux is somewhat of a phenomenon in the computer industry. Linux is distinguished by often being the first operating system to support new hardware.

Linux was created in 1991 by Linus Torvalds as a replacement for MINIX. A grass-roots effort then sprang up, whereby many developers across the world volunteered their time to use and enhance it.

The Ubuntu 12.04 distribution of Linux was one of the operating systems used to test the examples in this book. That distribution uses the 3.2.0 version of the Linux operating system kernel.

2.3.5 Mac OS X

Mac OS X is based on entirely different technology than prior versions. The core operating system is called ''Darwin,'' and is based on a combination of the Mach kernel (Accetta et al. [1986]), the FreeBSD operating system, and an object-oriented framework for drivers and other kernel extensions. As of version 10.5, the Intel port of Mac OS X has been certified to be a UNIX system. (For more information on UNIX certification, see `http://www.opengroup.org/certification/idx/unix.html`.)

Mac OS X version 10.6.8 (Darwin 10.8.0) was used as one of the operating systems to test the examples in this book.

2.3.6 Solaris

Solaris is the version of the UNIX System developed by Sun Microsystems (now Oracle). Solaris is based on System V Release 4, but includes more than fifteen years of enhancements from the engineers at Sun Microsystems. It is arguably the only commercially successful SVR4 descendant, and is formally certified to be a UNIX system.

In 2005, Sun Microsystems released most of the Solaris operating system source code to the public as part of the OpenSolaris open source operating system in an attempt to build an external developer community around Solaris.

The Solaris 10 UNIX system was one of the operating systems used to test the examples in this book.

2.3.7 Other UNIX Systems

Other versions of the UNIX system that have been certified in the past include

- AIX, IBM's version of the UNIX System
- HP-UX, Hewlett-Packard's version of the UNIX System
- IRIX, the UNIX System version shipped by Silicon Graphics
- UnixWare, the UNIX System descended from SVR4 sold by SCO

2.4 Relationship of Standards and Implementations

The standards that we've mentioned define a subset of any actual system. The focus of this book is on four real systems: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10. Although only Mac OS X and Solaris can call themselves UNIX systems, all four provide a similar programming environment. Because all four are POSIX compliant to varying degrees, we will also concentrate on the features required by the POSIX.1 standard, noting any differences between POSIX and the actual implementations of these four systems. Those features and routines that are specific to only a particular implementation are clearly marked. We'll also note any features that are required on UNIX systems but are optional on other POSIX-conforming systems.

Be aware that the implementations provide backward compatibility for features in earlier releases, such as SVR3.2 and 4.3BSD. For example, Solaris supports both the POSIX.1 specification for nonblocking I/O (`O_NONBLOCK`) and the traditional System V method (`O_NDELAY`). In this text, we'll use only the POSIX.1 feature, although we'll mention the nonstandard feature that it replaces. Similarly, both SVR3.2 and 4.3BSD provided reliable signals in a way that differs from the POSIX.1 standard. In Chapter 10 we describe only the POSIX.1 signal mechanism.

2.5 Limits

The implementations define many magic numbers and constants. Many of these have been hard coded into programs or were determined using ad hoc techniques. With the various standardization efforts that we've described, more portable methods are now provided to determine these magic numbers and implementation-defined limits, greatly improving the portability of software written for the UNIX environment.

Two types of limits are needed:

1. Compile-time limits (e.g., what's the largest value of a short integer?)
2. Runtime limits (e.g., how many bytes in a filename?)

Compile-time limits can be defined in headers that any program can include at compile time. But runtime limits require the process to call a function to obtain the limit's value.

Additionally, some limits can be fixed on a given implementation—and could therefore be defined statically in a header—yet vary on another implementation and would require a runtime function call. An example of this type of limit is the maximum number of bytes in a filename. Before SVR4, System V historically allowed only 14 bytes in a filename, whereas BSD-derived systems increased this number to 255. Most UNIX System implementations these days support multiple file system types, and each type has its own limit. This is the case of a runtime limit that depends on where in the file system the file in question is located. A filename in the root file system, for example, could have a 14-byte limit, whereas a filename in another file system could have a 255-byte limit.

To solve these problems, three types of limits are provided:

1. Compile-time limits (headers)

2. Runtime limits not associated with a file or directory (the `sysconf` function)
3. Runtime limits that are associated with a file or a directory (the `pathconf` and `fpathconf` functions)

To further confuse things, if a particular runtime limit does not vary on a given system, it can be defined statically in a header. If it is not defined in a header, however, the application must call one of the three `conf` functions (which we describe shortly) to determine its value at runtime.

Name	Description	Minimum acceptable value	Typical value
`CHAR_BIT`	bits in a `char`	8	8
`CHAR_MAX`	max value of `char`	(see later)	127
`CHAR_MIN`	min value of `char`	(see later)	–128
`SCHAR_MAX`	max value of `signed char`	127	127
`SCHAR_MIN`	min value of `signed char`	–127	–128
`UCHAR_MAX`	max value of `unsigned char`	255	255
`INT_MAX`	max value of `int`	32,767	2,147,483,647
`INT_MIN`	min value of `int`	–32,767	–2,147,483,648
`UINT_MAX`	max value of `unsigned int`	65,535	4,294,967,295
`SHRT_MAX`	max value of `short`	32,767	32,767
`SHRT_MIN`	min value of `short`	–32,767	–32,768
`USHRT_MAX`	max value of `unsigned short`	65,535	65,535
`LONG_MAX`	max value of `long`	2,147,483,647	2,147,483,647
`LONG_MIN`	min value of `long`	–2,147,483,647	–2,147,483,648
`ULONG_MAX`	max value of `unsigned long`	4,294,967,295	4,294,967,295
`LLONG_MAX`	max value of `long long`	9,223,372,036,854,775,807	9,223,372,036,854,775,807
`LLONG_MIN`	min value of `long long`	–9,223,372,036,854,775,807	–9,223,372,036,854,775,808
`ULLONG_MAX`	max value of `unsigned long long`	18,446,744,073,709,551,615	18,446,744,073,709,551,615
`MB_LEN_MAX`	max number of bytes in a multibyte character constant	1	6

Figure 2.6 Sizes of integral values from `<limits.h>`

2.5.1 ISO C Limits

All of the compile-time limits defined by ISO C are defined in the file `<limits.h>` (see Figure 2.6). These constants don't change in a given system. The third column in Figure 2.6 shows the minimum acceptable values from the ISO C standard. This allows for a system with 16-bit integers using one's-complement arithmetic. The fourth column shows the values from a Linux system with 32-bit integers using two's-complement arithmetic. Note that none of the unsigned data types has a minimum value, as this value must be 0 for an unsigned data type. On a 64-bit system, the values for `long` integer maximums match the maximum values for `long long` integers.

One difference that we will encounter is whether a system provides signed or unsigned character values. From the fourth column in Figure 2.6, we see that this particular system uses signed characters. We see that CHAR_MIN equals SCHAR_MIN and that CHAR_MAX equals SCHAR_MAX. If the system uses unsigned characters, we would have CHAR_MIN equal to 0 and CHAR_MAX equal to UCHAR_MAX.

The floating-point data types in the header <float.h> have a similar set of definitions. Anyone doing serious floating-point work should examine this file.

Although the ISO C standard specifies minimum acceptable values for integral data types, POSIX.1 makes extensions to the C standard. To conform to POSIX.1, an implementation must support a minimum value of 2,147,483,647 for INT_MAX, −2,147,483,647 for INT_MIN, and 4,294,967,295 for UINT_MAX. Because POSIX.1 requires implementations to support an 8-bit char, CHAR_BIT must be 8, SCHAR_MIN must be −128, SCHAR_MAX must be 127, and UCHAR_MAX must be 255.

Another ISO C constant that we'll encounter is FOPEN_MAX, the minimum number of standard I/O streams that the implementation guarantees can be open at once. This constant is found in the <stdio.h> header, and its minimum value is 8. The POSIX.1 value STREAM_MAX, if defined, must have the same value as FOPEN_MAX.

ISO C also defines the constant TMP_MAX in <stdio.h>. It is the maximum number of unique filenames generated by the tmpnam function. We'll have more to say about this constant in Section 5.13.

Although ISO C defines the constant FILENAME_MAX, we avoid using it, because POSIX.1 provides better alternatives (NAME_MAX and PATH_MAX). We'll see these constants shortly.

Figure 2.7 shows the values of FILENAME_MAX, FOPEN_MAX, and TMP_MAX on the four platforms we discuss in this book.

Limit	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
FOPEN_MAX	20	16	20	20
TMP_MAX	308,915,776	238,328	308,915,776	17,576
FILENAME_MAX	1024	4096	1024	1024

Figure 2.7 ISO limits on various platforms

2.5.2 POSIX Limits

POSIX.1 defines numerous constants that deal with implementation limits of the operating system. Unfortunately, this is one of the more confusing aspects of POSIX.1. Although POSIX.1 defines numerous limits and constants, we'll concern ourselves with only the ones that affect the base POSIX.1 interfaces. These limits and constants are divided into the following seven categories:

1. Numerical limits: LONG_BIT, SSIZE_MAX, and WORD_BIT
2. Minimum values: the 25 constants in Figure 2.8
3. Maximum value: _POSIX_CLOCKRES_MIN

4. Runtime increasable values: CHARCLASS_NAME_MAX, COLL_WEIGHTS_MAX, LINE_MAX, NGROUPS_MAX, and RE_DUP_MAX
5. Runtime invariant values, possibly indeterminate: the 17 constants in Figure 2.9 (plus an additional four constants introduced in Section 12.2 and three constants introduced in Section 14.5)
6. Other invariant values: NL_ARGMAX, NL_MSGMAX, NL_SETMAX, and NL_TEXTMAX
7. Pathname variable values: FILESIZEBITS, LINK_MAX, MAX_CANON, MAX_INPUT, NAME_MAX, PATH_MAX, PIPE_BUF, and SYMLINK_MAX

Name	Description: minimum acceptable value for maximum ...	Value
_POSIX_ARG_MAX	length of arguments to exec functions	4,096
_POSIX_CHILD_MAX	number of child processes at a time per real user ID	25
_POSIX_DELAYTIMER_MAX	number of timer expiration overruns	32
_POSIX_HOST_NAME_MAX	length of a host name as returned by gethostname	255
_POSIX_LINK_MAX	number of links to a file	8
_POSIX_LOGIN_NAME_MAX	length of a login name	9
_POSIX_MAX_CANON	number of bytes on a terminal's canonical input queue	255
_POSIX_MAX_INPUT	space available on a terminal's input queue	255
_POSIX_NAME_MAX	number of bytes in a filename, not including the terminating null	14
_POSIX_NGROUPS_MAX	number of simultaneous supplementary group IDs per process	8
_POSIX_OPEN_MAX	maximum number of open files per process	20
_POSIX_PATH_MAX	number of bytes in a pathname, including the terminating null	256
_POSIX_PIPE_BUF	number of bytes that can be written atomically to a pipe	512
_POSIX_RE_DUP_MAX	number of repeated occurrences of a basic regular expression permitted by the regexec and regcomp functions when using the interval notation \{m,n\}	255
_POSIX_RTSIG_MAX	number of real-time signal numbers reserved for applications	8
_POSIX_SEM_NSEMS_MAX	number of semaphores a process can have in use at one time	256
_POSIX_SEM_VALUE_MAX	value a semaphore can hold	32,767
_POSIX_SIGQUEUE_MAX	number of queued signals a process can send and have pending	32
_POSIX_SSIZE_MAX	value that can be stored in ssize_t object	32,767
_POSIX_STREAM_MAX	number of standard I/O streams a process can have open at once	8
_POSIX_SYMLINK_MAX	number of bytes in a symbolic link	255
_POSIX_SYMLOOP_MAX	number of symbolic links that can be traversed during pathname resolution	8
_POSIX_TIMER_MAX	number of timers per process	32
_POSIX_TTY_NAME_MAX	length of a terminal device name, including the terminating null	9
_POSIX_TZNAME_MAX	number of bytes for the name of a time zone	6

Figure 2.8 POSIX.1 minimum values from <limits.h>

Of these limits and constants, some may be defined in <limits.h>, and others may or may not be defined, depending on certain conditions. We describe the limits and constants that may or may not be defined in Section 2.5.4, when we describe the sysconf, pathconf, and fpathconf functions. The 25 minimum values are shown in Figure 2.8.

These minimum values do not change from one system to another. They specify the most restrictive values for these features. A conforming POSIX.1 implementation must provide values that are at least this large. This is why they are called minimums, although their names all contain MAX. Also, to ensure portability, a strictly conforming application must not require a larger value. We describe what each of these constants refers to as we proceed through the text.

A strictly conforming POSIX application is different from an application that is merely POSIX conforming. A POSIX-conforming application uses only interfaces defined in IEEE Standard 1003.1-2008. A strictly conforming POSIX application must meet further restrictions, such as not relying on any undefined behavior, not using any obsolescent interfaces, and not requiring values of constants larger than the minimums shown in Figure 2.8.

Name	Description	Minimum acceptable value
`ARG_MAX`	maximum length of arguments to `exec` functions	`_POSIX_ARG_MAX`
`ATEXIT_MAX`	maximum number of functions that can be registered with the `atexit` function	32
`CHILD_MAX`	maximum number of child processes per real user ID	`_POSIX_CHILD_MAX`
`DELAYTIMER_MAX`	maximum number of timer expiration overruns	`_POSIX_DELAYTIMER_MAX`
`HOST_NAME_MAX`	maximum length of a host name as returned by `gethostname`	`_POSIX_HOST_NAME_MAX`
`LOGIN_NAME_MAX`	maximum length of a login name	`_POSIX_LOGIN_NAME_MAX`
`OPEN_MAX`	one more than the maximum value assigned to a newly created file descriptor	`_POSIX_OPEN_MAX`
`PAGESIZE`	system memory page size, in bytes	1
`RTSIG_MAX`	maximum number of real-time signals reserved for application use	`_POSIX_RTSIG_MAX`
`SEM_NSEMS_MAX`	maximum number of semaphores a process can use	`_POSIX_SEM_NSEMS_MAX`
`SEM_VALUE_MAX`	maximum value of a semaphore	`_POSIX_SEM_VALUE_MAX`
`SIGQUEUE_MAX`	maximum number of signals that can be queued for a process	`_POSIX_SIGQUEUE_MAX`
`STREAM_MAX`	maximum number of standard I/O streams a process can have open at once	`_POSIX_STREAM_MAX`
`SYMLOOP_MAX`	number of symbolic links that can be traversed during pathname resolution	`_POSIX_SYMLOOP_MAX`
`TIMER_MAX`	maximum number of timers per process	`_POSIX_TIMER_MAX`
`TTY_NAME_MAX`	length of a terminal device name, including the terminating null	`_POSIX_TTY_NAME_MAX`
`TZNAME_MAX`	number of bytes for the name of a time zone	`_POSIX_TZNAME_MAX`

Figure 2.9 POSIX.1 runtime invariant values from `<limits.h>`

Unfortunately, some of these invariant minimum values are too small to be of practical use. For example, most UNIX systems today provide far more than 20 open files per process. Also, the minimum limit of 256 for `_POSIX_PATH_MAX` is too small. Pathnames can exceed this limit. This means that we can't use the two constants `_POSIX_OPEN_MAX` and `_POSIX_PATH_MAX` as array sizes at compile time.

Each of the 25 invariant minimum values in Figure 2.8 has an associated implementation value whose name is formed by removing the _POSIX_ prefix from the name in Figure 2.8. The names without the leading _POSIX_ were intended to be the actual values that a given implementation supports. (These 25 implementation values are from items 1, 4, 5, and 7 from our list earlier in this section: 2 of the runtime increasable values, 15 of the runtime invariant values, and 7 of the pathname variable values, along with SSIZE_MAX from the numeric values.) The problem is that not all of the 25 implementation values are guaranteed to be defined in the <limits.h> header.

For example, a particular value may not be included in the header if its actual value for a given process depends on the amount of memory on the system. If the values are not defined in the header, we can't use them as array bounds at compile time. To determine the actual implementation value at runtime, POSIX.1 decided to provide three functions for us to call—sysconf, pathconf, and fpathconf. There is still a problem, however, because some of the values are defined by POSIX.1 as being possibly "indeterminate" (logically infinite). This means that the value has no practical upper bound. On Solaris, for example, the number of functions you can register with atexit to be run when a process ends is limited only by the amount of memory on the system. Thus ATEXIT_MAX is considered indeterminate on Solaris. We'll return to this problem of indeterminate runtime limits in Section 2.5.5.

2.5.3 XSI Limits

The XSI option also defines constants representing implementation limits. They include:

1. Minimum values: the five constants in Figure 2.10
2. Runtime invariant values, possibly indeterminate: IOV_MAX and PAGE_SIZE

The minimum values are listed in Figure 2.10. The last two illustrate the situation in which the POSIX.1 minimums were too small—presumably to allow for embedded POSIX.1 implementations—so symbols with larger minimum values were added for XSI-conforming systems.

Name	Description	Minimum acceptable value	Typical value
NL_LANGMAX	maximum number of bytes in LANG environment variable	14	14
NZERO	default process priority	20	20
_XOPEN_IOV_MAX	maximum number of iovec structures that can be used with readv or writev	16	16
_XOPEN_NAME_MAX	number of bytes in a filename	255	255
_XOPEN_PATH_MAX	number of bytes in a pathname	1,024	1,024

Figure 2.10 XSI minimum values from <limits.h>

2.5.4 sysconf, pathconf, and fpathconf Functions

We've listed various minimum values that an implementation must support, but how do we find out the limits that a particular system actually supports? As we mentioned earlier, some of these limits might be available at compile time; others must be determined at runtime. We've also mentioned that some limits don't change in a given system, whereas others can change because they are associated with a file or directory. The runtime limits are obtained by calling one of the following three functions.

```
#include <unistd.h>

long sysconf(int name);

long pathconf(const char *pathname, int name);

long fpathconf(int fd, int name);
```

All three return: corresponding value if OK, −1 on error (see later)

The difference between the last two functions is that one takes a pathname as its argument and the other takes a file descriptor argument.

Figure 2.11 lists the *name* arguments that `sysconf` uses to identify system limits. Constants beginning with `_SC_` are used as arguments to `sysconf` to identify the runtime limit. Figure 2.12 lists the *name* arguments that are used by `pathconf` and `fpathconf` to identify system limits. Constants beginning with `_PC_` are used as arguments to `pathconf` and `fpathconf` to identify the runtime limit.

We need to look in more detail at the different return values from these three functions.

1. All three functions return −1 and set `errno` to `EINVAL` if the *name* isn't one of the appropriate constants. The third column in Figures 2.11 and 2.12 lists the limit constants we'll deal with throughout the rest of this book.
2. Some *name*s can return either the value of the variable (a return value ≥ 0) or an indication that the value is indeterminate. An indeterminate value is indicated by returning −1 and not changing the value of `errno`.
3. The value returned for `_SC_CLK_TCK` is the number of clock ticks per second, for use with the return values from the `times` function (Section 8.17).

Some restrictions apply to the `pathconf` *pathname* argument and the `fpathconf` *fd* argument. If any of these restrictions isn't met, the results are undefined.

1. The referenced file for `_PC_MAX_CANON` and `_PC_MAX_INPUT` must be a terminal file.
2. The referenced file for `_PC_LINK_MAX` and `_PC_TIMESTAMP_RESOLUTION` can be either a file or a directory. If the referenced file is a directory, the return value applies to the directory itself, not to the filename entries within the directory.
3. The referenced file for `_PC_FILESIZEBITS` and `_PC_NAME_MAX` must be a directory. The return value applies to filenames within the directory.

Name of limit	Description	*name* argument
ARG_MAX	maximum length, in bytes, of arguments to the exec functions	_SC_ARG_MAX
ATEXIT_MAX	maximum number of functions that can be registered with the atexit function	_SC_ATEXIT_MAX
CHILD_MAX	maximum number of processes per real user ID	_SC_CHILD_MAX
clock ticks/second	number of clock ticks per second	_SC_CLK_TCK
COLL_WEIGHTS_MAX	maximum number of weights that can be assigned to an entry of the LC_COLLATE order keyword in the locale definition file	_SC_COLL_WEIGHTS_MAX
DELAYTIMER_MAX	maximum number of timer expiration overruns	_SC_DELAYTIMER_MAX
HOST_NAME_MAX	maximum length of a host name as returned by gethostname	_SC_HOST_NAME_MAX
IOV_MAX	maximum number of iovec structures that can be used with readv or writev	_SC_IOV_MAX
LINE_MAX	maximum length of a utility's input line	_SC_LINE_MAX
LOGIN_NAME_MAX	maximum length of a login name	_SC_LOGIN_NAME_MAX
NGROUPS_MAX	maximum number of simultaneous supplementary process group IDs per process	_SC_NGROUPS_MAX
OPEN_MAX	one more than the maximum value assigned to a newly created file descriptor	_SC_OPEN_MAX
PAGESIZE	system memory page size, in bytes	_SC_PAGESIZE
PAGE_SIZE	system memory page size, in bytes	_SC_PAGE_SIZE
RE_DUP_MAX	number of repeated occurrences of a basic regular expression permitted by the regexec and regcomp functions when using the interval notation \{m,n\}	_SC_RE_DUP_MAX
RTSIG_MAX	maximum number of real-time signals reserved for application use	_SC_RTSIG_MAX
SEM_NSEMS_MAX	maximum number of semaphores a process can use at one time	_SC_SEM_NSEMS_MAX
SEM_VALUE_MAX	maximum value of a semaphore	_SC_SEM_VALUE_MAX
SIGQUEUE_MAX	maximum number of signals that can be queued for a process	_SC_SIGQUEUE_MAX
STREAM_MAX	maximum number of standard I/O streams per process at any given time; if defined, it must have the same value as FOPEN_MAX	_SC_STREAM_MAX
SYMLOOP_MAX	number of symbolic links that can be traversed during pathname resolution	_SC_SYMLOOP_MAX
TIMER_MAX	maximum number of timers per process	_SC_TIMER_MAX
TTY_NAME_MAX	length of a terminal device name, including the terminating null	_SC_TTY_NAME_MAX
TZNAME_MAX	maximum number of bytes for a time zone name	_SC_TZNAME_MAX

Figure 2.11 Limits and *name* arguments to sysconf

4. The referenced file for `_PC_PATH_MAX` must be a directory. The value returned is the maximum length of a relative pathname when the specified directory is the working directory. (Unfortunately, this isn't the real maximum length of an absolute pathname, which is what we want to know. We'll return to this problem in Section 2.5.5.)

Name of limit	Description	*name* argument
FILESIZEBITS	minimum number of bits needed to represent, as a signed integer value, the maximum size of a regular file allowed in the specified directory	_PC_FILESIZEBITS
LINK_MAX	maximum value of a file's link count	_PC_LINK_MAX
MAX_CANON	maximum number of bytes on a terminal's canonical input queue	_PC_MAX_CANON
MAX_INPUT	number of bytes for which space is available on terminal's input queue	_PC_MAX_INPUT
NAME_MAX	maximum number of bytes in a filename (does not include a null at end)	_PC_NAME_MAX
PATH_MAX	maximum number of bytes in a relative pathname, including the terminating null	_PC_PATH_MAX
PIPE_BUF	maximum number of bytes that can be written atomically to a pipe	_PC_PIPE_BUF
_POSIX_TIMESTAMP_RESOLUTION	resolution in nanoseconds for file timestamps	_PC_TIMESTAMP_RESOLUTION
SYMLINK_MAX	number of bytes in a symbolic link	_PC_SYMLINK_MAX

Figure 2.12 Limits and *name* arguments to `pathconf` and `fpathconf`

5. The referenced file for `_PC_PIPE_BUF` must be a pipe, FIFO, or directory. In the first two cases (pipe or FIFO), the return value is the limit for the referenced pipe or FIFO. For the other case (a directory), the return value is the limit for any FIFO created in that directory.
6. The referenced file for `_PC_SYMLINK_MAX` must be a directory. The value returned is the maximum length of the string that a symbolic link in that directory can contain.

Example

The `awk`(1) program shown in Figure 2.13 builds a C program that prints the value of each `pathconf` and `sysconf` symbol.

```
#!/usr/bin/awk -f
BEGIN   {
    printf("#include \"apue.h\"\n")
    printf("#include <errno.h>\n")
    printf("#include <limits.h>\n")
    printf("\n")
    printf("static void pr_sysconf(char *, int);\n")
    printf("static void pr_pathconf(char *, char *, int);\n")
    printf("\n")
    printf("int\n")
    printf("main(int argc, char *argv[])\n")
```

```
        printf("{\n")
        printf("\tif (argc != 2)\n")
        printf("\t\terr_quit(\"usage: a.out <dirname>\");\n\n")
        FS="\t+"
        while (getline <"sysconf.sym" > 0) {
            printf("#ifdef %s\n", $1)
            printf("\tprintf(\"%s defined to be %%ld\\n\", (long)%s+0);\n",
                 $1, $1)
            printf("#else\n")
            printf("\tprintf(\"no symbol for %s\\n\");\n", $1)
            printf("#endif\n")
            printf("#ifdef %s\n", $2)
            printf("\tpr_sysconf(\"%s =\", %s);\n", $1, $2)
            printf("#else\n")
            printf("\tprintf(\"no symbol for %s\\n\");\n", $2)
            printf("#endif\n")
        }
        close("sysconf.sym")
        while (getline <"pathconf.sym" > 0) {
            printf("#ifdef %s\n", $1)
            printf("\tprintf(\"%s defined to be %%ld\\n\", (long)%s+0);\n",
                $1, $1)
            printf("#else\n")
            printf("\tprintf(\"no symbol for %s\\n\");\n", $1)
            printf("#endif\n")
            printf("#ifdef %s\n", $2)
            printf("\tpr_pathconf(\"%s =\", argv[1], %s);\n", $1, $2)
            printf("#else\n")
            printf("\tprintf(\"no symbol for %s\\n\");\n", $2)
            printf("#endif\n")
        }
        close("pathconf.sym")
        exit
    }
    END {
        printf("\texit(0);\n")
        printf("}\n\n")
        printf("static void\n")
        printf("pr_sysconf(char *mesg, int name)\n")
        printf("{\n")
        printf("\tlong  val;\n\n")
        printf("\tfputs(mesg, stdout);\n")
        printf("\terrno = 0;\n")
        printf("\tif ((val = sysconf(name)) < 0) {\n")
        printf("\t\tif (errno != 0) {\n")
        printf("\t\t\tif (errno == EINVAL)\n")
        printf("\t\t\t\tfputs(\" (not supported)\\n\", stdout);\n")
        printf("\t\t\telse\n")
        printf("\t\t\t\terr_sys(\"sysconf error\");\n")
        printf("\t\t} else {\n")
        printf("\t\t\tfputs(\" (no limit)\\n\", stdout);\n")
        printf("\t\t}\n")
        printf("\t} else {\n")
        printf("\t\tprintf(\" %%ld\\n\", val);\n")
        printf("\t}\n")
        printf("}\n\n")
        printf("static void\n")
        printf("pr_pathconf(char *mesg, char *path, int name)\n")
        printf("{\n")
        printf("\tlong  val;\n")
        printf("\n")
        printf("\tfputs(mesg, stdout);\n")
        printf("\terrno = 0;\n")
        printf("\tif ((val = pathconf(path, name)) < 0) {\n")
        printf("\t\tif (errno != 0) {\n")
        printf("\t\t\tif (errno == EINVAL)\n")
        printf("\t\t\t\tfputs(\" (not supported)\\n\", stdout);\n")
        printf("\t\t\telse\n")
        printf("\t\t\t\terr_sys(\"pathconf error, path = %%s\", path);\n")
        printf("\t\t} else {\n")
        printf("\t\t\tfputs(\" (no limit)\\n\", stdout);\n")
        printf("\t\t}\n")
        printf("\t} else {\n")
        printf("\t\tprintf(\" %%ld\\n\", val);\n")
        printf("\t}\n")
        printf("}\n")
    }
```

Figure 2.13 Build C program to print all supported configuration limits

The awk program reads two input files—pathconf.sym and sysconf.sym—that contain lists of the limit name and symbol, separated by tabs. All symbols are not defined on every platform, so the awk program surrounds each call to pathconf and sysconf with the necessary #ifdef statements.

For example, the awk program transforms a line in the input file that looks like

```
NAME_MAX        _PC_NAME_MAX
```

into the following C code:

```
#ifdef NAME_MAX
    printf("NAME_MAX is defined to be %d\n", NAME_MAX+0);
#else
    printf("no symbol for NAME_MAX\n");
#endif
#ifdef _PC_NAME_MAX
    pr_pathconf("NAME_MAX =", argv[1], _PC_NAME_MAX);
#else
    printf("no symbol for _PC_NAME_MAX\n");
#endif
```

The program in Figure 2.14, generated by the awk program, prints all these limits, handling the case in which a limit is not defined.

