2

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.

2.10 Summary

Much has happened with the standardization of the UNIX programming environment over the past two and a half decades. We've described the dominant standards—ISO C, POSIX, and the Single UNIX Specification—and their effect on the four platforms that we'll examine in this text—FreeBSD, Linux, Mac OS X, and Solaris. These standards try to define certain parameters that can change with each implementation, but we've seen that these limits are imperfect. We'll encounter many of these limits and magic constants as we proceed through the text.

The bibliography specifies how to obtain copies of the standards discussed in this chapter.

Exercises

2.1 We mentioned in Section 2.8 that some of the primitive system data types are defined in more than one header. For example, in FreeBSD 8.0, `size_t` is defined in 29 different headers. Because all 29 headers could be included in a program and because ISO C does not allow multiple `typedef`s for the same name, how must the headers be written?

2.2 Examine your system's headers and list the actual data types used to implement the primitive system data types.

2.3 Update the program in Figure 2.17 to avoid the needless processing that occurs when `sysconf` returns `LONG_MAX` as the limit for `OPEN_MAX`.

3

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).

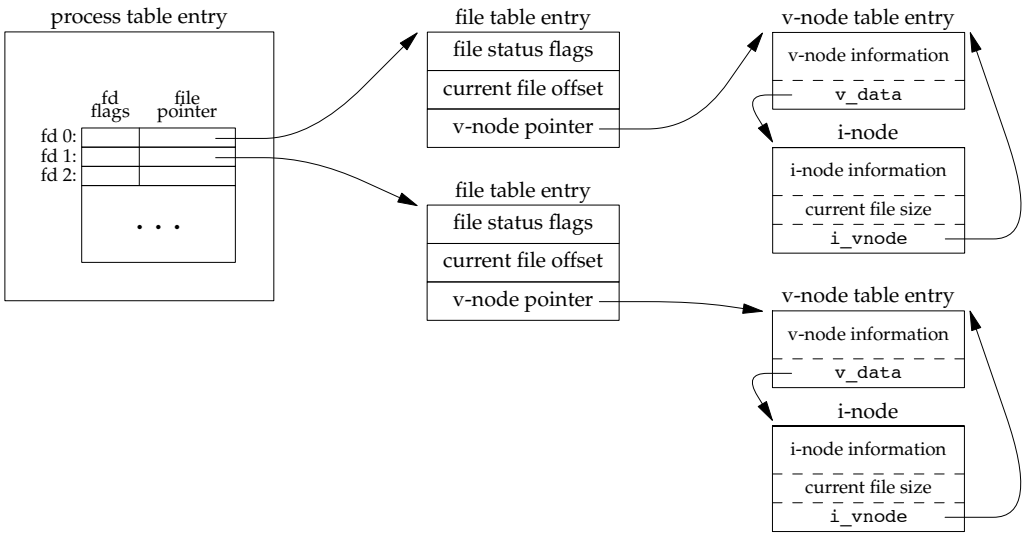

Figure 3.7 Kernel data structures for open files

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.

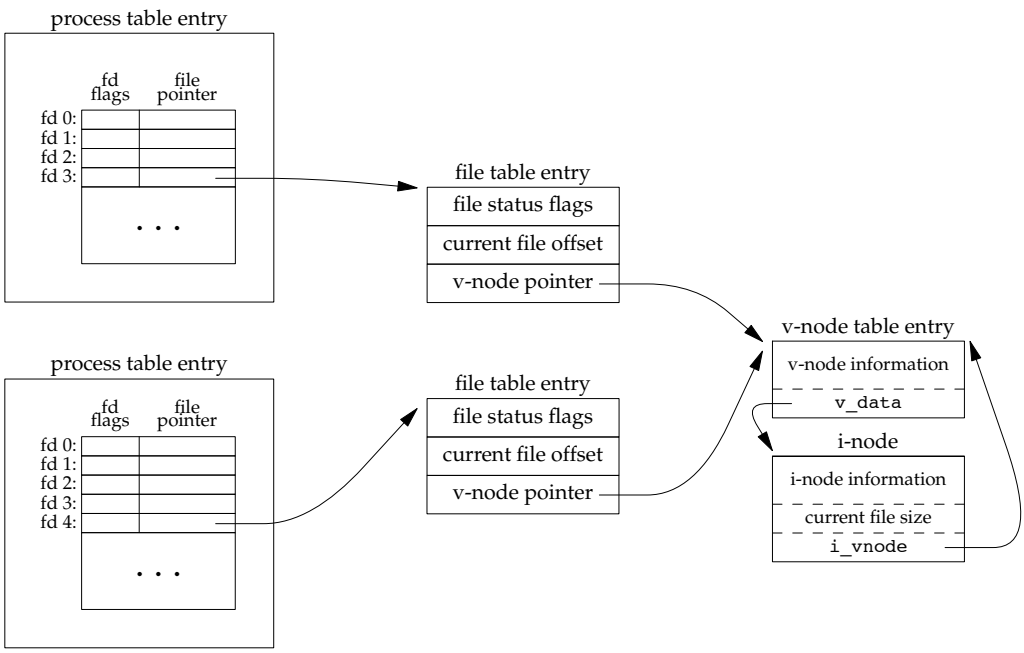

Figure 3.8 Two independent processes with the same file open

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.

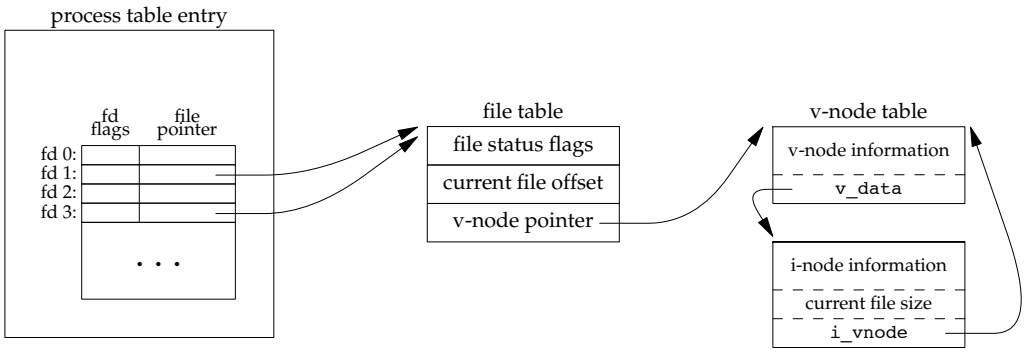

Figure 3.9 Kernel data structures after `dup(1)`

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 `#defined` 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.

3.17 Summary

This chapter has described the basic I/O functions provided by the UNIX System. These are often called the unbuffered I/O functions because each `read` or `write` invokes a system call into the kernel. Using only `read` and `write`, we looked at the effect of various I/O sizes on the amount of time required to read a file. We also looked at several ways to flush written data to disk and their effect on application performance.

Atomic operations were introduced when multiple processes append to the same file and when multiple processes create the same file. We also looked at the data structures used by the kernel to share information about open files. We'll return to these data structures later in the text.

We also described the `ioctl` and `fcntl` functions. We return to both of these functions later in the book. In Chapter 14, we'll use `fcntl` for record locking. In Chapter 18 and Chapter 19, we'll use `ioctl` when we deal with terminal devices.

Exercises

3.1 When reading or writing a disk file, are the functions described in this chapter really unbuffered? Explain.

3.2 Write your own `dup2` function that behaves the same way as the `dup2` function described in Section 3.12, without calling the `fcntl` function. Be sure to handle errors correctly.

3.3 Assume that a process executes the following three function calls:

```
fd1 = open(path, oflags);
fd2 = dup(fd1);
fd3 = open(path, oflags);
```

Draw the resulting picture, similar to Figure 3.9. Which descriptors are affected by an `fcntl` on `fd1` with a command of `F_SETFD`? Which descriptors are affected by an `fcntl` on `fd1` with a command of `F_SETFL`?

3.4 The following sequence of code has been observed in various programs:

```
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
if (fd > 2)
    close(fd);
```

To see why the `if` test is needed, assume that `fd` is 1 and draw a picture of what happens to the three descriptor entries and the corresponding file table entry with each call to `dup2`. Then assume that `fd` is 3 and draw the same picture.

3.5 The Bourne shell, Bourne-again shell, and Korn shell notation

digit1`>&`*digit2*

says to redirect descriptor *digit1* to the same file as descriptor *digit2*. What is the difference between the two commands shown below? (Hint: The shells process their command lines from left to right.)

```
./a.out > outfile 2>&1
./a.out 2>&1 > outfile
```

3.6 If you `open` a file for read–write with the append flag, can you still `read` from anywhere in the file using `lseek`? Can you use `lseek` to replace existing data in the file? Write a program to verify this.

This page intentionally left blank

4

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.

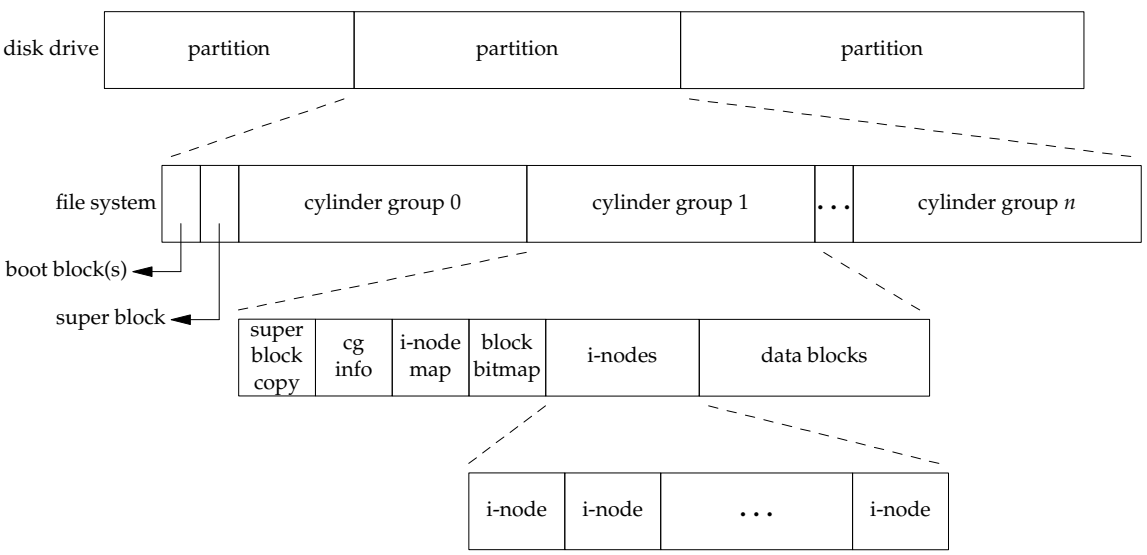

Figure 4.13 Disk drive, partitions, and a file system

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.

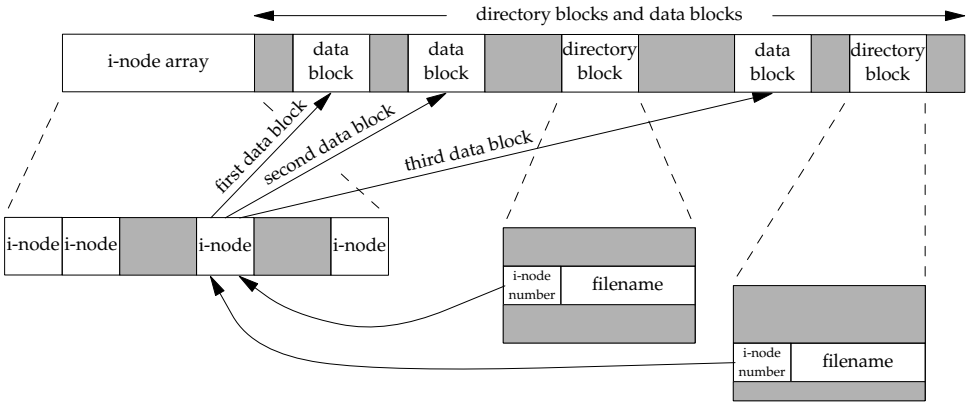

Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

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.

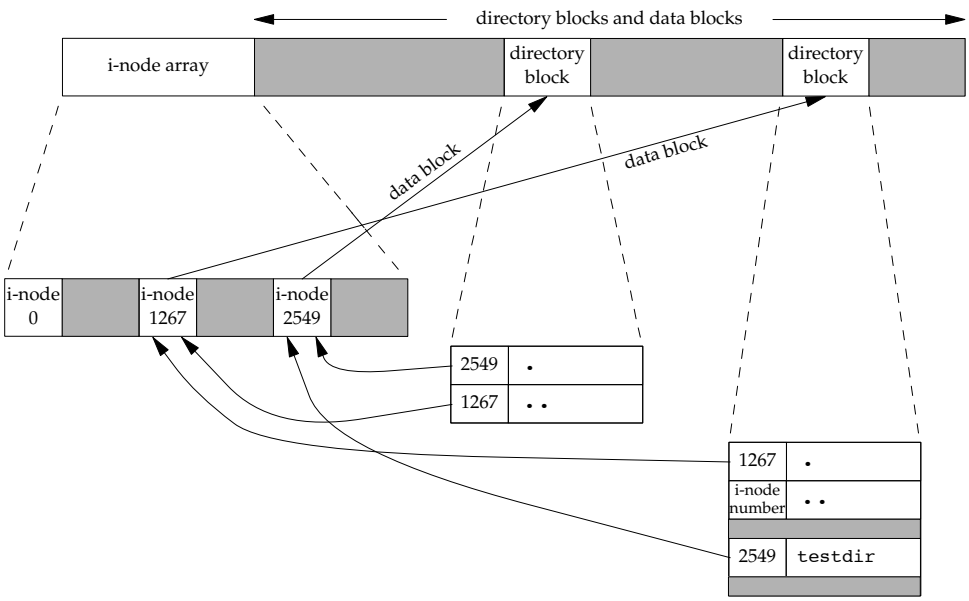

Figure 4.15 Sample cylinder group after creating the directory `testdir`

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.

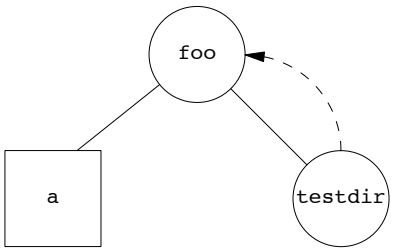

Figure 4.18 Symbolic link `testdir` that creates a loop

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
```

4.26 Summary

This chapter has centered on the `stat` function. We've gone through each member in the `stat` structure in detail. This, in turn, led us to examine all the attributes of UNIX files and directories. We've looked at how files and directories might be laid out in a file

7

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.

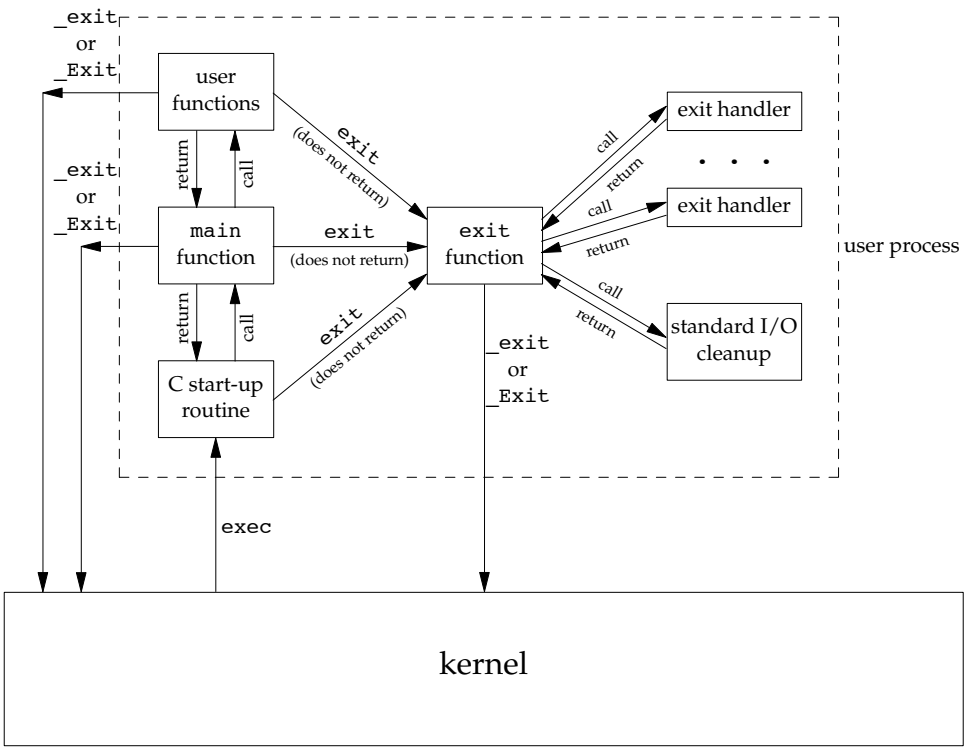

Figure 7.2 How a C program is started and how it terminates

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

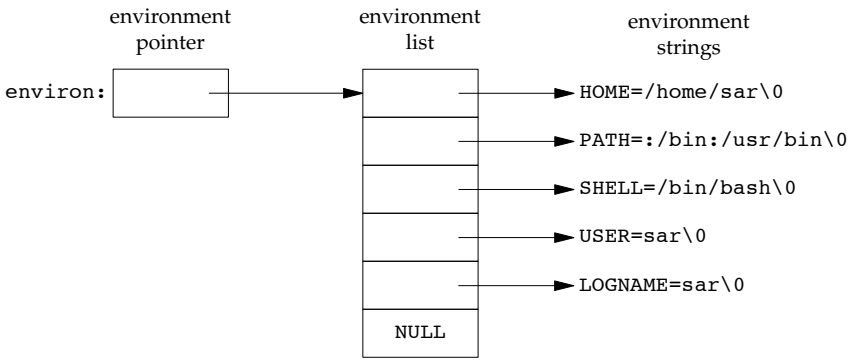

Figure 7.5 Environment consisting of five C character strings

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.

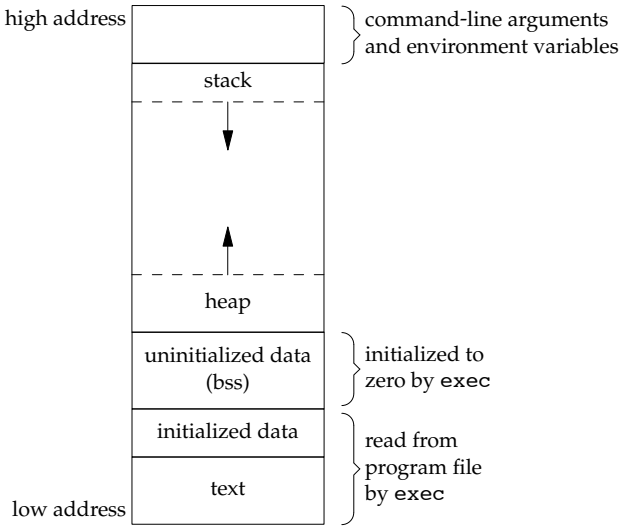

Figure 7.6 Typical memory arrangement

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

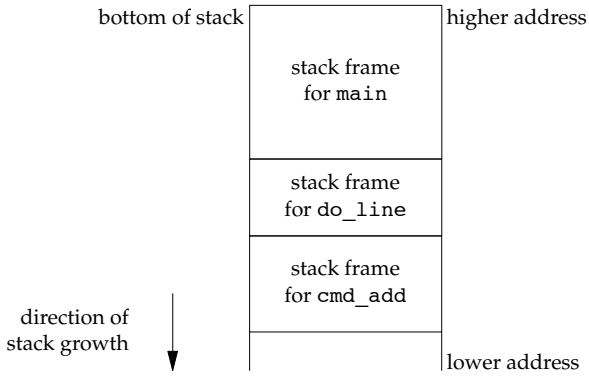

Figure 7.10 Stack frames after `cmd_add` has been called

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).

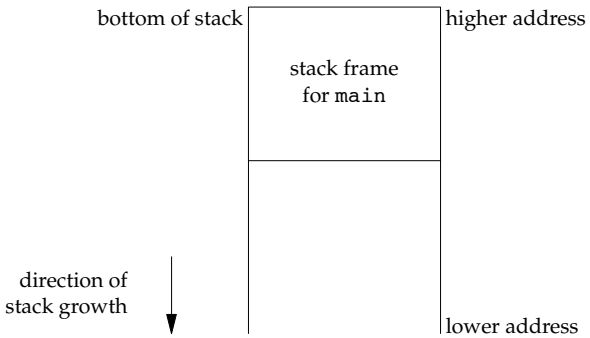

Figure 7.12 Stack frame after longjmp has been called

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)
```

□

Exercise 10.11 continues the discussion of resource limits, after we've covered signals.