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

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <dirname>");

#ifdef ARG_MAX
    printf("ARG_MAX defined to be %ld\n", (long)ARG_MAX+0);
#else
    printf("no symbol for ARG_MAX\n");
#endif
#ifdef _SC_ARG_MAX
    pr_sysconf("ARG_MAX =", _SC_ARG_MAX);
#else
    printf("no symbol for _SC_ARG_MAX\n");
#endif

/* similar processing for all the rest of the sysconf symbols... */

#ifdef MAX_CANON
    printf("MAX_CANON defined to be %ld\n", (long)MAX_CANON+0);
#else
    printf("no symbol for MAX_CANON\n");
#endif
#ifdef _PC_MAX_CANON
    pr_pathconf("MAX_CANON =", argv[1], _PC_MAX_CANON);
#else
    printf("no symbol for _PC_MAX_CANON\n");
#endif

/* similar processing for all the rest of the pathconf symbols... */

    exit(0);
}
static void
pr_sysconf(char *mesg, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = sysconf(name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs(" (not supported)\n", stdout);
            else
```

```
                err_sys("sysconf error");
        } else {
            fputs(" (no limit)\n", stdout);
        }
    } else {
        printf(" %ld\n", val);
    }
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = pathconf(path, name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs(" (not supported)\n", stdout);
            else
                err_sys("pathconf error, path = %s", path);
        } else {
            fputs(" (no limit)\n", stdout);
        }
    } else {
        printf(" %ld\n", val);
    }
}
```

Figure 2.14 Print all possible `sysconf` and `pathconf` values

Figure 2.15 summarizes the results from Figure 2.14 for the four systems we discuss in this book. The entry “no symbol” means that the system doesn’t provide a corresponding `_SC` or `_PC` symbol to query the value of the constant. Thus the limit is undefined in this case. In contrast, the entry “unsupported” means that the symbol is defined by the system but unrecognized by the `sysconf` or `pathconf` functions. The entry “no limit” means that the system defines no limit for the constant, but this doesn’t mean that the limit is infinite; it just means that the limit is indeterminite.

Beware that some limits are reported incorrectly. For example, on Linux, `SYMLOOP_MAX` is reportedly unlimited, but an examination of the source code reveals that there is actually a hard-coded limit of 40 for the number of consecutive symbolic links traversed in the absence of a loop (see the `follow_link` function in `fs/namei.c`).

Another potential source of inaccuracy in Linux is that the `pathconf` and `fpathconf` functions are implemented in the C library. The configuration limits returned by these functions depend on the underlying file system type, so if your file system is unknown to the C library, the functions return an educated guess.

We’ll see in Section 4.14 that `UFS` is the SVR4 implementation of the Berkeley fast file system. PCFS is the MS-DOS FAT file system implementation for Solaris. □

Limit	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	
				`UFS` file system	`PCFS` file system
`ARG_MAX`	262,144	2,097,152	262,144	2,096,640	2,096,640
`ATEXIT_MAX`	32	2,147,483,647	2,147,483,647	no limit	no limit
`CHARCLASS_NAME_MAX`	no symbol	2,048	14	14	14
`CHILD_MAX`	1,760	47,211	266	8,021	8,021
clock ticks/second	128	100	100	100	100
`COLL_WEIGHTS_MAX`	0	255	2	10	10
`FILESIZEBITS`	64	64	64	41	unsupported
`HOST_NAME_MAX`	255	64	255	255	255
`IOV_MAX`	1,024	1,024	1024	16	16
`LINE_MAX`	2,048	2,048	2,048	2,048	2,048
`LINK_MAX`	32,767	65,000	32,767	32,767	1
`LOGIN_NAME_MAX`	17	256	255	9	9
`MAX_CANON`	255	255	1,024	256	256
`MAX_INPUT`	255	255	1,024	512	512
`NAME_MAX`	255	255	255	255	8
`NGROUPS_MAX`	1,023	65,536	16	16	16
`OPEN_MAX`	3,520	1,024	256	256	256
`PAGESIZE`	4,096	4,096	4,096	8,192	8,192
`PAGE_SIZE`	4,096	4,096	4,096	8,192	8,192
`PATH_MAX`	1,024	4,096	1,024	1,024	1,024
`PIPE_BUF`	512	4,096	512	5,120	5,120
`RE_DUP_MAX`	255	32,767	255	255	255
`STREAM_MAX`	3,520	16	20	256	256
`SYMLINK_MAX`	1,024	no limit	255	1,024	1,024
`SYMLOOP_MAX`	32	no limit	32	20	20
`TTY_NAME_MAX`	255	32	255	128	128
`TZNAME_MAX`	255	6	255	no limit	no limit

Figure 2.15 Examples of configuration limits

2.5.5 Indeterminate Runtime Limits

We mentioned that some of the limits can be indeterminate. The problem we encounter is that if these limits aren't defined in the `<limits.h>` header, we can't use them at compile time. But they might not be defined at runtime if their value is indeterminate! Let's look at two specific cases: allocating storage for a pathname and determining the number of file descriptors.

Pathnames

Many programs need to allocate storage for a pathname. Typically, the storage has been allocated at compile time, and various magic numbers—few of which are the correct value—have been used by different programs as the array size: 256, 512, 1024, or the standard I/O constant `BUFSIZ`. The 4.3BSD constant `MAXPATHLEN` in the header `<sys/param.h>` is the correct value, but many 4.3BSD applications didn't use it.

POSIX.1 tries to help with the PATH_MAX value, but if this value is indeterminate, we're still out of luck. Figure 2.16 shows a function that we'll use throughout this text to allocate storage dynamically for a pathname.

If the constant PATH_MAX is defined in <limits.h>, then we're all set. If it's not, then we need to call pathconf. The value returned by pathconf is the maximum size of a relative pathname when the first argument is the working directory, so we specify the root as the first argument and add 1 to the result. If pathconf indicates that PATH_MAX is indeterminate, we have to punt and just guess a value.

Versions of POSIX.1 prior to 2001 were unclear as to whether PATH_MAX included a null byte at the end of the pathname. If the operating system implementation conforms to one of these prior versions and doesn't conform to any version of the Single UNIX Specification (which *does* require the terminating null byte to be included), we need to add 1 to the amount of memory we allocate for a pathname, just to be on the safe side.

The correct way to handle the case of an indeterminate result depends on how the allocated space is being used. If we are allocating space for a call to getcwd, for example—to return the absolute pathname of the current working directory; see Section 4.23—and if the allocated space is too small, an error is returned and errno is set to ERANGE. We could then increase the allocated space by calling realloc (see Section 7.8 and Exercise 4.16) and try again. We could keep doing this until the call to getcwd succeeded.

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

#ifdef  PATH_MAX
static long pathmax = PATH_MAX;
#else
static long pathmax = 0;
#endif

static long posix_version = 0;
static long xsi_version = 0;

/* If PATH_MAX is indeterminate, no guarantee this is adequate */
#define PATH_MAX_GUESS  1024

char *
path_alloc(size_t *sizep) /* also return allocated size, if nonnull */
{
    char    *ptr;
    size_t  size;

    if (posix_version == 0)
        posix_version = sysconf(_SC_VERSION);

    if (xsi_version == 0)
        xsi_version = sysconf(_SC_XOPEN_VERSION);

    if (pathmax == 0) {     /* first time through */
```

```
        errno = 0;
        if ((pathmax = pathconf("/", _PC_PATH_MAX)) < 0) {
            if (errno == 0)
                pathmax = PATH_MAX_GUESS;   /* it’s indeterminate */
            else
                err_sys("pathconf error for _PC_PATH_MAX");
        } else {
            pathmax++;      /* add one since it’s relative to root */
        }
    }

    /*
     * Before POSIX.1-2001, we aren’t guaranteed that PATH_MAX includes
     * the terminating null byte.  Same goes for XPG3.
     */
    if ((posix_version < 200112L) && (xsi_version < 4))
        size = pathmax + 1;
    else
        size = pathmax;

    if ((ptr = malloc(size)) == NULL)
        err_sys("malloc error for pathname");

    if (sizep != NULL)
        *sizep = size;
    return(ptr);
}
```

Figure 2.16 Dynamically allocate space for a pathname

Maximum Number of Open Files

A common sequence of code in a daemon process—a process that runs in the background, not connected to a terminal—is one that closes all open files. Some programs have the following code sequence, assuming the constant `NOFILE` was defined in the `<sys/param.h>` header:

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

for (i = 0; i < NOFILE; i++)
    close(i);
```

Other programs use the constant `_NFILE` that some versions of `<stdio.h>` provide as the upper limit. Some hard code the upper limit as 20. However, none of these approaches is portable.

We would hope to use the POSIX.1 value `OPEN_MAX` to determine this value portably, but if the value is indeterminate, we still have a problem. If we wrote the following code and if `OPEN_MAX` was indeterminate, the loop would never execute, since `sysconf` would return –1:

```
#include  <unistd.h>

for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
    close(i);
```

Our best option in this case is just to close all descriptors up to some arbitrary limit—say, 256. We show this technique in Figure 2.17. As with our pathname example, this strategy is not guaranteed to work for all cases, but it's the best we can do without using a more exotic approach.

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

#ifdef  OPEN_MAX
static long openmax = OPEN_MAX;
#else
static long openmax = 0;
#endif

/*
 * If OPEN_MAX is indeterminate, this might be inadequate.
 */
#define OPEN_MAX_GUESS  256

long
open_max(void)
{
    if (openmax == 0) {     /* first time through */
        errno = 0;
        if ((openmax = sysconf(_SC_OPEN_MAX)) < 0) {
            if (errno == 0)
                openmax = OPEN_MAX_GUESS;   /* it's indeterminate */
            else
                err_sys("sysconf error for _SC_OPEN_MAX");
        }
    }
    return(openmax);
}
```

Figure 2.17 Determine the number of file descriptors

We might be tempted to call `close` until we get an error return, but the error return from `close` (`EBADF`) doesn't distinguish between an invalid descriptor and a descriptor that wasn't open. If we tried this technique and descriptor 9 was not open but descriptor 10 was, we would stop on 9 and never close 10. The `dup` function (Section 3.12) does return a specific error when `OPEN_MAX` is exceeded, but duplicating a descriptor a couple of hundred times is an extreme way to determine this value.

Some implementations will return `LONG_MAX` for limit values that are effectively unlimited. Such is the case with the Linux limit for `ATEXIT_MAX` (see Figure 2.15). This isn't a good idea, because it can cause programs to behave badly.

For example, we can use the ulimit command built into the Bourne-again shell to change the maximum number of files our processes can have open at one time. This generally requires special (superuser) privileges if the limit is to be effectively unlimited. But once set to infinite, sysconf will report LONG_MAX as the limit for OPEN_MAX. A program that relies on this value as the upper bound of file descriptors to close, as shown in Figure 2.17, will waste a lot of time trying to close 2,147,483,647 file descriptors, most of which aren't even in use.

Systems that support the XSI option in the Single UNIX Specification will provide the getrlimit(2) function (Section 7.11). It can be used to return the maximum number of descriptors that a process can have open. With it, we can detect that there is no configured upper bound to the number of open files our processes can open, so we can avoid this problem.

The OPEN_MAX value is called runtime invariant by POSIX, meaning that its value should not change during the lifetime of a process. But on systems that support the XSI option, we can call the setrlimit(2) function (Section 7.11) to change this value for a running process. (This value can also be changed from the C shell with the limit command, and from the Bourne, Bourne-again, Debian Almquist, and Korn shells with the ulimit command.) If our system supports this functionality, we could change the function in Figure 2.17 to call sysconf every time it is called, not just the first time.

2.6 Options

We saw the list of POSIX.1 options in Figure 2.5 and discussed XSI option groups in Section 2.2.3. If we are to write portable applications that depend on any of these optionally supported features, we need a portable way to determine whether an implementation supports a given option.

Just as with limits (Section 2.5), POSIX.1 defines three ways to do this.

1. Compile-time options are defined in <unistd.h>.
2. Runtime options that are not associated with a file or a directory are identified with the sysconf function.
3. Runtime options that are associated with a file or a directory are discovered by calling either the pathconf or the fpathconf function.

The options include the symbols listed in the third column of Figure 2.5, as well as the symbols listed in Figures 2.19 and 2.18. If the symbolic constant is not defined, we must use sysconf, pathconf, or fpathconf to determine whether the option is supported. In this case, the *name* argument to the function is formed by replacing the _POSIX at the beginning of the symbol with _SC or _PC. For constants that begin with _XOPEN, the *name* argument is formed by prepending the string with _SC or _PC. For example, if the constant _POSIX_RAW_SOCKETS is undefined, we can call sysconf with the *name* argument set to _SC_RAW_SOCKETS to determine whether the platform supports the raw sockets option. If the constant _XOPEN_UNIX is undefined, we can call sysconf with the *name* argument set to _SC_XOPEN_UNIX to determine whether the platform supports the XSI option interfaces.

For each option, we have three possibilities for a platform's support status.

1. If the symbolic constant is either undefined or defined to have the value −1, then the corresponding option is unsupported by the platform at compile time. It is possible to run an old application on a newer system where the option *is* supported, so a runtime check might indicate the option is supported even though the option wasn't supported at the time the application was compiled.
2. If the symbolic constant is defined to be greater than zero, then the corresponding option is supported.
3. If the symbolic constant is defined to be equal to zero, then we must call `sysconf`, `pathconf`, or `fpathconf` to determine whether the option is supported.

The symbolic constants used with `pathconf` and `fpathconf` are summarized in Figure 2.18. Figure 2.19 summarizes the nonobsolete options and their symbolic constants that can be used with `sysconf`, in addition to those listed in Figure 2.5. Note that we omit options associated with utility commands.

As with the system limits, there are several points to note regarding how options are treated by `sysconf`, `pathconf`, and `fpathconf`.

1. The value returned for `_SC_VERSION` indicates the four-digit year and two-digit month of the standard. This value can be 198808L, 199009L, 199506L, or some other value for a later version of the standard. The value associated with Version 3 of the Single UNIX Specification is 200112L (the 2001 edition of POSIX.1). The value associated with Version 4 of the Single UNIX Specification (the 2008 edition of POSIX.1) is 200809L.
2. The value returned for `_SC_XOPEN_VERSION` indicates the version of the XSI that the system supports. The value associated with Version 3 of the Single UNIX Specification is 600. The value associated with Version 4 of the Single UNIX Specification (the 2008 edition of POSIX.1) is 700.
3. The values `_SC_JOB_CONTROL`, `_SC_SAVED_IDS`, and `_PC_VDISABLE` no longer represent optional features. Although XPG4 and prior versions of the Single UNIX Specification required that these features be supported, Version 3 of the Single UNIX Specification is the earliest version where these features are no longer optional in POSIX.1. These symbols are retained for backward compatibility.
4. Platforms conforming to POSIX.1-2008 are also required to support the following options:

   - `_POSIX_ASYNCHRONOUS_IO`
   - `_POSIX_BARRIERS`
   - `_POSIX_CLOCK_SELECTION`
   - `_POSIX_MAPPED_FILES`
   - `_POSIX_MEMORY_PROTECTION`

- _POSIX_READER_WRITER_LOCKS
- _POSIX_REALTIME_SIGNALS
- _POSIX_SEMAPHORES
- _POSIX_SPIN_LOCKS
- _POSIX_THREAD_SAFE_FUNCTIONS
- _POSIX_THREADS
- _POSIX_TIMEOUTS
- _POSIX_TIMERS

These constants are defined to have the value 200809L. Their corresponding _SC symbols are also retained for backward compatibility.

5. _PC_CHOWN_RESTRICTED and _PC_NO_TRUNC return −1 without changing errno if the feature is not supported for the specified *pathname* or *fd*. On all POSIX-conforming systems, the return value will be greater than zero (indicating that the feature is supported).
6. The referenced file for _PC_CHOWN_RESTRICTED must be either a file or a directory. If it is a directory, the return value indicates whether this option applies to files within that directory.
7. The referenced file for _PC_NO_TRUNC and _PC_2_SYMLINKS must be a directory.
8. For _PC_NO_TRUNC, the return value applies to filenames within the directory.
9. The referenced file for _PC_VDISABLE must be a terminal file.
10. For _PC_ASYNC_IO, _PC_PRIO_IO, and _PC_SYNC_IO, the referenced file must not be a directory.

Name of option	Indicates ...	*name* argument
_POSIX_CHOWN_RESTRICTED	whether use of chown is restricted	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	whether filenames longer than NAME_MAX generate an error	_PC_NO_TRUNC
_POSIX_VDISABLE	if defined, terminal special characters can be disabled with this value	_PC_VDISABLE
_POSIX_ASYNC_IO	whether asynchronous I/O can be used with the associated file	_PC_ASYNC_IO
_POSIX_PRIO_IO	whether prioritized I/O can be used with the associated file	_PC_PRIO_IO
_POSIX_SYNC_IO	whether synchronized I/O can be used with the associated file	_PC_SYNC_IO
_POSIX2_SYMLINKS	whether symbolic links are supported in the directory	_PC_2_SYMLINKS

Figure 2.18 Options and *name* arguments to pathconf and fpathconf

Name of option	Indicates ...	*name* argument
`_POSIX_ASYNCHRONOUS_IO`	whether the implementation supports POSIX asynchronous I/O	`_SC_ASYNCHRONOUS_IO`
`_POSIX_BARRIERS`	whether the implementation supports barriers	`_SC_BARRIERS`
`_POSIX_CLOCK_SELECTION`	whether the implementation supports clock selection	`_SC_CLOCK_SELECTION`
`_POSIX_JOB_CONTROL`	whether the implementation supports job control	`_SC_JOB_CONTROL`
`_POSIX_MAPPED_FILES`	whether the implementation supports memory-mapped files	`_SC_MAPPED_FILES`
`_POSIX_MEMORY_PROTECTION`	whether the implementation supports memory protection	`_SC_MEMORY_PROTECTION`
`_POSIX_READER_WRITER_LOCKS`	whether the implementation supports reader–writer locks	`_SC_READER_WRITER_LOCKS`
`_POSIX_REALTIME_SIGNALS`	whether the implementation supports real-time signals	`_SC_REALTIME_SIGNALS`
`_POSIX_SAVED_IDS`	whether the implementation supports the saved set-user-ID and the saved set-group-ID	`_SC_SAVED_IDS`
`_POSIX_SEMAPHORES`	whether the implementation supports POSIX semaphores	`_SC_SEMAPHORES`
`_POSIX_SHELL`	whether the implementation supports the POSIX shell	`_SC_SHELL`
`_POSIX_SPIN_LOCKS`	whether the implementation supports spin locks	`_SC_SPIN_LOCKS`
`_POSIX_THREAD_SAFE_FUNCTIONS`	whether the implementation supports thread-safe functions	`_SC_THREAD_SAFE_FUNCTIONS`
`_POSIX_THREADS`	whether the implementation supports threads	`_SC_THREADS`
`_POSIX_TIMEOUTS`	whether the implementation supports timeout-based variants of selected functions	`_SC_TIMEOUTS`
`_POSIX_TIMERS`	whether the implementation supports timers	`_SC_TIMERS`
`_POSIX_VERSION`	the POSIX.1 version	`_SC_VERSION`
`_XOPEN_CRYPT`	whether the implementation supports the XSI encryption option group	`_SC_XOPEN_CRYPT`
`_XOPEN_REALTIME`	whether the implementation supports the XSI real-time option group	`_SC_XOPEN_REALTIME`
`_XOPEN_REALTIME_THREADS`	whether the implementation supports the XSI real-time threads option group	`_SC_XOPEN_REALTIME_THREADS`
`_XOPEN_SHM`	whether the implementation supports the XSI shared memory option group	`_SC_XOPEN_SHM`
`_XOPEN_VERSION`	the XSI version	`_SC_XOPEN_VERSION`

Figure 2.19 Options and *name* arguments to `sysconf`

Figure 2.20 shows several configuration options and their corresponding values on the four sample systems we discuss in this text. An entry is "unsupported" if the system defines the symbolic constant but it has a value of −1, or if it has a value of 0 but the corresponding `sysconf` or `pathconf` call returned −1. It is interesting to see that some system implementations haven't yet caught up to the latest version of the Single UNIX Specification.

Limit	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	
				UFS file system	PCFS file system
`_POSIX_CHOWN_RESTRICTED`	1	1	200112	1	1
`_POSIX_JOB_CONTROL`	1	1	200112	1	1
`_POSIX_NO_TRUNC`	1	1	200112	1	unsupported
`_POSIX_SAVED_IDS`	unsupported	1	200112	1	1
`_POSIX_THREADS`	200112	200809	200112	200112	200112
`_POSIX_VDISABLE`	255	0	255	0	0
`_POSIX_VERSION`	200112	200809	200112	200112	200112
`_XOPEN_UNIX`	unsupported	1	1	1	1
`_XOPEN_VERSION`	unsupported	700	600	600	600

Figure 2.20 Examples of configuration options

Note that `pathconf` returns a value of −1 for `_PC_NO_TRUNC` when used with a file from a PCFS file system on Solaris. The PCFS file system supports the DOS format (for floppy disks), and DOS filenames are silently truncated to the 8.3 format limit that the DOS file system requires.

2.7 Feature Test Macros

The headers define numerous POSIX.1 and XSI symbols, as we've described. Even so, most implementations can add their own definitions to these headers, in addition to the POSIX.1 and XSI definitions. If we want to compile a program so that it depends only on the POSIX definitions and doesn't conflict with any implementation-defined constants, we need to define the constant `_POSIX_C_SOURCE`. All the POSIX.1 headers use this constant to exclude any implementation-defined definitions when `_POSIX_C_SOURCE` is defined.

Older versions of the POSIX.1 standard defined the `_POSIX_SOURCE` constant. This was superseded by the `_POSIX_C_SOURCE` constant in the 2001 version of POSIX.1.

The constants `_POSIX_C_SOURCE` and `_XOPEN_SOURCE` are called *feature test macros*. All feature test macros begin with an underscore. When used, they are typically defined in the `cc` command, as in

```
cc -D_POSIX_C_SOURCE=200809L file.c
```

This causes the feature test macro to be defined before any header files are included by the C program. If we want to use only the POSIX.1 definitions, we can also set the first line of a source file to

```
#define _POSIX_C_SOURCE  200809L
```

To enable the XSI option of Version 4 of the Single UNIX Specification, we need to define the constant `_XOPEN_SOURCE` to be 700. Besides enabling the XSI option, this has the same effect as defining `_POSIX_C_SOURCE` to be 200809L as far as POSIX.1 functionality is concerned.

The Single UNIX Specification defines the `c99` utility as the interface to the C compilation environment. With it we can compile a file as follows:

```
c99 -D_XOPEN_SOURCE=700 file.c -o file
```

To enable the 1999 ISO C extensions in the `gcc` C compiler, we use the `-std=c99` option, as in

```
gcc -D_XOPEN_SOURCE=700 -std=c99 file.c -o file
```

2.8 Primitive System Data Types

Historically, certain C data types have been associated with certain UNIX system variables. For example, major and minor device numbers have historically been stored in a 16-bit short integer, with 8 bits for the major device number and 8 bits for the minor device number. But many larger systems need more than 256 values for these device numbers, so a different technique is needed. (Indeed, the 32-bit version of Solaris uses 32 bits for the device number: 14 bits for the major and 18 bits for the minor.)

The header `<sys/types.h>` defines some implementation-dependent data types, called the *primitive system data types*. More of these data types are defined in other headers as well. These data types are defined in the headers with the C `typedef` facility. Most end in `_t`. Figure 2.21 lists many of the primitive system data types that we'll encounter in this text.

By defining these data types this way, we do not build into our programs implementation details that can change from one system to another. We describe what each of these data types is used for when we encounter them later in the text.

2.9 Differences Between Standards

All in all, these various standards fit together nicely. Our main concern is any differences between the ISO C standard and POSIX.1, since the Base Specifications of the Single UNIX Specification and POSIX.1 are one and the same. Conflicts are unintended, but if they should arise, POSIX.1 defers to the ISO C standard. However, there are some differences.

ISO C defines the function `clock` to return the amount of CPU time used by a process. The value returned is a `clock_t` value, but ISO C doesn't specify its units. To

Type	Description
`clock_t`	counter of clock ticks (process time) (Section 1.10)
`comp_t`	compressed clock ticks (not defined by POSIX.1; see Section 8.14)
`dev_t`	device numbers (major and minor) (Section 4.24)
`fd_set`	file descriptor sets (Section 14.4.1)
`fpos_t`	file position (Section 5.10)
`gid_t`	numeric group IDs
`ino_t`	i-node numbers (Section 4.14)
`mode_t`	file type, file creation mode (Section 4.5)
`nlink_t`	link counts for directory entries (Section 4.14)
`off_t`	file sizes and offsets (signed) (`lseek`, Section 3.6)
`pid_t`	process IDs and process group IDs (signed) (Sections 8.2 and 9.4)
`pthread_t`	thread IDs (Section 11.3)
`ptrdiff_t`	result of subtracting two pointers (signed)
`rlim_t`	resource limits (Section 7.11)
`sig_atomic_t`	data type that can be accessed atomically (Section 10.15)
`sigset_t`	signal set (Section 10.11)
`size_t`	sizes of objects (such as strings) (unsigned) (Section 3.7)
`ssize_t`	functions that return a count of bytes (signed) (`read`, `write`, Section 3.7)
`time_t`	counter of seconds of calendar time (Section 1.10)
`uid_t`	numeric user IDs
`wchar_t`	can represent all distinct character codes

Figure 2.21 Some common primitive system data types

convert this value to seconds, we divide it by `CLOCKS_PER_SEC`, which is defined in the `<time.h>` header. POSIX.1 defines the function `times` that returns both the CPU time (for the caller and all its terminated children) and the clock time. All these time values are `clock_t` values. The `sysconf` function is used to obtain the number of clock ticks per second for use with the return values from the `times` function. What we have is the same data type (`clock_t`) used to hold measurements of time defined with different units by ISO C and POSIX.1. The difference can be seen in Solaris, where `clock` returns microseconds (hence `CLOCKS_PER_SEC` is 1 million), whereas `sysconf` returns the value 100 for clock ticks per second. Thus we must take care when using variables of type `clock_t` so that we don't mix variables with different units.

Another area of potential conflict is when the ISO C standard specifies a function, but doesn't specify it as strongly as POSIX.1 does. This is the case for functions that require a different implementation in a POSIX environment (with multiple processes) than in an ISO C environment (where very little can be assumed about the host operating system). Nevertheless, POSIX-compliant systems implement the ISO C function for compatibility. The `signal` function is an example. If we unknowingly use the `signal` function provided by Solaris (hoping to write portable code that can be run in ISO C environments and under older UNIX systems), it will provide semantics different from the POSIX.1 `sigaction` function. We'll have more to say about the `signal` function in Chapter 10.

File I/O

3.1 Introduction

We'll start our discussion of the UNIX System by describing the functions available for file I/O—open a file, read a file, write a file, and so on. Most file I/O on a UNIX system can be performed using only five functions: `open`, `read`, `write`, `lseek`, and `close`. We then examine the effect of various buffer sizes on the `read` and `write` functions.

The functions described in this chapter are often referred to as *unbuffered I/O*, in contrast to the standard I/O routines, which we describe in Chapter 5. The term *unbuffered* means that each `read` or `write` invokes a system call in the kernel. These unbuffered I/O functions are not part of ISO C, but are part of POSIX.1 and the Single UNIX Specification.

Whenever we describe the sharing of resources among multiple processes, the concept of an atomic operation becomes important. We examine this concept with regard to file I/O and the arguments to the `open` function. This leads to a discussion of how files are shared among multiple processes and which kernel data structures are involved. After describing these features, we describe the `dup`, `fcntl`, `sync`, `fsync`, and `ioctl` functions.

3.2 File Descriptors

To the kernel, all open files are referred to by file descriptors. A file descriptor is a non-negative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by `open` or `creat` as an argument to either `read` or `write`.

By convention, UNIX System shells associate file descriptor 0 with the standard input of a process, file descriptor 1 with the standard output, and file descriptor 2 with the standard error. This convention is used by the shells and many applications; it is not a feature of the UNIX kernel. Nevertheless, many applications would break if these associations weren't followed.

Although their values are standardized by POSIX.1, the magic numbers 0, 1, and 2 should be replaced in POSIX-compliant applications with the symbolic constants `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` to improve readability. These constants are defined in the `<unistd.h>` header.

File descriptors range from 0 through `OPEN_MAX−1`. (Recall Figure 2.11.) Early historical implementations of the UNIX System had an upper limit of 19, allowing a maximum of 20 open files per process, but many systems subsequently increased this limit to 63.

With FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10, the limit is essentially infinite, bounded by the amount of memory on the system, the size of an integer, and any hard and soft limits configured by the system administrator.

3.3 open and openat Functions

A file is opened or created by calling either the `open` function or the `openat` function.

```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* mode_t mode */ );

int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );

                              Both return: file descriptor if OK, −1 on error
```

We show the last argument as `...`, which is the ISO C way to specify that the number and types of the remaining arguments may vary. For these functions, the last argument is used only when a new file is being created, as we describe later. We show this argument as a comment in the prototype.

The *path* parameter is the name of the file to open or create. This function has a multitude of options, which are specified by the *oflag* argument. This argument is formed by ORing together one or more of the following constants from the `<fcntl.h>` header:

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only.
`O_RDWR`	Open for reading and writing.

Most implementations define `O_RDONLY` as 0, `O_WRONLY` as 1, and `O_RDWR` as 2, for compatibility with older programs.

`O_EXEC`	Open for execute only.
`O_SEARCH`	Open for search only (applies to directories).

> The purpose of the `O_SEARCH` constant is to evaluate search permissions at the time a directory is opened. Further operations using the directory's file descriptor will not reevaluate permission to search the directory. None of the versions of the operating systems covered in this book support `O_SEARCH` yet.

One and only one of the previous five constants must be specified. The following constants are optional:

`O_APPEND`	Append to the end of file on each write. We describe this option in detail in Section 3.11.
`O_CLOEXEC`	Set the `FD_CLOEXEC` file descriptor flag. We discuss file descriptor flags in Section 3.14.
`O_CREAT`	Create the file if it doesn't exist. This option requires a third argument to the `open` function (a fourth argument to the `openat` function)—the *mode*, which specifies the access permission bits of the new file. (When we describe a file's access permission bits in Section 4.5, we'll see how to specify the *mode* and how it can be modified by the `umask` value of a process.)
`O_DIRECTORY`	Generate an error if *path* doesn't refer to a directory.
`O_EXCL`	Generate an error if `O_CREAT` is also specified and the file already exists. This test for whether the file already exists and the creation of the file if it doesn't exist is an atomic operation. We describe atomic operations in more detail in Section 3.11.
`O_NOCTTY`	If *path* refers to a terminal device, do not allocate the device as the controlling terminal for this process. We talk about controlling terminals in Section 9.6.
`O_NOFOLLOW`	Generate an error if *path* refers to a symbolic link. We discuss symbolic links in Section 4.17.
`O_NONBLOCK`	If *path* refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and subsequent I/O. We describe this mode in Section 14.2.

> In earlier releases of System V, the `O_NDELAY` (no delay) flag was introduced. This option is similar to the `O_NONBLOCK` (nonblocking) option, but an ambiguity was introduced in the return value from a read operation. The no-delay option causes a read operation to return 0 if there is no data to be read from a pipe, FIFO, or device, but this conflicts with a return value of 0, indicating an end of file. SVR4-based systems still support the no-delay option, with the old semantics, but new applications should use the nonblocking option instead.

`O_SYNC`	Have each `write` wait for physical I/O to complete, including I/O necessary to update file attributes modified as a result of the `write`. We use this option in Section 3.14.
`O_TRUNC`	If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0.

`O_TTY_INIT` When opening a terminal device that is not already open, set the nonstandard `termios` parameters to values that result in behavior that conforms to the Single UNIX Specification. We discuss the `termios` structure when we discuss terminal I/O in Chapter 18.

The following two flags are also optional. They are part of the synchronized input and output option of the Single UNIX Specification (and thus POSIX.1).

`O_DSYNC` Have each `write` wait for physical I/O to complete, but don't wait for file attributes to be updated if they don't affect the ability to read the data just written.

The `O_DSYNC` and `O_SYNC` flags are similar, but subtly different. The `O_DSYNC` flag affects a file's attributes only when they need to be updated to reflect a change in the file's data (for example, update the file's size to reflect more data). With the `O_SYNC` flag, data and attributes are always updated synchronously. When overwriting an existing part of a file opened with the `O_DSYNC` flag, the file times wouldn't be updated synchronously. In contrast, if we had opened the file with the `O_SYNC` flag, every `write` to the file would update the file's times before the `write` returns, regardless of whether we were writing over existing bytes or appending to the file.

`O_RSYNC` Have each `read` operation on the file descriptor wait until any pending writes for the same portion of the file are complete.

Solaris 10 supports all three synchronization flags. Historically, FreeBSD (and thus Mac OS X) have used the `O_FSYNC` flag, which has the same behavior as `O_SYNC`. Because the two flags are equivalent, they define the flags to have the same value. FreeBSD 8.0 doesn't support the `O_DSYNC` or `O_RSYNC` flags. Mac OS X doesn't support the `O_RSYNC` flag, but defines the `O_DSYNC` flag, treating it the same as the `O_SYNC` flag. Linux 3.2.0 supports the `O_DSYNC` flag, but treats the `O_RSYNC` flag the same as `O_SYNC`.

The file descriptor returned by `open` and `openat` is guaranteed to be the lowest-numbered unused descriptor. This fact is used by some applications to open a new file on standard input, standard output, or standard error. For example, an application might close standard output—normally, file descriptor 1—and then open another file, knowing that it will be opened on file descriptor 1. We'll see a better way to guarantee that a file is open on a given descriptor in Section 3.12, when we explore the `dup2` function.

The *fd* parameter distinguishes the `openat` function from the `open` function. There are three possibilities:

1. The *path* parameter specifies an absolute pathname. In this case, the *fd* parameter is ignored and the `openat` function behaves like the `open` function.
2. The *path* parameter specifies a relative pathname and the *fd* parameter is a file descriptor that specifies the starting location in the file system where the relative pathname is to be evaluated. The *fd* parameter is obtained by opening the directory where the relative pathname is to be evaluated.

3. The *path* parameter specifies a relative pathname and the *fd* parameter has the special value AT_FDCWD. In this case, the pathname is evaluated starting in the current working directory and the openat function behaves like the open function.

The openat function is one of a class of functions added to the latest version of POSIX.1 to address two problems. First, it gives threads a way to use relative pathnames to open files in directories other than the current working directory. As we'll see in Chapter 11, all threads in the same process share the same current working directory, so this makes it difficult for multiple threads in the same process to work in different directories at the same time. Second, it provides a way to avoid time-of-check-to-time-of-use (TOCTTOU) errors.

The basic idea behind TOCTTOU errors is that a program is vulnerable if it makes two file-based function calls where the second call depends on the results of the first call. Because the two calls are not atomic, the file can change between the two calls, thereby invalidating the results of the first call, leading to a program error. TOCTTOU errors in the file system namespace generally deal with attempts to subvert file system permissions by tricking a privileged program into either reducing permissions on a privileged file or modifying a privileged file to open up a security hole. Wei and Pu [2005] discuss TOCTTOU weaknesses in the UNIX file system interface.

Filename and Pathname Truncation

What happens if NAME_MAX is 14 and we try to create a new file in the current directory with a filename containing 15 characters? Traditionally, early releases of System V, such as SVR2, allowed this to happen, silently truncating the filename beyond the 14th character. BSD-derived systems, in contrast, returned an error status, with errno set to ENAMETOOLONG. Silently truncating the filename presents a problem that affects more than simply the creation of new files. If NAME_MAX is 14 and a file exists whose name is exactly 14 characters, any function that accepts a pathname argument, such as open or stat, has no way to determine what the original name of the file was, as the original name might have been truncated.

With POSIX.1, the constant _POSIX_NO_TRUNC determines whether long filenames and long components of pathnames are truncated or an error is returned. As we saw in Chapter 2, this value can vary based on the type of the file system, and we can use fpathconf or pathconf to query a directory to see which behavior is supported.

Whether an error is returned is largely historical. For example, SVR4-based systems do not generate an error for the traditional System V file system, S5. For the BSD-style file system (known as UFS), however, SVR4-based systems do generate an error. Figure 2.20 illustrates another example: Solaris will return an error for UFS, but not for PCFS, the DOS-compatible file system, as DOS silently truncates filenames that don't fit in an 8.3 format. BSD-derived systems and Linux always return an error.

If _POSIX_NO_TRUNC is in effect, errno is set to ENAMETOOLONG, and an error status is returned if any filename component of the pathname exceeds NAME_MAX.

Most modern file systems support a maximum of 255 characters for filenames. Because filenames are usually shorter than this limit, this constraint tends to not present problems for most applications.

3.4 creat Function

A new file can also be created by calling the `creat` function.

```
#include <fcntl.h>

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

Returns: file descriptor opened for write-only if OK, –1 on error

Note that this function is equivalent to

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

> Historically, in early versions of the UNIX System, the second argument to open could be only 0, 1, or 2. There was no way to open a file that didn't already exist. Therefore, a separate system call, `creat`, was needed to create new files. With the `O_CREAT` and `O_TRUNC` options now provided by open, a separate `creat` function is no longer needed.

We'll show how to specify *mode* in Section 4.5 when we describe a file's access permissions in detail.

One deficiency with `creat` is that the file is opened only for writing. Before the new version of `open` was provided, if we were creating a temporary file that we wanted to write and then read back, we had to call `creat`, `close`, and then `open`. A better way is to use the `open` function, as in

```
open(path, O_RDWR | O_CREAT | O_TRUNC, mode);
```

3.5 close Function

An open file is closed by calling the `close` function.

```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, –1 on error

Closing a file also releases any record locks that the process may have on the file. We'll discuss this point further in Section 14.3.

When a process terminates, all of its open files are closed automatically by the kernel. Many programs take advantage of this fact and don't explicitly close open files. See the program in Figure 1.4, for example.

3.6 lseek Function

Every open file has an associated "current file offset," normally a non-negative integer that measures the number of bytes from the beginning of the file. (We describe some exceptions to the "non-negative" qualifier later in this section.) Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written. By default, this offset is initialized to 0 when a file is opened, unless the `O_APPEND` option is specified.

An open file's offset can be set explicitly by calling `lseek`.

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Returns: new file offset if OK, –1 on error

The interpretation of the *offset* depends on the value of the *whence* argument.

- If *whence* is `SEEK_SET`, the file's offset is set to *offset* bytes from the beginning of the file.
- If *whence* is `SEEK_CUR`, the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative.
- If *whence* is `SEEK_END`, the file's offset is set to the size of the file plus the *offset*. The *offset* can be positive or negative.

Because a successful call to `lseek` returns the new file offset, we can seek zero bytes from the current position to determine the current offset:

```
off_t    currpos;

currpos = lseek(fd, 0, SEEK_CUR);
```

This technique can also be used to determine if a file is capable of seeking. If the file descriptor refers to a pipe, FIFO, or socket, `lseek` sets `errno` to `ESPIPE` and returns –1.

The three symbolic constants—`SEEK_SET`, `SEEK_CUR`, and `SEEK_END`—were introduced with System V. Prior to this, *whence* was specified as 0 (absolute), 1 (relative to the current offset), or 2 (relative to the end of file). Much software still exists with these numbers hard coded.

The character `l` in the name `lseek` means "long integer." Before the introduction of the `off_t` data type, the *offset* argument and the return value were long integers. `lseek` was introduced with Version 7 when long integers were added to C. (Similar functionality was provided in Version 6 by the functions `seek` and `tell`.)

Example

The program in Figure 3.1 tests its standard input to see whether it is capable of seeking.

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

Figure 3.1 Test whether standard input is capable of seeking

If we invoke this program interactively, we get

```
$ ./a.out < /etc/passwd
seek OK
$ cat < /etc/passwd | ./a.out
cannot seek
$ ./a.out < /var/spool/cron/FIFO
cannot seek
```

□

Normally, a file's current offset must be a non-negative integer. It is possible, however, that certain devices could allow negative offsets. But for regular files, the offset must be non-negative. Because negative offsets are possible, we should be careful to compare the return value from `lseek` as being equal to or not equal to −1, rather than testing whether it is less than 0.

> The `/dev/kmem` device on FreeBSD for the Intel x86 processor supports negative offsets.
>
> Because the offset (`off_t`) is a signed data type (Figure 2.21), we lose a factor of 2 in the maximum file size. If `off_t` is a 32-bit integer, the maximum file size is $2^{31}-1$ bytes.

`lseek` only records the current file offset within the kernel—it does not cause any I/O to take place. This offset is then used by the next read or write operation.

The file's offset can be greater than the file's current size, in which case the next `write` to the file will extend the file. This is referred to as creating a hole in a file and is allowed. Any bytes in a file that have not been written are read back as 0.

A hole in a file isn't required to have storage backing it on disk. Depending on the file system implementation, when you write after seeking past the end of a file, new disk blocks might be allocated to store the data, but there is no need to allocate disk blocks for the data between the old end of file and the location where you start writing.

Example

The program shown in Figure 3.2 creates a file with a hole in it.

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

char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int     fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */
    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}
```

Figure 3.2 Create a file with a hole in it

Running this program gives us

```
$ ./a.out
$ ls -l file.hole                        check its size
-rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
$ od -c file.hole                        let's look at the actual contents
0000000   a  b  c  d  e  f  g  h  i  j \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0040000   A  B  C  D  E  F  G  H  I  J
0040012
```

We use the od(1) command to look at the contents of the file. The -c flag tells it to print the contents as characters. We can see that the unwritten bytes in the middle are read back as zero. The seven-digit number at the beginning of each line is the byte offset in octal.

To prove that there is really a hole in the file, let's compare the file we just created with a file of the same size, but without holes:

```
$ ls -ls file.hole file.nohole      compare sizes
  8 -rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
 20 -rw-r--r--  1 sar       16394 Nov 25 01:03 file.nohole
```

Although both files are the same size, the file without holes consumes 20 disk blocks, whereas the file with holes consumes only 8 blocks.

In this example, we call the write function (Section 3.8). We'll have more to say about files with holes in Section 4.12. □

Because the offset address that lseek uses is represented by an off_t, implementations are allowed to support whatever size is appropriate on their particular platform. Most platforms today provide two sets of interfaces to manipulate file offsets: one set that uses 32-bit file offsets and another set that uses 64-bit file offsets.

The Single UNIX Specification provides a way for applications to determine which environments are supported through the sysconf function (Section 2.5.4). Figure 3.3 summarizes the sysconf constants that are defined.

Name of option	Description	*name* argument
`_POSIX_V7_ILP32_OFF32`	`int`, `long`, pointer, and `off_t` types are 32 bits.	`_SC_V7_ILP32_OFF32`
`_POSIX_V7_ILP32_OFFBIG`	`int`, `long`, and pointer types are 32 bits; `off_t` types are at least 64 bits.	`_SC_V7_ILP32_OFFBIG`
`_POSIX_V7_LP64_OFF64`	`int` types are 32 bits; `long`, pointer, and `off_t` types are 64 bits.	`_SC_V7_LP64_OFF64`
`_POSIX_V7_LP64_OFFBIG`	`int` types are at least 32 bits; `long`, pointer, and `off_t` types are at least 64 bits.	`_SC_V7_LP64_OFFBIG`

Figure 3.3 Data size options and *name* arguments to `sysconf`

The `c99` compiler requires that we use the `getconf`(1) command to map the desired data size model to the flags necessary to compile and link our programs. Different flags and libraries might be needed, depending on the environments supported by each platform.

Unfortunately, this is one area in which implementations haven't caught up to the standards. If your system does not match the latest version of the standard, the system might support the option names from the previous version of the Single UNIX Specification: `_POSIX_V6_ILP32_OFF32`, `_POSIX_V6_ILP32_OFFBIG`, `_POSIX_V6_LP64_OFF64`, and `_POSIX_V6_LP64_OFFBIG`.

To get around this, applications can set the `_FILE_OFFSET_BITS` constant to 64 to enable 64-bit offsets. Doing so changes the definition of `off_t` to be a 64-bit signed integer. Setting `_FILE_OFFSET_BITS` to 32 enables 32-bit file offsets. Be aware, however, that although all four platforms discussed in this text support both 32-bit and 64-bit file offsets, setting `_FILE_OFFSET_BITS` is not guaranteed to be portable and might not have the desired effect.

Figure 3.4 summarizes the size in bytes of the `off_t` data type for the platforms covered in this book when an application doesn't define `_FILE_OFFSET_BITS`, as well as the size when an application defines `_FILE_OFFSET_BITS` to have a value of either 32 or 64.

Operating system	CPU architecture	`_FILE_OFFSET_BITS` value		
		Undefined	32	64
FreeBSD 8.0	x86 32-bit	8	8	8
Linux 3.2.0	x86 64-bit	8	8	8
Mac OS X 10.6.8	x86 64-bit	8	8	8
Solaris 10	SPARC 64-bit	8	4	8

Figure 3.4 Size in bytes of `off_t` for different platforms

Note that even though you might enable 64-bit file offsets, your ability to create a file larger than 2 GB (2^{31}–1 bytes) depends on the underlying file system type.

3.7 `read` Function

Data is read from an open file with the `read` function.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);
```

Returns: number of bytes read, 0 if end of file, –1 on error

If the `read` is successful, the number of bytes read is returned. If the end of file is encountered, 0 is returned.

There are several cases in which the number of bytes actually read is less than the amount requested:

- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, `read` returns 30. The next time we call `read`, it will return 0 (end of file).
- When reading from a terminal device. Normally, up to one line is read at a time. (We'll see how to change this default in Chapter 18.)
- When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, `read` will return only what is available.
- When reading from a record-oriented device. Some record-oriented devices, such as magnetic tape, can return up to a single record at a time.
- When interrupted by a signal and a partial amount of data has already been read. We discuss this further in Section 10.5.

The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

POSIX.1 changed the prototype for this function in several ways. The classic definition is

```
int read(int fd, char *buf, unsigned nbytes);
```

- First, the second argument was changed from `char *` to `void *` to be consistent with ISO C: the type `void *` is used for generic pointers.
- Next, the return value was required to be a signed integer (`ssize_t`) to return a positive byte count, 0 (for end of file), or –1 (for an error).
- Finally, the third argument historically has been an unsigned integer, to allow a 16-bit implementation to read or write up to 65,534 bytes at a time. With the 1990 POSIX.1 standard, the primitive system data type `ssize_t` was introduced to provide the signed return value, and the unsigned `size_t` was used for the third argument. (Recall the `SSIZE_MAX` constant from Section 2.5.2.)

3.8 `write` Function

Data is written to an open file with the `write` function.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t nbytes);

                    Returns: number of bytes written if OK, −1 on error
```

The return value is usually equal to the *nbytes* argument; otherwise, an error has occurred. A common cause for a `write` error is either filling up a disk or exceeding the file size limit for a given process (Section 7.11 and Exercise 10.11).

For a regular file, the write operation starts at the file's current offset. If the `O_APPEND` option was specified when the file was opened, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

3.9 I/O Efficiency

The program in Figure 3.5 copies a file, using only the `read` and `write` functions.

```
#include "apue.h"

#define BUFFSIZE    4096

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

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

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

    exit(0);
}
```

Figure 3.5 Copy standard input to standard output

The following caveats apply to this program.

- It reads from standard input and writes to standard output, assuming that these have been set up by the shell before this program is executed. Indeed, all normal UNIX system shells provide a way to open a file for reading on standard input and to create (or rewrite) a file on standard output. This prevents the program from having to open the input and output files, and allows the user to take advantage of the shell's I/O redirection facilities.

- The program doesn't close the input file or output file. Instead, the program uses the feature of the UNIX kernel that closes all open file descriptors in a process when that process terminates.
- This example works for both text files and binary files, since there is no difference between the two to the UNIX kernel.

One question we haven't answered, however, is how we chose the `BUFFSIZE` value. Before answering that, let's run the program using different values for `BUFFSIZE`. Figure 3.6 shows the results for reading a 516,581,760-byte file, using 20 different buffer sizes.

`BUFFSIZE`	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

Figure 3.6 Timing results for reading with different buffer sizes on Linux

The file was read using the program shown in Figure 3.5, with standard output redirected to `/dev/null`. The file system used for this test was the Linux `ext4` file system with 4,096-byte blocks. (The `st_blksize` value, which we describe in Section 4.12, is 4,096.) This accounts for the minimum in the system time occurring at the few timing measurements starting around a `BUFFSIZE` of 4,096. Increasing the buffer size beyond this limit has little positive effect.

Most file systems support some kind of read-ahead to improve performance. When sequential reads are detected, the system tries to read in more data than an application requests, assuming that the application will read it shortly. The effect of read-ahead can be seen in Figure 3.6, where the elapsed time for buffer sizes as small as 32 bytes is as good as the elapsed time for larger buffer sizes.

We'll return to this timing example later in the text. In Section 3.14, we show the effect of synchronous writes; in Section 5.8, we compare these unbuffered I/O times with the standard I/O library.

Beware when trying to measure the performance of programs that read and write files. The operating system will try to cache the file incore, so if you measure the performance of the program repeatedly, the successive timings will likely be better than the first. This improvement occurs because the first run causes the file to be entered into the system's cache, and successive runs access the file from the system's cache instead of from the disk. (The term *incore* means *in main memory*. Back in the day, a computer's main memory was built out of ferrite core. This is where the phrase "core dump" comes from: the main memory image of a program stored in a file on disk for diagnosis.)

In the tests reported in Figure 3.6, each run with a different buffer size was made using a different copy of the file so that the current run didn't find the data in the cache from the previous run. The files are large enough that they all don't remain in the cache (the test system was configured with 6 GB of RAM).

3.10 File Sharing

The UNIX System supports the sharing of open files among different processes. Before describing the `dup` function, we need to describe this sharing. To do this, we'll examine the data structures used by the kernel for all I/O.

The following description is conceptual; it may or may not match a particular implementation. Refer to Bach [1986] for a discussion of these structures in System V. McKusick et al. [1996] describe these structures in 4.4BSD. McKusick and Neville-Neil [2005] cover FreeBSD 5.2. For a similar discussion of Solaris, see McDougall and Mauro [2007]. The Linux 2.6 kernel architecture is discussed in Bovet and Cesati [2006].

The kernel uses three data structures to represent an open file, and the relationships among them determine the effect one process has on another with regard to file sharing.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are
   (a) The file descriptor flags (close-on-exec; refer to Figure 3.7 and Section 3.14)
   (b) A pointer to a file table entry

2. The kernel maintains a file table for all open files. Each file table entry contains
   (a) The file status flags for the file, such as read, write, append, sync, and nonblocking; more on these in Section 3.14
   (b) The current file offset
   (c) A pointer to the v-node table entry for the file

3. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file. For most files, the

v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, and so on. (We talk more about i-nodes in Section 4.14 when we describe the typical UNIX file system in more detail.)

Linux has no v-node. Instead, a generic i-node structure is used. Although the implementations differ, the v-node is conceptually the same as a generic i-node. Both point to an i-node structure specific to the file system.

We're ignoring some implementation details that don't affect our discussion. For example, the table of open file descriptors can be stored in the user area (a separate per-process structure that can be paged out) instead of the process table. Also, these tables can be implemented in numerous ways—they need not be arrays; one alternate implementation is a linked lists of structures. Regardless of the implementation details, the general concepts remain the same.

Figure 3.7 shows a pictorial arrangement of these three tables for a single process that has two different files open: one file is open on standard input (file descriptor 0), and the other is open on standard output (file descriptor 1).

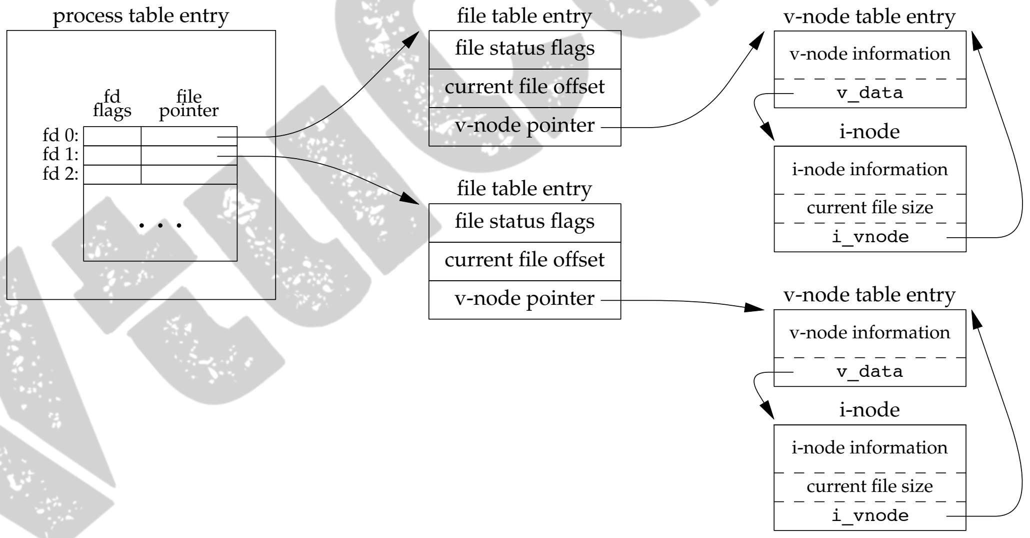

Figure 3.7 Kernel data structures for open files

The arrangement of these three tables has existed since the early versions of the UNIX System [Thompson 1978]. This arrangement is critical to the way files are shared among processes. We'll return to this figure in later chapters, when we describe additional ways that files are shared.

The v-node was invented to provide support for multiple file system types on a single computer system. This work was done independently by Peter Weinberger (Bell Laboratories) and Bill Joy (Sun Microsystems). Sun called this the Virtual File System and called the file system–independent portion of the i-node the v-node [Kleiman 1986]. The v-node propagated through various vendor implementations as support for Sun's Network File System (NFS) was added. The first release from Berkeley to provide v-nodes was the 4.3BSD Reno release, when NFS was added.

In SVR4, the v-node replaced the file system–independent i-node of SVR3. Solaris is derived from SVR4 and, therefore, uses v-nodes.

Instead of splitting the data structures into a v-node and an i-node, Linux uses a file system–independent i-node and a file system–dependent i-node.

If two independent processes have the same file open, we could have the arrangement shown in Figure 3.8.

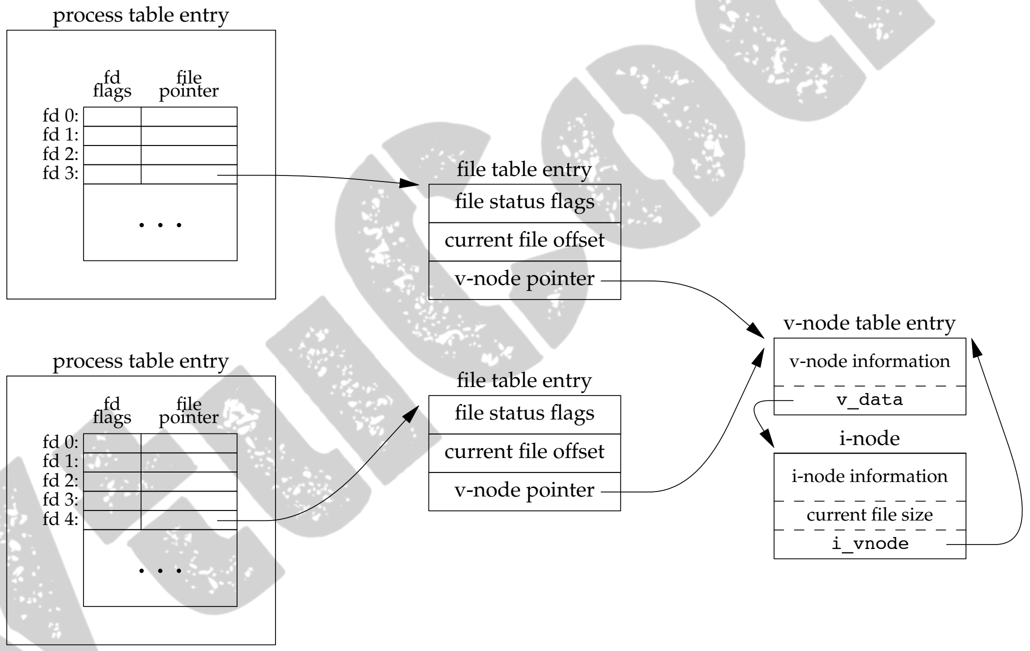

Figure 3.8 Two independent processes with the same file open

We assume here that the first process has the file open on descriptor 3 and that the second process has that same file open on descriptor 4. Each process that opens the file gets its own file table entry, but only a single v-node table entry is required for a given file. One reason each process gets its own file table entry is so that each process has its own current offset for the file.

Given these data structures, we now need to be more specific about what happens with certain operations that we've already described.

- After each write is complete, the current file offset in the file table entry is incremented by the number of bytes written. If this causes the current file offset to exceed the current file size, the current file size in the i-node table entry is set to the current file offset (for example, the file is extended).
- If a file is opened with the O_APPEND flag, a corresponding flag is set in the file status flags of the file table entry. Each time a write is performed for a file with this append flag set, the current file offset in the file table entry is first set to the current file size from the i-node table entry. This forces every write to be appended to the current end of file.
- If a file is positioned to its current end of file using lseek, all that happens is the current file offset in the file table entry is set to the current file size from the i-node table entry. (Note that this is not the same as if the file was opened with the O_APPEND flag, as we will see in Section 3.11.)
- The lseek function modifies only the current file offset in the file table entry. No I/O takes place.

It is possible for more than one file descriptor entry to point to the same file table entry, as we'll see when we discuss the dup function in Section 3.12. This also happens after a fork when the parent and the child share the same file table entry for each open descriptor (Section 8.3).

Note the difference in scope between the file descriptor flags and the file status flags. The former apply only to a single descriptor in a single process, whereas the latter apply to all descriptors in any process that point to the given file table entry. When we describe the fcntl function in Section 3.14, we'll see how to fetch and modify both the file descriptor flags and the file status flags.

Everything that we've described so far in this section works fine for multiple processes that are reading the same file. Each process has its own file table entry with its own current file offset. Unexpected results can arise, however, when multiple processes write to the same file. To see how to avoid some surprises, we need to understand the concept of atomic operations.

3.11 Atomic Operations

Appending to a File

Consider a single process that wants to append to the end of a file. Older versions of the UNIX System didn't support the O_APPEND option to open, so the program was coded as follows:

```
if (lseek(fd, 0L, 2) < 0)          /* position to EOF */
    err_sys("lseek error");
if (write(fd, buf, 100) != 100)    /* and write */
    err_sys("write error");
```

This works fine for a single process, but problems arise if multiple processes use this technique to append to the same file. (This scenario can arise if multiple instances of the same program are appending messages to a log file, for example.)

Assume that two independent processes, A and B, are appending to the same file. Each has opened the file but *without* the `O_APPEND` flag. This gives us the same picture as Figure 3.8. Each process has its own file table entry, but they share a single v-node table entry. Assume that process A does the `lseek` and that this sets the current offset for the file for process A to byte offset 1,500 (the current end of file). Then the kernel switches processes, and B continues running. Process B then does the `lseek`, which sets the current offset for the file for process B to byte offset 1,500 also (the current end of file). Then B calls `write`, which increments B's current file offset for the file to 1,600. Because the file's size has been extended, the kernel also updates the current file size in the v-node to 1,600. Then the kernel switches processes and A resumes. When A calls `write`, the data is written starting at the current file offset for A, which is byte offset 1,500. This overwrites the data that B wrote to the file.

The problem here is that our logical operation of "position to the end of file and write" requires two separate function calls (as we've shown it). The solution is to have the positioning to the current end of file and the write be an atomic operation with regard to other processes. Any operation that requires more than one function call cannot be atomic, as there is always the possibility that the kernel might temporarily suspend the process between the two function calls (as we assumed previously).

The UNIX System provides an atomic way to do this operation if we set the `O_APPEND` flag when a file is opened. As we described in the previous section, this causes the kernel to position the file to its current end of file before each `write`. We no longer have to call `lseek` before each `write`.

`pread` and `pwrite` Functions

The Single UNIX Specification includes two functions that allow applications to seek and perform I/O atomically: `pread` and `pwrite`.

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);

                     Returns: number of bytes read, 0 if end of file, –1 on error

ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);

                          Returns: number of bytes written if OK, –1 on error
```

Calling `pread` is equivalent to calling `lseek` followed by a call to `read`, with the following exceptions.

- There is no way to interrupt the two operations that occur when we call `pread`.
- The current file offset is not updated.

Calling pwrite is equivalent to calling lseek followed by a call to write, with similar exceptions.

Creating a File

We saw another example of an atomic operation when we described the O_CREAT and O_EXCL options for the open function. When both of these options are specified, the open will fail if the file already exists. We also said that the check for the existence of the file and the creation of the file was performed as an atomic operation. If we didn't have this atomic operation, we might try

```
if ((fd = open(path, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(path, mode)) < 0)
            err_sys("creat error");
    } else {
        err_sys("open error");
    }
}
```

The problem occurs if the file is created by another process between the open and the creat. If the file is created by another process between these two function calls, and if that other process writes something to the file, that data is erased when this creat is executed. Combining the test for existence and the creation into a single atomic operation avoids this problem.

In general, the term *atomic operation* refers to an operation that might be composed of multiple steps. If the operation is performed atomically, either all the steps are performed (on success) or none are performed (on failure). It must not be possible for only a subset of the steps to be performed. We'll return to the topic of atomic operations when we describe the link function (Section 4.15) and record locking (Section 14.3).

3.12 dup and dup2 Functions

An existing file descriptor is duplicated by either of the following functions:

```
#include <unistd.h>

int dup(int fd);

int dup2(int fd, int fd2);
```

Both return: new file descriptor if OK, –1 on error

The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor. With dup2, we specify the value of the new descriptor with the *fd2* argument. If *fd2* is already open, it is first closed. If *fd* equals *fd2*, then dup2 returns *fd2* without closing it. Otherwise, the FD_CLOEXEC file descriptor flag is cleared for *fd2*, so that *fd2* is left open if the process calls exec.

The new file descriptor that is returned as the value of the functions shares the same file table entry as the *fd* argument. We show this in Figure 3.9.