7.12 Summary

Understanding the environment of a C program within a UNIX system's environment is a prerequisite to understanding the process control features of the UNIX System. In this chapter, we've looked at how a process is started, how it can terminate, and how it's passed an argument list and an environment. Although both the argument list and the environment are uninterpreted by the kernel, it is the kernel that passes both from the caller of `exec` to the new process.

We've also examined the typical memory layout of a C program and seen how a process can dynamically allocate and free memory. It is worthwhile to look in detail at the functions available for manipulating the environment, since they involve memory allocation. The functions `setjmp` and `longjmp` were presented, providing a way to perform nonlocal branching within a process. We finished the chapter by describing the resource limits that various implementations provide.

Exercises

7.1 On an Intel x86 system under Linux, if we execute the program that prints "hello, world" and do not call `exit` or `return`, the termination status of the program—which we can examine with the shell—is 13. Why?

7.2 When is the output from the `printfs` in Figure 7.3 actually output?

7.3 Is there any way for a function that is called by `main` to examine the command-line arguments without (a) passing `argc` and `argv` as arguments from `main` to the function or (b) having `main` copy `argc` and `argv` into global variables?

7.4 Some UNIX system implementations purposely arrange that, when a program is executed, location 0 in the data segment is not accessible. Why?

7.5 Use the `typedef` facility of C to define a new data type `Exitfunc` for an exit handler. Redo the prototype for `atexit` using this data type.

7.6 If we allocate an array of `long`s using `calloc`, is the array initialized to 0? If we allocate an array of pointers using `calloc`, is the array initialized to null pointers?

7.7 In the output from the `size` command at the end of Section 7.6, why aren't any sizes given for the heap and the stack?

7.8 In Section 7.7, the two file sizes (879443 and 8378) don't equal the sums of their respective text and data sizes. Why?

7.9 In Section 7.7, why does the size of the executable file differ so dramatically when we use shared libraries for such a trivial program?

7.10 At the end of Section 7.10, we showed how a function can't return a pointer to an automatic variable. Is the following code correct?

```
int
f1(int val)
{
    int     num = 0;
    int     *ptr = &num;

    if (val == 0) {
        int     val;

        val = 5;
        ptr = &val;
    }
    return(*ptr + 1);
}
```

8

Process Control

8.1 Introduction

We now turn to the process control provided by the UNIX System. This includes the creation of new processes, program execution, and process termination. We also look at the various IDs that are the property of the process—real, effective, and saved; user and group IDs—and how they're affected by the process control primitives. Interpreter files and the `system` function are also covered. We conclude the chapter by looking at the process accounting provided by most UNIX systems. This lets us look at the process control functions from a different perspective.

8.2 Process Identifiers

Every process has a unique process ID, a non-negative integer. Because the process ID is the only well-known identifier of a process that is always unique, it is often used as a piece of other identifiers, to guarantee uniqueness. For example, applications sometimes include the process ID as part of a filename in an attempt to generate unique filenames.

Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse. Most UNIX systems implement algorithms to delay reuse, however, so that newly created processes are assigned IDs different from those used by processes that terminated recently. This prevents a new process from being mistaken for the previous process to have used the same ID.

There are some special processes, but the details differ from implementation to implementation. Process ID 0 is usually the scheduler process and is often known as the *swapper*. No program on disk corresponds to this process, which is part of the

kernel and is known as a system process. Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process was `/etc/init` in older versions of the UNIX System and is `/sbin/init` in newer versions. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped. `init` usually reads the system-dependent initialization files—the `/etc/rc*` files or `/etc/inittab` and the files in `/etc/init.d`—and brings the system to a certain state, such as multiuser. The `init` process never dies. It is a normal user process, not a system process within the kernel, like the swapper, although it does run with superuser privileges. Later in this chapter, we'll see how `init` becomes the parent process of any orphaned child process.

> In Mac OS X 10.4, the `init` process was replaced with the `launchd` process, which performs the same set of tasks as `init`, but has expanded functionality. See Section 5.10 in Singh [2006] for a discussion of how `launchd` operates.

Each UNIX System implementation has its own set of kernel processes that provide operating system services. For example, on some virtual memory implementations of the UNIX System, process ID 2 is the *pagedaemon*. This process is responsible for supporting the paging of the virtual memory system.

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.

```
#include <unistd.h>

pid_t getpid(void);
                                        Returns: process ID of calling process

pid_t getppid(void);
                                    Returns: parent process ID of calling process

uid_t getuid(void);
                                        Returns: real user ID of calling process

uid_t geteuid(void);
                                    Returns: effective user ID of calling process

gid_t getgid(void);
                                        Returns: real group ID of calling process

gid_t getegid(void);
                                   Returns: effective group ID of calling process
```

Note that none of these functions has an error return. We'll return to the parent process ID in the next section when we discuss the `fork` function. The real and effective user and group IDs were discussed in Section 4.4.

8.3 fork Function

An existing process can create a new one by calling the fork function.

```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, –1 on error

The new process created by fork is called the *child process*. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

Both the child and the parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory. The parent and the child do share the text segment, however (Section 7.6).

Modern implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an exec. Instead, a technique called *copy-on-write* (COW) is used. These regions are shared by the parent and the child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system. Section 9.2 of Bach [1986] and Sections 5.6 and 5.7 of McKusick et al. [1996] provide more detail on this feature.

> Variations of the fork function are provided by some platforms. All four platforms discussed in this book support the vfork(2) variant discussed in the next section.
>
> Linux 3.2.0 also provides new process creation through the clone(2) system call. This is a generalized form of fork that allows the caller to control what is shared between parent and child.
>
> FreeBSD 8.0 provides the rfork(2) system call, which is similar to the Linux clone system call. The rfork call is derived from the Plan 9 operating system (Pike et al. [1995]).
>
> Solaris 10 provides two threads libraries: one for POSIX threads (pthreads) and one for Solaris threads. In previous releases, the behavior of fork differed between the two thread libraries. For POSIX threads, fork created a process containing only the calling thread, but for Solaris threads, fork created a process containing copies of all threads from the process of the calling thread. In Solaris 10, this behavior has changed; fork creates a child containing a copy of the calling thread only, regardless of which thread library is used. Solaris also provides the fork1 function, which can be used to create a process that duplicates only the calling thread, and the forkall function, which can be used to create a process that duplicates all the threads in the process. Threads are discussed in detail in Chapters 11 and 12.

Example

The program in Figure 8.1 demonstrates the `fork` function, showing how changes to variables in a child process do not affect the value of the variables in the parent process.

```
#include "apue.h"

int     globvar = 6;        /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        globvar++;              /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }

    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar,
      var);
    exit(0);
}
```

Figure 8.1 Example of `fork` function

If we execute this program, we get

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89      child's variables were changed
pid = 429, glob = 6, var = 88      parent's copy was not changed
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

In general, we never know whether the child starts executing before the parent, or vice versa. The order depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize their actions, some form of interprocess

communication is required. In the program shown in Figure 8.1, we simply have the parent put itself to sleep for 2 seconds, to let the child execute. There is no guarantee that the length of this delay is adequate, and we talk about this and other types of synchronization in Section 8.9 when we discuss race conditions. In Section 10.16, we show how to use signals to synchronize a parent and a child after a `fork`.

When we write to standard output, we subtract 1 from the size of `buf` to avoid writing the terminating null byte. Although `strlen` will calculate the length of a string not including the terminating null byte, `sizeof` calculates the size of the buffer, which does include the terminating null byte. Another difference is that using `strlen` requires a function call, whereas `sizeof` calculates the buffer length at compile time, as the buffer is initialized with a known string and its size is fixed.

Note the interaction of `fork` with the I/O functions in the program in Figure 8.1. Recall from Chapter 3 that the `write` function is not buffered. Because `write` is called before the `fork`, its data is written once to standard output. The standard I/O library, however, is buffered. Recall from Section 5.12 that standard output is line buffered if it's connected to a terminal device; otherwise, it's fully buffered. When we run the program interactively, we get only a single copy of the first `printf` line, because the standard output buffer is flushed by the newline. When we redirect standard output to a file, however, we get two copies of the `printf` line. In this second case, the `printf` before the `fork` is called once, but the line remains in the buffer when `fork` is called. This buffer is then copied into the child when the parent's data space is copied to the child. Both the parent and the child now have a standard I/O buffer with this line in it. The second `printf`, right before the `exit`, just appends its data to the existing buffer. When each process terminates, its copy of the buffer is finally flushed. □

File Sharing

When we redirect the standard output of the parent from the program in Figure 8.1, the child's standard output is also redirected. Indeed, one characteristic of `fork` is that all file descriptors that are open in the parent are duplicated in the child. We say "duplicated" because it's as if the `dup` function had been called for each descriptor. The parent and the child share a file table entry for every open descriptor (recall Figure 3.9).

Consider a process that has three different files opened for standard input, standard output, and standard error. On return from `fork`, we have the arrangement shown in Figure 8.2.

It is important that the parent and the child share the same file offset. Consider a process that `fork`s a child, then `wait`s for the child to complete. Assume that both processes write to standard output as part of their normal processing. If the parent has its standard output redirected (by a shell, perhaps), it is essential that the parent's file offset be updated by the child when the child writes to standard output. In this case, the child can write to standard output while the parent is `wait`ing for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote. If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

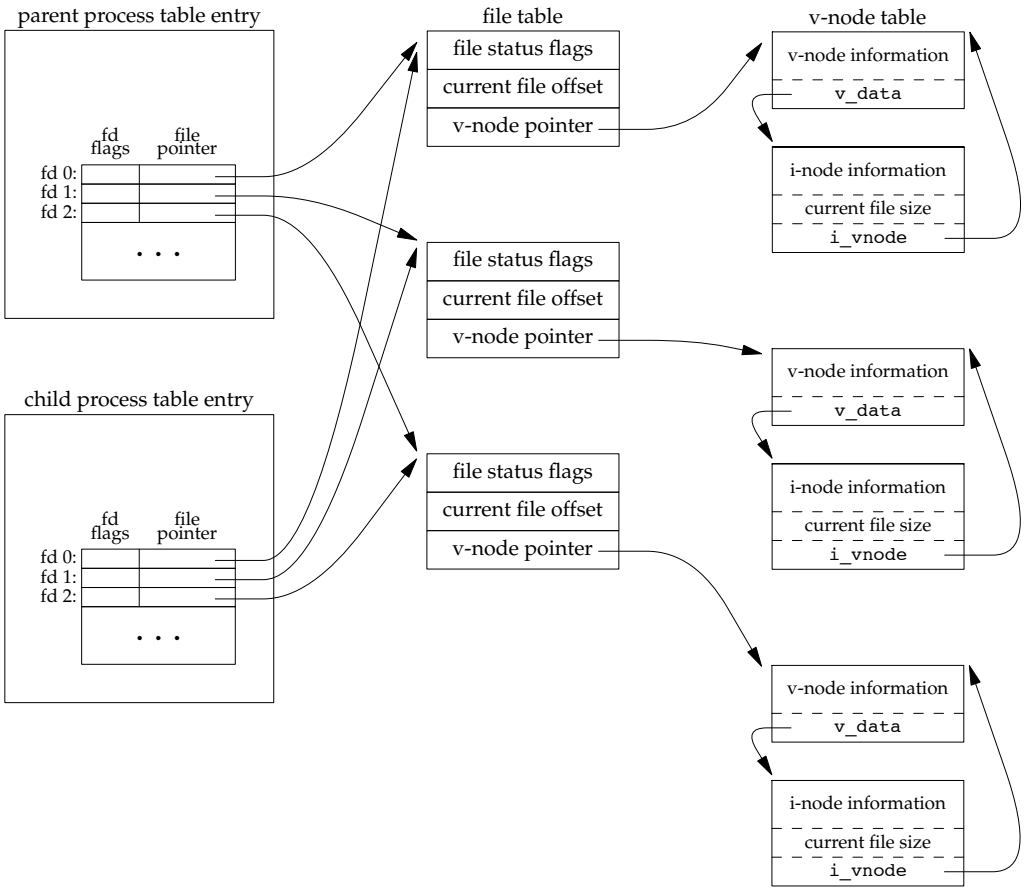

Figure 8.2 Sharing of open files between parent and child after `fork`

If both parent and child write to the same descriptor, without any form of synchronization, such as having the parent `wait` for the child, their output will be intermixed (assuming it's a descriptor that was open before the `fork`). Although this is possible—we saw it in Figure 8.2—it's not the normal mode of operation.

There are two normal cases for handling the descriptors after a `fork`.

1. The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
2. Both the parent and the child go their own ways. Here, after the `fork`, the parent closes the descriptors that it doesn't need, and the child does the same thing. This way, neither interferes with the other's open descriptors. This scenario is often found with network servers.

Besides the open files, numerous other properties of the parent are inherited by the child:

- Real user ID, real group ID, effective user ID, and effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

The differences between the parent and child are

- The return values from `fork` are different.
- The process IDs are different.
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change.
- The child's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` values are set to 0 (these times are discussed in Section 8.17).
- File locks set by the parent are not inherited by the child.
- Pending alarms are cleared for the child.
- The set of pending signals for the child is set to the empty set.

Many of these features haven't been discussed yet—we'll cover them in later chapters.

The two main reasons for `fork` to fail are (a) if too many processes are already in the system, which usually means that something else is wrong, or (b) if the total number of processes for this real user ID exceeds the system's limit. Recall from Figure 2.11 that `CHILD_MAX` specifies the maximum number of simultaneous processes per real user ID.

There are two uses for `fork`:

1. When a process wants to duplicate itself so that the parent and the child can each execute different sections of code at the same time. This is common for network servers—the parent waits for a service request from a client. When the request arrives, the parent calls `fork` and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
2. When a process wants to execute a different program. This is common for shells. In this case, the child does an `exec` (which we describe in Section 8.10) right after it returns from the `fork`.

Some operating systems combine the operations from step 2—a `fork` followed by an `exec`—into a single operation called a *spawn*. The UNIX System separates the two, as there are numerous cases where it is useful to `fork` without doing an `exec`. Also, separating the two operations allows the child to change the per-process attributes between the `fork` and the `exec`, such as I/O redirection, user ID, signal disposition, and so on. We'll see numerous examples of this in Chapter 15.

> The Single UNIX Specification does include `spawn` interfaces in the advanced real-time option group. These interfaces are not intended to be replacements for `fork` and `exec`, however. They are intended to support systems that have difficulty implementing `fork` efficiently, especially systems without hardware support for memory management.

8.4 `vfork` Function

The function `vfork` has the same calling sequence and same return values as `fork`, but the semantics of the two functions differ.

> The `vfork` function originated with 2.9BSD. Some consider the function a blemish, but all the platforms covered in this book support it. In fact, the BSD developers removed it from the 4.4BSD release, but all the open source BSD distributions that derive from 4.4BSD added support for it back into their own releases. The `vfork` function was marked as an obsolescent interface in Version 3 of the Single UNIX Specification and was removed entirely in Version 4. We include it here for historical reasons only. Portable applications should not use it.

The `vfork` function was intended to create a new process for the purpose of executing a new program (step 2 at the end of the previous section), similar to the method used by the bare-bones shell from Figure 1.7. The `vfork` function creates the new process, just like `fork`, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls `exec` (or `exit`) right after the `vfork`. Instead, the child runs in the address space of the parent until it calls either `exec` or `exit`. This optimization is more efficient on some implementations of the UNIX System, but leads to undefined results if the child modifies any data (except the variable used to hold the return value from `vfork`), makes function calls, or returns without calling `exec` or `exit`. (As we mentioned in the previous section, implementations use copy-on-write to improve the efficiency of a `fork` followed by an `exec`, but no copying is still faster than some copying.)

Another difference between the two functions is that `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`. When the child calls either of these functions, the parent resumes. (This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

Example

The program in Figure 8.3 is a modified version of the program from Figure 8.1. We've replaced the call to `fork` with `vfork` and removed the `write` to standard output. Also, we don't need to have the parent call `sleep`, as we're guaranteed that it is put to sleep by the kernel until the child calls either `exec` or `exit`.