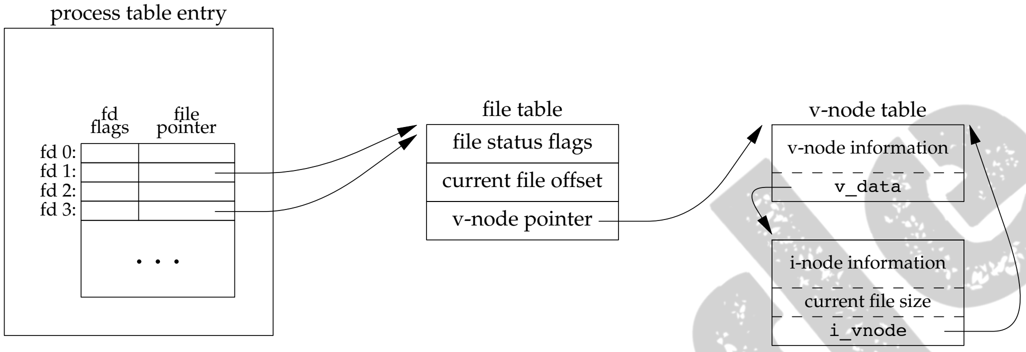

Figure 3.9 Kernel data structures after dup(1)

In this figure, we assume that when it's started, the process executes

```
newfd = dup(1);
```

We assume that the next available descriptor is 3 (which it probably is, since 0, 1, and 2 are opened by the shell). Because both descriptors point to the same file table entry, they share the same file status flags—read, write, append, and so on—and the same current file offset.

Each descriptor has its own set of file descriptor flags. As we describe in Section 3.14, the close-on-exec file descriptor flag for the new descriptor is always cleared by the dup functions.

Another way to duplicate a descriptor is with the fcntl function, which we describe in Section 3.14. Indeed, the call

```
dup(fd);
```

is equivalent to

```
fcntl(fd, F_DUPFD, 0);
```

Similarly, the call

```
dup2(fd, fd2);
```

is equivalent to

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

In this last case, the dup2 is not exactly the same as a close followed by an fcntl. The differences are as follows:

1. dup2 is an atomic operation, whereas the alternate form involves two function calls. It is possible in the latter case to have a signal catcher called between the close and the fcntl that could modify the file descriptors. (We describe signals in Chapter 10.) The same problem could occur if a different thread changes the file descriptors. (We describe threads in Chapter 11.)

2. There are some errno differences between dup2 and fcntl.

The dup2 system call originated with Version 7 and propagated through the BSD releases. The fcntl method for duplicating file descriptors appeared with System III and continued with System V. SVR3.2 picked up the dup2 function, and 4.2BSD picked up the fcntl function and the F_DUPFD functionality. POSIX.1 requires both dup2 and the F_DUPFD feature of fcntl.

3.13 sync, fsync, and fdatasync Functions

Traditional implementations of the UNIX System have a buffer cache or page cache in the kernel through which most disk I/O passes. When we write data to a file, the data is normally copied by the kernel into one of its buffers and queued for writing to disk at some later time. This is called *delayed write*. (Chapter 3 of Bach [1986] discusses this buffer cache in detail.)

The kernel eventually writes all the delayed-write blocks to disk, normally when it needs to reuse the buffer for some other disk block. To ensure consistency of the file system on disk with the contents of the buffer cache, the sync, fsync, and fdatasync functions are provided.

```
#include <unistd.h>

int fsync(int fd);

int fdatasync(int fd);

                                        Returns: 0 if OK, −1 on error

void sync(void);
```

The sync function simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place.

The function sync is normally called periodically (usually every 30 seconds) from a system daemon, often called update. This guarantees regular flushing of the kernel's block buffers. The command sync(1) also calls the sync function.

The function fsync refers only to a single file, specified by the file descriptor *fd*, and waits for the disk writes to complete before returning. This function is used when an application, such as a database, needs to be sure that the modified blocks have been written to the disk.

The fdatasync function is similar to fsync, but it affects only the data portions of a file. With fsync, the file's attributes are also updated synchronously.

All four of the platforms described in this book support sync and fsync. However, FreeBSD 8.0 does not support fdatasync.

3.14 `fcntl` Function

The `fcntl` function can change the properties of a file that is already open.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );
```

Returns: depends on *cmd* if OK (see following), −1 on error

In the examples in this section, the third argument is always an integer, corresponding to the comment in the function prototype just shown. When we describe record locking in Section 14.3, however, the third argument becomes a pointer to a structure.

The `fcntl` function is used for five different purposes.

1. Duplicate an existing descriptor (*cmd* = `F_DUPFD` or `F_DUPFD_CLOEXEC`)
2. Get/set file descriptor flags (*cmd* = `F_GETFD` or `F_SETFD`)
3. Get/set file status flags (*cmd* = `F_GETFL` or `F_SETFL`)
4. Get/set asynchronous I/O ownership (*cmd* = `F_GETOWN` or `F_SETOWN`)
5. Get/set record locks (*cmd* = `F_GETLK`, `F_SETLK`, or `F_SETLKW`)

We'll now describe the first 8 of these 11 *cmd* values. (We'll wait until Section 14.3 to describe the last 3, which deal with record locking.) Refer to Figure 3.7, as we'll discuss both the file descriptor flags associated with each file descriptor in the process table entry and the file status flags associated with each file table entry.

`F_DUPFD`	Duplicate the file descriptor *fd*. The new file descriptor is returned as the value of the function. It is the lowest-numbered descriptor that is not already open, and that is greater than or equal to the third argument (taken as an integer). The new descriptor shares the same file table entry as *fd*. (Refer to Figure 3.9.) But the new descriptor has its own set of file descriptor flags, and its `FD_CLOEXEC` file descriptor flag is cleared. (This means that the descriptor is left open across an `exec`, which we discuss in Chapter 8.)
`F_DUPFD_CLOEXEC`	Duplicate the file descriptor and set the `FD_CLOEXEC` file descriptor flag associated with the new descriptor. Returns the new file descriptor.
`F_GETFD`	Return the file descriptor flags for *fd* as the value of the function. Currently, only one file descriptor flag is defined: the `FD_CLOEXEC` flag.
`F_SETFD`	Set the file descriptor flags for *fd*. The new flag value is set from the third argument (taken as an integer).

Be aware that some existing programs that deal with the file descriptor flags don't use the constant FD_CLOEXEC. Instead, these programs set the flag to either 0 (don't close-on-exec, the default) or 1 (do close-on-exec).

F_GETFL Return the file status flags for *fd* as the value of the function. We described the file status flags when we described the open function. They are listed in Figure 3.10.

File status flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_EXEC	open for execute only
O_SEARCH	open directory for searching only
O_APPEND	append on each write
O_NONBLOCK	nonblocking mode
O_SYNC	wait for writes to complete (data and attributes)
O_DSYNC	wait for writes to complete (data only)
O_RSYNC	synchronize reads and writes
O_FSYNC	wait for writes to complete (FreeBSD and Mac OS X only)
O_ASYNC	asynchronous I/O (FreeBSD and Mac OS X only)

Figure 3.10 File status flags for fcntl

Unfortunately, the five access-mode flags—O_RDONLY, O_WRONLY, O_RDWR, O_EXEC, and O_SEARCH—are not separate bits that can be tested. (As we mentioned earlier, the first three often have the values 0, 1, and 2, respectively, for historical reasons. Also, these five values are mutually exclusive; a file can have only one of them enabled.) Therefore, we must first use the O_ACCMODE mask to obtain the access-mode bits and then compare the result against any of the five values.

F_SETFL Set the file status flags to the value of the third argument (taken as an integer). The only flags that can be changed are O_APPEND, O_NONBLOCK, O_SYNC, O_DSYNC, O_RSYNC, O_FSYNC, and O_ASYNC.

F_GETOWN Get the process ID or process group ID currently receiving the SIGIO and SIGURG signals. We describe these asynchronous I/O signals in Section 14.5.2.

F_SETOWN Set the process ID or process group ID to receive the SIGIO and SIGURG signals. A positive *arg* specifies a process ID. A negative *arg* implies a process group ID equal to the absolute value of *arg*.

The return value from fcntl depends on the command. All commands return −1 on an error or some other value if OK. The following four commands have special return values: F_DUPFD, F_GETFD, F_GETFL, and F_GETOWN. The first command returns the new file descriptor, the next two return the corresponding flags, and the final command returns a positive process ID or a negative process group ID.

Example

The program in Figure 3.11 takes a single command-line argument that specifies a file descriptor and prints a description of selected file flags for that descriptor.

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

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

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;

    case O_WRONLY:
        printf("write only");
        break;

    case O_RDWR:
        printf("read write");
        break;

    default:
        err_dump("unknown access mode");
    }

    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
    if (val & O_SYNC)
        printf(", synchronous writes");

#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC) && (O_FSYNC != O_SYNC)
    if (val & O_FSYNC)
        printf(", synchronous writes");
#endif

    putchar('\n');
    exit(0);
}
```

Figure 3.11 Print file flags for specified descriptor

Note that we use the feature test macro `_POSIX_C_SOURCE` and conditionally compile the file access flags that are not part of POSIX.1. The following script shows the operation of the program, when invoked from bash (the Bourne-again shell). Results will vary, depending on which shell you use.

```
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
$ ./a.out 5 5<>temp.foo
read write
```

The clause 5<>temp.foo opens the file temp.foo for reading and writing on file descriptor 5. □

Example

When we modify either the file descriptor flags or the file status flags, we must be careful to fetch the existing flag value, modify it as desired, and then set the new flag value. We can't simply issue an F_SETFD or an F_SETFL command, as this could turn off flag bits that were previously set.

Figure 3.12 shows a function that sets one or more of the file status flags for a descriptor.

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

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int     val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;       /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

Figure 3.12 Turn on one or more of the file status flags for a descriptor

If we change the middle statement to

```
val &= ~flags;      /* turn flags off */
```

we have a function named clr_fl, which we'll use in some later examples. This statement logically ANDs the one's complement of flags with the current val.

If we add the line

```
set_fl(STDOUT_FILENO, O_SYNC);
```

to the beginning of the program shown in Figure 3.5, we'll turn on the synchronous-write flag. This causes each `write` to wait for the data to be written to disk before returning. Normally in the UNIX System, a `write` only queues the data for writing; the actual disk write operation can take place sometime later. A database system is a likely candidate for using `O_SYNC`, so that it knows on return from a `write` that the data is actually on the disk, in case of an abnormal system failure.

We expect the `O_SYNC` flag to increase the system and clock times when the program runs. To test this, we can run the program in Figure 3.5, copying 492.6 MB of data from one file on disk to another and compare this with a version that does the same thing with the `O_SYNC` flag set. The results from a Linux system using the `ext4` file system are shown in Figure 3.13.

Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
read time from Figure 3.6 for `BUFFSIZE` = 4,096	0.03	0.58	8.62
normal `write` to disk file	0.00	1.05	9.70
`write` to disk file with `O_SYNC` set	0.02	1.09	10.28
`write` to disk followed by `fdatasync`	0.02	1.14	17.93
`write` to disk followed by `fsync`	0.00	1.19	18.17
`write` to disk with `O_SYNC` set followed by `fsync`	0.02	1.15	17.88

Figure 3.13 Linux `ext4` timing results using various synchronization mechanisms

The six rows in Figure 3.13 were all measured with a `BUFFSIZE` of 4,096 bytes. The results in Figure 3.6 were measured while reading a disk file and writing to `/dev/null`, so there was no disk output. The second row in Figure 3.13 corresponds to reading a disk file and writing to another disk file. This is why the first and second rows in Figure 3.13 are different. The system time increases when we write to a disk file, because the kernel now copies the data from our process and queues the data for writing by the disk driver. We expect the clock time to increase as well when we write to a disk file.

When we enable synchronous writes, the system and clock times should increase significantly. As the third row shows, the system time for writing synchronously is not much more expensive than when we used delayed writes. This implies that the Linux operating system is doing the same amount of work for delayed and synchronous writes (which is unlikely), or else the `O_SYNC` flag isn't having the desired effect. In this case, the Linux operating system isn't allowing us to set the `O_SYNC` flag using `fcntl`, instead failing without returning an error (but it would have honored the flag if we were able to specify it when the file was opened).

The clock time in the last three rows reflects the extra time needed to wait for all of the writes to be committed to disk. After writing a file synchronously, we expect that a call to `fsync` will have no effect. This case is supposed to be represented by the last row in Figure 3.13, but since the O_SYNC flag isn't having the intended effect, the last row behaves the same way as the fifth row.

Figure 3.14 shows timing results for the same tests run on Mac OS X 10.6.8, which uses the HFS file system. Note that the times match our expectations: synchronous writes are far more expensive than delayed writes, and using fsync with synchronous writes makes very little difference. Note also that adding a call to fsync at the end of the delayed writes makes little measurable difference. It is likely that the operating system flushed previously written data to disk as we were writing new data to the file, so by the time that we called fsync, very little work was left to be done.

Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
write to /dev/null	0.14	1.02	5.28
normal write to disk file	0.14	3.21	17.04
write to disk file with O_SYNC set	0.39	16.89	60.82
write to disk followed by fsync	0.13	3.07	17.10
write to disk with O_SYNC set followed by fsync	0.39	18.18	62.39

Figure 3.14 Mac OS X HFS timing results using various synchronization mechanisms

Compare fsync and fdatasync, both of which update a file's contents when we say so, with the O_SYNC flag, which updates a file's contents every time we write to the file. The performance of each alternative will depend on many factors, including the underlying operating system implementation, the speed of the disk drive, and the type of the file system. □

With this example, we see the need for fcntl. Our program operates on a descriptor (standard output), never knowing the name of the file that was opened on that descriptor. We can't set the O_SYNC flag when the file is opened, since the shell opened the file. With fcntl, we can modify the properties of a descriptor, knowing only the descriptor for the open file. We'll see another need for fcntl when we describe nonblocking pipes (Section 15.2), since all we have with a pipe is a descriptor.

3.15 ioctl Function

The ioctl function has always been the catchall for I/O operations. Anything that couldn't be expressed using one of the other functions in this chapter usually ended up being specified with an ioctl. Terminal I/O was the biggest user of this function. (When we get to Chapter 18, we'll see that POSIX.1 has replaced the terminal I/O operations with separate functions.)

```
#include <unistd.h>     /* System V */
#include <sys/ioctl.h>  /* BSD and Linux */

int ioctl(int fd, int request, ...);
```

Returns: −1 on error, something else if OK

The `ioctl` function was included in the Single UNIX Specification only as an extension for dealing with STREAMS devices [Rago 1993], but it was moved to obsolescent status in SUSv4. UNIX System implementations use `ioctl` for many miscellaneous device operations. Some implementations have even extended it for use with regular files.

The prototype that we show corresponds to POSIX.1. FreeBSD 8.0 and Mac OS X 10.6.8 declare the second argument as an `unsigned long`. This detail doesn't matter, since the second argument is always a `#define`d name from a header.

For the ISO C prototype, an ellipsis is used for the remaining arguments. Normally, however, there is only one more argument, and it's usually a pointer to a variable or a structure.

In this prototype, we show only the headers required for the function itself. Normally, additional device-specific headers are required. For example, the `ioctl` commands for terminal I/O, beyond the basic operations specified by POSIX.1, all require the `<termios.h>` header.

Each device driver can define its own set of `ioctl` commands. The system, however, provides generic `ioctl` commands for different classes of devices. Examples of some of the categories for these generic `ioctl` commands supported in FreeBSD are summarized in Figure 3.15.

Category	Constant names	Header	Number of `ioctl`s
disk labels	`DIOxxx`	`<sys/disklabel.h>`	4
file I/O	`FIOxxx`	`<sys/filio.h>`	14
mag tape I/O	`MTIOxxx`	`<sys/mtio.h>`	11
socket I/O	`SIOxxx`	`<sys/sockio.h>`	73
terminal I/O	`TIOxxx`	`<sys/ttycom.h>`	43

Figure 3.15 Common FreeBSD `ioctl` operations

The mag tape operations allow us to write end-of-file marks on a tape, rewind a tape, space forward over a specified number of files or records, and the like. None of these operations is easily expressed in terms of the other functions in the chapter (`read`, `write`, `lseek`, and so on), so the easiest way to handle these devices has always been to access their operations using `ioctl`.

We use the `ioctl` function in Section 18.12 to fetch and set the size of a terminal's window, and in Section 19.7 when we access the advanced features of pseudo terminals.

3.16 `/dev/fd`

Newer systems provide a directory named `/dev/fd` whose entries are files named `0`, `1`, `2`, and so on. Opening the file `/dev/fd/`*n* is equivalent to duplicating descriptor *n*, assuming that descriptor *n* is open.

The `/dev/fd` feature was developed by Tom Duff and appeared in the 8th Edition of the Research UNIX System. It is supported by all of the systems described in this book: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10. It is not part of POSIX.1.

In the function call

```
fd = open("/dev/fd/0", mode);
```

most systems ignore the specified `mode`, whereas others require that it be a subset of the mode used when the referenced file (standard input, in this case) was originally opened. Because the previous `open` is equivalent to

```
fd = dup(0);
```

the descriptors 0 and `fd` share the same file table entry (Figure 3.9). For example, if descriptor 0 was opened read-only, we can only read on `fd`. Even if the system ignores the open mode and the call

```
fd = open("/dev/fd/0", O_RDWR);
```

succeeds, we still can't write to `fd`.

The Linux implementation of `/dev/fd` is an exception. It maps file descriptors into symbolic links pointing to the underlying physical files. When you open `/dev/fd/0`, for example, you are really opening the file associated with your standard input. Thus the mode of the new file descriptor returned is unrelated to the mode of the `/dev/fd` file descriptor.

We can also call `creat` with a `/dev/fd` pathname argument as well as specify `O_CREAT` in a call to `open`. This allows a program that calls `creat` to still work if the pathname argument is `/dev/fd/1`, for example.

Beware of doing this on Linux. Because the Linux implementation uses symbolic links to the real files, using `creat` on a `/dev/fd` file will result in the underlying file being truncated.

Some systems provide the pathnames `/dev/stdin`, `/dev/stdout`, and `/dev/stderr`. These pathnames are equivalent to `/dev/fd/0`, `/dev/fd/1`, and `/dev/fd/2`, respectively.

The main use of the `/dev/fd` files is from the shell. It allows programs that use pathname arguments to handle standard input and standard output in the same manner as other pathnames. For example, the `cat`(1) program specifically looks for an input filename of `-` and uses it to mean standard input. The command

```
filter file2 | cat file1 - file3 | lpr
```

is an example. First, `cat` reads `file1`, then its standard input (the output of the `filter` program on `file2`), and then `file3`. If `/dev/fd` is supported, the special handling of `-` can be removed from `cat`, and we can enter

```
filter file2 | cat file1 /dev/fd/0 file3 | lpr
```

The special meaning of `-` as a command-line argument to refer to the standard input or the standard output is a kludge that has crept into many programs. There are also problems if we specify `-` as the first file, as it looks like the start of another command-line option. Using `/dev/fd` is a step toward uniformity and cleanliness.

Files and Directories

4.1 Introduction

In the previous chapter we covered the basic functions that perform I/O. The discussion centered on I/O for regular files—opening a file, and reading or writing a file. We'll now look at additional features of the file system and the properties of a file. We'll start with the `stat` functions and go through each member of the `stat` structure, looking at all the attributes of a file. In this process, we'll also describe each of the functions that modify these attributes: change the owner, change the permissions, and so on. We'll also look in more detail at the structure of a UNIX file system and symbolic links. We finish this chapter with the functions that operate on directories, and we develop a function that descends through a directory hierarchy.

4.2 `stat`, `fstat`, `fstatat`, and `lstat` Functions

The discussion in this chapter centers on the four `stat` functions and the information they return.

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

int stat(const char *restrict pathname, struct stat *restrict buf);

int fstat(int fd, struct stat *buf);

int lstat(const char *restrict pathname, struct stat *restrict buf);

int fstatat(int fd, const char *restrict pathname,
            struct stat *restrict buf, int flag);

                                All four return: 0 if OK, –1 on error
```

Given a *pathname*, the stat function returns a structure of information about the named file. The fstat function obtains information about the file that is already open on the descriptor *fd*. The lstat function is similar to stat, but when the named file is a symbolic link, lstat returns information about the symbolic link, not the file referenced by the symbolic link. (We'll need lstat in Section 4.22 when we walk down a directory hierarchy. We describe symbolic links in more detail in Section 4.17.)

The fstatat function provides a way to return the file statistics for a pathname relative to an open directory represented by the *fd* argument. The *flag* argument controls whether symbolic links are followed; when the AT_SYMLINK_NOFOLLOW flag is set, fstatat will not follow symbolic links, but rather returns information about the link itself. Otherwise, the default is to follow symbolic links, returning information about the file to which the symbolic link points. If the *fd* argument has the value AT_FDCWD and the *pathname* argument is a relative pathname, then fstatat evaluates the *pathname* argument relative to the current directory. If the *pathname* argument is an absolute pathname, then the *fd* argument is ignored. In these two cases, fstatat behaves like either stat or lstat, depending on the value of *flag*.

The *buf* argument is a pointer to a structure that we must supply. The functions fill in the structure. The definition of the structure can differ among implementations, but it could look like

```
struct stat {
  mode_t          st_mode;    /* file type & mode (permissions) */
  ino_t           st_ino;     /* i-node number (serial number) */
  dev_t           st_dev;     /* device number (file system) */
  dev_t           st_rdev;    /* device number for special files */
  nlink_t         st_nlink;   /* number of links */
  uid_t           st_uid;     /* user ID of owner */
  gid_t           st_gid;     /* group ID of owner */
  off_t           st_size;    /* size in bytes, for regular files */
  struct timespec st_atim;    /* time of last access */
  struct timespec st_mtim;    /* time of last modification */
  struct timespec st_ctim;    /* time of last file status change */
  blksize_t       st_blksize; /* best I/O block size */
  blkcnt_t        st_blocks;  /* number of disk blocks allocated */
};
```

The st_rdev, st_blksize, and st_blocks fields are not required by POSIX.1. They are defined as part of the XSI option in the Single UNIX Specification.

The timespec structure type defines time in terms of seconds and nanoseconds. It includes at least the following fields:

```
time_t tv_sec;
long   tv_nsec;
```

Prior to the 2008 version of the standard, the time fields were named st_atime, st_mtime, and st_ctime, and were of type time_t (expressed in seconds). The timespec structure enables higher-resolution timestamps. The old names can be defined in terms of the tv_sec members for compatibility. For example, st_atime can be defined as st_atim.tv_sec.

Note that most members of the `stat` structure are specified by a primitive system data type (see Section 2.8). We'll go through each member of this structure to examine the attributes of a file.

The biggest user of the `stat` functions is probably the `ls -l` command, to learn all the information about a file.

4.3 File Types

We've talked about two different types of files so far: regular files and directories. Most files on a UNIX system are either regular files or directories, but there are additional types of files. The types are

1. Regular file. The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary. Any interpretation of the contents of a regular file is left to the application processing the file.

   One notable exception to this is with binary executable files. To execute a program, the kernel must understand its format. All binary executable files conform to a format that allows the kernel to identify where to load a program's text and data.

2. Directory file. A file that contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of the directory, but only the kernel can write directly to a directory file. Processes must use the functions described in this chapter to make changes to a directory.
3. Block special file. A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.

   Note that FreeBSD no longer supports block special files. All access to devices is through the character special interface.

4. Character special file. A type of file providing unbuffered I/O access in variable-sized units to devices. All devices on a system are either block special files or character special files.
5. FIFO. A type of file used for communication between processes. It's sometimes called a named pipe. We describe FIFOs in Section 15.5.
6. Socket. A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host. We use sockets for interprocess communication in Chapter 16.
7. Symbolic link. A type of file that points to another file. We talk more about symbolic links in Section 4.17.

The type of a file is encoded in the `st_mode` member of the `stat` structure. We can determine the file type with the macros shown in Figure 4.1. The argument to each of these macros is the `st_mode` member from the `stat` structure.

Macro	Type of file
`S_ISREG()`	regular file
`S_ISDIR()`	directory file
`S_ISCHR()`	character special file
`S_ISBLK()`	block special file
`S_ISFIFO()`	pipe or FIFO
`S_ISLNK()`	symbolic link
`S_ISSOCK()`	socket

Figure 4.1 File type macros in `<sys/stat.h>`

POSIX.1 allows implementations to represent interprocess communication (IPC) objects, such as message queues and semaphores, as files. The macros shown in Figure 4.2 allow us to determine the type of IPC object from the `stat` structure. Instead of taking the `st_mode` member as an argument, these macros differ from those in Figure 4.1 in that their argument is a pointer to the `stat` structure.

Macro	Type of object
`S_TYPEISMQ()`	message queue
`S_TYPEISSEM()`	semaphore
`S_TYPEISSHM()`	shared memory object

Figure 4.2 IPC type macros in `<sys/stat.h>`

Message queues, semaphores, and shared memory objects are discussed in Chapter 15. However, none of the various implementations of the UNIX System discussed in this book represent these objects as files.

Example

The program in Figure 4.3 prints the type of file for each command-line argument.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))
```

```
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

Figure 4.3 Print type of file for each command-line argument

Sample output from Figure 4.3 is

```
$ ./a.out /etc/passwd /etc /dev/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/log: socket
/dev/tty: character special
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: block special
/dev/cdrom: symbolic link
```

(Here, we have explicitly entered a backslash at the end of the first command line, telling the shell that we want to continue entering the command on another line. The shell then prompted us with its secondary prompt, >, on the next line.) We have specifically used the `lstat` function instead of the `stat` function to detect symbolic links. If we used the `stat` function, we would never see symbolic links. □

Historically, early versions of the UNIX System didn't provide the `S_ISxxx` macros. Instead, we had to logically AND the `st_mode` value with the mask `S_IFMT` and then compare the result with the constants whose names are `S_IFxxx`. Most systems define this mask and the related constants in the file `<sys/stat.h>`. If we examine this file, we'll find the `S_ISDIR` macro defined something like

```
#define  S_ISDIR(mode)  (((mode) & S_IFMT) == S_IFDIR)
```

We've said that regular files are predominant, but it is interesting to see what percentage of the files on a given system are of each file type. Figure 4.4 shows the counts and percentages for a Linux system that is used as a single-user workstation. This data was obtained from the program shown in Section 4.22.

File type	Count	Percentage
regular file	415,803	79.77 %
directory	62,197	11.93
symbolic link	40,018	8.25
character special	155	0.03
block special	47	0.01
socket	45	0.01
FIFO	0	0.00

Figure 4.4 Counts and percentages of different file types

4.4 Set-User-ID and Set-Group-ID

Every process has six or more IDs associated with it. These are shown in Figure 4.5.

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by `exec` functions

Figure 4.5 User IDs and group IDs associated with each process

- The real user ID and real group ID identify who we really are. These two fields are taken from our entry in the password file when we log in. Normally, these values don't change during a login session, although there are ways for a superuser process to change them, which we describe in Section 8.11.
- The effective user ID, effective group ID, and supplementary group IDs determine our file access permissions, as we describe in the next section. (We defined supplementary group IDs in Section 1.8.)
- The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and the effective group ID, respectively, when a program is executed. We describe the function of these two saved values when we describe the `setuid` function in Section 8.11.

The saved IDs are required as of the 2001 version of POSIX.1. They were optional in older versions of POSIX. An application can test for the constant `_POSIX_SAVED_IDS` at compile time or can call `sysconf` with the `_SC_SAVED_IDS` argument at runtime, to see whether the implementation supports this feature.

Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID.

Every file has an owner and a group owner. The owner is specified by the `st_uid` member of the `stat` structure; the group owner, by the `st_gid` member.

When we execute a program file, the effective user ID of the process is usually the real user ID, and the effective group ID is usually the real group ID. However, we can also set a special flag in the file's mode word (`st_mode`) that says, "When this file is executed, set the effective user ID of the process to be the owner of the file (`st_uid`)." Similarly, we can set another bit in the file's mode word that causes the effective group ID to be the group owner of the file (`st_gid`). These two bits in the file's mode word are called the *set-user-ID* bit and the *set-group-ID* bit.

For example, if the owner of the file is the superuser and if the file's set-user-ID bit is set, then while that program file is running as a process, it has superuser privileges. This happens regardless of the real user ID of the process that executes the file. As an example, the UNIX System program that allows anyone to change his or her password, `passwd`(1), is a set-user-ID program. This is required so that the program can write the new password to the password file, typically either `/etc/passwd` or `/etc/shadow`, files that should be writable only by the superuser. Because a process that is running set-user-ID to some other user usually assumes extra permissions, it must be written carefully. We'll discuss these types of programs in more detail in Chapter 8.

Returning to the `stat` function, the set-user-ID bit and the set-group-ID bit are contained in the file's `st_mode` value. These two bits can be tested against the constants `S_ISUID` and `S_ISGID`, respectively.

4.5 File Access Permissions

The `st_mode` value also encodes the access permission bits for the file. When we say *file*, we mean any of the file types that we described earlier. All the file types—directories, character special files, and so on—have permissions. Many people think of only regular files as having access permissions.

There are nine permission bits for each file, divided into three categories. They are shown in Figure 4.6.

`st_mode` mask	Meaning
`S_IRUSR`	user-read
`S_IWUSR`	user-write
`S_IXUSR`	user-execute
`S_IRGRP`	group-read
`S_IWGRP`	group-write
`S_IXGRP`	group-execute
`S_IROTH`	other-read
`S_IWOTH`	other-write
`S_IXOTH`	other-execute

Figure 4.6 The nine file access permission bits, from `<sys/stat.h>`

The term *user* in the first three rows in Figure 4.6 refers to the owner of the file. The `chmod`(1) command, which is typically used to modify these nine permission bits, allows us to specify `u` for user (owner), `g` for group, and `o` for other. Some books refer to these three as owner, group, and world; this is confusing, as the `chmod` command uses o to mean other, not owner. We'll use the terms *user*, *group*, and *other*, to be consistent with the chmod command.

The three categories in Figure 4.6—read, write, and execute—are used in various ways by different functions. We'll summarize them here, and return to them when we describe the actual functions.

- The first rule is that *whenever* we want to open any type of file by name, we must have execute permission in each directory mentioned in the name, including the current directory, if it is implied. This is why the execute permission bit for a directory is often called the search bit.

  For example, to open the file /usr/include/stdio.h, we need execute permission in the directory /, execute permission in the directory /usr, and execute permission in the directory /usr/include. We then need appropriate permission for the file itself, depending on how we're trying to open it: read-only, read–write, and so on.

  If the current directory is /usr/include, then we need execute permission in the current directory to open the file stdio.h. This is an example of the current directory being implied, not specifically mentioned. It is identical to our opening the file ./stdio.h.

  Note that read permission for a directory and execute permission for a directory mean different things. Read permission lets us read the directory, obtaining a list of all the filenames in the directory. Execute permission lets us pass through the directory when it is a component of a pathname that we are trying to access. (We need to search the directory to look for a specific filename.)

  Another example of an implicit directory reference is if the PATH environment variable, described in Section 8.10, specifies a directory that does not have execute permission enabled. In this case, the shell will never find executable files in that directory.

- The read permission for a file determines whether we can open an existing file for reading: the O_RDONLY and O_RDWR flags for the open function.

- The write permission for a file determines whether we can open an existing file for writing: the O_WRONLY and O_RDWR flags for the open function.

- We must have write permission for a file to specify the O_TRUNC flag in the open function.

- We cannot create a new file in a directory unless we have write permission and execute permission in the directory.

- To delete an existing file, we need write permission and execute permission in the directory containing the file. We do not need read permission or write permission for the file itself.

- Execute permission for a file must be on if we want to execute the file using any of the seven exec functions (Section 8.10). The file also has to be a regular file.

The file access tests that the kernel performs each time a process opens, creates, or deletes a file depend on the owners of the file (`st_uid` and `st_gid`), the effective IDs of the process (effective user ID and effective group ID), and the supplementary group IDs of the process, if supported. The two owner IDs are properties of the file, whereas the two effective IDs and the supplementary group IDs are properties of the process. The tests performed by the kernel are as follows:

1. If the effective user ID of the process is 0 (the superuser), access is allowed. This gives the superuser free rein throughout the entire file system.
2. If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied. By *appropriate access permission bit*, we mean that if the process is opening the file for reading, the user-read bit must be on. If the process is opening the file for writing, the user-write bit must be on. If the process is executing the file, the user-execute bit must be on.
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied.
4. If the appropriate other access permission bit is set, access is allowed. Otherwise, permission is denied.

These four steps are tried in sequence. Note that if the process owns the file (step 2), access is granted or denied based only on the user access permissions; the group permissions are never looked at. Similarly, if the process does not own the file but belongs to an appropriate group, access is granted or denied based only on the group access permissions; the other permissions are not looked at.

4.6 Ownership of New Files and Directories

When we described the creation of a new file in Chapter 3 using either `open` or `creat`, we never said which values were assigned to the user ID and group ID of the new file. We'll see how to create a new directory in Section 4.21 when we describe the `mkdir` function. The rules for the ownership of a new directory are identical to the rules in this section for the ownership of a new file.

The user ID of a new file is set to the effective user ID of the process. POSIX.1 allows an implementation to choose one of the following options to determine the group ID of a new file:

1. The group ID of a new file can be the effective group ID of the process.
2. The group ID of a new file can be the group ID of the directory in which the file is being created.

FreeBSD 8.0 and Mac OS X 10.6.8 always copy the new file's group ID from the directory. Several Linux file systems allow the choice between the two options to be selected using a `mount(1)` command option. The default behavior for Linux 3.2.0 and Solaris 10 is to determine the group ID of a new file depending on whether the set-group-ID bit is set for the directory in which the file is created. If this bit is set, the new file's group ID is copied from the directory; otherwise, the new file's group ID is set to the effective group ID of the process.

Using the second option—inheriting the directory's group ID—assures us that all files and directories created in that directory will have the same group ID as the directory. This group ownership of files and directories will then propagate down the hierarchy from that point. This is used in the Linux directory `/var/mail`, for example.

As we mentioned earlier, this option for group ownership is the default for FreeBSD 8.0 and Mac OS X 10.6.8, but an option for Linux and Solaris. Under Solaris 10, and by default under Linux 3.2.0, we have to enable the set-group-ID bit, and the `mkdir` function has to propagate a directory's set-group-ID bit automatically for this to work. (This is described in Section 4.21.)

4.7 access and faccessat Functions

As we described earlier, when we open a file, the kernel performs its access tests based on the effective user and group IDs. Sometimes, however, a process wants to test accessibility based on the real user and group IDs. This is useful when a process is running as someone else, using either the set-user-ID or the set-group-ID feature. Even though a process might be set-user-ID to root, it might still want to verify that the real user can access a given file. The `access` and `faccessat` functions base their tests on the real user and group IDs. (Replace *effective* with *real* in the four steps at the end of Section 4.5.)

```
#include <unistd.h>

int access(const char *pathname, int mode);

int faccessat(int fd, const char *pathname, int mode, int flag);

                                        Both return: 0 if OK, –1 on error
```

The *mode* is either the value `F_OK` to test if a file exists, or the bitwise OR of any of the flags shown in Figure 4.7.

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission

Figure 4.7 The *mode* flags for `access` function, from `<unistd.h>`

The `faccessat` function behaves like `access` when the *pathname* argument is absolute or when the *fd* argument has the value `AT_FDCWD` and the *pathname* argument is relative. Otherwise, `faccessat` evaluates the *pathname* relative to the open directory referenced by the *fd* argument.

The *flag* argument can be used to change the behavior of `faccessat`. If the `AT_EACCESS` flag is set, the access checks are made using the effective user and group IDs of the calling process instead of the real user and group IDs.

Example

Figure 4.8 shows the use of the `access` function.

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

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");
    exit(0);
}
```

Figure 4.8 Example of access function

Here is a sample session with this program:

```
$ ls -l a.out
-rwxrwxr-x  1 sar         15945 Nov 30 12:10 a.out
$ ./a.out a.out
read access OK
open for reading OK
$ ls -l /etc/shadow
-r--------  1 root         1315 Jul 17  2002 /etc/shadow
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open error for /etc/shadow: Permission denied
$ su                                  become superuser
Password:                             enter superuser password
# chown root a.out                    change file's user ID to root
# chmod u+s a.out                     and turn on set-user-ID bit
# ls -l a.out                         check owner and SUID bit
-rwsrwxr-x  1 root        15945 Nov 30 12:10 a.out
# exit                                go back to normal user
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open for reading OK
```

In this example, the set-user-ID program can determine that the real user cannot normally read the file, even though the open function will succeed. □

In the preceding example and in Chapter 8, we'll sometimes switch to become the superuser to demonstrate how something works. If you're on a multiuser system and do not have superuser permission, you won't be able to duplicate these examples completely.

4.8 umask Function

Now that we've described the nine permission bits associated with every file, we can describe the file mode creation mask that is associated with every process.

The umask function sets the file mode creation mask for the process and returns the previous value. (This is one of the few functions that doesn't have an error return.)

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

mode_t umask(mode_t cmask);

                              Returns: previous file mode creation mask
```

The *cmask* argument is formed as the bitwise OR of any of the nine constants from Figure 4.6: S_IRUSR, S_IWUSR, and so on.

The file mode creation mask is used whenever the process creates a new file or a new directory. (Recall from Sections 3.3 and 3.4 our description of the open and creat functions. Both accept a *mode* argument that specifies the new file's access permission bits.) We describe how to create a new directory in Section 4.21. Any bits that are *on* in the file mode creation mask are turned *off* in the file's *mode*.

Example

The program in Figure 4.9 creates two files: one with a umask of 0 and one with a umask that disables all the group and other permission bits.

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

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

Figure 4.9 Example of umask function

If we run this program, we can see how the permission bits have been set.

```
$ umask                          first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw-------  1 sar                0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar                0 Dec  7 21:20 foo
$ umask                          see if the file mode creation mask changed
002
```

□

Most users of UNIX systems never deal with their `umask` value. It is usually set once, on login, by the shell's start-up file, and never changed. Nevertheless, when writing programs that create new files, if we want to ensure that specific access permission bits are enabled, we must modify the `umask` value while the process is running. For example, if we want to ensure that anyone can read a file, we should set the `umask` to 0. Otherwise, the `umask` value that is in effect when our process is running can cause permission bits to be turned off.

In the preceding example, we use the shell's `umask` command to print the file mode creation mask both before we run the program and after it completes. This shows us that changing the file mode creation mask of a process doesn't affect the mask of its parent (often a shell). All of the shells have a built-in `umask` command that we can use to set or print the current file mode creation mask.

Users can set the `umask` value to control the default permissions on the files they create. This value is expressed in octal, with one bit representing one permission to be masked off, as shown in Figure 4.10. Permissions can be denied by setting the corresponding bits. Some common `umask` values are 002 to prevent others from writing your files, 022 to prevent group members and others from writing your files, and 027 to prevent group members from writing your files and others from reading, writing, or executing your files.

Mask bit	Meaning
0400	user-read
0200	user-write
0100	user-execute
0040	group-read
0020	group-write
0010	group-execute
0004	other-read
0002	other-write
0001	other-execute

Figure 4.10 The `umask` file access permission bits

The Single UNIX Specification requires that the `umask` command support a symbolic mode of operation. Unlike the octal format, the symbolic format specifies which permissions are to be allowed (i.e., clear in the file creation mask) instead of which ones are to be denied (i.e., set in the file creation mask). Compare both forms of the command, shown below.

```
$ umask                              first print the current file mode creation mask
002
$ umask -S                           print the symbolic form
u=rwx,g=rwx,o=rx
$ umask 027                          change the file mode creation mask
$ umask -S                           print the symbolic form
u=rwx,g=rx,o=
```

4.9 chmod, fchmod, and fchmodat Functions

The chmod, fchmod, and fchmodat functions allow us to change the file access permissions for an existing file.

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

int chmod(const char *pathname, mode_t mode);

int fchmod(int fd, mode_t mode);

int fchmodat(int fd, const char *pathname, mode_t mode, int flag);

                                   All three return: 0 if OK, −1 on error
```

The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened. The fchmodat function behaves like chmod when the *pathname* argument is absolute or when the *fd* argument has the value AT_FDCWD and the *pathname* argument is relative. Otherwise, fchmodat evaluates the *pathname* relative to the open directory referenced by the *fd* argument. The *flag* argument can be used to change the behavior of fchmodat—when the AT_SYMLINK_NOFOLLOW flag is set, fchmodat doesn't follow symbolic links.

To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have superuser permissions.

The *mode* is specified as the bitwise OR of the constants shown in Figure 4.11.

mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

Figure 4.11 The *mode* constants for chmod functions, from <sys/stat.h>

Note that nine of the entries in Figure 4.11 are the nine file access permission bits from Figure 4.6. We've added the two set-ID constants (S_ISUID and S_ISGID), the saved-text constant (S_ISVTX), and the three combined constants (S_IRWXU, S_IRWXG, and S_IRWXO).

The saved-text bit (S_ISVTX) is not part of POSIX.1. It is defined in the XSI option in the Single UNIX Specification. We describe its purpose in the next section.

Example

Recall the final state of the files foo and bar when we ran the program in Figure 4.9 to demonstrate the umask function:

```
$ ls -l foo bar
-rw-------  1 sar             0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar             0 Dec  7 21:20 foo
```

The program shown in Figure 4.12 modifies the mode of these two files.

```
#include "apue.h"

int
main(void)
{
    struct stat     statbuf;

    /* turn on set-group-ID and turn off group-execute */

    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");

    /* set absolute mode to "rw-r--r--" */

    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("chmod error for bar");

    exit(0);
}
```

Figure 4.12 Example of chmod function

After running the program in Figure 4.12, we see that the final state of the two files is

```
$ ls -l foo bar
-rw-r--r--  1 sar             0 Dec  7 21:20 bar
-rw-rwSrw-  1 sar             0 Dec  7 21:20 foo
```

In this example, we have set the permissions of the file bar to an absolute value, regardless of the current permission bits. For the file foo, we set the permissions relative to their current state. To do this, we first call stat to obtain the current permissions and then modify them. We have explicitly turned on the set-group-ID bit and turned off the group-execute bit. Note that the ls command lists the group-execute permission as S to signify that the set-group-ID bit is set without the group-execute bit being set.

On Solaris, the `ls` command displays an `l` instead of an `S` to indicate that mandatory file and record locking has been enabled for this file. This behavior applies only to regular files, but we'll discuss this more in Section 14.3.

Finally, note that the time and date listed by the `ls` command did not change after we ran the program in Figure 4.12. We'll see in Section 4.19 that the `chmod` function updates only the time that the i-node was last changed. By default, the `ls -l` lists the time when the contents of the file were last modified. □

The `chmod` functions automatically clear two of the permission bits under the following conditions:

- On systems, such as Solaris, that place special meaning on the sticky bit when used with regular files, if we try to set the sticky bit (`S_ISVTX`) on a regular file and do not have superuser privileges, the sticky bit in the *mode* is automatically turned off. (We describe the sticky bit in the next section.) To prevent malicious users from setting the sticky bit and adversely affecting system performance, only the superuser can set the sticky bit of a regular file.

  In FreeBSD 8.0 and Solaris 10, only the superuser can set the sticky bit on a regular file. Linux 3.2.0 and Mac OS X 10.6.8 place no such restriction on the setting of the sticky bit, because the bit has no meaning when applied to regular files on these systems. Although the bit also has no meaning when applied to regular files on FreeBSD, everyone except the superuser is prevented from setting it on a regular file.

- The group ID of a newly created file might potentially be a group that the calling process does not belong to. Recall from Section 4.6 that it's possible for the group ID of the new file to be the group ID of the parent directory. Specifically, if the group ID of the new file does not equal either the effective group ID of the process or one of the process's supplementary group IDs and if the process does not have superuser privileges, then the set-group-ID bit is automatically turned off. This prevents a user from creating a set-group-ID file owned by a group that the user doesn't belong to.

  FreeBSD 8.0 fails an attempt to set the set-group-ID in this case. The other systems silently turn the bit off, but don't fail the attempt to change the file access permissions.

  FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 add another security feature to try to prevent misuse of some of the protection bits. If a process that does not have superuser privileges writes to a file, the set-user-ID and set-group-ID bits are automatically turned off. If malicious users find a set-group-ID or a set-user-ID file they can write to, even though they can modify the file, they lose the special privileges of the file.

4.10 Sticky Bit

The `S_ISVTX` bit has an interesting history. On versions of the UNIX System that predated demand paging, this bit was known as the *sticky bit*. If it was set for an executable program file, then the first time the program was executed, a copy of the program's text was saved in the swap area when the process terminated. (The text portion of a program is the machine instructions.) The program would then load into memory more quickly the next time it was executed, because the swap area was handled as a contiguous file, as compared to the possibly random location of data blocks in a normal UNIX file system. The sticky bit was often set for common application programs, such as the text editor and the passes of the C compiler. Naturally, there was a limit to the number of sticky files that could be contained in the swap area before running out of swap space, but it was a useful technique. The name *sticky* came about because the text portion of the file stuck around in the swap area until the system was rebooted. Later versions of the UNIX System referred to this as the *saved-text* bit; hence the constant `S_ISVTX`. With today's newer UNIX systems, most of which have a virtual memory system and a faster file system, the need for this technique has disappeared.

On contemporary systems, the use of the sticky bit has been extended. The Single UNIX Specification allows the sticky bit to be set for a directory. If the bit is set for a directory, a file in the directory can be removed or renamed only if the user has write permission for the directory and meets one of the following criteria:

- Owns the file
- Owns the directory
- Is the superuser

The directories `/tmp` and `/var/tmp` are typical candidates for the sticky bit—they are directories in which any user can typically create files. The permissions for these two directories are often read, write, and execute for everyone (user, group, and other). But users should not be able to delete or rename files owned by others.

The saved-text bit is not part of POSIX.1. It is part of the XSI option defined in the Single UNIX Specification, and is supported by FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10.

Solaris 10 places special meaning on the sticky bit if it is set on a regular file. In this case, if none of the execute bits is set, the operating system will not cache the contents of the file.

4.11 chown, fchown, fchownat, and lchown Functions

The `chown` functions allow us to change a file's user ID and group ID, but if either of the arguments *owner* or *group* is −1, the corresponding ID is left unchanged.

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group);

int fchownat(int fd, const char *pathname, uid_t owner, gid_t group,
             int flag);

int lchown(const char *pathname, uid_t owner, gid_t group);

                                    All four return: 0 if OK, −1 on error
```

These four functions operate similarly unless the referenced file is a symbolic link. In that case, `lchown` and `fchownat` (with the `AT_SYMLINK_NOFOLLOW` flag set) change the owners of the symbolic link itself, not the file pointed to by the symbolic link.

The `fchown` function changes the ownership of the open file referenced by the *fd* argument. Since it operates on a file that is already open, it can't be used to change the ownership of a symbolic link.

The `fchownat` function behaves like either `chown` or `lchown` when the *pathname* argument is absolute or when the *fd* argument has the value `AT_FDCWD` and the *pathname* argument is relative. In these cases, `fchownat` acts like `lchown` if the `AT_SYMLINK_NOFOLLOW` flag is set in the *flag* argument, or it acts like `chown` if the `AT_SYMLINK_NOFOLLOW` flag is clear. When the *fd* argument is set to the file descriptor of an open directory and the *pathname* argument is a relative pathname, `fchownat` evaluates the *pathname* relative to the open directory.

Historically, BSD-based systems have enforced the restriction that only the superuser can change the ownership of a file. This is to prevent users from giving away their files to others, thereby defeating any disk space quota restrictions. System V, however, has allowed all users to change the ownership of any files they own.

> POSIX.1 allows either form of operation, depending on the value of `_POSIX_CHOWN_RESTRICTED`.
>
> With Solaris 10, this functionality is a configuration option, whose default value is to enforce the restriction. FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 always enforce the `chown` restriction.

Recall from Section 2.6 that the `_POSIX_CHOWN_RESTRICTED` constant can optionally be defined in the header `<unistd.h>`, and can always be queried using either the `pathconf` function or the `fpathconf` function. Also recall that this option can depend on the referenced file; it can be enabled or disabled on a per file system basis. We'll use the phrase "if `_POSIX_CHOWN_RESTRICTED` is in effect," to mean "if it applies to the particular file that we're talking about," regardless of whether this actual constant is defined in the header.

If `_POSIX_CHOWN_RESTRICTED` is in effect for the specified file, then

1. Only a superuser process can change the user ID of the file.
2. A nonsuperuser process can change the group ID of the file if the process owns the file (the effective user ID equals the user ID of the file), *owner* is specified as −1 or equals the user ID of the file, and *group* equals either the effective group ID of the process or one of the process's supplementary group IDs.

This means that when `_POSIX_CHOWN_RESTRICTED` is in effect, you can't change the user ID of your files. You can change the group ID of files that you own, but only to groups that you belong to.

If these functions are called by a process other than a superuser process, on successful return, both the set-user-ID and the set-group-ID bits are cleared.

4.12 File Size

The `st_size` member of the `stat` structure contains the size of the file in bytes. This field is meaningful only for regular files, directories, and symbolic links.

FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10 also define the file size for a pipe as the number of bytes that are available for reading from the pipe. We'll discuss pipes in Section 15.2.

For a regular file, a file size of 0 is allowed. We'll get an end-of-file indication on the first read of the file. For a directory, the file size is usually a multiple of a number, such as 16 or 512. We talk about reading directories in Section 4.22.

For a symbolic link, the file size is the number of bytes in the filename. For example, in the following case, the file size of 7 is the length of the pathname `usr/lib`:

```
lrwxrwxrwx  1 root           7 Sep 25 07:14 lib -> usr/lib
```

(Note that symbolic links do not contain the normal C null byte at the end of the name, as the length is always specified by `st_size`.)

Most contemporary UNIX systems provide the fields `st_blksize` and `st_blocks`. The first is the preferred block size for I/O for the file, and the latter is the actual number of 512-byte blocks that are allocated. Recall from Section 3.9 that we encountered the minimum amount of time required to read a file when we used `st_blksize` for the `read` operations. The standard I/O library, which we describe in Chapter 5, also tries to read or write `st_blksize` bytes at a time, for efficiency.

Be aware that different versions of the UNIX System use units other than 512-byte blocks for `st_blocks`. Use of this value is nonportable.

Holes in a File

In Section 3.6, we mentioned that a regular file can contain "holes." We showed an example of this in Figure 3.2. Holes are created by seeking past the current end of file and writing some data. As an example, consider the following:

```
$ ls -l core
-rw-r--r--  1 sar   8483248 Nov 18 12:18 core
$ du -s core
272     core
```

The size of the file `core` is slightly more than 8 MB, yet the `du` command reports that the amount of disk space used by the file is 272 512-byte blocks (139,264 bytes). Obviously, this file has many holes.

The du command on many BSD-derived systems reports the number of 1,024-byte blocks. Solaris reports the number of 512-byte blocks. On Linux, the units reported depend on the whether the `POSIXLY_CORRECT` environment is set. When it is set, the du command reports 1,024-byte block units; when it is not set, the command reports 512-byte block units.

As we mentioned in Section 3.6, the `read` function returns data bytes of 0 for any byte positions that have not been written. If we execute the following command, we can see that the normal I/O operations read up through the size of the file:

```
$ wc -c core
 8483248 core
```

The wc(1) command with the -c option counts the number of characters (bytes) in the file.

If we make a copy of this file, using a utility such as cat(1), all these holes are written out as actual data bytes of 0:

```
$ cat core > core.copy
$ ls -l core*
-rw-r--r--  1 sar   8483248 Nov 18 12:18 core
-rw-rw-r--  1 sar   8483248 Nov 18 12:27 core.copy
$ du -s core*
272     core
16592   core.copy
```

Here, the actual number of bytes used by the new file is 8,495,104 (512 × 16,592). The difference between this size and the size reported by `ls` is caused by the number of blocks used by the file system to hold pointers to the actual data blocks.

Interested readers should refer to Section 4.2 of Bach [1986], Sections 7.2 and 7.3 of McKusick et al. [1996] (or Sections 8.2 and 8.3 in McKusick and Neville-Neil [2005]), Section 15.2 of McDougall and Mauro [2007], and Chapter 12 in Singh [2006] for additional details on the physical layout of files.

4.13 File Truncation

Sometimes we would like to truncate a file by chopping off data at the end of the file. Emptying a file, which we can do with the `O_TRUNC` flag to `open`, is a special case of truncation.

```
#include <unistd.h>

int truncate(const char *pathname, off_t length);

int ftruncate(int fd, off_t length);

                                        Both return: 0 if OK, −1 on error
```

These two functions truncate an existing file to *length* bytes. If the previous size of the file was greater than *length*, the data beyond *length* is no longer accessible. Otherwise, if the previous size was less than *length*, the file size will increase and the data between the old end of file and the new end of file will read as 0 (i.e., a hole is probably created in the file).

BSD releases prior to 4.4BSD could only make a file smaller with `truncate`.

Solaris also includes an extension to `fcntl` (`F_FREESP`) that allows us to free any part of a file, not just a chunk at the end of the file.

We use `ftruncate` in the program shown in Figure 13.6 when we need to empty a file after obtaining a lock on the file.

4.14 File Systems

To appreciate the concept of links to a file, we need a conceptual understanding of the structure of the UNIX file system. Understanding the difference between an i-node and a directory entry that points to an i-node is also useful.

Various implementations of the UNIX file system are in use today. Solaris, for example, supports several types of disk file systems: the traditional BSD-derived UNIX file system (called `UFS`), a file system (called `PCFS`) to read and write DOS-formatted diskettes, and a file system (called `HSFS`) to read CD file systems. We saw one difference between file system types in Figure 2.20. `UFS` is based on the Berkeley fast file system, which we describe in this section.

Each file system type has its own characteristic features—and some of these features can be confusing. For example, most UNIX file systems support case-sensitive filenames. Thus, if you create one file named `file.txt` and another named `file.TXT`, then two distinct files are created. On Mac OS X, however, the `HFS` file system is case-preserving with case-insensitive comparisons. Thus, if you create `file.txt`, when you try to create `file.TXT`, you will overwrite `file.txt`. However, only the name used when the file was created is stored in the file system (the case-preserving aspect). In fact, any permutation of uppercase and lowercase letters in the sequence `f`, `i`, `l`, `e`, `.`, `t`, `x`, `t` will match when searching for the file (the case-insensitive comparison aspect). As a consequence, besides `file.txt` and `file.TXT`, we can access the file with the names `File.txt`, `fILE.tXt`, and `FiLe.TxT`.

We can think of a disk drive being divided into one or more partitions. Each partition can contain a file system, as shown in Figure 4.13. The i-nodes are fixed-length entries that contain most of the information about a file.

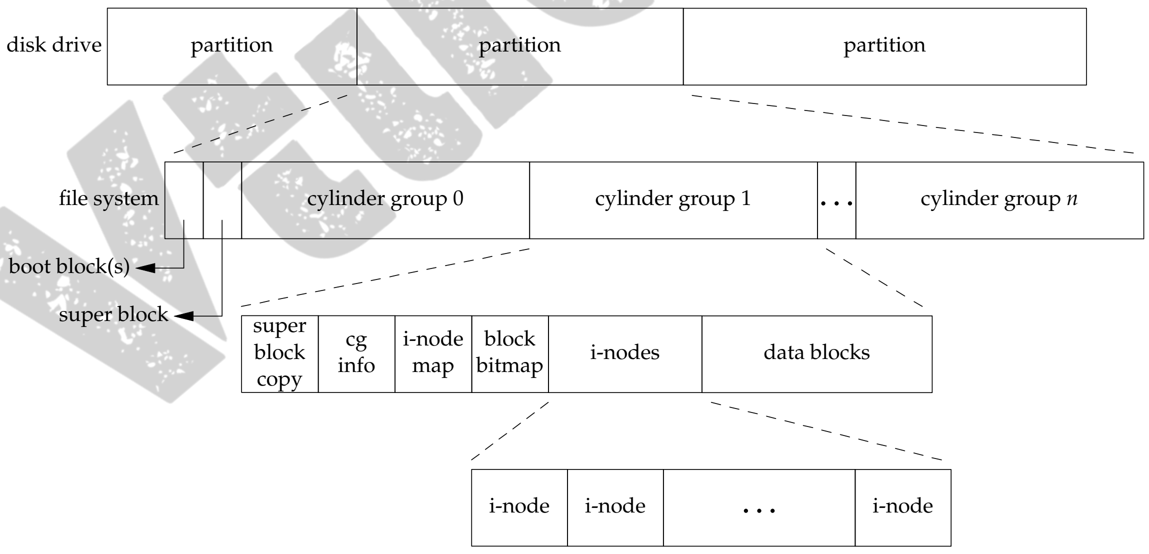

Figure 4.13 Disk drive, partitions, and a file system

If we examine the i-node and data block portion of a cylinder group in more detail, we could have the arrangement shown in Figure 4.14.

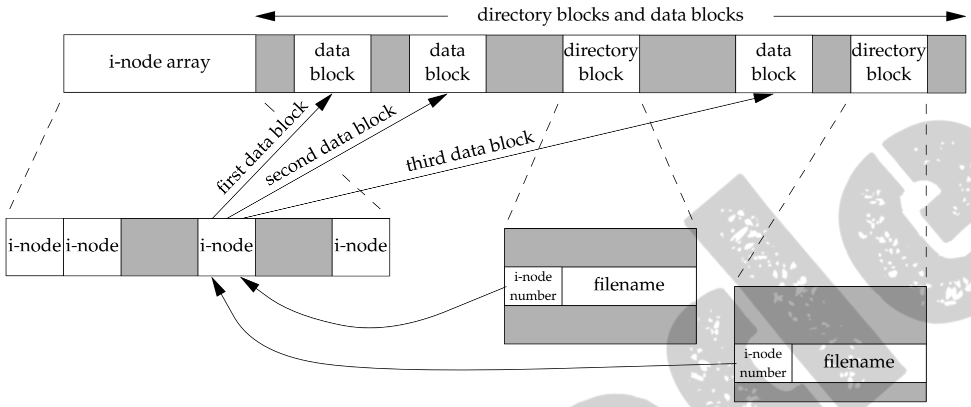

Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

Note the following points from Figure 4.14.

- Two directory entries point to the same i-node entry. Every i-node has a link count that contains the number of directory entries that point to it. Only when the link count goes to 0 can the file be deleted (thereby releasing the data blocks associated with the file). This is why the operation of "unlinking a file" does not always mean "deleting the blocks associated with the file." This is why the function that removes a directory entry is called `unlink`, not delete. In the `stat` structure, the link count is contained in the `st_nlink` member. Its primitive system data type is `nlink_t`. These types of links are called hard links. Recall from Section 2.5.2 that the POSIX.1 constant `LINK_MAX` specifies the maximum value for a file's link count.
- The other type of link is called a *symbolic link*. With a symbolic link, the actual contents of the file—the data blocks—store the name of the file that the symbolic link points to. In the following example, the filename in the directory entry is the three-character string `lib` and the 7 bytes of data in the file are `usr/lib`:

```
lrwxrwxrwx  1 root      7 Sep 25 07:14 lib -> usr/lib
```

  The file type in the i-node would be `S_IFLNK` so that the system knows that this is a symbolic link.
- The i-node contains all the information about the file: the file type, the file's access permission bits, the size of the file, pointers to the file's data blocks, and so on. Most of the information in the `stat` structure is obtained from the i-node. Only two items of interest are stored in the directory entry: the filename and the i-node number. The other items—the length of the filename and the length of the directory record—are not of interest to this discussion. The data type for the i-node number is `ino_t`.

- Because the i-node number in the directory entry points to an i-node in the same file system, a directory entry can't refer to an i-node in a different file system. This is why the `ln`(1) command (make a new directory entry that points to an existing file) can't cross file systems. We describe the `link` function in the next section.
- When renaming a file without changing file systems, the actual contents of the file need not be moved—all that needs to be done is to add a new directory entry that points to the existing i-node and then unlink the old directory entry. The link count will remain the same. For example, to rename the file `/usr/lib/foo` to `/usr/foo`, the contents of the file `foo` need not be moved if the directories `/usr/lib` and `/usr` are on the same file system. This is how the `mv`(1) command usually operates.

We've talked about the concept of a link count for a regular file, but what about the link count field for a directory? Assume that we make a new directory in the working directory, as in

```
$ mkdir testdir
```

Figure 4.15 shows the result. Note that in this figure, we explicitly show the entries for dot and dot-dot.

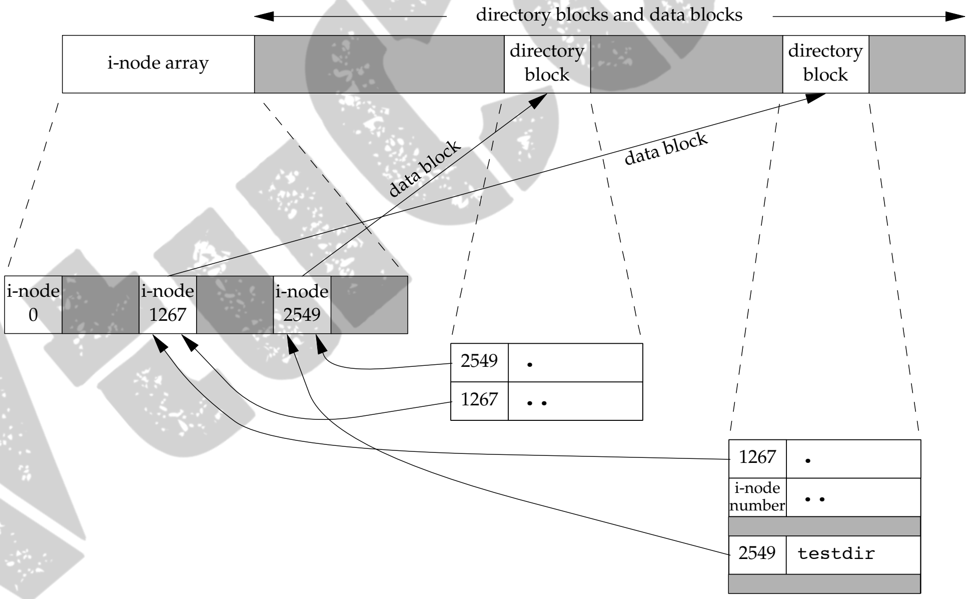

Figure 4.15 Sample cylinder group after creating the directory `testdir`

The i-node whose number is 2549 has a type field of "directory" and a link count equal to 2. Any leaf directory (a directory that does not contain any other directories) always has a link count of 2. The value of 2 comes from the directory entry that names the directory (`testdir`) and from the entry for dot in that directory. The i-node whose number is 1267 has a type field of "directory" and a link count that is greater than or equal to 3. We know that this link count is greater than or equal to 3 because, at a minimum, the i-node is pointed to from the directory entry that names it (which we don't show in Figure 4.15), from dot, and from dot-dot in the `testdir` directory. Note that every subdirectory in a parent directory causes the parent directory's link count to be increased by 1.

This format is similar to the classic format of the UNIX file system, which is described in detail in Chapter 4 of Bach [1986]. Refer to Chapter 7 of McKusick et al. [1996] or Chapter 8 of McKusick and Neville-Neil [2005] for additional information on the changes made with the Berkeley fast file system. See Chapter 15 of McDougall and Mauro [2007] for details on `UFS`, the Solaris version of the Berkeley fast file system. For information on the `HFS` file system format used in Mac OS X, see Chapter 12 of Singh [2006].

4.15 `link`, `linkat`, `unlink`, `unlinkat`, and `remove` Functions

As we saw in the previous section, a file can have multiple directory entries pointing to its i-node. We can use either the `link` function or the `linkat` function to create a link to an existing file.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);

int linkat(int efd, const char *existingpath, int nfd, const char *newpath,
           int flag);

                                        Both return: 0 if OK, −1 on error
```

These functions create a new directory entry, *newpath*, that references the existing file *existingpath*. If the *newpath* already exists, an error is returned. Only the last component of the *newpath* is created. The rest of the path must already exist.