```
#include "apue.h"

int     globvar = 6;        /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        globvar++;              /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }

    /* parent continues here */
    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar,
      var);
    exit(0);
}
```

Figure 8.3 Example of `vfork` function

Running this program gives us

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

Here, the incrementing of the variables done by the child changes the values in the parent. Because the child runs in the address space of the parent, this doesn't surprise us. This behavior, however, differs from the behavior of `fork`.

Note in Figure 8.3 that we call `_exit` instead of `exit`. As we described in Section 7.3, `_exit` does not perform any flushing of standard I/O buffers. If we call `exit` instead, the results are indeterminate. Depending on the implementation of the standard I/O library, we might see no difference in the output, or we might find that the output from the first `printf` in the parent has disappeared.

If the child calls `exit`, the implementation flushes the standard I/O streams. If this is the only action taken by the library, then we will see no difference from the output generated if the child called `_exit`. If the implementation also closes the standard I/O streams, however, the memory representing the `FILE` object for the standard output will be cleared out. Because the child is borrowing the parent's address space, when the parent resumes and calls `printf`, no output will appear and `printf` will return –1. Note that the parent's `STDOUT_FILENO` is still valid, as the child gets a copy of the parent's file descriptor array (refer back to Figure 8.2).

Most modern implementations of `exit` do not bother to close the streams. Because the process is about to exit, the kernel will close all the file descriptors open in the process. Closing them in the library simply adds overhead without any benefit.

□

Section 5.6 of McKusick et al. [1996] contains additional information on the implementation issues of `fork` and `vfork`. Exercises 8.1 and 8.2 continue the discussion of `vfork`.

8.5 `exit` Functions

As we described in Section 7.3, a process can terminate normally in five ways:

1. Executing a `return` from the `main` function. As we saw in Section 7.3, this is equivalent to calling `exit`.
2. Calling the `exit` function. This function is defined by ISO C and includes the calling of all exit handlers that have been registered by calling `atexit` and closing all standard I/O streams. Because ISO C does not deal with file descriptors, multiple processes (parents and children), and job control, the definition of this function is incomplete for a UNIX system.
3. Calling the `_exit` or `_Exit` function. ISO C defines `_Exit` to provide a way for a process to terminate without running exit handlers or signal handlers. Whether standard I/O streams are flushed depends on the implementation. On UNIX systems, `_Exit` and `_exit` are synonymous and do not flush standard I/O streams. The `_exit` function is called by `exit` and handles the UNIX system-specific details; `_exit` is specified by POSIX.1.

   In most UNIX system implementations, `exit`(3) is a function in the standard C library, whereas `_exit`(2) is a system call.

4. Executing a `return` from the start routine of the last thread in the process. The return value of the thread is not used as the return value of the process, however. When the last thread returns from its start routine, the process exits with a termination status of 0.
5. Calling the `pthread_exit` function from the last thread in the process. As with the previous case, the exit status of the process in this situation is always 0, regardless of the argument passed to `pthread_exit`. We'll say more about `pthread_exit` in Section 11.5.

The three forms of abnormal termination are as follows:

1. Calling `abort`. This is a special case of the next item, as it generates the `SIGABRT` signal.
2. When the process receives certain signals. (We describe signals in more detail in Chapter 10.) The signal can be generated by the process itself (e.g., by calling the `abort` function), by some other process, or by the kernel. Examples of

signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.

3. The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later the target thread terminates. We discuss cancellation requests in detail in Sections 11.5 and 12.7.

Regardless of how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and so on.

For any of the preceding cases, we want the terminating process to be able to notify its parent how it terminated. For the three exit functions (`exit`, `_exit`, and `_Exit`), this is done by passing an exit status as the argument to the function. In the case of an abnormal termination, however, the kernel—not the process—generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the `wait` or the `waitpid` function (described in the next section).

Note that we differentiate between the exit status, which is the argument to one of the three exit functions or the return value from `main`, and the termination status. The exit status is converted into a termination status by the kernel when `_exit` is finally called (recall Figure 7.2). Figure 8.4 describes the various ways the parent can examine the termination status of a child. If the child terminated normally, the parent can obtain the exit status of the child.

When we described the `fork` function, it was obvious that the child has a parent process after the call to `fork`. Now we're talking about returning a termination status to the parent. But what happens if the parent terminates before the child? The answer is that the `init` process becomes the parent process of any process whose parent terminates. In such a case, we say that the process has been inherited by `init`. What normally happens is that whenever a process terminates, the kernel goes through all active processes to see whether the terminating process is the parent of any process that still exists. If so, the parent process ID of the surviving process is changed to be 1 (the process ID of `init`). This way, we're guaranteed that every process has a parent.

Another condition we have to worry about is when a child terminates before its parent. If the child completely disappeared, the parent wouldn't be able to fetch its termination status when and if the parent was finally ready to check if the child had terminated. The kernel keeps a small amount of information for every terminating process, so that the information is available when the parent of the terminating process calls `wait` or `waitpid`. Minimally, this information consists of the process ID, the termination status of the process, and the amount of CPU time taken by the process. The kernel can discard all the memory used by the process and close its open files. In UNIX System terminology, a process that has terminated, but whose parent has not yet waited for it, is called a *zombie*. The `ps`(1) command prints the state of a zombie process as *Z*. If we write a long-running program that `fork`s many child processes, they become zombies unless we wait for them and fetch their termination status.

Some systems provide ways to prevent the creation of zombies, as we describe in Section 10.7.

The final condition to consider is this: What happens when a process that has been inherited by `init` terminates? Does it become a zombie? The answer is "no," because `init` is written so that whenever one of its children terminates, `init` calls one of the `wait` functions to fetch the termination status. By doing this, `init` prevents the system from being clogged by zombies. When we say "one of `init`'s children," we mean either a process that `init` generates directly (such as `getty`, which we describe in Section 9.2) or a process whose parent has terminated and has been subsequently inherited by `init`.

8.6 `wait` and `waitpid` Functions

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent. Because the termination of a child is an asynchronous event—it can happen at any time while the parent is running—this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored. We describe these options in Chapter 10. For now, we need to be aware that a process that calls `wait` or `waitpid` can

- Block, if all of its children are still running
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- Return immediately with an error, if it doesn't have any child processes

If the process is calling `wait` because it received the `SIGCHLD` signal, we expect `wait` to return immediately. But if we call it at any random point in time, it can block.

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

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 (see later), or –1 on error

The differences between these two functions are as follows:

- The `wait` function can block the caller until a child process terminates, whereas `waitpid` has an option that prevents it from blocking.
- The `waitpid` function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, `wait` returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, `wait` returns when one terminates. We can always tell which child terminated, because the process ID is returned by the function.

For both functions, the argument *statloc* is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument. If we don't care about the termination status, we simply pass a null pointer as this argument.

Traditionally, the integer status that these two functions return has been defined by the implementation, with certain bits indicating the exit status (for a normal return), other bits indicating the signal number (for an abnormal return), one bit indicating whether a core file was generated, and so on. POSIX.1 specifies that the termination status is to be looked at using various macros that are defined in `<sys/wait.h>`. Four mutually exclusive macros tell us how the process terminated, and they all begin with `WIF`. Based on which of these four macros is true, other macros are used to obtain the exit status, signal number, and the like. The four mutually exclusive macros are shown in Figure 8.4.

Macro	Description
`WIFEXITED(`*status*`)`	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS(`*status*`)` to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`, or `_Exit`.

`WIFSIGNALED(`*status*`)`	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute
	`WTERMSIG(`*status*`)`
	to fetch the signal number that caused the termination.
	Additionally, some implementations (but not the Single UNIX Specification) define the macro
	`WCOREDUMP(`*status*`)`
	that returns true if a core file of the terminated process was generated.

`WIFSTOPPED(`*status*`)`	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG(`*status*`)` to fetch the signal number that caused the child to stop.
`WIFCONTINUED(`*status*`)`	True if status was returned for a child that has been continued after a job control stop (XSI option; `waitpid` only).

Figure 8.4 Macros to examine the termination status returned by `wait` and `waitpid`

We'll discuss how a process can be stopped in Section 9.8 when we discuss job control.

Example

The function `pr_exit` in Figure 8.5 uses the macros from Figure 8.4 to print a description of the termination status. We'll call this function from numerous programs in the text. Note that this function handles the `WCOREDUMP` macro, if it is defined.

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

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

Figure 8.5 Print a description of the `exit` status

FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 all support the `WCOREDUMP` macro. However, some platforms hide its definition if the `_POSIX_C_SOURCE` constant is defined (recall Section 2.7).

The program shown in Figure 8.6 calls the `pr_exit` function, demonstrating the various values for the termination status. If we run the program in Figure 8.6, we get

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

For now, we print the signal number from `WTERMSIG`. We can look at the `<signal.h>` header to verify that `SIGABRT` has a value of 6 and that `SIGFPE` has a value of 8. We'll see a portable way to map a signal number to a descriptive name in Section 10.22. □

As we mentioned, if we have more than one child, `wait` returns on termination of any of the children. But what if we want to wait for a specific process to terminate (assuming we know which process ID we want to wait for)? In older versions of the UNIX System, we would have to call `wait` and compare the returned process ID with the one we're interested in. If the terminated process wasn't the one we wanted, we would have to save the process ID and termination status and call `wait` again. We would need to continue doing this until the desired process terminated. The next time we wanted to wait for a specific process, we would go through the list of already terminated processes to see whether we had already waited for it, and if not, call `wait`

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

int
main(void)
{
    pid_t   pid;
    int     status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        abort();                    /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        status /= 0;                /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

Figure 8.6 Demonstrate various `exit` statuses

again. What we need is a function that waits for a specific process. This functionality (and more) is provided by the POSIX.1 `waitpid` function.

The interpretation of the *pid* argument for `waitpid` depends on its value:

pid == –1	Waits for any child process. In this respect, `waitpid` is equivalent to `wait`.
pid > 0	Waits for the child whose process ID equals *pid*.
pid == 0	Waits for any child whose process group ID equals that of the calling process. (We discuss process groups in Section 9.4.)
pid < –1	Waits for any child whose process group ID equals the absolute value of *pid*.

The `waitpid` function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by *statloc*. With `wait`, the only real error is if the calling process has no children. (Another error return is possible, in case the function call is interrupted by a signal. We'll discuss this in Chapter 10.) With `waitpid`, however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

The *options* argument lets us further control the operation of `waitpid`. This argument either is 0 or is constructed from the bitwise OR of the constants in Figure 8.7.

> FreeBSD 8.0 and Solaris 10 support one additional, but nonstandard, *option* constant. `WNOWAIT` has the system keep the process whose termination status is returned by `waitpid` in a wait state, so that it may be waited for again.

Constant	Description
`WCONTINUED`	If the implementation supports job control, the status of any child specified by *pid* that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI option).
`WNOHANG`	The `waitpid` function will not block if a child specified by *pid* is not immediately available. In this case, the return value is 0.
`WUNTRACED`	If the implementation supports job control, the status of any child specified by *pid* that has stopped, and whose status has not been reported since it has stopped, is returned. The `WIFSTOPPED` macro determines whether the return value corresponds to a stopped child process.

Figure 8.7 The *options* constants for `waitpid`

The `waitpid` function provides three features that aren't provided by the `wait` function.

1. The `waitpid` function lets us wait for one particular process, whereas the `wait` function returns the status of any terminated child. We'll return to this feature when we discuss the `popen` function.
2. The `waitpid` function provides a nonblocking version of `wait`. There are times when we want to fetch a child's status, but we don't want to block.
3. The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

Example

Recall our discussion in Section 8.5 about zombie processes. If we want to write a process so that it `fork`s a child but we don't want to wait for the child to complete and we don't want the child to become a zombie until we terminate, the trick is to call `fork` twice. The program in Figure 8.8 does this.

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

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */

        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child, parent pid = %ld\n", (long)getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)   /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

Figure 8.8 Avoid zombie processes by calling `fork` twice

We call `sleep` in the second child to ensure that the first child terminates before printing the parent process ID. After a `fork`, either the parent or the child can continue executing; we never know which will resume execution first. If we didn't put the second child to sleep, and if it resumed execution after the `fork` before its parent, the parent process ID that it printed would be that of its parent, not process ID 1.

Executing the program in Figure 8.8 gives us

```
$ ./a.out
$ second child, parent pid = 1
```

Note that the shell prints its prompt when the original process terminates, which is before the second child prints its parent process ID. □

8.7 waitid Function

The Single UNIX Specification includes an additional function to retrieve the exit status of a process. The `waitid` function is similar to `waitpid`, but provides extra flexibility.

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns: 0 if OK, –1 on error

Like `waitpid`, `waitid` allows a process to specify which children to wait for. Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used. The *id* parameter is interpreted based on the value of *idtype*. The types supported are summarized in Figure 8.9.

Constant	Description
`P_PID`	Wait for a particular process: *id* contains the process ID of the child to wait for.
`P_PGID`	Wait for any child process in a particular process group: *id* contains the process group ID of the children to wait for.
`P_ALL`	Wait for any child process: *id* is ignored.

Figure 8.9 The *idtype* constants for `waitid`

The *options* argument is a bitwise OR of the flags shown in Figure 8.10. These flags indicate which state changes the caller is interested in.

Constant	Description
`WCONTINUED`	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
`WEXITED`	Wait for processes that have exited.
`WNOHANG`	Return immediately instead of blocking if there is no child exit status available.
`WNOWAIT`	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to `wait`, `waitid`, or `waitpid`.
`WSTOPPED`	Wait for a process that has stopped and whose status has not yet been reported.

Figure 8.10 The *options* constants for `waitid`

At least one of `WCONTINUED`, `WEXITED`, or `WSTOPPED` must be specified in the *options* argument.

The *infop* argument is a pointer to a `siginfo` structure. This structure contains detailed information about the signal generated that caused the state change in the child process. The `siginfo` structure is discussed further in Section 10.14.

> Of the four platforms covered in this book, only Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 provide support for `waitid`. Note, however, that Mac OS X 10.6.8 doesn't set all the information we expect in the `siginfo` structure.

8.8 wait3 and wait4 Functions

Most UNIX system implementations provide two additional functions: wait3 and wait4. Historically, these two variants descend from the BSD branch of the UNIX System. The only feature provided by these two functions that isn't provided by the wait, waitid, and waitpid functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK, 0, or –1 on error

The resource information includes such statistics as the amount of user CPU time, amount of system CPU time, number of page faults, number of signals received, and the like. Refer to the getrusage(2) manual page for additional details. (This resource information differs from the resource limits we described in Section 7.11.) Figure 8.11 details the various arguments supported by the wait functions.

Function	*pid*	*options*	*rusage*	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
wait				•	•	•	•	•
waitid	•	•		•		•	•	•
waitpid	•	•		•	•	•	•	•
wait3		•	•		•	•	•	•
wait4	•	•	•		•	•	•	•

Figure 8.11 Arguments supported by wait functions on various systems

The wait3 function was included in earlier versions of the Single UNIX Specification. In Version 2, wait3 was moved to the legacy category; wait3 was removed from the specification in Version 3.

8.9 Race Conditions

For our purposes, a race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork. In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

We saw a potential race condition in the program in Figure 8.8 when the second child printed its parent process ID. If the second child runs before the first child, then its parent process will be the first child. But if the first child runs first and has enough time to `exit`, then the parent process of the second child is `init`. Even calling `sleep`, as we did, guarantees nothing. If the system was heavily loaded, the second child could resume after `sleep` returns, before the first child has a chance to run. Problems of this form can be difficult to debug because they tend to work "most of the time."

A process that wants to wait for a child to terminate must call one of the `wait` functions. If a process wants to wait for its parent to terminate, as in the program from Figure 8.8, a loop of the following form could be used:

```
while (getppid() != 1)
    sleep(1);
```

The problem with this type of loop, called *polling*, is that it wastes CPU time, as the caller is awakened every second to test the condition.

To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes. Signals can be used for this purpose, and we describe one way to do this in Section 10.16. Various forms of interprocess communication (IPC) can also be used. We'll discuss some of these options in Chapters 15 and 17.

For a parent and child relationship, we often have the following scenario. After the `fork`, both the parent and the child have something to do. For example, the parent could update a record in a log file with the child's process ID, and the child might have to create a file for the parent. In this example, we require that each process tell the other when it has finished its initial set of operations, and that each wait for the other to complete, before heading off on its own. The following code illustrates this scenario:

```
#include  "apue.h"

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {            /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */
    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */

    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);            /* tell child we're done */
WAIT_CHILD();               /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

We assume that the header `apue.h` defines whatever variables are required. The five routines `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` can be either macros or functions.

We'll show various ways to implement these `TELL` and `WAIT` routines in later chapters: Section 10.16 shows an implementation using signals; Figure 15.7 shows an implementation using pipes. Let's look at an example that uses these five routines.

Example

The program in Figure 8.12 outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and the length of time for which each process runs.

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Figure 8.12 Program with a race condition

We set the standard output unbuffered, so every character output generates a `write`. The goal in this example is to allow the kernel to switch between the two processes as often as possible to demonstrate the race condition. (If we didn't do this, we might never see the type of output that follows. Not seeing the erroneous output doesn't

mean that the race condition doesn't exist; it simply means that we can't see it on this particular system.) The following actual output shows how the results can vary:

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

We need to change the program in Figure 8.12 to use the `TELL` and `WAIT` functions. The program in Figure 8.13 does this. The lines preceded by a plus sign are new lines.

```
  #include "apue.h"

  static void charatatime(char *);

  int
  main(void)
  {
      pid_t   pid;

+     TELL_WAIT();
+
      if ((pid = fork()) < 0) {
          err_sys("fork error");
      } else if (pid == 0) {
+         WAIT_PARENT();      /* parent goes first */
          charatatime("output from child\n");
      } else {
          charatatime("output from parent\n");
+         TELL_CHILD(pid);
      }
      exit(0);
  }

  static void
  charatatime(char *str)
  {
      char    *ptr;
      int     c;

      setbuf(stdout, NULL);           /* set unbuffered */
      for (ptr = str; (c = *ptr++) != 0; )
          putc(c, stdout);
  }
```

Figure 8.13 Modification of Figure 8.12 to avoid race condition

When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

In the program shown in Figure 8.13, the parent goes first. The child goes first if we change the lines following the fork to be

```
} else if (pid == 0) {
    charatatime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();        /* child goes first */
    charatatime("output from parent\n");
}
```

Exercise 8.4 continues this example. □

8.10 exec Functions

We mentioned in Section 8.3 that one use of the fork function is to create a new process (the child) that then causes another program to be executed by calling one of the exec functions. When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process—its text, data, heap, and stack segments—with a brand-new program from disk.

There are seven different exec functions, but we'll often simply refer to "the exec function," which means that we could use any of the seven functions. These seven functions round out the UNIX System process control primitives. With fork, we can create new processes; and with the exec functions, we can initiate new programs. The exit function and the wait functions handle termination and waiting for termination. These are the only process control primitives we need. We'll use these primitives in later sections to build additional functions, such as popen and system.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

All seven return: –1 on error, no return on success

The first difference in these functions is that the first four take a pathname argument, the next two take a filename argument, and the last one takes a file descriptor argument. When a *filename* argument is specified,

- If *filename* contains a slash, it is taken as a pathname.
- Otherwise, the executable file is searched for in the directories specified by the `PATH` environment variable.