With the `linkat` function, the existing file is specified by both the *efd* and *existingpath* arguments, and the new pathname is specified by both the *nfd* and *newpath* arguments. By default, if either pathname is relative, it is evaluated relative to the corresponding file descriptor. If either file descriptor is set to `AT_FDCWD`, then the corresponding pathname, if it is a relative pathname, is evaluated relative to the current directory. If either pathname is absolute, then the corresponding file descriptor argument is ignored.

When the existing file is a symbolic link, the *flag* argument controls whether the `linkat` function creates a link to the symbolic link or to the file to which the symbolic link points. If the `AT_SYMLINK_FOLLOW` flag is set in the *flag* argument, then a link is created to the target of the symbolic link. If this flag is clear, then a link is created to the symbolic link itself.

The creation of the new directory entry and the increment of the link count must be an atomic operation. (Recall the discussion of atomic operations in Section 3.11.)

Most implementations require that both pathnames be on the same file system, although POSIX.1 allows an implementation to support linking across file systems. If an implementation supports the creation of hard links to directories, it is restricted to only the superuser. This constraint exists because such hard links can cause loops in the file system, which most utilities that process the file system aren't capable of handling. (We show an example of a loop introduced by a symbolic link in Section 4.17.) Many file system implementations disallow hard links to directories for this reason.

To remove an existing directory entry, we call the `unlink` function.

```
#include <unistd.h>

int unlink(const char *pathname);

int unlinkat(int fd, const char *pathname, int flag);

                                    Both return: 0 if OK, –1 on error
```

These functions remove the directory entry and decrement the link count of the file referenced by *pathname*. If there are other links to the file, the data in the file is still accessible through the other links. The file is not changed if an error occurs.

As mentioned earlier, to unlink a file, we must have write permission and execute permission in the directory containing the directory entry, as it is the directory entry that we will be removing. Also, as mentioned in Section 4.10, if the sticky bit is set in this directory we must have write permission for the directory and meet one of the following criteria:

- Own the file
- Own the directory
- Have superuser privileges

Only when the link count reaches 0 can the contents of the file be deleted. One other condition prevents the contents of a file from being deleted: as long as some process has the file open, its contents will not be deleted. When a file is closed, the kernel first checks the count of the number of processes that have the file open. If this count has reached 0, the kernel then checks the link count; if it is 0, the file's contents are deleted.

If the *pathname* argument is a relative pathname, then the `unlinkat` function evaluates the pathname relative to the directory represented by the *fd* file descriptor argument. If the *fd* argument is set to the value `AT_FDCWD`, then the pathname is evaluated relative to the current working directory of the calling process. If the *pathname* argument is an absolute pathname, then the *fd* argument is ignored.

The *flag* argument gives callers a way to change the default behavior of the `unlinkat` function. When the `AT_REMOVEDIR` flag is set, then the `unlinkat` function can be used to remove a directory, similar to using `rmdir`. If this flag is clear, then `unlinkat` operates like `unlink`.

Example

The program shown in Figure 4.16 opens a file and then unlinks it. The program then goes to sleep for 15 seconds before terminating.

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

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");
    if (unlink("tempfile") < 0)
        err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
```

Figure 4.16 Open a file and then `unlink` it

Running this program gives us

```
$ ls -l tempfile                   look at how big the file is
-rw-r-----  1 sar     413265408 Jan 21 07:14 tempfile
$ df /home                         check how much free space is available
Filesystem   1K-blocks     Used  Available  Use%  Mounted on
/dev/hda4     11021440  1956332    9065108   18%  /home
$ ./a.out &                        run the program in Figure 4.16 in the background
1364                               the shell prints its process ID
$ file unlinked                    the file is unlinked
ls -l tempfile                     see if the filename is still there
ls: tempfile: No such file or directory     the directory entry is gone
$ df /home                         see if the space is available yet
Filesystem   1K-blocks     Used  Available  Use%  Mounted on
/dev/hda4     11021440  1956332    9065108   18%  /home
$ done                             the program is done, all open files are closed
df /home                           now the disk space should be available
Filesystem   1K-blocks     Used  Available  Use%  Mounted on
/dev/hda4     11021440  1552352    9469088   15%  /home
                                   now the 394.1 MB of disk space are available
```
□

This property of `unlink` is often used by a program to ensure that a temporary file it creates won't be left around in case the program crashes. The process creates a file using either `open` or `creat` and then immediately calls `unlink`. The file is not deleted, however, because it is still open. Only when the process either closes the file or terminates, which causes the kernel to close all its open files, is the file deleted.

If *pathname* is a symbolic link, `unlink` removes the symbolic link, not the file referenced by the link. There is no function to remove the file referenced by a symbolic link given the name of the link.

The superuser can call `unlink` with *pathname* specifying a directory if the file system supports it, but the function `rmdir` should be used instead to unlink a directory. We describe the `rmdir` function in Section 4.21.

We can also unlink a file or a directory with the `remove` function. For a file, `remove` is identical to `unlink`. For a directory, `remove` is identical to `rmdir`.

```
#include <stdio.h>

int remove(const char *pathname);

                                        Returns: 0 if OK, −1 on error
```

ISO C specifies the `remove` function to delete a file. The name was changed from the historical UNIX name of `unlink` because most non-UNIX systems that implement the C standard didn't support the concept of links to a file at the time.

4.16 rename and renameat Functions

A file or a directory is renamed with either the `rename` or `renameat` function.

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);

int renameat(int oldfd, const char *oldname, int newfd,
             const char *newname);

                                   Both return: 0 if OK, −1 on error
```

The `rename` function is defined by ISO C for files. (The C standard doesn't deal with directories.) POSIX.1 expanded the definition to include directories and symbolic links.

There are several conditions to describe for these functions, depending on whether *oldname* refers to a file, a directory, or a symbolic link. We must also describe what happens if *newname* already exists.

1. If *oldname* specifies a file that is not a directory, then we are renaming a file or a symbolic link. In this case, if *newname* exists, it cannot refer to a directory. If *newname* exists and is not a directory, it is removed, and *oldname* is renamed to *newname*. We must have write permission for the directory containing *oldname* and the directory containing *newname*, since we are changing both directories.
2. If *oldname* specifies a directory, then we are renaming a directory. If *newname* exists, it must refer to a directory, and that directory must be empty. (When we say that a directory is empty, we mean that the only entries in the directory are dot and dot-dot.) If *newname* exists and is an empty directory, it is removed, and *oldname* is renamed to *newname*. Additionally, when we're renaming a directory, *newname* cannot contain a path prefix that names *oldname*. For example, we can't rename `/usr/foo` to `/usr/foo/testdir`, because the old name (`/usr/foo`) is a path prefix of the new name and cannot be removed.

3. If either *oldname* or *newname* refers to a symbolic link, then the link itself is processed, not the file to which it resolves.
4. We can't rename dot or dot-dot. More precisely, neither dot nor dot-dot can appear as the last component of *oldname* or *newname*.
5. As a special case, if *oldname* and *newname* refer to the same file, the function returns successfully without changing anything.

If *newname* already exists, we need permissions as if we were deleting it. Also, because we're removing the directory entry for *oldname* and possibly creating a directory entry for *newname*, we need write permission and execute permission in the directory containing *oldname* and in the directory containing *newname*.

The `renameat` function provides the same functionality as the `rename` function, except when either *oldname* or *newname* refers to a relative pathname. If *oldname* specifies a relative pathname, it is evaluated relative to the directory referenced by *oldfd*. Similarly, *newname* is evaluated relative to the directory referenced by *newfd* if *newname* specifies a relative pathname. Either the *oldfd* or *newfd* arguments (or both) can be set to `AT_FDCWD` to evaluate the corresponding pathname relative to the current directory.

4.17 Symbolic Links

A symbolic link is an indirect pointer to a file, unlike the hard links described in the previous section, which pointed directly to the i-node of the file. Symbolic links were introduced to get around the limitations of hard links.

- Hard links normally require that the link and the file reside in the same file system.
- Only the superuser can create a hard link to a directory (when supported by the underlying file system).

There are no file system limitations on a symbolic link and what it points to, and anyone can create a symbolic link to a directory. Symbolic links are typically used to "move" a file or an entire directory hierarchy to another location on a system.

When using functions that refer to a file by name, we always need to know whether the function follows a symbolic link. If the function follows a symbolic link, a pathname argument to the function refers to the file pointed to by the symbolic link. Otherwise, a pathname argument refers to the link itself, not the file pointed to by the link. Figure 4.17 summarizes whether the functions described in this chapter follow a symbolic link. The functions `mkdir`, `mkfifo`, `mknod`, and `rmdir` do not appear in this figure, as they return an error when the pathname is a symbolic link. Also, the functions that take a file descriptor argument, such as `fstat` and `fchmod`, are not listed, as the function that returns the file descriptor (usually `open`) handles the symbolic link. Historically, implementations have differed in whether `chown` follows symbolic links. In all modern systems, however, `chown` does follow symbolic links.

Symbolic links were introduced with 4.2BSD. Initially, chown didn't follow symbolic links, but this behavior was changed in 4.4BSD. System V included support for symbolic links in SVR4, but diverged from the original BSD behavior by implementing chown to follow symbolic links. In older versions of Linux (those before version 2.1.81), chown didn't follow symbolic links. From version 2.1.81 onward, chown follows symbolic links. With FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10, chown follows symbolic links. All of these platforms provide implementations of lchown to change the ownership of symbolic links themselves.

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown		•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

Figure 4.17 Treatment of symbolic links by various functions

One exception to the behavior summarized in Figure 4.17 occurs when the open function is called with both O_CREAT and O_EXCL set. In this case, if the pathname refers to a symbolic link, open will fail with errno set to EEXIST. This behavior is intended to close a security hole so that privileged processes can't be fooled into writing to the wrong files.

Example

It is possible to introduce loops into the file system by using symbolic links. Most functions that look up a pathname return an errno of ELOOP when this occurs. Consider the following commands:

```
$ mkdir foo                       make a new directory
$ touch foo/a                     create a 0-length file
$ ln -s ../foo foo/testdir        create a symbolic link
$ ls -l foo
total 0
-rw-r-----  1 sar          0 Jan 22 00:16 a
lrwxrwxrwx  1 sar          6 Jan 22 00:16 testdir -> ../foo
```

This creates a directory `foo` that contains the file `a` and a symbolic link that points to `foo`. We show this arrangement in Figure 4.18, drawing a directory as a circle and a file as a square.

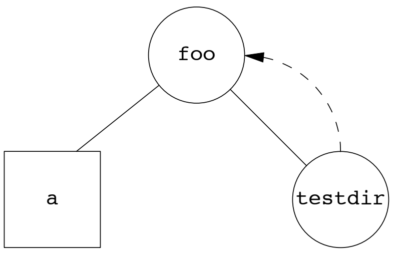

Figure 4.18 Symbolic link `testdir` that creates a loop

If we write a simple program that uses the standard function `ftw`(3) on Solaris to descend through a file hierarchy, printing each pathname encountered, the output is

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
```

(many more lines until we encounter an `ELOOP` error)

In Section 4.22, we provide our own version of the `ftw` function that uses `lstat` instead of `stat`, to prevent it from following symbolic links.

Note that on Linux, the `ftw` and `nftw` functions record all directories seen and avoid processing a directory more than once, so they don't display this behavior.

A loop of this form is easy to remove. We can `unlink` the file `foo/testdir`, as `unlink` does not follow a symbolic link. But if we create a hard link that forms a loop of this type, its removal is much more difficult. This is why the `link` function will not form a hard link to a directory unless the process has superuser privileges.

Indeed, Rich Stevens did this on his own system as an experiment while writing the original version of this section. The file system got corrupted and the normal `fsck`(1) utility couldn't fix things. The deprecated tools `clri`(8) and `dcheck`(8) were needed to repair the file system.

The need for hard links to directories has long since passed. With symbolic links and the `mkdir` function, there is no longer any need for users to create hard links to directories.

When we open a file, if the pathname passed to `open` specifies a symbolic link, `open` follows the link to the specified file. If the file pointed to by the symbolic link doesn't exist, `open` returns an error saying that it can't open the file. This response can confuse users who aren't familiar with symbolic links. For example,

```
$ ln -s /no/such/file myfile              create a symbolic link
$ ls myfile
myfile                                    ls says it's there
$ cat myfile                              so we try to look at it
cat: myfile: No such file or directory
$ ls -l myfile                            try -l option
lrwxrwxrwx  1 sar       13 Jan 22 00:26 myfile -> /no/such/file
```

The file `myfile` does exist, yet `cat` says there is no such file, because `myfile` is a symbolic link and the file pointed to by the symbolic link doesn't exist. The `-l` option to `ls` gives us two hints: the first character is an `l`, which means a symbolic link, and the sequence `->` also indicates a symbolic link. The `ls` command has another option (`-F`) that appends an at-sign (@) to filenames that are symbolic links, which can help us spot symbolic links in a directory listing without the `-l` option. □

4.18 Creating and Reading Symbolic Links

A symbolic link is created with either the `symlink` or `symlinkat` function.

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);

int symlinkat(const char *actualpath, int fd, const char *sympath);

                                   Both return: 0 if OK, −1 on error
```

A new directory entry, *sympath*, is created that points to *actualpath*. It is not required that *actualpath* exist when the symbolic link is created. (We saw this in the example at the end of the previous section.) Also, *actualpath* and *sympath* need not reside in the same file system.

The `symlinkat` function is similar to `symlink`, but the *sympath* argument is evaluated relative to the directory referenced by the open file descriptor for that directory (specified by the *fd* argument). If the *sympath* argument specifies an absolute pathname or if the *fd* argument has the special value `AT_FDCWD`, then `symlinkat` behaves the same way as `symlink`.

Because the `open` function follows a symbolic link, we need a way to open the link itself and read the name in the link. The `readlink` and `readlinkat` functions do this.

```
#include <unistd.h>

ssize_t readlink(const char* restrict pathname, char *restrict buf,
                 size_t bufsize);

ssize_t readlinkat(int fd, const char* restrict pathname,
                   char *restrict buf, size_t bufsize);

                  Both return: number of bytes read if OK, −1 on error
```

These functions combine the actions of `open`, `read`, and `close`. If successful, they return the number of bytes placed into *buf*. The contents of the symbolic link that are returned in *buf* are not null terminated.

The `readlinkat` function behaves the same way as the `readlink` function when the *pathname* argument specifies an absolute pathname or when the *fd* argument has the special value `AT_FDCWD`. However, when the *fd* argument is a valid file descriptor of an open directory and the *pathname* argument is a relative pathname, then `readlinkat` evaluates the pathname relative to the open directory represented by *fd*.

4.19 File Times

In Section 4.2, we discussed how the 2008 version of the Single UNIX Specification increased the resolution of the time fields in the `stat` structure from seconds to seconds plus nanoseconds. The actual resolution stored with each file's attributes depends on the file system implementation. For file systems that store timestamps in second granularity, the nanoseconds fields will be filled with zeros. For file systems that store timestamps in a resolution higher than seconds, the partial seconds value will be converted into nanoseconds and returned in the nanoseconds fields.

Three time fields are maintained for each file. Their purpose is summarized in Figure 4.19.

Field	Description	Example	`ls`(1) option
`st_atim`	last-access time of file data	`read`	`-u`
`st_mtim`	last-modification time of file data	`write`	default
`st_ctim`	last-change time of i-node status	`chmod, chown`	`-c`

Figure 4.19 The three time values associated with each file

Note the difference between the modification time (`st_mtim`) and the changed-status time (`st_ctim`). The modification time indicates when the contents of the file were last modified. The changed-status time indicates when the i-node of the file was last modified. In this chapter, we've described many operations that affect the i-node without changing the actual contents of the file: changing the file access permissions, changing the user ID, changing the number of links, and so on. Because all the information in the i-node is stored separately from the actual contents of the file, we need the changed-status time, in addition to the modification time.

Note that the system does not maintain the last-access time for an i-node. This is why the functions `access` and `stat`, for example, don't change any of the three times.

The access time is often used by system administrators to delete files that have not been accessed for a certain amount of time. The classic example is the removal of files named `a.out` or `core` that haven't been accessed in the past week. The `find`(1) command is often used for this type of operation.

The modification time and the changed-status time can be used to archive only those files that have had their contents modified or their i-node modified.

The `ls` command displays or sorts only on one of the three time values. By default, when invoked with either the `-l` or the `-t` option, it uses the modification time of a file. The `-u` option causes the `ls` command to use the access time, and the `-c` option causes it to use the changed-status time.

Figure 4.20 summarizes the effects of the various functions that we've described on these three times. Recall from Section 4.14 that a directory is simply a file containing directory entries: filenames and associated i-node numbers. Adding, deleting, or modifying these directory entries can affect the three times associated with that directory. This is why Figure 4.20 contains one column for the three times associated with the file or directory and another column for the three times associated with the parent directory of the referenced file or directory. For example, creating a new file affects the directory that contains the new file, and it affects the i-node for the new file. Reading or writing a file, however, affects only the i-node of the file and has no effect on the directory.

Function	Referenced file or directory			Parent directory of referenced file or directory			Section	Note
	a	m	c	a	m	c		
`chmod`, `fchmod`			•				4.9	
`chown`, `fchown`			•				4.11	
`creat`	•	•	•		•	•	3.4	`O_CREAT` new file
`creat`		•	•				3.4	`O_TRUNC` existing file
`exec`	•						8.10	
`lchown`			•				4.11	
`link`			•		•	•	4.15	parent of second argument
`mkdir`	•	•	•		•	•	4.21	
`mkfifo`	•	•	•		•	•	15.5	
`open`	•	•	•		•	•	3.3	`O_CREAT` new file
`open`		•	•				3.3	`O_TRUNC` existing file
`pipe`	•	•	•				15.2	
`read`	•						3.7	
`remove`			•		•	•	4.15	remove file = `unlink`
`remove`					•	•	4.15	remove directory = `rmdir`
`rename`			•		•	•	4.16	for both arguments
`rmdir`					•	•	4.21	
`truncate`, `ftruncate`		•	•				4.13	
`unlink`			•		•	•	4.15	
`utimes`, `utimensat`, `futimens`	•	•	•				4.20	
`write`		•	•				3.8	

Figure 4.20 Effect of various functions on the access, modification, and changed-status times

(The `mkdir` and `rmdir` functions are covered in Section 4.21. The `utimes`, `utimensat`, and `futimens` functions are covered in the next section. The seven `exec` functions are described in Section 8.10. We describe the `mkfifo` and `pipe` functions in Chapter 15.)

4.20 futimens, utimensat, and utimes Functions

Several functions are available to change the access time and the modification time of a file. The futimens and utimensat functions provide nanosecond granularity for specifying timestamps, using the timespec structure (the same structure used by the stat family of functions; see Section 4.2).

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

int futimens(int fd, const struct timespec times[2]);

int utimensat(int fd, const char *path, const struct timespec times[2],
              int flag);

                                        Both return: 0 if OK, −1 on error
```

In both functions, the first element of the *times* array argument contains the access time, and the second element contains the modification time. The two time values are calendar times, which count seconds since the Epoch, as described in Section 1.10. Partial seconds are expressed in nanoseconds.

Timestamps can be specified in one of four ways:

1. The *times* argument is a null pointer. In this case, both timestamps are set to the current time.
2. The *times* argument points to an array of two timespec structures. If either tv_nsec field has the special value UTIME_NOW, the corresponding timestamp is set to the current time. The corresponding tv_sec field is ignored.
3. The *times* argument points to an array of two timespec structures. If either tv_nsec field has the special value UTIME_OMIT, then the corresponding timestamp is unchanged. The corresponding tv_sec field is ignored.
4. The *times* argument points to an array of two timespec structures and the tv_nsec field contains a value other than UTIME_NOW or UTIME_OMIT. In this case, the corresponding timestamp is set to the value specified by the corresponding tv_sec and tv_nsec fields.

The privileges required to execute these functions depend on the value of the *times* argument.

- If *times* is a null pointer or if either tv_nsec field is set to UTIME_NOW, either the effective user ID of the process must equal the owner ID of the file, the process must have write permission for the file, or the process must be a superuser process.
- If *times* is a non-null pointer and either tv_nsec field has a value other than UTIME_NOW or UTIME_OMIT, the effective user ID of the process must equal the owner ID of the file, or the process must be a superuser process. Merely having write permission for the file is not adequate.

- If *times* is a non-null pointer and both `tv_nsec` fields are set to `UTIME_OMIT`, no permissions checks are performed.

With `futimens`, you need to open the file to change its times. The `utimensat` function provides a way to change a file's times using the file's name. The *pathname* argument is evaluated relative to the *fd* argument, which is either a file descriptor of an open directory or the special value `AT_FDCWD` to force evaluation relative to the current directory of the calling process. If *pathname* specifies an absolute pathname, then the *fd* argument is ignored.

The *flag* argument to `utimensat` can be used to further modify the default behavior. If the `AT_SYMLINK_NOFOLLOW` flag is set, then the times of the symbolic link itself are changed (if the pathname refers to a symbolic link). The default behavior is to follow a symbolic link and modify the times of the file to which the link refers.

Both `futimens` and `utimensat` are included in POSIX.1. A third function, `utimes`, is included in the Single UNIX Specification as part of the XSI option.

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

int utimes(const char *pathname, const struct timeval times[2]);

                                        Returns: 0 if OK, –1 on error
```

The `utimes` function operates on a pathname. The *times* argument is a pointer to an array of two timestamps—access time and modification time—but they are expressed in seconds and microseconds:

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

Note that we are unable to specify a value for the changed-status time, `st_ctim`—the time the i-node was last changed—as this field is automatically updated when the `utime` function is called.

On some versions of the UNIX System, the `touch`(1) command uses one of these functions. Also, the standard archive programs, `tar`(1) and `cpio`(1), optionally call these functions to set a file's times to the time values saved when the file was archived.

Example

The program shown in Figure 4.21 truncates files to zero length using the `O_TRUNC` option of the `open` function, but does not change their access time or modification time. To do this, the program first obtains the times with the `stat` function, truncates the file, and then resets the times with the `futimens` function.

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

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

```
{
    int             i, fd;
    struct stat     statbuf;
    struct timespec times[2];

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) {  /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* truncate */
            err_ret("%s: open error", argv[i]);
            continue;
        }
        times[0] = statbuf.st_atim;
        times[1] = statbuf.st_mtim;
        if (futimens(fd, times) < 0)        /* reset times */
            err_ret("%s: futimens error", argv[i]);
        close(fd);
    }
    exit(0);
}
```

Figure 4.21 Example of `futimens` function

We can demonstrate the program in Figure 4.21 on Linux with the following commands:

```
$ ls -l changemod times          look at sizes and last-modification times
-rwxr-xr-x  1 sar   13792 Jan 22 01:26 changemod
-rwxr-xr-x  1 sar   13824 Jan 22 01:26 times
$ ls -lu changemod times         look at last-access times
-rwxr-xr-x  1 sar   13792 Jan 22 22:22 changemod
-rwxr-xr-x  1 sar   13824 Jan 22 22:22 times
$ date                           print today's date
Fri Jan 27 20:53:46 EST 2012
$ ./a.out changemod times        run the program in Figure 4.21
$ ls -l changemod times          and check the results
-rwxr-xr-x  1 sar       0 Jan 22 01:26 changemod
-rwxr-xr-x  1 sar       0 Jan 22 01:26 times
$ ls -lu changemod times         check the last-access times also
-rwxr-xr-x  1 sar       0 Jan 22 22:22 changemod
-rwxr-xr-x  1 sar       0 Jan 22 22:22 times
$ ls -lc changemod times         and the changed-status times
-rwxr-xr-x  1 sar       0 Jan 27 20:53 changemod
-rwxr-xr-x  1 sar       0 Jan 27 20:53 times
```

As we would expect, the last-modification times and the last-access times have not changed. The changed-status times, however, have changed to the time that the program was run. □

4.21 mkdir, mkdirat, and rmdir Functions

Directories are created with the mkdir and mkdirat functions, and deleted with the rmdir function.

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

int mkdir(const char *pathname, mode_t mode);

int mkdirat(int fd, const char *pathname, mode_t mode);

                                        Both return: 0 if OK, −1 on error
```

These functions create a new, empty directory. The entries for dot and dot-dot are created automatically. The specified file access permissions, *mode*, are modified by the file mode creation mask of the process.

A common mistake is to specify the same *mode* as for a file: read and write permissions only. But for a directory, we normally want at least one of the execute bits enabled, to allow access to filenames within the directory. (See Exercise 4.16.)

The user ID and group ID of the new directory are established according to the rules we described in Section 4.6.

Solaris 10 and Linux 3.2.0 also have the new directory inherit the set-group-ID bit from the parent directory. Files created in the new directory will then inherit the group ID of that directory. With Linux, the file system implementation determines whether this behavior is supported. For example, the ext2, ext3, and ext4 file systems allow this behavior to be controlled by an option to the mount(1) command. With the Linux implementation of the UFS file system, however, the behavior is not selectable; it inherits the set-group-ID bit to mimic the historical BSD implementation, where the group ID of a directory is inherited from the parent directory.

BSD-based implementations don't propagate the set-group-ID bit; they simply inherit the group ID as a matter of policy. Because FreeBSD 8.0 and Mac OS X 10.6.8 are based on 4.4BSD, they do not require inheriting the set-group-ID bit. On these platforms, newly created files and directories always inherit the group ID of the parent directory, regardless of whether the set-group-ID bit is set.

Earlier versions of the UNIX System did not have the mkdir function; it was introduced with 4.2BSD and SVR3. In the earlier versions, a process had to call the mknod function to create a new directory—but use of the mknod function was restricted to superuser processes. To circumvent this constraint, the normal command that created a directory, mkdir(1), had to be owned by root with the set-user-ID bit on. To create a directory from a process, the mkdir(1) command had to be invoked with the system(3) function.

The mkdirat function is similar to the mkdir function. When the *fd* argument has the special value AT_FDCWD, or when the *pathname* argument specifies an absolute pathname, mkdirat behaves exactly like mkdir. Otherwise, the *fd* argument is an open directory from which relative pathnames will be evaluated.

An empty directory is deleted with the rmdir function. Recall that an empty directory is one that contains entries only for dot and dot-dot.

```
#include <unistd.h>

int rmdir(const char *pathname);

                                        Returns: 0 if OK, −1 on error
```

If the link count of the directory becomes 0 with this call, and if no other process has the directory open, then the space occupied by the directory is freed. If one or more processes have the directory open when the link count reaches 0, the last link is removed and the dot and dot-dot entries are removed before this function returns. Additionally, no new files can be created in the directory. The directory is not freed, however, until the last process closes it. (Even though some other process has the directory open, it can't be doing much in the directory, as the directory had to be empty for the `rmdir` function to succeed.)

4.22 Reading Directories

Directories can be read by anyone who has access permission to read the directory. But only the kernel can write to a directory, to preserve file system sanity. Recall from Section 4.5 that the write permission bits and execute permission bits for a directory determine if we can create new files in the directory and remove files from the directory—they don't specify if we can write to the directory itself.

The actual format of a directory depends on the UNIX System implementation and the design of the file system. Earlier systems, such as Version 7, had a simple structure: each directory entry was 16 bytes, with 14 bytes for the filename and 2 bytes for the i-node number. When longer filenames were added to 4.2BSD, each entry became variable length, which means that any program that reads a directory is now system dependent. To simplify the process of reading a directory, a set of directory routines were developed and are part of POSIX.1. Many implementations prevent applications from using the `read` function to access the contents of directories, thereby further isolating applications from the implementation-specific details of directory formats.

```
#include <dirent.h>

DIR *opendir(const char *pathname);

DIR *fdopendir(int fd);

                            Both return: pointer if OK, NULL on error

struct dirent *readdir(DIR *dp);

              Returns: pointer if OK, NULL at end of directory or error

void rewinddir(DIR *dp);

int closedir(DIR *dp);

                                        Returns: 0 if OK, −1 on error

long telldir(DIR *dp);

             Returns: current location in directory associated with dp

void seekdir(DIR *dp, long loc);
```

The fdopendir function first appeared in version 4 of the Single UNIX Specification. It provides a way to convert an open file descriptor into a DIR structure for use by the directory handling functions.

The telldir and seekdir functions are not part of the base POSIX.1 standard. They are included in the XSI option in the Single UNIX Specification, so all conforming UNIX System implementations are expected to provide them.

Recall our use of several of these functions in the program shown in Figure 1.3, our bare-bones implementation of the ls command.

The dirent structure defined in <dirent.h> is implementation dependent. Implementations define the structure to contain at least the following two members:

```
ino_t  d_ino;                    /* i-node number */
char   d_name[];                 /* null-terminated filename */
```

The d_ino entry is not defined by POSIX.1, because it is an implementation feature, but it is defined as part of the XSI option in POSIX.1. POSIX.1 defines only the d_name entry in this structure.

Note that the size of the d_name entry isn't specified, but it is guaranteed to hold at least NAME_MAX characters, not including the terminating null byte (recall Figure 2.15.) Since the filename is null terminated, however, it doesn't matter how d_name is defined in the header, because the array size doesn't indicate the length of the filename.

The DIR structure is an internal structure used by these seven functions to maintain information about the directory being read. The purpose of the DIR structure is similar to that of the FILE structure maintained by the standard I/O library, which we describe in Chapter 5.

The pointer to a DIR structure returned by opendir and fdopendir is then used with the other five functions. The opendir function initializes things so that the first readdir returns the first entry in the directory. When the DIR structure is created by fdopendir, the first entry returned by readdir depends on the file offset associated with the file descriptor passed to fdopendir. Note that the ordering of entries within the directory is implementation dependent and is usually not alphabetical.

Example

We'll use these directory routines to write a program that traverses a file hierarchy. The goal is to produce a count of the various types of files shown in Figure 4.4. The program shown in Figure 4.22 takes a single argument—the starting pathname—and recursively descends the hierarchy from that point. Solaris provides a function, ftw(3), that performs the actual traversal of the hierarchy, calling a user-defined function for each file. The problem with this function is that it calls the stat function for each file, which causes the program to follow symbolic links. For example, if we start at the root and have a symbolic link named /lib that points to /usr/lib, all the files in the directory /usr/lib are counted twice. To correct this problem, Solaris provides an additional function, nftw(3), with an option that stops it from following symbolic links. Although we could use nftw, we'll write our own simple file walker to show the use of the directory routines.

In SUSv4, `nftw` is included as part of the XSI option. Implementations are included in FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10. (In SUSv4, the `ftw` function has been marked as obsolescent.) BSD-based systems have a different function, `fts`(3), that provides similar functionality. It is available in FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8.

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

/* function type that is called for each filename */
typedef int Myfunc(const char *, const struct stat *, int);

static Myfunc   myfunc;
static int      myftw(char *, Myfunc *);
static int      dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

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

    if (argc != 2)
        err_quit("usage:  ftw  <starting-pathname>");

    ret = myftw(argv[1], myfunc);       /* does it all */

    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1;       /* avoid divide by 0; print 0 for all counts */
    printf("regular files  = %7ld, %5.2f %%\n", nreg,
      nreg*100.0/ntot);
    printf("directories    = %7ld, %5.2f %%\n", ndir,
      ndir*100.0/ntot);
    printf("block special  = %7ld, %5.2f %%\n", nblk,
      nblk*100.0/ntot);
    printf("char special   = %7ld, %5.2f %%\n", nchr,
      nchr*100.0/ntot);
    printf("FIFOs          = %7ld, %5.2f %%\n", nfifo,
      nfifo*100.0/ntot);
    printf("symbolic links = %7ld, %5.2f %%\n", nslink,
      nslink*100.0/ntot);
    printf("sockets        = %7ld, %5.2f %%\n", nsock,
      nsock*100.0/ntot);
    exit(ret);
}

/*
 * Descend through the hierarchy, starting at "pathname".
 * The caller's func() is called for every file.
 */
#define FTW_F   1       /* file other than directory */
#define FTW_D   2       /* directory */
```

```
#define FTW_DNR 3       /* directory that can't be read */
#define FTW_NS  4       /* file that we can't stat */

static char *fullpath;      /* contains full pathname for every file */
static size_t pathlen;

static int                  /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    fullpath = path_alloc(&pathlen);    /* malloc PATH_MAX+1 bytes */
                                        /* (Figure 2.16) */
    if (pathlen <= strlen(pathname)) {
        pathlen = strlen(pathname) * 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    strcpy(fullpath, pathname);
    return(dopath(func));
}

/*
 * Descend through the hierarchy, starting at "fullpath".
 * If "fullpath" is anything other than a directory, we lstat() it,
 * call func(), and return.  For a directory, we call ourself
 * recursively for each name in the directory.
 */
static int                  /* we return whatever func() returns */
dopath(Myfunc* func)
{
    struct stat     statbuf;
    struct dirent   *dirp;
    DIR             *dp;
    int             ret, n;

    if (lstat(fullpath, &statbuf) < 0)  /* stat error */
        return(func(fullpath, &statbuf, FTW_NS));
    if (S_ISDIR(statbuf.st_mode) == 0)  /* not a directory */
        return(func(fullpath, &statbuf, FTW_F));

    /*
     * It's a directory.  First call func() for the directory,
     * then process each filename in the directory.
     */
    if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    n = strlen(fullpath);
    if (n + NAME_MAX + 2 > pathlen) {   /* expand path buffer */
        pathlen *= 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    fullpath[n++] = '/';
    fullpath[n] = 0;

    if ((dp = opendir(fullpath)) == NULL)   /* can't read directory */
        return(func(fullpath, &statbuf, FTW_DNR));

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0  ||
            strcmp(dirp->d_name, "..") == 0)
                continue;       /* ignore dot and dot-dot */
        strcpy(&fullpath[n], dirp->d_name); /* append name after "/" */
        if ((ret = dopath(func)) != 0)      /* recursive */
            break;  /* time to leave */
    }
    fullpath[n-1] = 0;  /* erase everything from slash onward */

    if (closedir(dp) < 0)
        err_ret("can't close directory %s", fullpath);
    return(ret);
}

static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;     break;
        case S_IFBLK:   nblk++;     break;
        case S_IFCHR:   nchr++;     break;
        case S_IFIFO:   nfifo++;    break;
        case S_IFLNK:   nslink++;   break;
        case S_IFSOCK:  nsock++;    break;
        case S_IFDIR:   /* directories should have type = FTW_D */
            err_dump("for S_IFDIR for %s", pathname);
        }
        break;
    case FTW_D:
        ndir++;
        break;
    case FTW_DNR:
        err_ret("can't read directory %s", pathname);
        break;
    case FTW_NS:
        err_ret("stat error for %s", pathname);
        break;
    default:
        err_dump("unknown type %d for pathname %s", type, pathname);
    }
    return(0);
}
```

Figure 4.22 Recursively descend a directory hierarchy, counting file types

To illustrate the ftw and nftw functions, we have provided more generality in this program than needed. For example, the function myfunc always returns 0, even though the function that calls it is prepared to handle a nonzero return. □

For additional information on descending through a file system and using this technique in many standard UNIX System commands—find, ls, tar, and so on—refer to Fowler, Korn, and Vo [1989].

4.23 chdir, fchdir, and getcwd Functions

Every process has a current working directory. This directory is where the search for all relative pathnames starts (i.e., with all pathnames that do not begin with a slash). When a user logs in to a UNIX system, the current working directory normally starts at the directory specified by the sixth field in the /etc/passwd file—the user's home directory. The current working directory is an attribute of a process; the home directory is an attribute of a login name.

We can change the current working directory of the calling process by calling the chdir or fchdir function.

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int fd);

                                    Both return: 0 if OK, –1 on error
```

We can specify the new current working directory either as a *pathname* or through an open file descriptor.

Example

Because it is an attribute of a process, the current working directory cannot affect processes that invoke the process that executes the chdir. (We describe the relationship between processes in more detail in Chapter 8.) As a result, the program in Figure 4.23 doesn't do what we might expect.

```
#include "apue.h"

int
main(void)
{
    if (chdir("/tmp") < 0)
        err_sys("chdir failed");
    printf("chdir to /tmp succeeded\n");
    exit(0);
}
```

Figure 4.23 Example of chdir function

If we compile this program, call the executable mycd, and run it, we get the following:

```
$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib
```

The current working directory for the shell that executed the mycd program didn't change. This is a side effect of the way that the shell executes programs. Each program is run in a separate process, so the current working directory of the shell is unaffected by the call to chdir in the program. For this reason, the chdir function has to be called directly from the shell, so the cd command is built into the shells. □

Because the kernel must maintain knowledge of the current working directory, we should be able to fetch its current value. Unfortunately, the kernel doesn't maintain the full pathname of the directory. Instead, the kernel keeps information about the directory, such as a pointer to the directory's v-node.

The Linux kernel can determine the full pathname. Its components are distributed throughout the mount table and the dcache table, and are reassembled, for example, when you read the /proc/self/cwd symbolic link.

What we need is a function that starts at the current working directory (dot) and works its way up the directory hierarchy, using dot-dot to move up one level. At each level, the function reads the directory entries until it finds the name that corresponds to the i-node of the directory that it just came from. Repeating this procedure until the root is encountered yields the entire absolute pathname of the current working directory. Fortunately, a function already exists that does this work for us.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);

                                Returns: buf if OK, NULL on error
```

We must pass to this function the address of a buffer, *buf*, and its *size* (in bytes). The buffer must be large enough to accommodate the absolute pathname plus a terminating null byte, or else an error will be returned. (Recall the discussion of allocating space for a maximum-sized pathname in Section 2.5.5.)

Some older implementations of getcwd allow the first argument *buf* to be NULL. In this case, the function calls malloc to allocate *size* number of bytes dynamically. This is not part of POSIX.1 or the Single UNIX Specification and should be avoided.

Example

The program in Figure 4.24 changes to a specific directory and then calls getcwd to print the working directory. If we run the program, we get

```
$ ./a.out
cwd = /var/spool/uucppublic
$ ls -l /usr/spool
lrwxrwxrwx  1 root  12 Jan 31 07:57 /usr/spool -> ../var/spool
```

```
#include "apue.h"

int
main(void)
{
    char    *ptr;
    size_t      size;

    if (chdir("/usr/spool/uucppublic") < 0)
        err_sys("chdir failed");

    ptr = path_alloc(&size);    /* our own function */
    if (getcwd(ptr, size) == NULL)
        err_sys("getcwd failed");

    printf("cwd = %s\n", ptr);
    exit(0);
}
```

Figure 4.24 Example of getcwd function

Note that `chdir` follows the symbolic link—as we expect it to, from Figure 4.17—but when it goes up the directory tree, `getcwd` has no idea when it hits the `/var/spool` directory that it is pointed to by the symbolic link `/usr/spool`. This is a characteristic of symbolic links. □

The `getcwd` function is useful when we have an application that needs to return to the location in the file system where it started out. We can save the starting location by calling `getcwd` before we change our working directory. After we complete our processing, we can pass the pathname obtained from `getcwd` to `chdir` to return to our starting location in the file system.

The `fchdir` function provides us with an easy way to accomplish this task. Instead of calling `getcwd`, we can `open` the current directory and save the file descriptor before we change to a different location in the file system. When we want to return to where we started, we can simply pass the file descriptor to `fchdir`.

4.24 Device Special Files

The two fields `st_dev` and `st_rdev` are often confused. We'll need to use these fields in Section 18.9 when we write the `ttyname` function. The rules for their use are simple.

- Every file system is known by its major and minor device numbers, which are encoded in the primitive system data type `dev_t`. The major number identifies the device driver and sometimes encodes which peripheral board to communicate with; the minor number identifies the specific subdevice. Recall from Figure 4.13 that a disk drive often contains several file systems. Each file system on the same disk drive would usually have the same major number, but a different minor number.

- We can usually access the major and minor device numbers through two macros defined by most implementations: `major` and `minor`. Consequently, we don't care how the two numbers are stored in a `dev_t` object.

  Early systems stored the device number in a 16-bit integer, with 8 bits for the major number and 8 bits for the minor number. FreeBSD 8.0 and Mac OS X 10.6.8 use a 32-bit integer, with 8 bits for the major number and 24 bits for the minor number. On 32-bit systems, Solaris 10 uses a 32-bit integer for `dev_t`, with 14 bits designated as the major number and 18 bits designated as the minor number. On 64-bit systems, Solaris 10 represents `dev_t` as a 64-bit integer, with 32 bits for each number. On Linux 3.2.0, although `dev_t` is a 64-bit integer, only 12 bits are used for the major number and 20 bits are used for the minor number.

  POSIX.1 states that the `dev_t` type exists, but doesn't define what it contains or how to get at its contents. The macros `major` and `minor` are defined by most implementations. Which header they are defined in depends on the system. They can be found in `<sys/types.h>` on BSD-based systems. Solaris defines their function prototypes in `<sys/mkdev.h>`, because the macro definitions in `<sys/sysmacros.h>` are considered obsolete in Solaris. Linux defines these macros in `<sys/sysmacros.h>`, which is included by `<sys/types.h>`.

- The `st_dev` value for every filename on a system is the device number of the file system containing that filename and its corresponding i-node.
- Only character special files and block special files have an `st_rdev` value. This value contains the device number for the actual device.

Example

The program in Figure 4.25 prints the device number for each command-line argument. Additionally, if the argument refers to a character special file or a block special file, the `st_rdev` value for the special file is printed.

```
#include "apue.h"
#ifdef SOLARIS
#include <sys/mkdev.h>
#endif

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (stat(argv[i], &buf) < 0) {
            err_ret("stat error");
            continue;
        }

        printf("dev = %d/%d", major(buf.st_dev),  minor(buf.st_dev));
        if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                    (S_ISCHR(buf.st_mode)) ? "character" : "block",
                    major(buf.st_rdev), minor(buf.st_rdev));
        }
        printf("\n");
    }

    exit(0);
}
```

Figure 4.25 Print st_dev and st_rdev values

Running this program on Linux gives us the following output:

```
$ ./a.out / /home/sar /dev/tty[01]
/: dev = 8/3
/home/sar: dev = 8/4
/dev/tty0: dev = 0/5 (character) rdev = 4/0
/dev/tty1: dev = 0/5 (character) rdev = 4/1
$ mount                              which directories are mounted on which devices?
/dev/sda3 on / type ext3 (rw,errors=remount-ro,commit=0)
/dev/sda4 on /home type ext3 (rw,commit=0)
$ ls -l /dev/tty[01] /dev/sda[34]
brw-rw----  1 root       8, 3 2011-07-01 11:08 /dev/sda3
brw-rw----  1 root       8, 4 2011-07-01 11:08 /dev/sda4
crw--w----  1 root       4, 0 2011-07-01 11:08 /dev/tty0
crw-------  1 root       4, 1 2011-07-01 11:08 /dev/tty1
```

The first two arguments to the program are directories (/ and /home/sar), and the next two are the device names /dev/tty[01]. (We use the shell's regular expression language to shorten the amount of typing we need to do. The shell will expand the string /dev/tty[01] to /dev/tty0 /dev/tty1.)

We expect the devices to be character special files. The output from the program shows that the root directory has a different device number than does the /home/sar directory, which indicates that they are on different file systems. Running the mount(1) command verifies this.

We then use ls to look at the two disk devices reported by mount and the two terminal devices. The two disk devices are block special files, and the two terminal devices are character special files. (Normally, the only types of devices that are block special files are those that can contain random-access file systems—disk drives, floppy disk drives, and CD-ROMs, for example. Some older UNIX systems supported magnetic tapes for file systems, but this was never widely used.)

Note that the filenames and i-nodes for the two terminal devices (st_dev) are on device 0/5—the devtmpfs pseudo file system, which implements the /dev—but that their actual device numbers are 4/0 and 4/1. □

4.25 Summary of File Access Permission Bits

We’ve covered all the file access permission bits, some of which serve multiple purposes. Figure 4.26 summarizes these permission bits and their interpretation when applied to a regular file and a directory.

Constant	Description	Effect on regular file	Effect on directory
`S_ISUID`	set-user-ID	set effective user ID on execution	(not used)
`S_ISGID`	set-group-ID	if group-execute set, then set effective group ID on execution; otherwise, enable mandatory record locking (if supported)	set group ID of new files created in directory to group ID of directory
`S_ISVTX`	sticky bit	control caching of file contents (if supported)	restrict removal and renaming of files in directory
`S_IRUSR`	user-read	user permission to read file	user permission to read directory entries
`S_IWUSR`	user-write	user permission to write file	user permission to remove and create files in directory
`S_IXUSR`	user-execute	user permission to execute file	user permission to search for given pathname in directory
`S_IRGRP`	group-read	group permission to read file	group permission to read directory entries
`S_IWGRP`	group-write	group permission to write file	group permission to remove and create files in directory
`S_IXGRP`	group-execute	group permission to execute file	group permission to search for given pathname in directory
`S_IROTH`	other-read	other permission to read file	other permission to read directory entries
`S_IWOTH`	other-write	other permission to write file	other permission to remove and create files in directory
`S_IXOTH`	other-execute	other permission to execute file	other permission to search for given pathname in directory

Figure 4.26 Summary of file access permission bits

The final nine constants can also be grouped into threes, as follows:

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

Process Environment

7.1 Introduction

Before looking at the process control primitives in the next chapter, we need to examine the environment of a single process. In this chapter, we'll see how the `main` function is called when the program is executed, how command-line arguments are passed to the new program, what the typical memory layout looks like, how to allocate additional memory, how the process can use environment variables, and various ways for the process to terminate. Additionally, we'll look at the `longjmp` and `setjmp` functions and their interaction with the stack. We finish the chapter by examining the resource limits of a process.

7.2 `main` Function

A C program starts execution with a function called `main`. The prototype for the `main` function is

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

where *argc* is the number of command-line arguments, and *argv* is an array of pointers to the arguments. We describe these arguments in Section 7.4.

When a C program is executed by the kernel—by one of the `exec` functions, which we describe in Section 8.10—a special start-up routine is called before the `main` function is called. The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler. This start-up routine takes values from the kernel—the command-line arguments and the environment—and sets things up so that the `main` function is called as shown earlier.

7.3 Process Termination

There are eight ways for a process to terminate. Normal termination occurs in five ways:

1. Return from `main`
2. Calling `exit`
3. Calling `_exit` or `_Exit`
4. Return of the last thread from its start routine (Section 11.5)
5. Calling `pthread_exit` (Section 11.5) from the last thread

Abnormal termination occurs in three ways:

6. Calling `abort` (Section 10.17)
7. Receipt of a signal (Section 10.2)
8. Response of the last thread to a cancellation request (Sections 11.5 and 12.7)

For now, we'll ignore the three termination methods specific to threads until we discuss threads in Chapters 11 and 12.

The start-up routine that we mentioned in the previous section is also written so that if the `main` function returns, the `exit` function is called. If the start-up routine were coded in C (it is often coded in assembly language) the call to `main` could look like

```
exit(main(argc, argv));
```

Exit Functions

Three functions terminate a program normally: `_exit` and `_Exit`, which return to the kernel immediately, and `exit`, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

We'll discuss the effect of these three functions on other processes, such as the children and the parent of the terminating process, in Section 8.5.

The reason for the different headers is that `exit` and `_Exit` are specified by ISO C, whereas `_exit` is specified by POSIX.1.

Historically, the `exit` function has always performed a clean shutdown of the standard I/O library: the `fclose` function is called for all open streams. Recall from Section 5.5 that this causes all buffered output data to be flushed (written to the file).

All three exit functions expect a single integer argument, which we call the *exit status*. Most UNIX System shells provide a way to examine the exit status of a process. If (a) any of these functions is called without an exit status, (b) `main` does a `return` without a return value, or (c) the `main` function is not declared to return an integer, the exit status of the process is undefined. However, if the return type of `main` is an integer and `main` "falls off the end" (an implicit return), the exit status of the process is 0.

> This behavior is new with the 1999 version of the ISO C standard. Historically, the exit status was undefined if the end of the `main` function was reached without an explicit `return` statement or a call to the `exit` function.

Returning an integer value from the `main` function is equivalent to calling `exit` with the same value. Thus

```
exit(0);
```

is the same as

```
return(0);
```

from the `main` function.

Example

The program in Figure 7.1 is the classic "hello, world" example.

```
#include    <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Figure 7.1 Classic C program

When we compile and run the program in Figure 7.1, we see that the exit code is random. If we compile the same program on different systems, we are likely to get different exit codes, depending on the contents of the stack and register contents at the time that the `main` function returns:

```
$ gcc hello.c
$ ./a.out
hello, world
$ echo $?                    print the exit status
13
```

Now if we enable the 1999 ISO C compiler extensions, we see that the exit code changes:

```
$ gcc -std=c99 hello.c        enable gcc's 1999 ISO C extensions
hello.c:4: warning: return type defaults to 'int'
$ ./a.out
hello, world
$ echo $?                     print the exit status
0
```

Note the compiler warning when we enable the 1999 ISO C extensions. This warning is printed because the type of the `main` function is not explicitly declared to be an integer. If we were to add this declaration, the message would go away. However, if we were to enable all recommended warnings from the compiler (with the `-Wall` flag), then we would see a warning message something like "control reaches end of nonvoid function."

The declaration of `main` as returning an integer and the use of `exit` instead of `return` produces needless warnings from some compilers and the `lint`(1) program. The problem is that these compilers don't know that an `exit` from `main` is the same as a `return`. One way around these warnings, which become annoying after a while, is to use `return` instead of `exit` from `main`. But doing this prevents us from using the UNIX System's `grep` utility to locate all calls to `exit` from a program. Another solution is to declare `main` as returning `void`, instead of `int`, and continue calling `exit`. This gets rid of the compiler warning but doesn't look right (especially in a programming text), and can generate other compiler warnings, since the return type of `main` is supposed to be a signed integer. In this text, we show `main` as returning an integer, since that is the definition specified by both ISO C and POSIX.1.

Different compilers vary in the verbosity of their warnings. Note that the GNU C compiler usually doesn't emit these extraneous compiler warnings unless additional warning options are used.

□

In the next chapter, we'll see how any process can cause a program to be executed, wait for the process to complete, and then fetch its exit status.

`atexit` Function

With ISO C, a process can register at least 32 functions that are automatically called by `exit`. These are called *exit handlers* and are registered by calling the `atexit` function.

```
#include <stdlib.h>

int atexit(void (*func)(void));

                                   Returns: 0 if OK, nonzero on error
```

This declaration says that we pass the address of a function as the argument to `atexit`. When this function is called, it is not passed any arguments and is not expected to return a value. The `exit` function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

These exit handlers first appeared in the ANSI C Standard in 1989. Systems that predate ANSI C, such as SVR3 and 4.3BSD, did not provide these exit handlers.

ISO C requires that systems support at least 32 exit handlers, but implementations often support more (see Figure 2.15). The `sysconf` function can be used to determine the maximum number of exit handlers supported by a given platform, as illustrated in Figure 2.14.

With ISO C and POSIX.1, `exit` first calls the exit handlers and then closes (via `fclose`) all open streams. POSIX.1 extends the ISO C standard by specifying that any exit handlers installed will be cleared if the program calls any of the `exec` family of functions. Figure 7.2 summarizes how a C program is started and the various ways it can terminate.

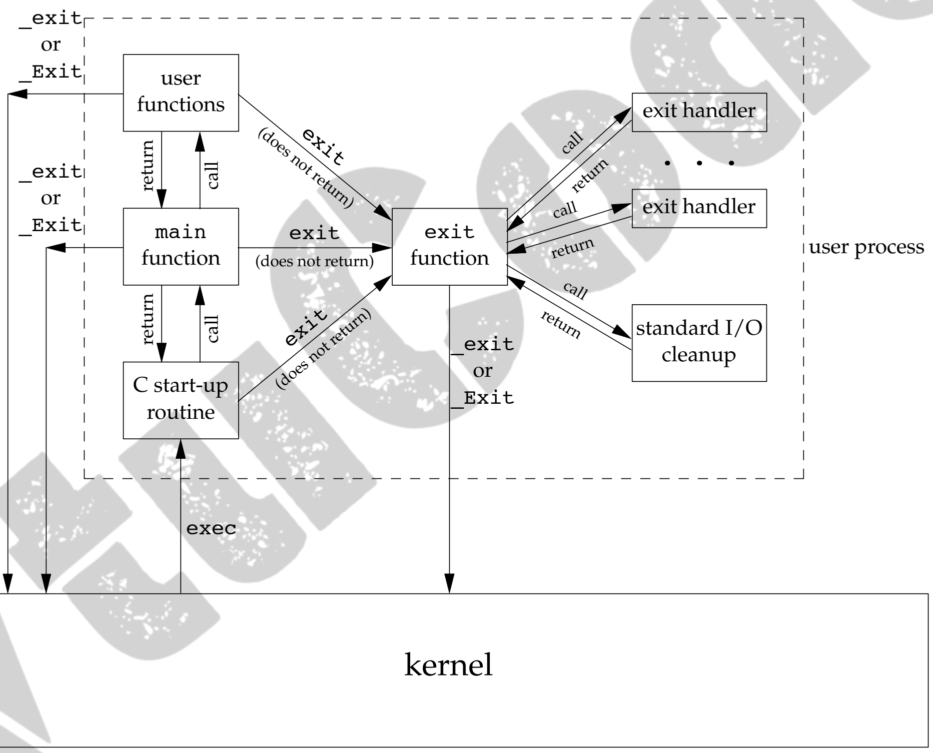

Figure 7.2 How a C program is started and how it terminates

The only way a program can be executed by the kernel is if one of the `exec` functions is called. The only way a process can voluntarily terminate is if `_exit` or `_Exit` is called, either explicitly or implicitly (by calling `exit`). A process can also be involuntarily terminated by a signal (not shown in Figure 7.2).

Example

The program in Figure 7.3 demonstrates the use of the atexit function.

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

Figure 7.3 Example of exit handlers

Executing the program in Figure 7.3 yields

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

An exit handler is called once for each time it is registered. In Figure 7.3, the first exit handler is registered twice, so it is called two times. Note that we don't call exit; instead, we return from main. □

7.4 Command-Line Arguments

When a program is executed, the process that does the exec can pass command-line arguments to the new program. This is part of the normal operation of the UNIX system shells. We have already seen this in many of the examples from earlier chapters.

Example

The program in Figure 7.4 echoes all its command-line arguments to standard output. Note that the normal echo(1) program doesn't echo the zeroth argument.

```
#include "apue.h"

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

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

Figure 7.4 Echo all command-line arguments to standard output

If we compile this program and name the executable echoarg, we have

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

We are guaranteed by both ISO C and POSIX.1 that argv[argc] is a null pointer. This lets us alternatively code the argument-processing loop as

```
for (i = 0; argv[i] != NULL; i++)
```

□

7.5 Environment List

Each program is also passed an *environment list*. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string. The address of the array of pointers is contained in the global variable environ:

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

For example, if the environment consisted of five strings, it could look like Figure 7.5. Here we explicitly show the null bytes at the end of each string. We'll call environ the

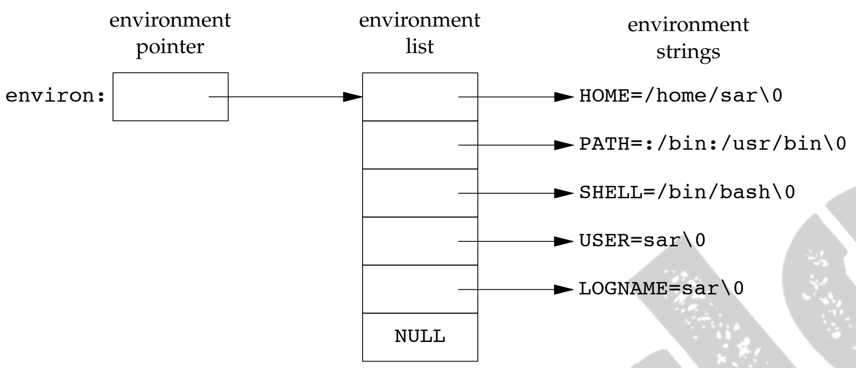

Figure 7.5 Environment consisting of five C character strings

environment pointer, the array of pointers the environment list, and the strings they point to the *environment strings*.

By convention, the environment consists of

name=*value*

strings, as shown in Figure 7.5. Most predefined names are entirely uppercase, but this is only a convention.

Historically, most UNIX systems have provided a third argument to the `main` function that is the address of the environment list:

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

Because ISO C specifies that the `main` function be written with two arguments, and because this third argument provides no benefit over the global variable `environ`, POSIX.1 specifies that `environ` should be used instead of the (possible) third argument. Access to specific environment variables is normally through the `getenv` and `putenv` functions, described in Section 7.9, instead of through the `environ` variable. But to go through the entire environment, the `environ` pointer must be used.

7.6 Memory Layout of a C Program

Historically, a C program has been composed of the following pieces:

- Text segment, consisting of the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

- Initialized data segment, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

  ```
  int   maxcount = 99;
  ```

  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

- Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

  ```
  long   sum[1000];
  ```

  appearing outside any function causes this variable to be stored in the uninitialized data segment.

- Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

- Heap, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.

Figure 7.6 shows the typical arrangement of these segments. This is a logical picture of how a program looks; there is no requirement that a given implementation arrange its memory in this fashion. Nevertheless, this gives us a typical arrangement to describe. With Linux on a 32-bit Intel x86 processor, the text segment starts at location `0x08048000`, and the bottom of the stack starts just below `0xC0000000`. (The stack grows from higher-numbered addresses to lower-numbered addresses on this particular architecture.) The unused virtual address space between the top of the heap and the top of the stack is large.