The `PATH` variable contains a list of directories, called path prefixes, that are separated by colons. For example, the *name=value* environment string

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

specifies four directories to search. The last path prefix specifies the current directory. (A zero-length prefix also means the current directory. It can be specified as a colon at the beginning of the *value*, two colons in a row, or a colon at the end of the *value*.)

> There are security reasons for *never* including the current directory in the search path. See Garfinkel et al. [2003].

If either `execlp` or `execvp` finds an executable file using one of the path prefixes, but the file isn't a machine executable that was generated by the link editor, the function assumes that the file is a shell script and tries to invoke `/bin/sh` with the *filename* as input to the shell.

With `fexecve`, we avoid the issue of finding the correct executable file altogether and rely on the caller to do this. By using a file descriptor, the caller can verify the file is in fact the intended file and execute it without a race. Otherwise, a malicious user with appropriate privileges could replace the executable file (or a portion of the path to the executable file) after it has been located and verified, but before the caller can execute it (recall the discussion of TOCTTOU errors in Section 3.3).

The next difference concerns the passing of the argument list (`l` stands for list and `v` stands for vector). The functions `execl`, `execlp`, and `execle` require each of the command-line arguments to the new program to be specified as separate arguments. We mark the end of the arguments with a null pointer. For the other four functions (`execv`, `execvp`, `execve`, and `fexecve`), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

Before using ISO C prototypes, the normal way to show the command-line arguments for the three functions `execl`, `execle`, and `execlp` was

```
char *arg0, char *arg1, ..., char *argn, (char *)0
```

This syntax explicitly shows that the final command-line argument is followed by a null pointer. If this null pointer is specified by the constant `0`, we must cast it to a pointer; if we don't, it's interpreted as an integer argument. If the size of an integer is different from the size of a `char *`, the actual arguments to the `exec` function will be wrong.

The final difference is the passing of the environment list to the new program. The three functions whose names end in an `e` (`execle`, `execve`, and `fexecve`) allow us to pass a pointer to an array of pointers to the environment strings. The other four

functions, however, use the environ variable in the calling process to copy the existing environment for the new program. (Recall our discussion of the environment strings in Section 7.9 and Figure 7.8. We mentioned that if the system supported such functions as setenv and putenv, we could change the current environment and the environment of any subsequent child processes, but we couldn't affect the environment of the parent process.) Normally, a process allows its environment to be propagated to its children, but in some cases, a process wants to specify a certain environment for a child. One example of the latter is the login program when a new login shell is initiated. Normally, login creates a specific environment with only a few variables defined and lets us, through the shell start-up file, add variables to the environment when we log in.

Before using ISO C prototypes, the arguments to execle were shown as

```
char *pathname, char *arg0, ..., char *argn, (char *)0, char *envp[]
```

This syntax specifically shows that the final argument is the address of the array of character pointers to the environment strings. The ISO C prototype doesn't show this, as all the command-line arguments, the null pointer, and the *envp* pointer are shown with the ellipsis notation (...).

The arguments for these seven exec functions are difficult to remember. The letters in the function names help somewhat. The letter p means that the function takes a *filename* argument and uses the PATH environment variable to find the executable file. The letter l means that the function takes a list of arguments and is mutually exclusive with the letter v, which means that it takes an *argv*[] vector. Finally, the letter e means that the function takes an *envp*[] array instead of using the current environment. Figure 8.14 shows the differences among these seven functions.

Function	*pathname*	*filename*	*fd*	Arg list	*argv*[]	environ	*envp*[]
execl	•			•		•	
execlp		•		•		•	
execle	•			•			•
execv	•				•	•	
execvp		•			•	•	
execve	•				•		•
fexecve			•		•		•
(letter in name)		p	f	l	v		e

Figure 8.14 Differences among the seven exec functions

Every system has a limit on the total size of the argument list and the environment list. From Section 2.5.2 and Figure 2.8, this limit is given by ARG_MAX. This value must be at least 4,096 bytes on a POSIX.1 system. We sometimes encounter this limit when using the shell's filename expansion feature to generate a list of filenames. On some systems, for example, the command

```
grep getrlimit /usr/share/man/*/*
```

can generate a shell error of the form

```
Argument list too long
```

> Historically, the limit in older System V implementations was 5,120 bytes. Older BSD systems had a limit of 20,480 bytes. The limit in current systems is much higher. (See the output from the program in Figure 2.14, which is summarized in Figure 2.15.)

To get around the limitation in argument list size, we can use the `xargs`(1) command to break up long argument lists. To look for all the occurrences of `getrlimit` in the man pages on our system, we could use

```
find /usr/share/man -type f -print | xargs grep getrlimit
```

If the man pages on our system are compressed, however, we could try

```
find /usr/share/man -type f -print | xargs bzgrep getrlimit
```

We use the `type -f` option to the `find` command to restrict the list so that it contains only regular files, because the `grep` commands can't search for patterns in directories, and we want to avoid unnecessary error messages.

We've mentioned that the process ID does not change after an `exec`, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Nice value (on XSI-conformant systems; see Section 8.16)
- Values for `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`

The handling of open files depends on the value of the close-on-exec flag for each descriptor. Recall from Figure 3.7 and our mention of the `FD_CLOEXEC` flag in Section 3.14 that every open descriptor in a process has a close-on-exec flag. If this flag is set, the descriptor is closed across an `exec`. Otherwise, the descriptor is left open across the `exec`. The default is to leave the descriptor open across the `exec` unless we specifically set the close-on-exec flag using `fcntl`.

POSIX.1 specifically requires that open directory streams (recall the `opendir`

function from Section 4.22) be closed across an exec. This is normally done by the opendir function calling fcntl to set the close-on-exec flag for the descriptor corresponding to the open directory stream.

Note that the real user ID and the real group ID remain the same across the exec, but the effective IDs can change, depending on the status of the set-user-ID and the set-group-ID bits for the program file that is executed. If the set-user-ID bit is set for the new program, the effective user ID becomes the owner ID of the program file. Otherwise, the effective user ID is not changed (it's not set to the real user ID). The group ID is handled in the same way.

In many UNIX system implementations, only one of these seven functions, execve, is a system call within the kernel. The other six are just library functions that eventually invoke this system call. We can illustrate the relationship among these seven functions as shown in Figure 8.15.

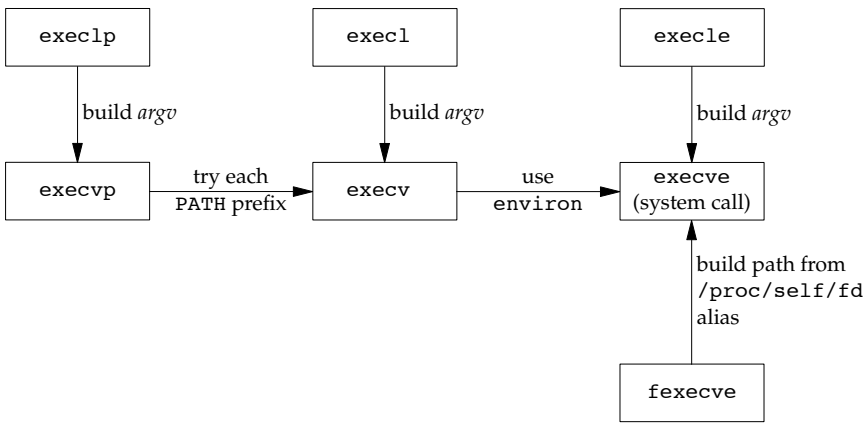

Figure 8.15 Relationship of the seven exec functions

In this arrangement, the library functions execlp and execvp process the PATH environment variable, looking for the first path prefix that contains an executable file named *filename*. The fexecve library function uses /proc to convert the file descriptor argument into a pathname that can be used by execve to execute the program.

> This describes how fexecve is implemented in FreeBSD 8.0 and Linux 3.2.0. Other systems might take a different approach. For example, a system without /proc or /dev/fd could implement fexecve as a system call veneer that translates the file descriptor argument into an i-node pointer, implement execve as a system call veneer that translates the pathname argument into an i-node pointer, and place all the rest of the exec code common to both execve and fexecve in a separate function to be called with an i-node pointer for the file to be executed.

Example

The program in Figure 8.16 demonstrates the exec functions.

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

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

Figure 8.16 Example of exec functions

We first call `execle`, which requires a pathname and a specific environment. The next call is to `execlp`, which uses a filename and passes the caller's environment to the new program. The only reason the call to `execlp` works is that the directory `/home/sar/bin` is one of the current path prefixes. Note also that we set the first argument, `argv[0]` in the new program, to be the filename component of the pathname. Some shells set this argument to be the complete pathname. This is a convention only; we can set `argv[0]` to any string we like. The `login` command does this when it executes the shell. Before executing the shell, `login` adds a dash as a prefix to `argv[0]` to indicate to the shell that it is being invoked as a login shell. A login shell will execute the start-up profile commands, whereas a nonlogin shell will not.

The program `echoall` that is executed twice in the program in Figure 8.16 is shown in Figure 8.17. It is a trivial program that echoes all its command-line arguments and its entire environment list.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

Figure 8.17 Echo all command-line arguments and all environment strings

When we execute the program from Figure 8.16, we get

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
                                    47 more lines that aren't shown
HOME=/home/sar
```

Note that the shell prompt appeared before the printing of `argv[0]` from the second `exec`. This occurred because the parent did not `wait` for this child process to finish. □

8.11 Changing User IDs and Group IDs

In the UNIX System, privileges, such as being able to change the system's notion of the current date, and access control, such as being able to read or write a particular file, are based on user and group IDs. When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

In general, we try to use the *least-privilege* model when we design our applications. According to this model, our programs should use the least privilege necessary to accomplish any given task. This reduces the risk that security might be compromised by a malicious user trying to trick our programs into using their privileges in unintended ways.

We can set the real user ID and effective user ID with the `setuid` function. Similarly, we can set the real group ID and the effective group ID with the `setgid` function.

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

Both return: 0 if OK, –1 on error

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

1. If the process has superuser privileges, the `setuid` function sets the real user ID, effective user ID, and saved set-user-ID to *uid*.
2. If the process does not have superuser privileges, but *uid* equals either the real user ID or the saved set-user-ID, `setuid` sets only the effective user ID to *uid*. The real user ID and the saved set-user-ID are not changed.
3. If neither of these two conditions is true, `errno` is set to `EPERM` and –1 is returned.

Here, we are assuming that `_POSIX_SAVED_IDS` is true. If this feature isn't provided, then delete all preceding references to the saved set-user-ID.

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

We can make a few statements about the three user IDs that the kernel maintains.

1. Only a superuser process can change the real user ID. Normally, the real user ID is set by the `login`(1) program when we log in and never changes. Because `login` is a superuser process, it sets all three user IDs when it calls `setuid`.
2. The effective user ID is set by the `exec` functions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the `exec` functions leave the effective user ID as its current value. We can call `setuid` at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.
3. The saved set-user-ID is copied from the effective user ID by `exec`. If the file's set-user-ID bit is set, this copy is saved after `exec` stores the effective user ID from the file's user ID.

Figure 8.18 summarizes the various ways these three user IDs can be changed.

ID	exec		setuid(*uid*)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

Figure 8.18 Ways to change the three user IDs

Note that we can obtain only the current value of the real user ID and the effective user ID with the functions `getuid` and `geteuid` from Section 8.2. We have no portable way to obtain the current value of the saved set-user-ID.

> FreeBSD 8.0 and LINUX 3.2.0 provide the `getresuid` and `getresgid` functions, which can be used to get the saved set-user-ID and saved set-group-ID, respectively.

`setreuid` and `setregid` Functions

Historically, BSD supported the swapping of the real user ID and the effective user ID with the `setreuid` function.

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

Both return: 0 if OK, –1 on error

We can supply a value of –1 for any of the arguments to indicate that the corresponding ID should remain unchanged.

The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user-ID operations. When the saved set-user-ID feature was introduced with POSIX.1, the rule was enhanced to also allow an unprivileged user to set its effective user ID to its saved set-user-ID.

> Both `setreuid` and `setregid` are included in the XSI option in POSIX.1. As such, all UNIX System implementations are expected to provide support for them.
>
> 4.3BSD didn't have the saved set-user-ID feature described earlier; it used `setreuid` and `setregid` instead. This allowed an unprivileged user to swap back and forth between the two values. Be aware, however, that when programs that used this feature spawned a shell, they had to set the real user ID to the normal user ID before the `exec`. If they didn't do this, the real user ID could be privileged (from the swap done by `setreuid`) and the shell process could call `setreuid` to swap the two and assume the permissions of the more privileged user. As a defensive programming measure to solve this problem, programs set both the real user ID and the effective user ID to the normal user ID before the call to `exec` in the child.

`seteuid` and `setegid` Functions

POSIX.1 includes the two functions `seteuid` and `setegid`. These functions are similar to `setuid` and `setgid`, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
                                        Both return: 0 if OK, –1 on error
```

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user, only the effective user ID is set to *uid*. (This behavior differs from that of the `setuid` function, which changes all three user IDs.)

Figure 8.19 summarizes all the functions that we've described here that modify the three user IDs.

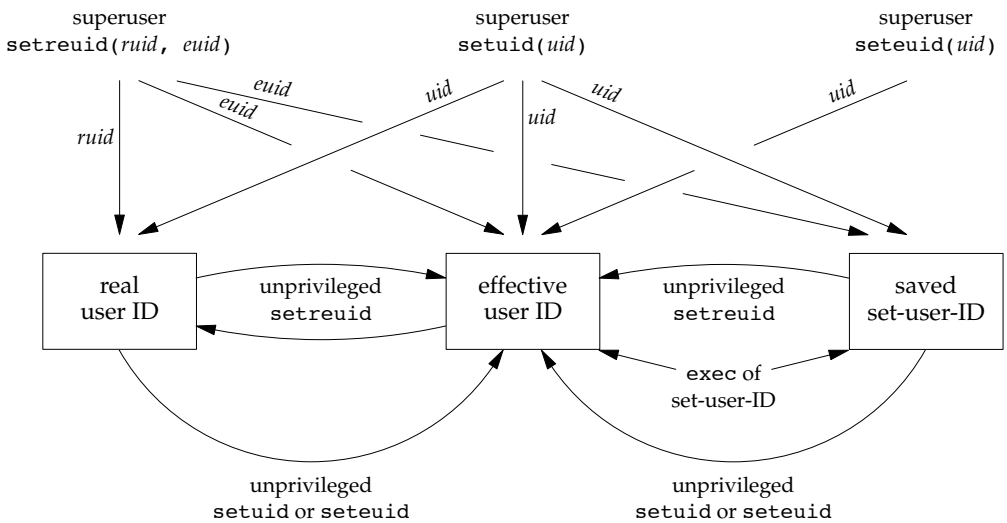

Figure 8.19 Summary of all the functions that set the various user IDs

Group IDs

Everything that we've said so far in this section also applies in a similar fashion to group IDs. The supplementary group IDs are not affected by `setgid`, `setregid`, or `setegid`.

Example

To see the utility of the saved set-user-ID feature, let's examine the operation of a program that uses it. We'll look at the `at`(1) program, which we can use to schedule commands to be run at some time in the future.

> On Linux 3.2.0, the `at` program is installed set-user-ID to user `daemon`. On FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10, the `at` program is installed set-user-ID to user `root`. This allows the `at` command to write privileged files owned by the daemon that will run the commands on behalf of the user running the `at` command. On Linux 3.2.0, the programs are run by the `atd`(8) daemon. On FreeBSD 8.0 and Solaris 10, the programs are run by the `cron`(1M) daemon. On Mac OS X 10.6.8, the programs are run by the `launchd`(8) daemon.

To prevent being tricked into running commands that we aren't allowed to run, or reading or writing files that we aren't allowed to access, the `at` command and the daemon that ultimately runs the commands on our behalf have to switch between sets of privileges: ours and those of the daemon. The following steps take place.

1. Assuming that the `at` program file is owned by `root` and has its set-user-ID bit set, when we run it, we have

   real user ID = our user ID (unchanged)
   effective user ID = `root`
   saved set-user-ID = `root`

2. The first thing the `at` command does is reduce its privileges so that it runs with our privileges. It calls the `seteuid` function to set the effective user ID to our real user ID. After this, we have

   real user ID = our user ID (unchanged)
   effective user ID = our user ID
   saved set-user-ID = `root` (unchanged)

3. The `at` program runs with our privileges until it needs to access the configuration files that control which commands are to be run and the time at which they need to run. These files are owned by the daemon that will run the commands for us. The `at` command calls `seteuid` to set the effective user ID to `root`. This call is allowed because the argument to `seteuid` equals the saved set-user-ID. (This is why we need the saved set-user-ID.) After this, we have

   real user ID = our user ID (unchanged)
   effective user ID = `root`
   saved set-user-ID = `root` (unchanged)

   Because the effective user ID is `root`, file access is allowed.

4. After the files are modified to record the commands to be run and the time at which they are to be run, the `at` command lowers its privileges by calling

`seteuid` to set its effective user ID to our user ID. This prevents any accidental misuse of privilege. At this point, we have

```
real user ID = our user ID (unchanged)
effective user ID = our user ID
saved set-user-ID = root (unchanged)
```

5. The daemon starts out running with `root` privileges. To run commands on our behalf, the daemon calls `fork` and the child calls `setuid` to change its user ID to our user ID. Because the child is running with `root` privileges, this changes all of the IDs. We have

```
real user ID = our user ID
effective user ID = our user ID
saved set-user-ID = our user ID
```

   Now the daemon can safely execute commands on our behalf, because it can access only the files to which we normally have access. We have no additional permissions.

By using the saved set-user-ID in this fashion, we can use the extra privileges granted to us by the set-user-ID of the program file only when we need elevated privileges. Any other time, however, the process runs with our normal permissions. If we weren't able to switch back to the saved set-user-ID at the end, we might be tempted to retain the extra permissions the whole time we were running (which is asking for trouble). □

8.12 Interpreter Files

All contemporary UNIX systems support interpreter files. These files are text files that begin with a line of the form

```
#! pathname [ optional-argument ]
```

The space between the exclamation point and the *pathname* is optional. The most common of these interpreter files begin with the line

```
#!/bin/sh
```

The *pathname* is normally an absolute pathname, since no special operations are performed on it (i.e., `PATH` is not used). The recognition of these files is done within the kernel as part of processing the `exec` system call. The actual file that gets executed by the kernel is not the interpreter file, but rather the file specified by the *pathname* on the first line of the interpreter file. Be sure to differentiate between the interpreter file—a text file that begins with `#!`—and the interpreter, which is specified by the *pathname* on the first line of the interpreter file.

Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the `#!`, the *pathname*, the optional argument, the terminating newline, and any spaces.

> On FreeBSD 8.0, this limit is 4,097 bytes. On Linux 3.2.0, the limit is 128 bytes. Mac OS X 10.6.8 supports a limit of 513 bytes, whereas Solaris 10 places the limit at 1,024 bytes.

Example

Let's look at an example to see what the kernel does with the arguments to the exec function when the file being executed is an interpreter file and the optional argument on the first line of the interpreter file. The program in Figure 8.20 execs an interpreter file.

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

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0)  /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Figure 8.20 A program that execs an interpreter file

The following shows the contents of the one-line interpreter file that is executed and the result from running the program in Figure 8.20:

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

The program echoarg (the interpreter) just echoes each of its command-line arguments. (This is the program from Figure 7.4.) Note that when the kernel execs the interpreter (/home/sar/bin/echoarg), argv[0] is the *pathname* of the interpreter, argv[1] is the optional argument from the interpreter file, and the remaining arguments are the *pathname* (/home/sar/bin/testinterp) and the second and third arguments from the call to execl in the program shown in Figure 8.20 (myarg1 and MY ARG2). Both argv[1] and argv[2] from the call to execl have been shifted right two positions. Note that the kernel takes the *pathname* from the execl call instead of the first argument (testinterp), on the assumption that the *pathname* might contain more information than the first argument. □

Example

A common use for the optional argument following the interpreter *pathname* is to specify the `-f` option for programs that support this option. For example, an `awk`(1) program can be executed as

```
awk -f myfile
```

which tells `awk` to read the `awk` program from the file `myfile`.

> Systems derived from UNIX System V often include two versions of the `awk` language. On these systems, `awk` is often called "old `awk`" and corresponds to the original version distributed with Version 7. In contrast, `nawk` (new `awk`) contains numerous enhancements and corresponds to the language described in Aho, Kernighan, and Weinberger [1988]. This newer version provides access to the command-line arguments, which we need for the example that follows. Solaris 10 provides both versions.
>
> The `awk` program is one of the utilities included by POSIX in its 1003.2 standard, which is now part of the base POSIX.1 specification in the Single UNIX Specification. This utility is also based on the language described in Aho, Kernighan, and Weinberger [1988].
>
> The version of `awk` in Mac OS X 10.6.8 is based on the Bell Laboratories version, which has been placed in the public domain. FreeBSD 8.0 and some Linux distributions ship with GNU `awk`, called `gawk`, which is linked to the name `awk`. `gawk` conforms to the POSIX standard, but also includes other extensions. Because they are more up-to-date, `gawk` and the version of `awk` from Bell Laboratories are preferred to either `nawk` or old `awk`. (The Bell Labs version of `awk` is available at `http://cm.bell-labs.com/cm/cs/awkbook/index.html`.)

Using the `-f` option with an interpreter file lets us write

```
#!/bin/awk -f
(awk program follows in the interpreter file)
```

For example, Figure 8.21 shows `/usr/local/bin/awkexample` (an interpreter file).

```
#!/usr/bin/awk -f
# Note: on Solaris, use nawk instead
BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

Figure 8.21 An awk program as an interpreter file

If one of the path prefixes is `/usr/local/bin`, we can execute the program in Figure 8.21 (assuming that we've turned on the execute bit for the file) as

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

When `/bin/awk` is executed, its command-line arguments are

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

The pathname of the interpreter file (`/usr/local/bin/awkexample`) is passed to the interpreter. The filename portion of this pathname (what we typed to the shell) isn't adequate, because the interpreter (`/bin/awk` in this example) can't be expected to use the `PATH` variable to locate files. When it reads the interpreter file, `awk` ignores the first line, since the pound sign is `awk`'s comment character.

We can verify these command-line arguments with the following commands:

```
$ /bin/su                                        become superuser
Password:                                        enter superuser password
# mv /usr/bin/awk /usr/bin/awk.save              save the original program
# cp /home/sar/bin/echoarg /usr/bin/awk          and replace it temporarily
# suspend                                        suspend the superuser shell
[1] + Stopped          /bin/su                   using job control
$ awkexample file1 FILENAME2 f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILENAME2
argv[5]: f3
$ fg                                             resume superuser shell using job control
/bin/su
# mv /usr/bin/awk.save /usr/bin/awk              restore the original program
# exit                                           and exit the superuser shell
```

In this example, the `-f` option for the interpreter is required. As we said, this tells `awk` where to look for the `awk` program. If we remove the `-f` option from the interpreter file, an error message usually results when we try to run it. The exact text of the message varies, depending on where the interpreter file is stored and whether the remaining arguments represent existing files. This is because the command-line arguments in this case are

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

and `awk` is trying to interpret the string `/usr/local/bin/awkexample` as an `awk` program. If we couldn't pass at least a single optional argument to the interpreter (`-f` in this case), these interpreter files would be usable only with the shells. □

Are interpreter files required? Not really. They provide an efficiency gain for the user at some expense in the kernel (since it's the kernel that recognizes these files). Interpreter files are useful for the following reasons.

1. They hide that certain programs are scripts in some other language. For example, to execute the program in Figure 8.21, we just say

   `awkexample` *optional-arguments*

instead of needing to know that the program is really an `awk` script that we would otherwise have to execute as

```
awk -f awkexample optional-arguments
```

2. Interpreter scripts provide an efficiency gain. Consider the previous example again. We could still hide that the program is an `awk` script, by wrapping it in a shell script:

```
awk 'BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}' $*
```

The problem with this solution is that more work is required. First, the shell reads the command and tries to `execlp` the filename. Because the shell script is an executable file but isn't a machine executable, an error is returned and `execlp` assumes that the file is a shell script (which it is). Then `/bin/sh` is executed with the pathname of the shell script as its argument. The shell correctly runs our script, but to run the `awk` program, the shell does a `fork`, `exec`, and `wait`. Thus there is more overhead involved in replacing an interpreter script with a shell script.

3. Interpreter scripts let us write shell scripts using shells other than `/bin/sh`. When it finds an executable file that isn't a machine executable, `execlp` has to choose a shell to invoke, and it always uses `/bin/sh`. Using an interpreter script, however, we can simply write

```
#!/bin/csh
(C shell script follows in the interpreter file)
```

Again, we could wrap all of this in a `/bin/sh` script (that invokes the C shell), as we described earlier, but more overhead is required.

None of this would work as we've shown here if the three shells and `awk` didn't use the pound sign as their comment character.

8.13 `system` Function

It is convenient to execute a command string from within a program. For example, assume that we want to put a time-and-date stamp into a certain file. We could use the functions described in Section 6.10 to do this: call `time` to get the current calendar time, then call `localtime` to convert it to a broken-down time, then call `strftime` to format the result, and finally write the result to the file. It is much easier, however, to say

```
system("date > file");
```

ISO C defines the `system` function, but its operation is strongly system dependent. POSIX.1 includes the `system` interface, expanding on the ISO C definition to describe its behavior in a POSIX environment.

```
#include <stdlib.h>

int system(const char *cmdstring);

                                                    Returns: (see below)
```

If *cmdstring* is a null pointer, `system` returns nonzero only if a command processor is available. This feature determines whether the `system` function is supported on a given operating system. Under the UNIX System, `system` is always available.

Because `system` is implemented by calling `fork`, `exec`, and `waitpid`, there are three types of return values.

1. If either the `fork` fails or `waitpid` returns an error other than `EINTR`, `system` returns –1 with `errno` set to indicate the error.
2. If the `exec` fails, implying that the shell can't be executed, the return value is as if the shell had executed `exit(127)`.
3. Otherwise, all three functions—`fork`, `exec`, and `waitpid`—succeed, and the return value from `system` is the termination status of the shell, in the format specified for `waitpid`.

> Some older implementations of `system` returned an error (`EINTR`) if `waitpid` was interrupted by a caught signal. Because there is no strategy that an application can use to recover from this type of error (the process ID of the child is hidden from the caller), POSIX later added the requirement that `system` not return an error in this case. (We discuss interrupted system calls in Section 10.5.)

Figure 8.22 shows an implementation of the `system` function. The one feature that it doesn't handle is signals. We'll update this function with signal handling in Section 10.18.

The shell's `-c` option tells it to take the next command-line argument—*cmdstring*, in this case—as its command input instead of reading from standard input or from a given file. The shell parses this null-terminated C string and breaks it up into separate command-line arguments for the command. The actual command string that is passed to the shell can contain any valid shell commands. For example, input and output redirection using < and > can be used.

If we didn't use the shell to execute the command, but tried to execute the command ourself, it would be more difficult. First, we would want to call `execlp`, instead of `execl`, to use the `PATH` variable, like the shell. We would also have to break up the null-terminated C string into separate command-line arguments for the call to `execlp`. Finally, we wouldn't be able to use any of the shell metacharacters.

Note that we call `_exit` instead of `exit`. We do this to prevent any standard I/O buffers, which would have been copied from the parent to the child across the `fork`, from being flushed in the child.

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)   /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

Figure 8.22 The `system` function, without signal handling

We can test this version of `system` with the program shown in Figure 8.23. (The `pr_exit` function was defined in Figure 8.5.) Running the program in Figure 8.23 gives us

```
$ ./a.out
Sat Feb 25 19:36:59 EST 2012
normal termination, exit status = 0     for date
sh: nosuchcommand: command not found
normal termination, exit status = 127   for nosuchcommand
sar      console  Jan  1 14:59
sar      ttys000  Feb  7 19:08
sar      ttys001  Jan 15 15:28
sar      ttys002  Jan 15 21:50
sar      ttys003  Jan 21 16:02
normal termination, exit status = 44    for exit
```

The advantage in using `system`, instead of using `fork` and `exec` directly, is that `system` does all the required error handling and (in our next version of this function in Section 10.18) all the required signal handling.

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

int
main(void)
{
    int     status;

    if ((status = system("date")) < 0)
        err_sys("system() error");

    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("system() error");

    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("system() error");

    pr_exit(status);

    exit(0);
}
```

Figure 8.23 Calling the system function

Earlier systems, including SVR3.2 and 4.3BSD, didn't have the waitpid function available. Instead, the parent waited for the child, using a statement such as

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
    ;
```

A problem occurs if the process that calls system has spawned its own children before calling system. Because the while statement above keeps looping until the child that was generated by system terminates, if any children of the process terminate before the process identified by pid, then the process ID and termination status of these other children are discarded by the while statement. Indeed, this inability to wait for a specific child is one of the reasons given in the POSIX.1 Rationale for including the waitpid function. We'll see in Section 15.3 that the same problem occurs with the popen and pclose functions if the system doesn't provide a waitpid function.

Set-User-ID Programs

What happens if we call system from a set-user-ID program? Doing so creates a security hole and should never be attempted. Figure 8.24 shows a simple program that just calls system for its command-line argument.

```
#include "apue.h"

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

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");

    pr_exit(status);

    exit(0);
}
```

Figure 8.24 Execute the command-line argument using `system`

We'll compile this program into the executable file `tsys`.

Figure 8.25 shows another simple program that prints its real and effective user IDs.

```
#include "apue.h"

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

Figure 8.25 Print real and effective user IDs

We'll compile this program into the executable file `printuids`. Running both programs gives us the following:

```
$ tsys printuids                        normal execution, no special privileges
real uid = 205, effective uid = 205
normal termination, exit status = 0
$ su                                    become superuser
Password:                               enter superuser password
# chown root tsys                       change owner
# chmod u+s tsys                        make set-user-ID
# ls -l tsys                            verify file's permissions and owner
-rwsrwxr-x  1 root    7888 Feb 25 22:13 tsys
# exit                                  leave superuser shell
$ tsys printuids
real uid = 205, effective uid = 0       oops, this is a security hole
normal termination, exit status = 0
```

The superuser permissions that we gave the `tsys` program are retained across the `fork` and `exec` that are done by `system`.

> Some implementations have closed this security hole by changing `/bin/sh` to reset the effective user ID to the real user ID when they don't match. On these systems, the previous example doesn't work as shown. Instead, the same effective user ID will be printed regardless of the status of the set-user-ID bit on the program calling `system`.

If it is running with special permissions—either set-user-ID or set-group-ID—and wants to spawn another process, a process should use `fork` and `exec` directly, being certain to change back to normal permissions after the `fork`, before calling `exec`. The `system` function should *never* be used from a set-user-ID or a set-group-ID program.

> One reason for this admonition is that `system` invokes the shell to parse the command string, and the shell uses its `IFS` variable as the input field separator. Older versions of the shell didn't reset this variable to a normal set of characters when invoked. As a result, a malicious user could set `IFS` before `system` was called, causing `system` to execute a different program.

8.14 Process Accounting

Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates. These accounting records typically contain a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on. We'll take a closer look at these accounting records in this section, as it gives us a chance to look at processes again and to use the `fread` function from Section 5.9.

> Process accounting is not specified by any of the standards. Thus all the implementations have annoying differences. For example, the I/O counts maintained on Solaris 10 are in units of bytes, whereas FreeBSD 8.0 and Mac OS X 10.6.8 maintain units of blocks, although there is no distinction between different block sizes, making the counter effectively useless. Linux 3.2.0, on the other hand, doesn't try to maintain I/O statistics at all.
>
> Each implementation also has its own set of administrative commands to process raw accounting data. For example, Solaris provides `runacct`(1m) and `acctcom`(1), whereas FreeBSD provides the `sa`(8) command to process and summarize the raw accounting data.

A function we haven't described (`acct`) enables and disables process accounting. The only use of this function is from the `accton`(8) command (which happens to be one of the few similarities among platforms). A superuser executes `accton` with a pathname argument to enable accounting. The accounting records are written to the specified file, which is usually `/var/account/acct` on FreeBSD and Mac OS X, `/var/log/account/pacct` on Linux, and `/var/adm/pacct` on Solaris. Accounting is turned off by executing `accton` without any arguments.

The structure of the accounting records is defined in the header `<sys/acct.h>`. Although the implementation of each system differs, the accounting records look something like

```
typedef  u_short comp_t;  /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char   ac_flag;    /* flag (see Figure 8.26) */
  char   ac_stat;    /* termination status (signal & core flag only) */
                     /* (Solaris only) */
  uid_t  ac_uid;     /* real user ID */
  gid_t  ac_gid;     /* real group ID */
  dev_t  ac_tty;     /* controlling terminal */
  time_t ac_btime;   /* starting calendar time */
  comp_t ac_utime;   /* user CPU time */
  comp_t ac_stime;   /* system CPU time */
  comp_t ac_etime;   /* elapsed time */
  comp_t ac_mem;     /* average memory usage */
  comp_t ac_io;      /* bytes transferred (by read and write) */
                     /* "blocks" on BSD systems */
  comp_t ac_rw;      /* blocks read or written */
                     /* (not present on BSD systems) */
  char   ac_comm[8]; /* command name: [8] for Solaris, */
                     /* [10] for Mac OS X, [16] for FreeBSD, and */
                     /* [17] for Linux */
};
```

Times are recorded in units of clock ticks on most platforms, but FreeBSD stores microseconds instead. The `ac_flag` member records certain events during the execution of the process. These events are described in Figure 8.26.

The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a `fork`. Each accounting record is written when the process terminates. This has two consequences.

First, we don't get accounting records for processes that never terminate. Processes like `init` that run for the lifetime of the system don't generate accounting records. This also applies to kernel daemons, which normally don't exit.

Second, the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started. To know the starting order, we would have to go through the accounting file and sort by the starting calendar time. But this isn't perfect, since calendar times are in units of seconds (Section 1.10), and it's possible for many processes to be started in any given second. Alternatively, the elapsed time is given in clock ticks, which are usually between 60 and 128 ticks per second. But we don't know the ending time of a process; all we know is its starting time and ending order. Thus, even though the elapsed time is more accurate than the starting time, we still can't reconstruct the exact starting order of various processes, given the data in the accounting file.

The accounting records correspond to processes, not programs. A new record is initialized by the kernel for the child after a `fork`, not when a new program is executed. Although `exec` doesn't create a new accounting record, the command name changes, and the `AFORK` flag is cleared. This means that if we have a chain of three programs—A

ac_flag	Description	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
AFORK	process is the result of fork, but never called exec	•	•	•	•
ASU	process used superuser privileges		•	•	•
ACORE	process dumped core	•	•	•	
AXSIG	process was killed by a signal	•	•	•	
AEXPND	expanded accounting entry				•
ANVER	new record format	•			

Figure 8.26 Values for ac_flag from accounting record

execs B, then B execs C, and C exits—only a single accounting record is written. The command name in the record corresponds to program C, but the CPU times, for example, are the sum for programs A, B, and C.

Example

To have some accounting data to examine, we'll create a test program to implement the diagram shown in Figure 8.27.

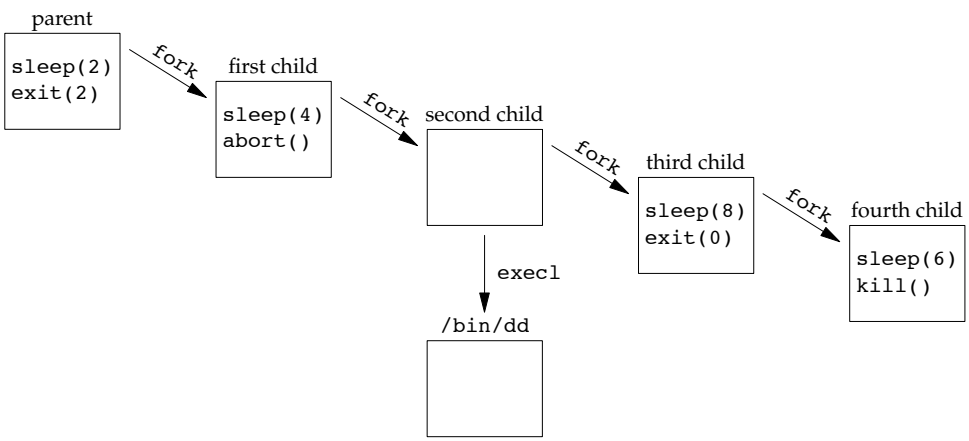

Figure 8.27 Process structure for accounting example

The source for the test program is shown in Figure 8.28. It calls fork four times. Each child does something different and then terminates.

```
#include "apue.h"

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
```

```
    else if (pid != 0) {        /* parent */
        sleep(2);
        exit(2);                /* terminate with exit status 2 */
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* first child */
        sleep(4);
        abort();                /* terminate with core dump */
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* second child */
        execl("/bin/dd", "dd", "if=/etc/passwd", "of=/dev/null", NULL);
        exit(7);                /* shouldn't get here */
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* third child */
        sleep(8);
        exit(0);                /* normal exit */
    }

    sleep(6);                   /* fourth child */
    kill(getpid(), SIGKILL);    /* terminate w/signal, no core dump */
    exit(6);                    /* shouldn't get here */
}
```

Figure 8.28 Program to generate accounting data

We'll run the test program on Solaris and then use the program in Figure 8.29 to print out selected fields from the accounting records.