Several more segment types exist in an `a.out`, containing the symbol table, debugging information, linkage tables for dynamic shared libraries, and the like. These additional sections don't get loaded as part of the program's image executed by a process.

Note from Figure 7.6 that the contents of the uninitialized data segment are not stored in the program file on disk, because the kernel sets the contents to 0 before the program starts running. The only portions of the program that need to be saved in the program file are the text segment and the initialized data.

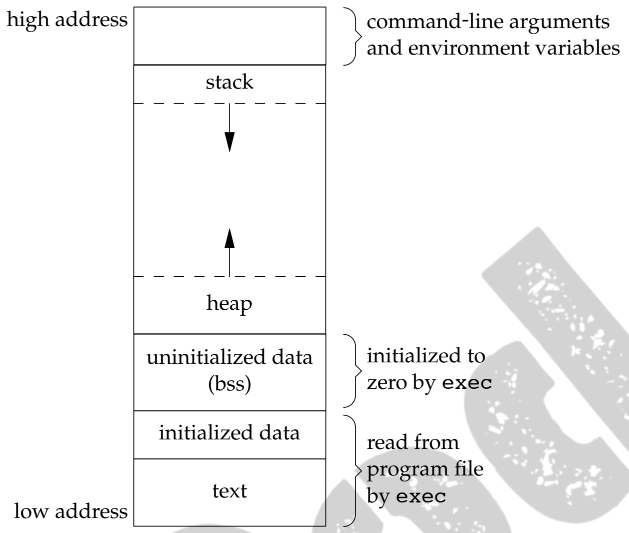
high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Figure 7.6 Typical memory arrangement

The `size`(1) command reports the sizes (in bytes) of the text, data, and bss segments. For example:

```
$ size /usr/bin/cc /bin/sh
   text    data     bss     dec     hex   filename
 346919    3576    6680  357175   57337   /usr/bin/cc
 102134    1776   11272  115182   1c1ee   /bin/sh
```

The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively.

7.7 Shared Libraries

Most UNIX systems today support shared libraries. Arnold [1986] describes an early implementation under System V, and Gingell et al. [1987] describe a different implementation under SunOS. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called. Another advantage of shared libraries is that library functions can be replaced with new versions without having to relink edit every program that uses the library (assuming that the number and type of arguments haven't changed).

Different systems provide different ways for a program to say that it wants to use or not use the shared libraries. Options for the `cc`(1) and `ld`(1) commands are typical. As an example of the size differences, the following executable file—the classic `hello.c` program—was first created without shared libraries:

```
$ gcc -static hello1.c         prevent gcc from using shared libraries
$ ls -l a.out
-rwxr-xr-x  1 sar     879443 Sep 2 10:39 a.out
$ size a.out
   text    data    bss     dec     hex  filename
 787775    6128  11272  805175   c4937  a.out
```

If we compile this program to use shared libraries, the text and data sizes of the executable file are greatly decreased:

```
$ gcc hello1.c                 gcc defaults to use shared libraries
$ ls -l a.out
-rwxr-xr-x  1 sar       8378 Sep 2 10:39 a.out
$ size a.out
   text    data    bss     dec     hex  filename
   1176     504     16    1696     6a0  a.out
```

7.8 Memory Allocation

ISO C specifies three functions for memory allocation:

1. `malloc`, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
2. `calloc`, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
3. `realloc`, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);

                     All three return: non-null pointer if OK, NULL on error

void free(void *ptr);
```

The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. For example, if the most restrictive alignment requirement on a particular system requires that `doubles` must start at memory locations that are multiples of 8, then all pointers returned by these three functions would be so aligned.

Because the three `alloc` functions return a generic `void *` pointer, if we `#include <stdlib.h>` (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type. The default return value for undeclared functions is `int`, so using a cast without the proper function declaration could hide an error on systems where the size of type `int` differs from the size of a function's return value (a pointer in this case).

The function `free` causes the space pointed to by *ptr* to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three `alloc` functions.

The `realloc` function lets us change the size of a previously allocated area. (The most common usage is to increase an area's size.) For example, if we allocate room for 512 elements in an array that we fill in at runtime but later find that we need more room, we can call `realloc`. If there is room beyond the end of the existing region for the requested space, then `realloc` simply allocates this additional area at the end and returns the same pointer that we passed it. But if there isn't room, `realloc` allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area. Because the area may move, we shouldn't have any pointers into this area. Exercise 4.16 and Figure C.3 show the use of `realloc` with `getcwd` to handle a pathname of any length. Figure 17.27 shows an example that uses `realloc` to avoid arrays with fixed, compile-time sizes.

Note that the final argument to `realloc` is the new size of the region, not the difference between the old and new sizes. As a special case, if *ptr* is a null pointer, `realloc` behaves like `malloc` and allocates a region of the specified *newsize*.

Older versions of these routines allowed us to `realloc` a block that we had `free`d since the last call to `malloc`, `realloc`, or `calloc`. This trick dates back to Version 7 and exploited the search strategy of `malloc` to perform storage compaction. Solaris still supports this feature, but many other platforms do not. This feature is deprecated and should not be used.

The allocation routines are usually implemented with the `sbrk`(2) system call. This system call expands (or contracts) the heap of the process. (Refer to Figure 7.6.) A sample implementation of `malloc` and `free` is given in Section 8.7 of Kernighan and Ritchie [1988].

Although `sbrk` can expand or contract the memory of a process, most versions of `malloc` and `free` never decrease their memory size. The space that we free is available for a later allocation, but the freed space is not usually returned to the kernel; instead, that space is kept in the `malloc` pool.

Most implementations allocate more space than requested and use the additional space for record keeping—the size of the block, a pointer to the next allocated block, and the like. As a consequence, writing past the end or before the start of an allocated area could overwrite this record-keeping information in another block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later.

Writing past the end or before the beginning of a dynamically allocated buffer can corrupt more than internal record-keeping information. The memory before and after a dynamically allocated buffer can potentially be used for other dynamically allocated objects. These objects can be unrelated to the code corrupting them, making it even more difficult to find the source of the corruption.

Other possible errors that can be fatal are freeing a block that was already freed and calling `free` with a pointer that was not obtained from one of the three `alloc` functions. If a process calls `malloc` but forgets to call `free`, its memory usage will continually increase; this is called leakage. If we do not call `free` to return unused space, the size of a process's address space will slowly increase until no free space is left. During this time, performance can degrade from excess paging overhead.

Because memory allocation errors are difficult to track down, some systems provide versions of these functions that do additional error checking every time one of the three `alloc` functions or `free` is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources that you can compile with special flags to enable additional runtime checking.

> FreeBSD, Mac OS X, and Linux support additional debugging through the setting of environment variables. In addition, options can be passed to the FreeBSD library through the symbolic link `/etc/malloc.conf`.

Alternate Memory Allocators

Many replacements for `malloc` and `free` are available. Some systems already include libraries providing alternative memory allocator implementations. Other systems provide only the standard allocator, leaving it up to software developers to download alternatives, if desired. We discuss some of the alternatives here.

`libmalloc`

SVR4-based systems, such as Solaris, include the `libmalloc` library, which provides a set of interfaces matching the ISO C memory allocation functions. The `libmalloc` library includes `mallopt`, a function that allows a process to set certain variables that control the operation of the storage allocator. A function called `mallinfo` is also available to provide statistics on the memory allocator.

`vmalloc`

Vo [1996] describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory. In addition to the functions specific to `vmalloc`, the library provides emulations of the ISO C memory allocation functions.

`quick-fit`

Historically, the standard `malloc` algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Weinstock and Wulf [1988] describe the algorithm, which is based on splitting up memory into buffers of various sizes and maintaining unused buffers on different free lists, depending on the buffer sizes. Most modern allocators are based on quick-fit.

jemalloc

The jemalloc implementation of the malloc family of library functions is the default memory allocator in FreeBSD 8.0. It was designed to scale well when used with multithreaded applications running on multiprocessor systems. Evans [2006] describes the implementation and evaluates its performance.

TCMalloc

TCMalloc was designed as a replacement for the malloc family of functions to provide high performance, scalability, and memory efficiency. It uses thread-local caches to avoid locking overhead when allocating buffers from and releasing buffers to the cache. It also has a heap checker and a heap profiler built in to aid in debugging and analyzing dynamic memory usage. The TCMalloc library is available as open source from Google. It is briefly described by Ghemawat and Menage [2005].

alloca Function

One additional function is also worth mentioning. The function alloca has the same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The alloca function increases the size of the stack frame. The disadvantage is that some systems can't support alloca, if it's impossible to increase the size of the stack frame after the function has been called. Nevertheless, many software packages use it, and implementations exist for a wide variety of systems.

All four platforms discussed in this text provide the alloca function.

7.9 Environment Variables

As we mentioned earlier, the environment strings are usually of the form

name=value

The UNIX kernel never looks at these strings; their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some, such as HOME and USER, are set automatically at login; others are left for us to set. We normally set environment variables in a shell start-up file to control the shell's actions. If we set the environment variable MAILPATH, for example, it tells the Bourne shell, GNU Bourne-again shell, and Korn shell where to look for mail.

ISO C defines a function that we can use to fetch values from the environment, but this standard says that the contents of the environment are implementation defined.

```
#include <stdlib.h>

char *getenv(const char *name);
```

Returns: pointer to *value* associated with *name*, NULL if not found

Note that this function returns a pointer to the *value* of a *name=value* string. We should always use `getenv` to fetch a specific value from the environment, instead of accessing `environ` directly.

Some environment variables are defined by POSIX.1 in the Single UNIX Specification, whereas others are defined only if the XSI option is supported. Figure 7.7 lists the environment variables defined by the Single UNIX Specification and notes which implementations support the variables. Any environment variable defined by POSIX.1 is marked with •; otherwise, it is part of the XSI option. Many additional implementation-dependent environment variables are used in the four implementations described in this book. Note that ISO C doesn't define any environment variables.

Variable	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
`COLUMNS`	•	•	•	•	•	terminal width
`DATEMSK`	XSI		•	•	•	`getdate`(3) template file pathname
`HOME`	•	•	•	•	•	home directory
`LANG`	•	•	•	•	•	name of locale
`LC_ALL`	•	•	•	•	•	name of locale
`LC_COLLATE`	•	•	•	•	•	name of locale for collation
`LC_CTYPE`	•	•	•	•	•	name of locale for character classification
`LC_MESSAGES`	•	•	•	•	•	name of locale for messages
`LC_MONETARY`	•	•	•	•	•	name of locale for monetary editing
`LC_NUMERIC`	•	•	•	•	•	name of locale for numeric editing
`LC_TIME`	•	•	•	•	•	name of locale for date/time formatting
`LINES`	•	•	•	•	•	terminal height
`LOGNAME`	•	•	•	•	•	login name
`MSGVERB`	XSI	•	•	•	•	`fmtmsg`(3) message components to process
`NLSPATH`	•	•	•	•	•	sequence of templates for message catalogs
`PATH`	•	•	•	•	•	list of path prefixes to search for executable file
`PWD`	•	•	•	•	•	absolute pathname of current working directory
`SHELL`	•	•	•	•	•	name of user's preferred shell
`TERM`	•	•	•	•	•	terminal type
`TMPDIR`	•	•	•	•	•	pathname of directory for creating temporary files
`TZ`	•	•	•	•	•	time zone information

Figure 7.7 Environment variables defined in the Single UNIX Specification

In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to change the value of an existing variable or add a new variable to the environment. (In the next chapter, we'll see that we can affect the environment of only the current process and any child processes that we invoke. We cannot affect the environment of the parent process, which is often a shell. Nevertheless, it is still useful to be able to modify the environment list.) Unfortunately, not all systems support this capability. Figure 7.8 shows the functions that are supported by the various standards and implementations.

Function	ISO C	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
getenv	•	•	•	•	•	•
putenv		XSI	•	•	•	•
setenv		•	•	•	•	
unsetenv		•	•	•	•	
clearenv				•		

Figure 7.8 Support for various environment list functions

The `clearenv` function is not part of the Single UNIX Specification. It is used to remove all entries from the environment list.

The prototypes for the middle three functions listed in Figure 7.8 are

```
#include <stdlib.h>

int putenv(char *str);

                                        Returns: 0 if OK, nonzero on error

int setenv(const char *name, const char *value, int rewrite);

int unsetenv(const char *name);

                                        Both return: 0 if OK, –1 on error
```

The operation of these three functions is as follows:

- The `putenv` function takes a string of the form *name=value* and places it in the environment list. If *name* already exists, its old definition is first removed.
- The `setenv` function sets *name* to *value*. If *name* already exists in the environment, then (a) if *rewrite* is nonzero, the existing definition for *name* is first removed; or (b) if *rewrite* is 0, an existing definition for *name* is not removed, *name* is not set to the new *value*, and no error occurs.
- The `unsetenv` function removes any definition of *name*. It is not an error if such a definition does not exist.

Note the difference between `putenv` and `setenv`. Whereas `setenv` must allocate memory to create the *name=value* string from its arguments, `putenv` is free to place the string passed to it directly into the environment. Indeed, many implementations do exactly this, so it would be an error to pass `putenv` a string allocated on the stack, since the memory would be reused after we return from the current function.

It is interesting to examine how these functions must operate when modifying the environment list. Recall Figure 7.6: the environment list—the array of pointers to the actual *name=value* strings—and the environment strings are typically stored at the top of a process's memory space, above the stack. Deleting a string is simple; we just find the pointer in the environment list and move all subsequent pointers down one. But adding a string or modifying an existing string is more difficult. The space at the top of the stack cannot be expanded, because it is often at the top of the address space of the process and so can't expand upward; it can't be expanded downward, because all the stack frames below it can't be moved.

1. If we're modifying an existing *name:*
   a. If the size of the new *value* is less than or equal to the size of the existing *value*, we can just copy the new string over the old string.
   b. If the size of the new *value* is larger than the old one, however, we must `malloc` to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for *name* with the pointer to this allocated area.
2. If we're adding a new *name*, it's more complicated. First, we have to call `malloc` to allocate room for the *name=value* string and copy the string to this area.
   a. Then, if it's the first time we've added a new *name*, we have to call `malloc` to obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the *name=value* string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set `environ` to point to this new list of pointers. Note from Figure 7.6 that if the original environment list was contained above the top of the stack, as is common, then we have moved this list of pointers to the heap. But most of the pointers in this list still point to *name=value* strings above the top of the stack.
   b. If this isn't the first time we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call `realloc` to allocate room for one more pointer. The pointer to the new *name=value* string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

7.10 `setjmp` and `longjmp` Functions

In C, we can't `goto` a label that's in another function. Instead, we must use the `setjmp` and `longjmp` functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

Consider the skeleton in Figure 7.9. It consists of a main loop that reads lines from standard input and calls the function `do_line` to process each line. This function then calls `get_token` to fetch the next token from the input line. The first token of a line is assumed to be a command of some form, and a `switch` statement selects each command. For the single command shown, the function `cmd_add` is called.

The skeleton in Figure 7.9 is typical for programs that read commands, determine the command type, and then call functions to process each command. Figure 7.10 shows what the stack could look like after `cmd_add` has been called.

```
#include "apue.h"

#define TOK_ADD    5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

int
main(void)
{
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char    *tok_ptr;       /* global pointer for get_token() */

void
do_line(char *ptr)      /* process one line of input */
{
    int     cmd;

    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) {  /* one case for each command */
        case TOK_ADD:
                cmd_add();
                break;
        }
    }
}

void
cmd_add(void)
{
    int     token;

    token = get_token();
    /* rest of processing for this command */
}

int
get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}
```

Figure 7.9 Typical program skeleton for command processing

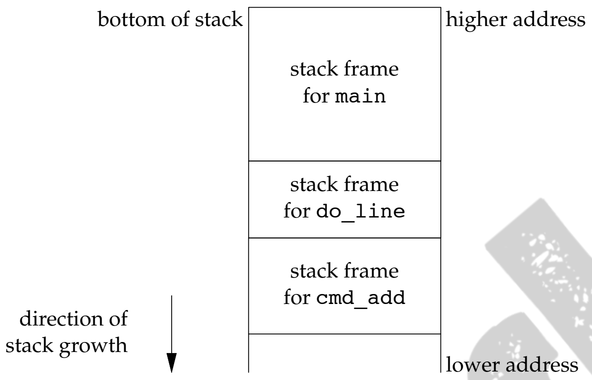

Figure 7.10 Stack frames after cmd_add has been called

Storage for the automatic variables is within the stack frame for each function. The array `line` is in the stack frame for `main`, the integer `cmd` is in the stack frame for `do_line`, and the integer `token` is in the stack frame for `cmd_add`.

As we've said, this type of arrangement of the stack is typical, but not required. Stacks do not have to grow toward lower memory addresses. On systems that don't have built-in hardware support for stacks, a C implementation might use a linked list for its stack frames.

The coding problem that's often encountered with programs like the one shown in Figure 7.9 is how to handle nonfatal errors. For example, if the `cmd_add` function encounters an error—say, an invalid number—it might want to print an error message, ignore the rest of the input line, and return to the `main` function to read the next input line. But when we're deeply nested numerous levels down from the `main` function, this is difficult to do in C. (In this example, the `cmd_add` function is only two levels down from `main`, but it's not uncommon to be five or more levels down from the point to which we want to return.) It becomes messy if we have to code each function with a special return value that tells it to return one level.

The solution to this problem is to use a nonlocal `goto`: the `setjmp` and `longjmp` functions. The adjective "nonlocal" indicates that we're not doing a normal C `goto` statement within a function; instead, we're branching back through the call frames to a function that is in the call path of the current function.

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

Returns: 0 if called directly, nonzero if returning from a call to `longjmp`

```
void longjmp(jmp_buf env, int val);
```

We call setjmp from the location that we want to return to, which in this example is in the main function. In this case, setjmp returns 0 because we called it directly. In the call to setjmp, the argument *env* is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp. Normally, the *env* variable is a global variable, since we'll need to reference it from another function.

When we encounter an error—say, in the cmd_add function—we call longjmp with two arguments. The first is the same *env* that we used in a call to setjmp, and the second, *val*, is a nonzero value that becomes the return value from setjmp. The second argument allows us to use more than one longjmp for each setjmp. For example, we could longjmp from cmd_add with a *val* of 1 and also call longjmp from get_token with a *val* of 2. In the main function, the return value from setjmp is either 1 or 2, and we can test this value, if we want, and determine whether the longjmp was from cmd_add or get_token.

Let's return to the example. Figure 7.11 shows both the main and cmd_add functions. (The other two functions, do_line and get_token, haven't changed.)

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

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

 . . .

void
cmd_add(void)
{
    int     token;

    token = get_token();
    if (token < 0)      /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

Figure 7.11 Example of setjmp and longjmp

When main is executed, we call setjmp, which records whatever information it needs to in the variable jmpbuffer and returns 0. We then call do_line, which calls cmd_add, and assume that an error of some form is detected. Before the call to longjmp in cmd_add, the stack looks like that in Figure 7.10. But longjmp causes the stack to be "unwound" back to the main function, throwing away the stack frames for cmd_add and do_line (Figure 7.12). Calling longjmp causes the setjmp in main to return, but this time it returns with a value of 1 (the second argument for longjmp).

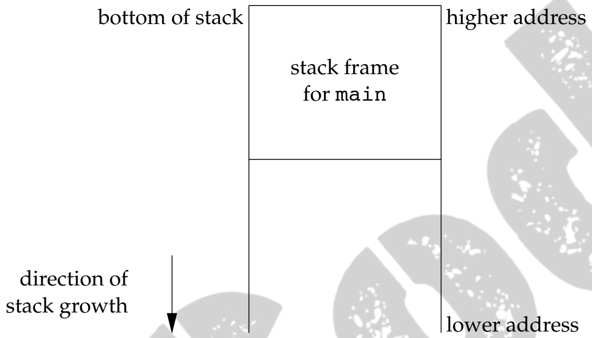

Figure 7.12 Stack frame after longjmp has been called

Automatic, Register, and Volatile Variables

We've seen what the stack looks like after calling longjmp. The next question is, "What are the states of the automatic variables and register variables in the main function?" When we return to main as a result of the longjmp, do these variables have values corresponding to those when the setjmp was previously called (i.e., are their values rolled back), or are their values left alone so that their values are whatever they were when do_line was called (which caused cmd_add to be called, which caused longjmp to be called)? Unfortunately, the answer is "It depends." Most implementations do not try to roll back these automatic variables and register variables, but the standards say only that their values are indeterminate. If you have an automatic variable that you don't want rolled back, define it with the volatile attribute. Variables that are declared as global or static are left alone when longjmp is executed.

Example

The program in Figure 7.13 demonstrates the different behavior that can be seen with automatic, global, register, static, and volatile variables after calling longjmp.

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

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf  jmpbuffer;
static int      globval;

int
main(void)
{
    int             autoval;
    register int    regival;
    volatile int    volaval;
    static int      statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }

    /*
     * Change variables after setjmp, but before longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns */
    exit(0);
}

static void
f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
        " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}

static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

Figure 7.13 Effect of `longjmp` on various types of variables

If we compile and test the program in Figure 7.13, with and without compiler optimizations, the results are different:

```
$ gcc testjmp.c                 compile without any optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ gcc -O testjmp.c              compile with full optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

Note that the optimizations don't affect the global, static, and volatile variables; their values after the `longjmp` are the last values that they assumed. The `setjmp`(3) manual page on one system states that variables stored in memory will have values as of the time of the `longjmp`, whereas variables in the CPU and floating-point registers are restored to their values when `setjmp` was called. This is indeed what we see when we run the program in Figure 7.13. Without optimization, all five variables are stored in memory (the `register` hint is ignored for `regival`). When we enable optimization, both `autoval` and `regival` go into registers, even though the former wasn't declared `register`, and the `volatile` variable stays in memory. The important thing to realize with this example is that you must use the `volatile` attribute if you're writing portable code that uses nonlocal jumps. Anything else can change from one system to the next.

Some `printf` format strings in Figure 7.13 are longer than will fit comfortably for display in a programming text. Instead of making multiple calls to `printf`, we rely on ISO C's string concatenation feature, where the sequence

```
"string1" "string2"
```

is equivalent to

```
"string1string2"
```

□

We'll return to these two functions, `setjmp` and `longjmp`, in Chapter 10 when we discuss signal handlers and their signal versions: `sigsetjmp` and `siglongjmp`.

Potential Problem with Automatic Variables

Having looked at the way stack frames are usually handled, it is worth looking at a potential error in dealing with automatic variables. The basic rule is that an automatic variable can never be referenced after the function that declared it returns. Numerous warnings about this can be found throughout the UNIX System manuals.

Figure 7.14 shows a function called `open_data` that opens a standard I/O stream and sets the buffering for the stream.

```
#include    <stdio.h>

FILE *
open_data(void)
{
    FILE    *fp;
    char    databuf[BUFSIZ];  /* setvbuf makes this the stdio buffer */

    if ((fp = fopen("datafile", "r")) == NULL)
        return(NULL);
    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);
    return(fp);     /* error */
}
```

Figure 7.14 Incorrect usage of an automatic variable

The problem is that when `open_data` returns, the space it used on the stack will be used by the stack frame for the next function that is called. But the standard I/O library will still be using that portion of memory for its stream buffer. Chaos is sure to result. To correct this problem, the array `databuf` needs to be allocated from global memory, either statically (`static` or `extern`) or dynamically (one of the `alloc` functions).

7.11 getrlimit and setrlimit Functions

Every process has a set of resource limits, some of which can be queried and changed by the `getrlimit` and `setrlimit` functions.

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

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);

                                    Both return: 0 if OK, –1 on error
```

These two functions are defined in the XSI option in the Single UNIX Specification. The resource limits for a process are normally established by process 0 when the system is initialized and then inherited by each successive process. Each implementation has its own way of tuning the various limits.

Each call to these two functions specifies a single *resource* and a pointer to the following structure:

```
struct rlimit {
  rlim_t  rlim_cur;  /* soft limit: current limit */
  rlim_t  rlim_max;  /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.
2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
3. Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant `RLIM_INFINITY`.

The *resource* argument takes on one of the following values. Figure 7.15 shows which limits are defined by the Single UNIX Specification and supported by each implementation.

`RLIMIT_AS`	The maximum size in bytes of a process's total available memory. This affects the `sbrk` function (Section 1.11) and the `mmap` function (Section 14.8).
`RLIMIT_CORE`	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
`RLIMIT_CPU`	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the `SIGXCPU` signal is sent to the process.
`RLIMIT_DATA`	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap from Figure 7.6.
`RLIMIT_FSIZE`	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the `SIGXFSZ` signal.
`RLIMIT_MEMLOCK`	The maximum amount of memory in bytes that a process can lock into memory using `mlock`(2).
`RLIMIT_MSGQUEUE`	The maximum amount of memory in bytes that a process can allocate for POSIX message queues.
`RLIMIT_NICE`	The limit to which a process's nice value (Section 8.16) can be raised to affect its scheduling priority.
`RLIMIT_NOFILE`	The maximum number of open files per process. Changing this limit affects the value returned by the `sysconf` function for its `_SC_OPEN_MAX` argument (Section 2.5.4). See Figure 2.17 also.
`RLIMIT_NPROC`	The maximum number of child processes per real user ID. Changing this limit affects the value returned for `_SC_CHILD_MAX` by the `sysconf` function (Section 2.5.4).
`RLIMIT_NPTS`	The maximum number of pseudo terminals (Chapter 19) that a user can have open at one time.

`RLIMIT_RSS`	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
`RLIMIT_SBSIZE`	The maximum size in bytes of socket buffers that a user can consume at any given time.
`RLIMIT_SIGPENDING`	The maximum number of signals that can be queued for a process. This limit is enforced by the `sigqueue` function (Section 10.20).
`RLIMIT_STACK`	The maximum size in bytes of the stack. See Figure 7.6.
`RLIMIT_SWAP`	The maximum amount of swap space in bytes that a user can consume.
`RLIMIT_VMEM`	This is a synonym for `RLIMIT_AS`.

Limit	XSI	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
RLIMIT_AS	•	•	•		•
RLIMIT_CORE	•	•	•	•	•
RLIMIT_CPU	•	•	•	•	•
RLIMIT_DATA	•	•	•	•	•
RLIMIT_FSIZE	•	•	•	•	•
RLIMIT_MEMLOCK		•	•	•	
RLIMIT_MSGQUEUE			•		
RLIMIT_NICE			•		
RLIMIT_NOFILE	•	•	•	•	•
RLIMIT_NPROC		•	•	•	
RLIMIT_NPTS		•			
RLIMIT_RSS		•	•	•	
RLIMIT_SBSIZE		•			
RLIMIT_SIGPENDING			•		
RLIMIT_STACK	•	•	•	•	•
RLIMIT_SWAP		•			
RLIMIT_VMEM					•

Figure 7.15 Support for resource limits

The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes. Indeed, the Bourne shell, the GNU Bourne-again shell, and the Korn shell have the built-in `ulimit` command, and the C shell has the built-in `limit` command. (The `umask` and `chdir` functions also have to be handled as shell built-ins.)

Example

The program in Figure 7.16 prints out the current soft limit and hard limit for all the resource limits supported on the system. To compile this program on all the various implementations, we have conditionally included the resource names that differ. Note that some systems define `rlim_t` to be an `unsigned long long` instead of an `unsigned long`. This definition can even change on the same system, depending on whether we compile the program to support 64-bit files. Some limits apply to file size, so the `rlim_t` type has to be large enough to represent a file size limit. To avoid compiler warnings that use the wrong format specification, we first copy the limit into a 64-bit integer so that we have to deal with only one format.

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

#define doit(name)  pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
#ifdef  RLIMIT_AS
    doit(RLIMIT_AS);
#endif

    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);

#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif

#ifdef RLIMIT_MSGQUEUE
    doit(RLIMIT_MSGQUEUE);
#endif

#ifdef RLIMIT_NICE
    doit(RLIMIT_NICE);
#endif

    doit(RLIMIT_NOFILE);

#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif

#ifdef RLIMIT_NPTS
    doit(RLIMIT_NPTS);
#endif

#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif

#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#endif

#ifdef RLIMIT_SIGPENDING
    doit(RLIMIT_SIGPENDING);
#endif

    doit(RLIMIT_STACK);

#ifdef RLIMIT_SWAP
    doit(RLIMIT_SWAP);
#endif

#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif

    exit(0);
}

static void
pr_limits(char *name, int resource)
{
    struct rlimit       limit;
    unsigned long long  lim;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY) {
        printf("(infinite)  ");
    } else {
        lim = limit.rlim_cur;
        printf("%10lld  ", lim);
    }
    if (limit.rlim_max == RLIM_INFINITY) {
        printf("(infinite)");
    } else {
        lim = limit.rlim_max;
        printf("%10lld", lim);
    }
    putchar((int)'\n');
}
```

Figure 7.16 Print the current resource limits

Note that we've used the ISO C string-creation operator (#) in the `doit` macro, to generate the string value for each resource name. When we say

```
doit(RLIMIT_CORE);
```

the C preprocessor expands this into

```
pr_limits("RLIMIT_CORE", RLIMIT_CORE);
```

Running this program under FreeBSD gives us the following output:

```
$ ./a.out
RLIMIT_AS       (infinite)  (infinite)
RLIMIT_CORE     (infinite)  (infinite)
RLIMIT_CPU      (infinite)  (infinite)
RLIMIT_DATA      536870912   536870912
RLIMIT_FSIZE    (infinite)  (infinite)
RLIMIT_MEMLOCK  (infinite)  (infinite)
RLIMIT_NOFILE         3520        3520
RLIMIT_NPROC          1760        1760
RLIMIT_NPTS     (infinite)  (infinite)
RLIMIT_RSS      (infinite)  (infinite)
RLIMIT_SBSIZE   (infinite)  (infinite)
RLIMIT_STACK      67108864    67108864
RLIMIT_SWAP     (infinite)  (infinite)
RLIMIT_VMEM     (infinite)  (infinite)
```

Solaris gives us the following results:

```
$ ./a.out
RLIMIT_AS       (infinite)  (infinite)
RLIMIT_CORE     (infinite)  (infinite)
RLIMIT_CPU      (infinite)  (infinite)
RLIMIT_DATA     (infinite)  (infinite)
RLIMIT_FSIZE    (infinite)  (infinite)
RLIMIT_NOFILE          256       65536
RLIMIT_STACK       8388608  (infinite)
RLIMIT_VMEM     (infinite)  (infinite)
```