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

#if defined(BSD)    /* different structure in FreeBSD */
#define acct acctv2
#define ac_flag ac_trailer.ac_flag
#define FMT "%-*.*s  e = %.0f, chars = %.0f, %c %c %c %c\n"
#elif defined(HAS_AC_STAT)
#define FMT "%-*.*s  e = %6ld, chars = %7ld, stat = %3u: %c %c %c %c\n"
#else
#define FMT "%-*.*s  e = %6ld, chars = %7ld, %c %c %c %c\n"
#endif
#if defined(LINUX)
#define acct acct_v3    /* different structure in Linux */
#endif

#if !defined(HAS_ACORE)
#define ACORE 0
```

```
#endif
#if !defined(HAS_AXSIG)
#define AXSIG 0
#endif

#if !defined(BSD)
static unsigned long
compt2ulong(comp_t comptime)    /* convert comp_t to unsigned long */
{
    unsigned long   val;
    int             exp;

    val = comptime & 0x1fff;    /* 13-bit fraction */
    exp = (comptime >> 13) & 7; /* 3-bit exponent (0-7) */
    while (exp-- > 0)
        val *= 8;
    return(val);
}
#endif

int
main(int argc, char *argv[])
{
    struct acct     acdata;
    FILE            *fp;

    if (argc != 2)
        err_quit("usage: pracct filename");
    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf(FMT, (int)sizeof(acdata.ac_comm),
            (int)sizeof(acdata.ac_comm), acdata.ac_comm,
#if defined(BSD)
            acdata.ac_etime, acdata.ac_io,
#else
            compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
#endif
#if defined(HAS_AC_STAT)
            (unsigned char) acdata.ac_stat,
#endif
            acdata.ac_flag & ACORE ? 'D' : ' ',
            acdata.ac_flag & AXSIG ? 'X' : ' ',
            acdata.ac_flag & AFORK ? 'F' : ' ',
            acdata.ac_flag & ASU   ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("read error");
    exit(0);
}
```

Figure 8.29 Print selected fields from system's accounting file

BSD-derived platforms don't support the `ac_stat` member, so we define the `HAS_AC_STAT` constant on the platforms that do support this member. Basing the defined symbol on the feature instead of on the platform makes the code read better and allows us to modify the program simply by adding the new definition to our compilation command. The alternative would be to use

```
#if !defined(BSD) && !defined(MACOS)
```

which becomes unwieldy as we port our application to additional platforms.

We define similar constants to determine whether the platform supports the `ACORE` and `AXSIG` accounting flags. We can't use the flag symbols themselves, because on Linux, they are defined as `enum` values, which we can't use in a `#ifdef` expression.

To perform our test, we do the following:

1. Become superuser and enable accounting, with the `accton` command. Note that when this command terminates, accounting should be on; therefore, the first record in the accounting file should be from this command.
2. Exit the superuser shell and run the program in Figure 8.28. This should append six records to the accounting file: one for the superuser shell, one for the test parent, and one for each of the four test children.

   A new process is not created by the `execl` in the second child. There is only a single accounting record for the second child.
3. Become superuser and turn accounting off. Since accounting is off when this `accton` command terminates, it should not appear in the accounting file.
4. Run the program in Figure 8.29 to print the selected fields from the accounting file.

The output from step 4 follows. We have appended the description of the process in italics to selected lines, for the discussion later.

```
accton     e =      1, chars =     336, stat =   0:       S
sh         e =   1550, chars =   20168, stat =   0:       S
dd         e =      2, chars =    1585, stat =   0:           second child
a.out      e =    202, chars =       0, stat =   0:           parent
a.out      e =    420, chars =       0, stat = 134:     F     first child
a.out      e =    600, chars =       0, stat =   9:     F     fourth child
a.out      e =    801, chars =       0, stat =   0:     F     third child
```

For this system, the elapsed time values are measured in units of clock ticks. Figure 2.15 shows that this system generates 100 clock ticks per second. For example, the `sleep(2)` in the parent corresponds to the elapsed time of 202 clock ticks. For the first child, the `sleep(4)` becomes 420 clock ticks. Note that the amount of time a process sleeps is not exact. (We'll return to the `sleep` function in Chapter 10.) Also, the calls to `fork` and `exit` take some amount of time.

Note that the `ac_stat` member is not the true termination status of the process, but rather corresponds to a portion of the termination status that we discussed in

Section 8.6. The only information in this byte is a core-flag bit (usually the high-order bit) and the signal number (usually the seven low-order bits), if the process terminated abnormally. If the process terminated normally, we are not able to obtain the `exit` status from the accounting file. For the first child, this value is 128 + 6. The 128 is the core flag bit, and 6 happens to be the value on this system for `SIGABRT`, which is generated by the call to `abort`. The value 9 for the fourth child corresponds to the value of `SIGKILL`. We can't tell from the accounting data that the parent's argument to `exit` was 2 and that the third child's argument to `exit` was 0.

The size of the file `/etc/passwd` that the `dd` process copies in the second child is 777 bytes. The number of characters of I/O is just over twice this value. It is twice the value, as 777 bytes are read in, then 777 bytes are written out. Even though the output goes to the null device, the bytes are still accounted for. The 31 additional bytes come from the `dd` command reporting the summary of bytes read and written, which it prints to `stdout`.

The `ac_flag` values are are what we would expect. The `F` flag is set for all the child processes except the second child, which does the `execl`. The `F` flag is not set for the parent, because the interactive shell that executed the parent did a `fork` and then an `exec` of the `a.out` file. The first child process calls `abort`, which generates a `SIGABRT` signal to generate the core dump. Note that neither the `X` flag nor the `D` flag is on, as they are not supported on Solaris; the information they represent can be derived from the `ac_stat` field. The fourth child also terminates because of a signal, but the `SIGKILL` signal does not generate a core dump; it just terminates the process.

As a final note, the first child has a 0 count for the number of characters of I/O, yet this process generated a `core` file. It appears that the I/O required to write the `core` file is not charged to the process. □

8.15 User Identification

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call `getpwuid(getuid())`, but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in under (Section 6.8), and the `getlogin` function provides a way to fetch that login name.

```
#include <unistd.h>

char *getlogin(void);
```

Returns: pointer to string giving login name if OK, `NULL` on error

This function can fail if the process is not attached to a terminal that a user logged in to. We normally call these processes *daemons*. We discuss them in Chapter 13.

Given the login name, we can then use it to look up the user in the password file—to determine the login shell, for example—using `getpwnam`.

> To find the login name, UNIX systems have historically called the `ttyname` function (Section 18.9) and then tried to find a matching entry in the `utmp` file (Section 6.8). FreeBSD and Mac OS X store the login name in the session structure associated with the process table entry and provide system calls to fetch and store this name.
>
> System V provided the `cuserid` function to return the login name. This function called `getlogin` and, if that failed, did a `getpwuid(getuid())`. The IEEE Standard 1003.1-1988 specified `cuserid`, but it called for the effective user ID to be used, instead of the real user ID. The 1990 version of POSIX.1 dropped the `cuserid` function.
>
> The environment variable `LOGNAME` is usually initialized with the user's login name by `login`(1) and inherited by the login shell. Realize, however, that a user can modify an environment variable, so we shouldn't use `LOGNAME` to validate the user in any way. Instead, we should use `getlogin`.

8.16 Process Scheduling

Historically, the UNIX System provided processes with only coarse control over their scheduling priority. The scheduling policy and priority were determined by the kernel. A process could choose to run with lower priority by adjusting its *nice value* (thus a process could be "nice" and reduce its share of the CPU by adjusting its nice value). Only a privileged process was allowed to increase its scheduling priority.

The real-time extensions in POSIX added interfaces to select among multiple scheduling classes and fine-tune their behavior. We discuss only the interfaces used to adjust the nice value here; they are part of the XSI option in POSIX.1. Refer to Gallmeister [1995] for more information on the real-time scheduling extensions.

In the Single UNIX Specification, nice values range from `0` to `(2*NZERO)-1`, although some implementations support a range from `0` to `2*NZERO`. Lower nice values have higher scheduling priority. Although this might seem backward, it actually makes sense: the more nice you are, the lower your scheduling priority is. `NZERO` is the default nice value of the system.

> Be aware that the header file defining `NZERO` differs among systems. In addition to the header file, Linux 3.2.0 makes the value of `NZERO` accessible through a nonstandard `sysconf` argument (`_SC_NZERO`).

A process can retrieve and change its nice value with the `nice` function. With this function, a process can affect only its own nice value; it can't affect the nice value of any other process.

```
#include <unistd.h>

int nice(int incr);
```

Returns: new nice value – `NZERO` if OK, –1 on error

The *incr* argument is added to the nice value of the calling process. If *incr* is too large, the system silently reduces it to the maximum legal value. Similarly, if *incr* is too small, the system silently increases it to the minimum legal value. Because –1 is a legal successful return value, we need to clear `errno` before calling `nice` and check its value if `nice` returns –1. If the call to `nice` succeeds and the return value is –1, then `errno` will still be zero. If `errno` is nonzero, it means that the call to `nice` failed.

The `getpriority` function can be used to get the nice value for a process, just like the `nice` function. However, `getpriority` can also get the nice value for a group of related processes.

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

int getpriority(int which, id_t who);
```

Returns: nice value between –NZERO and NZERO–1 if OK, –1 on error

The *which* argument can take on one of three values: `PRIO_PROCESS` to indicate a process, `PRIO_PGRP` to indicate a process group, and `PRIO_USER` to indicate a user ID. The *which* argument controls how the *who* argument is interpreted and the *who* argument selects the process or processes of interest. If the *who* argument is 0, then it indicates the calling process, process group, or user (depending on the value of the *which* argument). When *which* is set to `PRIO_USER` and *who* is 0, the real user ID of the calling process is used. When the *which* argument applies to more than one process, the highest priority (lowest value) of all the applicable processes is returned.

The `setpriority` function can be used to set the priority of a process, a process group, or all the processes belonging to a particular user ID.

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

int setpriority(int which, id_t who, int value);
```

Returns: 0 if OK, –1 on error

The *which* and *who* arguments are the same as in the `getpriority` function. The *value* is added to `NZERO` and this becomes the new nice value.

> The `nice` system call originated with an early PDP-11 version of the Research UNIX System. The `getpriority` and `setpriority` functions originated with 4.2BSD.

The Single UNIX Specification leaves it up to the implementation whether the nice value is inherited by a child process after a `fork`. However, XSI-compliant systems are required to preserve the nice value across a call to `exec`.

> A child process inherits the nice value from its parent process in FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10.

Example

The program in Figure 8.30 measures the effect of adjusting the nice value of a process. Two processes run in parallel, each incrementing its own counter. The parent runs with the default nice value, and the child runs with an adjusted nice value as specified by the

optional command argument. After running for 10 seconds, both processes print the value of their counter and exit. By comparing the counter values for different nice values, we can get an idea how the nice value affects process scheduling.

```
#include "apue.h"
#include <errno.h>
#include <sys/time.h>

#if defined(MACOS)
#include <sys/syslimits.h>
#elif defined(SOLARIS)
#include <limits.h>
#elif defined(BSD)
#include <sys/param.h>
#endif

unsigned long long count;
struct timeval end;

void
checktime(char *str)
{
    struct timeval  tv;

    gettimeofday(&tv, NULL);
    if (tv.tv_sec >= end.tv_sec && tv.tv_usec >= end.tv_usec) {
        printf("%s count = %lld\n", str, count);
        exit(0);
    }
}

int
main(int argc, char *argv[])
{
    pid_t   pid;
    char    *s;
    int     nzero, ret;
    int     adj = 0;

    setbuf(stdout, NULL);
#if defined(NZERO)
    nzero = NZERO;
#elif defined(_SC_NZERO)
    nzero = sysconf(_SC_NZERO);
#else
#error NZERO undefined
#endif
    printf("NZERO = %d\n", nzero);
    if (argc == 2)
        adj = strtol(argv[1], NULL, 10);
    gettimeofday(&end, NULL);
    end.tv_sec += 10;   /* run for 10 seconds */

    if ((pid = fork()) < 0) {
```

```
        err_sys("fork failed");
    } else if (pid == 0) {  /* child */
        s = "child";
        printf("current nice value in child is %d, adjusting by %d\n",
          nice(0)+nzero, adj);
        errno = 0;
        if ((ret = nice(adj)) == -1 && errno != 0)
            err_sys("child set scheduling priority");
        printf("now child nice value is %d\n", ret+nzero);
    } else {        /* parent */
        s = "parent";
        printf("current nice value in parent is %d\n", nice(0)+nzero);
    }
    for(;;) {
        if (++count == 0)
            err_quit("%s counter wrap", s);
        checktime(s);
    }
}
```

Figure 8.30 Evaluate the effect of changing the nice value

We run the program twice: once with the default nice value, and once with the highest valid nice value (the lowest scheduling priority). We run this on a uniprocessor Linux system to show how the scheduler shares the CPU among processes with different nice values. With an otherwise idle system, a multiprocessor system (or a multicore CPU) would allow both processes to run without the need to share a CPU, and we wouldn't see much difference between two processes with different nice values.

```
$ ./a.out
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 0
now child nice value is 20
child count = 1859362
parent count = 1845338
$ ./a.out 20
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 20
now child nice value is 39
parent count = 3595709
child count = 52111
```

When both processes have the same nice value, the parent process gets 50.2% of the CPU and the child gets 49.8% of the CPU. Note that the two processes are effectively treated equally. The percentages aren't exactly equal, because process scheduling isn't exact, and because the child and parent perform different amounts of processing between the time that the end time is calculated and the time that the processing loop begins.

In contrast, when the child has the highest possible nice value (the lowest priority), we see that the parent gets 98.5% of the CPU, while the child gets only 1.5% of the CPU. These values will vary based on how the process scheduler uses the nice value, so a different UNIX system will produce different ratios. □

8.17 Process Times

In Section 1.10, we described three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the `times` function to obtain these values for itself and any terminated children.

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

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, –1 on error

This function fills in the `tms` structure pointed to by *buf*:

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value. For example, we call `times` and save the return value. At some later time, we call `times` again and subtract the earlier return value from the new return value. The difference is the wall clock time. (It is possible, though unlikely, for a long-running process to overflow the wall clock time; see Exercise 1.5.)

The two structure fields for child processes contain values only for children that we have waited for with one of the `wait` functions discussed earlier in this chapter.

All the `clock_t` values returned by this function are converted to seconds using the number of clock ticks per second—the `_SC_CLK_TCK` value returned by `sysconf` (Section 2.5.4).

> Most implementations provide the `getrusage`(2) function. This function returns the CPU times and 14 other values indicating resource usage. Historically, this function originated with the BSD operating system, so BSD-derived implementations generally support more of the fields than do other implementations.

Example

The program in Figure 8.31 executes each command-line argument as a shell command string, timing the command and printing the values from the `tms` structure.

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

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

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

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]);    /* once for each command-line arg */
    exit(0);
}

static void
do_cmd(char *cmd)       /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmsend;
    clock_t     start, end;
    int         status;

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1)   /* starting values */
        err_sys("times error");

    if ((status = system(cmd)) < 0)         /* execute command */
        err_sys("system() error");

    if ((end = times(&tmsend)) == -1)       /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
    static long     clktck = 0;

    if (clktck == 0)    /* fetch clock ticks per second first time */
        if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");

    printf("  real:  %7.2f\n", real / (double) clktck);
    printf("  user:  %7.2f\n",
      (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf("  sys:   %7.2f\n",
      (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf("  child user:  %7.2f\n",
```

```
      (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf("  child sys:   %7.2f\n",
      (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

Figure 8.31 Time and execute all command-line arguments

If we run this program, we get

```
$ ./a.out "sleep 5" "date" "man bash >/dev/null"

command: sleep 5
  real:     5.01
  user:     0.00
  sys:      0.00
  child user:     0.00
  child sys:      0.00
normal termination, exit status = 0

command: date
Sun Feb 26 18:39:23 EST 2012
  real:     0.00
  user:     0.00
  sys:      0.00
  child user:     0.00
  child sys:      0.00
normal termination, exit status = 0

command: man bash >/dev/null
  real:     1.46
  user:     0.00
  sys:      0.00
  child user:     1.32
  child sys:      0.07
normal termination, exit status = 0
```

In the first two commands, execution is fast enough to avoid registering any CPU time at the reported resolution. In the third command, however, we run a command that takes enough processing time to note that all the CPU time appears in the child process, which is where the shell and the command execute. □

8.18 Summary

A thorough understanding of the UNIX System's process control is essential for advanced programming. There are only a few functions to master: `fork`, the `exec` family, `_exit`, `wait`, and `waitpid`. These primitives are used in many applications. The `fork` function also gave us an opportunity to look at race conditions.

Our examination of the `system` function and process accounting gave us another look at all these process control functions. We also looked at another variation of the

15

Interprocess Communication

15.1 Introduction

In Chapter 8, we described the process control primitives and saw how to work with multiple processes. But the only way for these processes to exchange information is by passing open files across a `fork` or an `exec` or through the file system. We'll now describe other techniques for processes to communicate with one another: interprocess communication (IPC).

In the past, UNIX System IPC was a hodgepodge of various approaches, few of which were portable across all UNIX system implementations. Through the POSIX and The Open Group (formerly X/Open) standardization efforts, the situation has since improved, but differences still exist. Figure 15.1 summarizes the various forms of IPC that are supported by the four implementations discussed in this text.

Note that the Single UNIX Specification (the "SUS" column) allows an implementation to support full-duplex pipes, but requires only half-duplex pipes. An implementation that supports full-duplex pipes will still work with correctly written applications that assume that the underlying operating system supports only half-duplex pipes. We use "(full)" instead of a bullet to show implementations that support half-duplex pipes by using full-duplex pipes.

In Figure 15.1, we show a bullet where basic functionality is supported. For full-duplex pipes, if the feature can be provided through UNIX domain sockets (Section 17.2), we show "UDS" in the column. Some implementations support the feature with pipes and UNIX domain sockets, so these entries have both "UDS" and a bullet.

The IPC interfaces introduced as part of the real-time extensions to POSIX.1 were included as options in the Single UNIX Specification. In SUSv4, the semaphore interfaces were moved from an option to the base specification.

IPC type	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
half-duplex pipes	•	(full)	•	•	(full)
FIFOs	•	•	•	•	•
full-duplex pipes	allowed	•, UDS	UDS	UDS	•, UDS
named full-duplex pipes	obsolescent	UDS	UDS	UDS	•, UDS
XSI message queues	XSI	•	•	•	•
XSI semaphores	XSI	•	•	•	•
XSI shared memory	XSI	•	•	•	•
message queues (real-time)	MSG option	•	•		•
semaphores	•	•	•	•	•
shared memory (real-time)	SHM option	•	•	•	•
sockets	•	•	•	•	•
STREAMS	obsolescent				•

Figure 15.1 Summary of UNIX System IPC

Named full-duplex pipes are provided as mounted STREAMS-based pipes, but are marked obsolescent in the Single UNIX Specification.

> Although support for STREAMS on Linux is available in the "Linux Fast-STREAMS" package from the OpenSS7 project, the package hasn't been updated recently. The latest release of the package from 2008 claims to work with kernels up to Linux 2.6.26.

The first ten forms of IPC in Figure 15.1 are usually restricted to IPC between processes on the same host. The final two rows—sockets and STREAMS—are the only two forms that are generally supported for IPC between processes on different hosts.

We have divided the discussion of IPC into three chapters. In this chapter, we examine classical IPC: pipes, FIFOs, message queues, semaphores, and shared memory. In the next chapter, we take a look at network IPC using the sockets mechanism. In Chapter 17, we take a look at some advanced features of IPC.

15.2 Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls `fork`, and the pipe is used between the parent and the child.

We'll see that FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the `pipe` function.

```
#include <unistd.h>

int pipe(int fd[2]);
```

Returns: 0 if OK, –1 on error

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

> Originally in 4.3BSD and 4.4BSD, pipes were implemented using UNIX domain sockets. Even though UNIX domain sockets are full duplex by default, these operating systems hobbled the sockets used with pipes so that they operated in half-duplex mode only.
>
> POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

Two ways to picture a half-duplex pipe are shown in Figure 15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

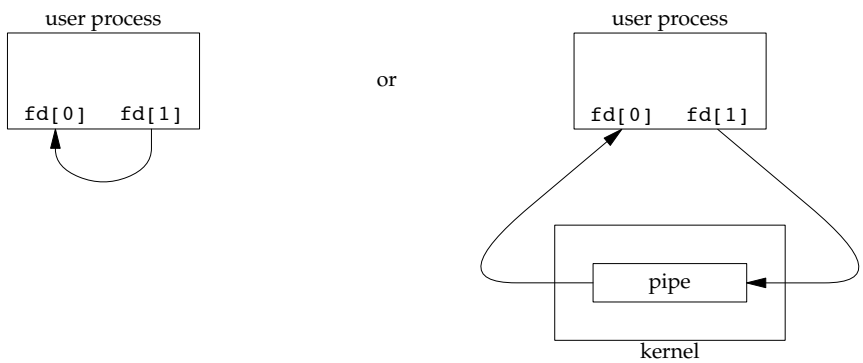

Figure 15.2 Two ways to view a half-duplex pipe

The `fstat` function (Section 4.2) returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

> POSIX.1 states that the `st_size` member of the `stat` structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe. This is, however, nonportable.

A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa. Figure 15.3 shows this scenario.

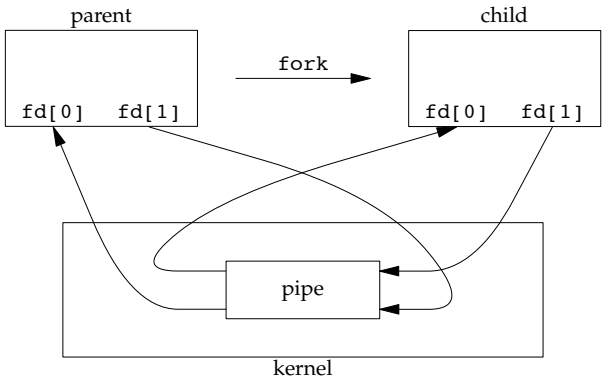

Figure 15.3 Half-duplex pipe after a `fork`

What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (`fd[0]`), and the child closes the write end (`fd[1]`). Figure 15.4 shows the resulting arrangement of descriptors.

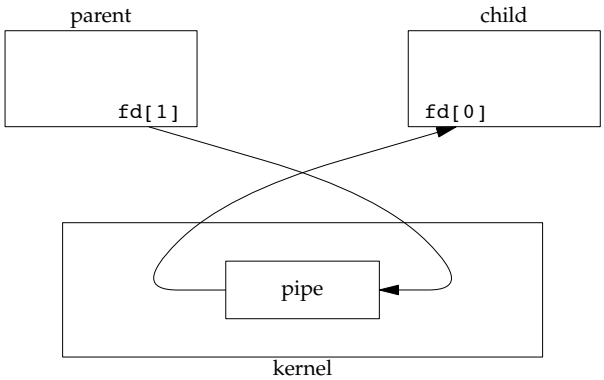

Figure 15.4 Pipe from parent to child

For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`.

When one end of a pipe is closed, two rules apply.

1. If we `read` from a pipe whose write end has been closed, `read` returns 0 to indicate an end of file after all the data has been read. (Technically, we should say that this end of file is not generated until there are no more writers for the pipe. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing. Normally, however, there is a single reader and a single writer for a pipe. When we get to FIFOs in the next section, we'll see that often there are multiple writers for a single FIFO.)

2. If we `write` to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, `write` returns –1 with `errno` set to `EPIPE`.

When we're writing to a pipe (or FIFO), the constant `PIPE_BUF` specifies the kernel's pipe buffer size. A `write` of `PIPE_BUF` bytes or less will not be interleaved with the `write`s from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we `write` more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers. We can determine the value of `PIPE_BUF` by using `pathconf` or `fpathconf` (recall Figure 2.12).

Example

Figure 15.5 shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Figure 15.5 Send data from parent to child over a pipe

Note that the pipe direction here matches the orientation shown in Figure 15.4. □

In the previous example, we called `read` and `write` directly on the pipe descriptors. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

Example

Consider a program that displays some output that it has created, one page at a time. Rather than reinvent the pagination done by several UNIX system utilities, we want to invoke the user's favorite pager. To avoid writing all the data to a temporary file and calling `system` to display that file, we want to pipe the output directly to the pager. To do this, we create a pipe, `fork` a child process, set up the child's standard input to be the read end of the pipe, and `exec` the user's pager program. Figure 15.6 shows how to do this. (This example takes a command-line argument to specify the name of a file to display. Often, a program of this type would already have the data to display to the terminal in memory.)

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

#define DEF_PAGER   "/bin/more"     /* default pager program */

int
main(int argc, char *argv[])
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    *pager, *argv0;
    char    line[MAXLINE];
    FILE    *fp;

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

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                                   /* parent */
        close(fd[0]);       /* close read end */

        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);   /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
```

```
        exit(0);
    } else {                                        /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

        /* get arguments for execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *)0) < 0)
            err_sys("execl error for %s", pager);
    }
    exit(0);
}
```

Figure 15.6 Copy file to pager program

Before calling `fork`, we create a pipe. After the `fork`, the parent closes its read end, and the child closes its write end. The child then calls `dup2` to have its standard input be the read end of the pipe. When the pager program is executed, its standard input will be the read end of the pipe.

When we duplicate one descriptor onto another (`fd[0]` onto standard input in the child), we have to be careful that the descriptor doesn't already have the desired value. If the descriptor already had the desired value and we called `dup2` and `close`, the single copy of the descriptor would be closed. (Recall the operation of `dup2` when its two arguments are equal, discussed in Section 3.12.) In this program, if standard input had not been opened by the shell, the `fopen` at the beginning of the program should have used descriptor 0, the lowest unused descriptor, so `fd[0]` should never equal standard input. Nevertheless, whenever we call `dup2` and `close` to duplicate one descriptor onto another, we'll always compare the descriptors first, as a defensive programming measure.

Note how we try to use the environment variable `PAGER` to obtain the name of the user's pager program. If this doesn't work, we use a default. This is a common usage of environment variables. □

Example

Recall the five functions `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` from Section 8.9. In Figure 10.24, we showed an implementation using signals. Figure 15.7 shows an implementation using pipes.

```
#include "apue.h"

static int  pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("write error");
}

void
WAIT_PARENT(void)
{
    char    c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'p')
        err_quit("WAIT_PARENT: incorrect data");
}

void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

Figure 15.7 Routines to let a parent and child synchronize

We create two pipes before the `fork`, as shown in Figure 15.8. The parent writes the character "p" across the top pipe when `TELL_CHILD` is called, and the child writes the character "c" across the bottom pipe when `TELL_PARENT` is called. The corresponding `WAIT_xxx` functions do a blocking `read` for the single character.

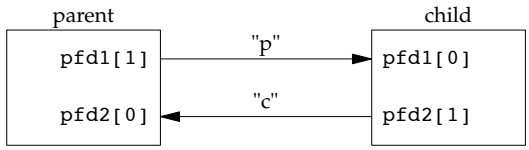

Figure 15.8 Using two pipes for parent–child synchronization

Note that each pipe has an extra reader, which doesn't matter. That is, in addition to the child reading from `pfd1[0]`, the parent has this end of the top pipe open for reading. This doesn't affect us, since the parent doesn't try to read from this pipe. □

15.3 `popen` and `pclose` Functions

Since a common operation is to create a pipe to another process to either read its output or send it input, the standard I/O library has historically provided the `popen` and `pclose` functions. These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, `fork`ing a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
                                        Returns: file pointer if OK, NULL on error

int pclose(FILE *fp);
                                Returns: termination status of cmdstring, or –1 on error
```

The function `popen` does a `fork` and `exec` to execute the *cmdstring* and returns a standard I/O file pointer. If *type* is `"r"`, the file pointer is connected to the standard output of *cmdstring* (Figure 15.9).

Figure 15.9 Result of `fp = popen(`*cmdstring*`, "r")`

If *type* is `"w"`, the file pointer is connected to the standard input of *cmdstring*, as shown in Figure 15.10.

Figure 15.10 Result of `fp = popen(`*cmdstring*`, "w")`

One way to remember the final argument to `popen` is to remember that, like `fopen`, the returned file pointer is readable if *type* is `"r"` or writable if *type* is `"w"`.

The `pclose` function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell. (We described the termination status in Section 8.6. The `system` function, described in Section 8.13, also returns the termination status.) If the shell cannot be executed, the termination status returned by `pclose` is as if the shell had executed `exit(127)`.

The *cmdstring* is executed by the Bourne shell, as in

```
sh -c cmdstring
```

This means that the shell expands any of its special characters in *cmdstring*. This allows us to say, for example,

```
fp = popen("ls *.c", "r");
```

or

```
fp = popen("cmd 2>&1", "r");
```

Example

Let's redo the program from Figure 15.6, using `popen`. This is shown in Figure 15.11.

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

#define PAGER   "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    FILE    *fpin, *fpout;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("popen error");

    /* copy argv[1] to pager */
```

```
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (ferror(fpin))
        err_sys("fgets error");
    if (pclose(fpout) == -1)
        err_sys("pclose error");

    exit(0);
}
```

Figure 15.11 Copy file to pager program using popen

Using popen reduces the amount of code we have to write.

The shell command `${PAGER:-more}` says to use the value of the shell variable PAGER if it is defined and non-null; otherwise, use the string more. □

Example — popen and pclose Functions

Figure 15.12 shows our version of popen and pclose.

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

/*
 * Pointer to array allocated at run-time.
 */
static pid_t    *childpid = NULL;

/*
 * From our open_max(), Figure 2.17.
 */
static int      maxfd;

FILE *
popen(const char *cmdstring, const char *type)
{
    int     i;
    int     pfd[2];
    pid_t   pid;
    FILE    *fp;

    /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;
        return(NULL);
    }
```

```
if (childpid == NULL) {     /* first time through */
    /* allocate zeroed out array for child pids */
    maxfd = open_max();
    if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
        return(NULL);
}

if (pipe(pfd) < 0)
    return(NULL);   /* errno set by pipe() */
if (pfd[0] >= maxfd || pfd[1] >= maxfd) {
    close(pfd[0]);
    close(pfd[1]);
    errno = EMFILE;
    return(NULL);
}

if ((pid = fork()) < 0) {
    return(NULL);   /* errno set by fork() */
} else if (pid == 0) {                          /* child */
    if (*type == 'r') {
        close(pfd[0]);
        if (pfd[1] != STDOUT_FILENO) {
            dup2(pfd[1], STDOUT_FILENO);
            close(pfd[1]);
        }
    } else {
        close(pfd[1]);
        if (pfd[0] != STDIN_FILENO) {
            dup2(pfd[0], STDIN_FILENO);
            close(pfd[0]);
        }
    }

    /* close all descriptors in childpid[] */
    for (i = 0; i < maxfd; i++)
        if (childpid[i] > 0)
            close(i);

    execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
    _exit(127);
}

/* parent continues... */
if (*type == 'r') {
    close(pfd[1]);
    if ((fp = fdopen(pfd[0], type)) == NULL)
        return(NULL);
} else {
    close(pfd[0]);
    if ((fp = fdopen(pfd[1], type)) == NULL)
        return(NULL);
```

```
    }

    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}

int
pclose(FILE *fp)
{
    int     fd, stat;
    pid_t   pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);     /* popen() has never been called */
    }

    fd = fileno(fp);
    if (fd >= maxfd) {
        errno = EINVAL;
        return(-1);     /* invalid file descriptor */
    }
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);     /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

Figure 15.12 The popen and pclose functions

Although the core of popen is similar to the code we've used earlier in this chapter, there are many details that we need to take care of. First, each time popen is called, we have to remember the process ID of the child that we create and either its file descriptor or FILE pointer. We choose to save the child's process ID in the array childpid, which we index by the file descriptor. This way, when pclose is called with the FILE pointer as its argument, we call the standard I/O function fileno to get the file descriptor and then have the child process ID for the call to waitpid. Since it's possible for a given process to call popen more than once, we dynamically allocate the childpid array (the first time popen is called), with room for as many children as there are file descriptors.

Note that our open_max function from Figure 2.17 can return a guess of the maximum number of open files if this value is indeterminate for the system. We need to be careful not to use a pipe file descriptor whose value is larger than (or equal to) what the open_max function returns. In popen, if the value returned by open_max happens to be too small, we close the pipe file descriptors, set errno to EMFILE to indicate too many file descriptors are open, and return –1. In pclose, if the file descriptor corresponding to the file pointer argument is larger than expected, we set errno to EINVAL and return –1.

Calling pipe and fork and then duplicating the appropriate descriptors for each process in the popen function is similar to what we did earlier in this chapter.

POSIX.1 requires that popen close any streams that are still open in the child from previous calls to popen. To do this, we go through the childpid array in the child, closing any descriptors that are still open.

What happens if the caller of pclose has established a signal handler for SIGCHLD? The call to waitpid from pclose would return an error of EINTR. Since the caller is allowed to catch this signal (or any other signal that might interrupt the call to waitpid), we simply call waitpid again if it is interrupted by a caught signal.

Note that if the application calls waitpid and obtains the exit status of the child created by popen, we will call waitpid when the application calls pclose, find that the child no longer exists, and return –1 with errno set to ECHILD. This is the behavior required by POSIX.1 in this situation.

> Some early versions of pclose returned an error of EINTR if a signal interrupted the wait. Also, some early versions of pclose blocked or ignored the signals SIGINT, SIGQUIT, and SIGHUP during the wait. This is not allowed by POSIX.1.
>
> □

Note that popen should never be called by a set-user-ID or set-group-ID program. When it executes the command, popen does the equivalent of

```
execl("/bin/sh", "sh", "-c", command, NULL);
```

which executes the shell and *command* with the environment inherited by the caller. A malicious user can manipulate the environment so that the shell executes commands other than those intended, with the elevated permissions granted by the set-ID file mode.

One thing that popen is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.

Example

Consider an application that writes a prompt to standard output and reads a line from standard input. With the popen function, we can interpose a program between the application and its input to transform the input. Figure 15.13 shows the arrangement of processes in this situation.

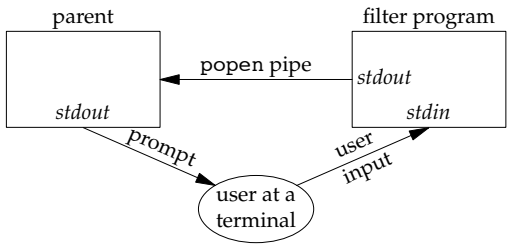

Figure 15.13 Transforming input using popen

The transformation could be pathname expansion, for example, or providing a history mechanism (remembering previously entered commands).

Figure 15.14 shows a simple filter to demonstrate this operation. The filter copies standard input to standard output, converting any uppercase character to lowercase. The reason we're careful to `fflush` standard output after writing a newline is discussed in the next section when we talk about coprocesses.

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

int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

Figure 15.14 Filter to convert uppercase characters to lowercase

We compile this filter into the executable file `myuclc`, which we then invoke from the program in Figure 15.15 using `popen`.

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

int
main(void)
{
    char    line[MAXLINE];
    FILE    *fpin;
```

```
    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error");
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

Figure 15.15 Invoke uppercase/lowercase filter to read commands

We need to call `fflush` after writing the prompt, because the standard output is normally line buffered, and the prompt does not contain a newline. □

15.4 Coprocesses

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a *coprocess* when the same program generates the filter's input and reads the filter's output.

The Korn shell provides coprocesses [Bolsky and Korn 1995]. The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as coprocesses. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe. Although the shell syntax required to initiate a coprocess and connect its input and output to other processes is quite contorted (see pp. 62–63 of Bolsky and Korn [1995] for all the details), coprocesses are also useful from a C program.

Whereas `popen` gives us a one-way pipe to the standard input or from the standard output of another process, with a coprocess we have two one-way pipes to the other process: one to its standard input and one from its standard output. We want to write to its standard input, let it operate on the data, and then read from its standard output.

Example

Let's look at coprocesses with an example. The process creates two pipes: one is the standard input of the coprocess and the other is the standard output of the coprocess. Figure 15.16 shows this arrangement.

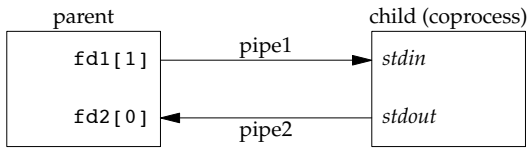

Figure 15.16 Driving a coprocess by writing its standard input and reading its standard output

The program in Figure 15.17 is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output. (Coprocesses usually do more interesting work than we illustrate here. This example is admittedly contrived so that we can study the plumbing needed to connect the processes.)

```
#include "apue.h"

int
main(void)
{
    int     n, int1, int2;
    char    line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;        /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

Figure 15.17 Simple filter to add two numbers

We compile this program and leave the executable in the file `add2`.

The program in Figure 15.18 invokes the `add2` coprocess after reading two numbers from its standard input. The value from the coprocess is written to its standard output.

```
#include "apue.h"

static void sig_pipe(int);      /* our signal handler */

int
```

```
main(void)
{
    int     n, fd1[2], fd2[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                           /* parent */
        close(fd1[0]);
        close(fd2[1]);

        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;    /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }

        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                        /* child */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
            if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd1[0]);
        }

        if (fd2[1] != STDOUT_FILENO) {
            if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("execl error");
```

```
    }
    exit(0);
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

Figure 15.18 Program to drive the add2 filter

Here, we create two pipes, with the parent and the child closing the ends they don't need. We have to use two pipes: one for the standard input of the coprocess and one for its standard output. The child then calls `dup2` to move the pipe descriptors onto its standard input and standard output, before calling `execl`.

If we compile and run the program in Figure 15.18, it works as expected. Furthermore, if we `kill` the `add2` coprocess while the program in Figure 15.18 is waiting for our input and then enter two numbers, the signal handler is invoked when the program writes to the pipe that has no reader. (See Exercise 15.4.) □

Example

In the coprocess `add2` (Figure 15.17), we purposely used low-level I/O (UNIX system calls): `read` and `write`. What happens if we rewrite this coprocess to use standard I/O? Figure 15.19 shows the new version.

```
#include "apue.h"

int
main(void)
{
    int     int1, int2;
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            if (printf("%d\n", int1 + int2) == EOF)
                err_sys("printf error");
        } else {
            if (printf("invalid args\n") == EOF)
                err_sys("printf error");
        }
    }
    exit(0);
}
```

Figure 15.19 Filter to add two numbers, using standard I/O

If we invoke this new coprocess from the program in Figure 15.18, it no longer works. The problem is the default standard I/O buffering. When the program in Figure 15.19 is invoked, the first `fgets` on the standard input causes the standard I/O library to allocate a buffer and choose the type of buffering. Since the standard input is a pipe, the standard I/O library defaults to fully buffered. The same thing happens with the standard output. While `add2` is blocked reading from its standard input, the program in Figure 15.18 is blocked reading from the pipe. We have a deadlock.

Here, we have control over the coprocess that's being run. We can change the program in Figure 15.19 by adding the following four lines before the `while` loop:

```
if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
```

These lines cause `fgets` to return when a line is available and cause `printf` to do an `fflush` when a newline is output (refer to Section 5.4 for the details on standard I/O buffering). Making these explicit calls to `setvbuf` fixes the program in Figure 15.19.

If we aren't able to modify the program that we're piping the output into, other techniques are required. For example, if we use `awk`(1) as a coprocess from our program (instead of the `add2` program), the following won't work:

```
#! /bin/awk -f
{ print $1 + $2 }
```

The reason this won't work is again the standard I/O buffering. But in this case, we cannot change the way `awk` works (unless we have the source code for it). We are unable to modify the executable of `awk` in any way to change the way the standard I/O buffering is handled.

The solution for this general problem is to make the coprocess being invoked (`awk` in this case) think that its standard input and standard output are connected to a terminal. That causes the standard I/O routines in the coprocess to line buffer these two I/O streams, similar to what we did with the explicit calls to `setvbuf` previously. We use pseudo terminals to do this in Chapter 19. □

15.5 FIFOs

FIFOs are sometimes called named pipes. Unnamed pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated processes can exchange data.

We saw in Chapter 4 that a FIFO is a type of file. One of the encodings of the `st_mode` member of the `stat` structure (Section 4.2) indicates that a file is a FIFO. We can test for this with the `S_ISFIFO` macro.

Creating a FIFO is similar to creating a file. Indeed, the *pathname* for a FIFO exists in the file system.

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

int mkfifo(const char *path, mode_t mode);

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

Both return: 0 if OK, –1 on error

The specification of the *mode* argument is the same as for the open function (Section 3.3). The rules for the user and group ownership of the new FIFO are the same as we described in Section 4.6.

The mkfifoat function is similar to the mkfifo function, except that it can be used to create a FIFO in a location relative to the directory represented by the *fd* file descriptor argument. Like the other *at functions, there are three cases:

1. If the *path* parameter specifies an absolute pathname, then the *fd* parameter is ignored and the mkfifoat function behaves like the mkfifo function.
2. If the *path* parameter specifies a relative pathname and the *fd* parameter is a valid file descriptor for an open directory, the pathname is evaluated relative to this directory.
3. If the *path* parameter specifies a relative pathname and the *fd* parameter has the special value AT_FDCWD, the pathname is evaluated starting in the current working directory, and mkfifoat behaves like mkfifo.

Once we have used mkfifo or mkfifoat to create a FIFO, we open it using open. Indeed, the normal file I/O functions (e.g., close, read, write, unlink) all work with FIFOs.

> Applications can create FIFOs with the mknod and mknodat functions. Because POSIX.1 originally didn't include mknod, the mkfifo function was invented specifically for POSIX.1. The mknod and mknodat functions are included in the XSI option in POSIX.1.
>
> POSIX.1 also includes support for the mkfifo(1) command. All four platforms discussed in this text provide this command. As a result, we can create a FIFO using a shell command and then access it with the normal shell I/O redirection.

When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

- In the normal case (without O_NONBLOCK), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.
- If O_NONBLOCK is specified, an open for read-only returns immediately. But an open for write-only returns –1 with errno set to ENXIO if no process has the FIFO open for reading.

As with a pipe, if we write to a FIFO that no process has open for reading, the signal SIGPIPE is generated. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.

It is common to have multiple writers for a given FIFO. This means that we have to worry about atomic writes if we don't want the writes from multiple processes to be

interleaved. As with pipes, the constant `PIPE_BUF` specifies the maximum amount of data that can be written atomically to a FIFO.

There are two uses for FIFOs.

1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers.

We discuss each of these uses with an example.

Example — Using FIFOs to Duplicate Output Streams

FIFOs can be used to duplicate an output stream in a series of shell commands. This prevents writing the data to an intermediate disk file (similar to using pipes to avoid intermediate disk files). But whereas pipes can be used only for linear connections between processes, a FIFO has a name, so it can be used for nonlinear connections.

Consider a procedure that needs to process a filtered input stream twice. Figure 15.20 shows this arrangement.

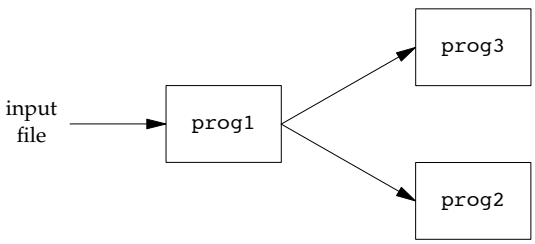

Figure 15.20 Procedure that processes a filtered input stream twice

With a FIFO and the UNIX program `tee`(1), we can accomplish this procedure without using a temporary file. (The `tee` program copies its standard input to both its standard output and the file named on its command line.)

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

We create the FIFO and then start `prog3` in the background, reading from the FIFO. We then start `prog1` and use `tee` to send its input to both the FIFO and `prog2`. Figure 15.21 shows the process arrangement. □

Example — Client–Server Communication Using a FIFO

Another use for FIFOs is to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known

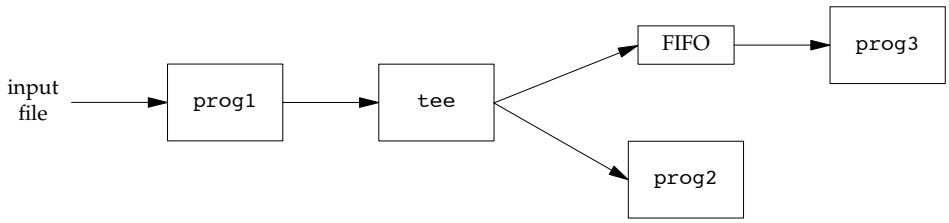

Figure 15.21 Using a FIFO and `tee` to send a stream to two different processes

FIFO that the server creates. (By "well-known," we mean that the pathname of the FIFO is known to all the clients that need to contact the server.) Figure 15.22 shows this arrangement.

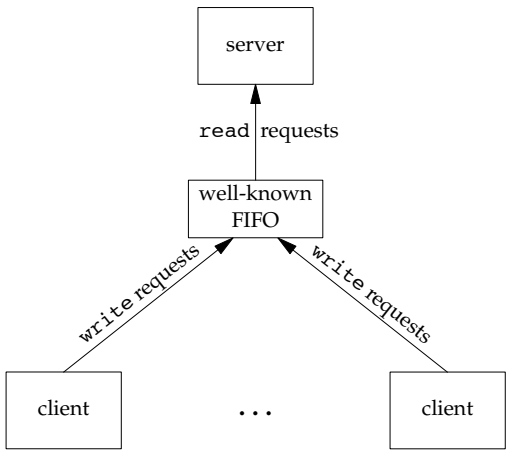

Figure 15.22 Clients sending requests to a server using a FIFO

Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than `PIPE_BUF` bytes in size. This prevents any interleaving of the client `write`s.

The problem in using FIFOs for this type of client–server communication is how to send replies back from the server to each client. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID. For example, the server can create a FIFO with the name `/tmp/serv1.XXXXX`, where `XXXXX` is replaced with the client's process ID. This arrangement is shown in Figure 15.23.

This arrangement works, although it is impossible for the server to tell whether a client crashes. A client crash leaves the client-specific FIFO in the file system. The

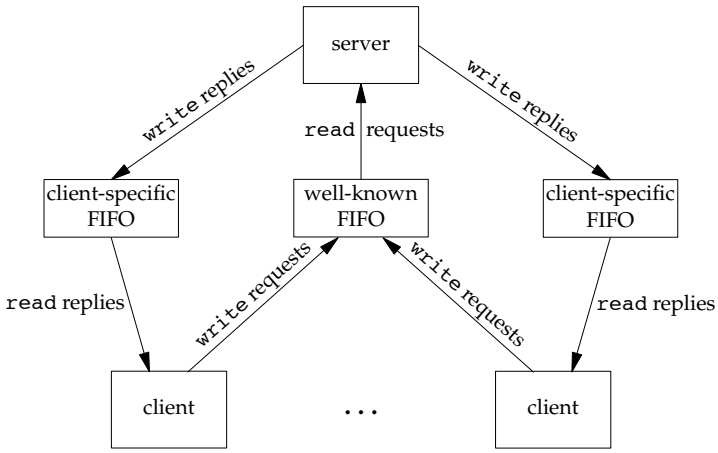

Figure 15.23 Client–server communication using FIFOs

server also must catch `SIGPIPE`, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

With the arrangement shown in Figure 15.23, if the server opens its well-known FIFO read-only (since it only `read`s from it) each time the number of clients goes from 1 to 0, the server will `read` an end of file on the FIFO. To prevent the server from having to handle this case, a common trick is just to have the server `open` its well-known FIFO for read–write. (See Exercise 15.10.) □

15.6 XSI IPC

The three types of IPC that we call XSI IPC—message queues, semaphores, and shared memory—have many similarities. In this section, we cover these similar features; in the following sections, we look at the specific functions for each of the three IPC types.

> The XSI IPC functions are based closely on the System V IPC functions. These three types of IPC originated in the 1970s in an internal AT&T version of the UNIX System called "Columbus UNIX." These IPC features were later added to System V. They are often criticized for inventing their own namespace instead of using the file system.

15.6.1 Identifiers and Keys

Each *IPC structure* (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer *identifier*. To send a message to or fetch a message from a message queue, for example, all we need know is the identifier for the queue. Unlike file descriptors, IPC identifiers are not small integers. Indeed, when a

given IPC structure is created and then removed, the identifier associated with that structure continually increases until it reaches the maximum positive value for an integer, and then wraps around to 0.

The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a *key* that acts as an external name.

Whenever an IPC structure is being created (by calling `msgget`, `semget`, or `shmget`), a key must be specified. The data type of this key is the primitive system data type `key_t`, which is often defined as a long integer in the header `<sys/types.h>`. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

1. The server can create a new IPC structure by specifying a key of `IPC_PRIVATE` and store the returned identifier somewhere (such as a file) for the client to obtain. The key `IPC_PRIVATE` guarantees that the server creates a new IPC structure. The disadvantage of this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.

   The `IPC_PRIVATE` key is also used in a parent–child relationship. The parent creates a new IPC structure specifying `IPC_PRIVATE`, and the resulting identifier is then available to the child after the `fork`. The child can pass the identifier to a new program as an argument to one of the `exec` functions.

2. The client and the server can agree on a key by defining the key in a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the `get` function (`msgget`, `semget`, or `shmget`) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.

3. The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function `ftok` to convert these two values into a key. This key is then used in step 2. The only service provided by `ftok` is a way of generating a key from a pathname and project ID.

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

key_t ftok(const char *path, int id);
```

Returns: key if OK, `(key_t)`–1 on error

The *path* argument must refer to an existing file. Only the lower 8 bits of *id* are used when generating the key.

The key created by `ftok` is usually formed by taking parts of the `st_dev` and `st_ino` fields in the `stat` structure (Section 4.2) corresponding to the given pathname and combining them with the project ID. If two pathnames refer to two different files,

then ftok usually returns two different keys for the two pathnames. However, because both i-node numbers and keys are often stored in long integers, information loss can occur when creating a key. This means that two different pathnames to different files can generate the same key if the same project ID is used.

The three get functions (msgget, semget, and shmget) all have two similar arguments: a *key* and an integer *flag*. A new IPC structure is created (normally by a server) if either *key* is IPC_PRIVATE or *key* is not currently associated with an IPC structure of the particular type and the IPC_CREAT bit of *flag* is specified. To reference an existing queue (normally done by a client), *key* must equal the key that was specified when the queue was created, and IPC_CREAT must not be specified.

Note that it's never possible to specify IPC_PRIVATE to reference an existing queue, since this special *key* value always creates a new queue. To reference an existing queue that was created with a *key* of IPC_PRIVATE, we must know the associated identifier and then use that identifier in the other IPC calls (such as msgsnd and msgrcv), bypassing the get function.

If we want to create a new IPC structure, making sure that we don't reference an existing one with the same identifier, we must specify a *flag* with both the IPC_CREAT and IPC_EXCL bits set. Doing this causes an error return of EEXIST if the IPC structure already exists. (This is similar to an open that specifies the O_CREAT and O_EXCL flags.)

15.6.2 Permission Structure

XSI IPC associates an ipc_perm structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
struct ipc_perm {
  uid_t  uid;  /* owner's effective user ID */
  gid_t  gid;  /* owner's effective group ID */
  uid_t  cuid; /* creator's effective user ID */
  gid_t  cgid; /* creator's effective group ID */
  mode_t mode; /* access modes */
  .
  .
  .
};
```

Each implementation includes additional members. See <sys/ipc.h> on your system for the complete definition.

All the fields are initialized when the IPC structure is created. At a later time, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling chown or chmod for a file.

The values in the mode field are similar to the values we saw in Figure 4.6, but there is nothing corresponding to execute permission for any of the IPC structures. Also, message queues and shared memory use the terms *read* and *write*, but semaphores use the terms *read* and *alter*. Figure 15.24 shows the six permissions for each form of IPC.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Figure 15.24 XSI IPC permissions

Some implementations define symbolic constants to represent each permission, but these constants are not standardized by the Single UNIX Specification.

15.6.3 Configuration Limits

All three forms of XSI IPC have built-in limits that we may encounter. Most of these limits can be changed by reconfiguring the kernel. We describe the limits when we describe each of the three forms of IPC.

> Each platform provides its own way to report and modify a particular limit. FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 provide the `sysctl` command to view and modify kernel configuration parameters. On Solaris 10, changes to kernel IPC limits are made with the `prctl` command.
>
> On Linux, you can display the IPC-related limits by running `ipcs -l`. On FreeBSD and Mac OS X, the equivalent command is `ipcs -T`. On Solaris, you can discover the tunable parameters by running `sysdef -i`.

15.6.4 Advantages and Disadvantages

A fundamental problem with XSI IPC is that the IPC structures are systemwide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling `msgrcv` or `msgctl`, by someone executing the `ipcrm`(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.

Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions we described in Chapters 3 and 4. Almost a dozen new system calls (`msgget`, `semop`, `shmat`, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an `ls` command, we can't remove them with the `rm` command, and we can't change their permissions with the `chmod` command. Instead, two new commands—`ipcs`(1) and `ipcrm`(1)—were added.

Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (`select` and `poll`) with them. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O. For example, we can't have a server wait for a message to be placed on one of two message queues without some form of busy–wait loop.

An overview of a transaction processing system built using System V IPC is given in Andrade, Carges, and Kovach [1989]. They claim that the namespace used by System V IPC (the identifiers) is an advantage, not a problem as we said earlier, because using identifiers allows a process to send a message to a message queue with a single function call (`msgsnd`), whereas other forms of IPC normally require an `open`, `write`, and `close`. This argument is false. Clients still have to obtain the identifier for the server's queue somehow, to avoid using a key and calling `msgget`. The identifier assigned to a particular queue depends on how many other message queues exist when the queue is created and how many times the table in the kernel assigned to the new queue has been used since the kernel was bootstrapped. This is a dynamic value that can't be guessed or stored in a header. As we mentioned in Section 15.6.1, minimally a server has to write the assigned queue identifier to a file for its clients to read.

Other advantages listed by these authors for message queues are that they're reliable, flow controlled, and record oriented, and that they can be processed in other than first-in, first-out order. Figure 15.25 compares some of the features of these various forms of IPC.

IPC type	Connectionless?	Reliable?	Flow control?	Records?	Message types or priorities?
message queues	no	yes	yes	yes	yes
STREAMS	no	yes	yes	yes	yes
UNIX domain stream socket	no	yes	yes	no	no
UNIX domain datagram socket	yes	yes	no	yes	no
FIFOs (non-STREAMS)	no	yes	yes	no	no

Figure 15.25 Comparison of features of various forms of IPC

(We describe stream and datagram sockets in Chapter 16. We describe UNIX domain sockets in Section 17.2.) By "connectionless," we mean the ability to send a message without having to call some form of an open function first. As described previously, we don't consider message queues connectionless, since some technique is required to obtain the identifier for a queue. Since all these forms of IPC are restricted to a single host, all are reliable. When the messages are sent across a network, the possibility of messages being lost becomes a concern. "Flow control" means that the sender is put to sleep if there is a shortage of system resources (buffers) or if the receiver can't accept any more messages. When the flow control condition subsides (i.e., when there is room in the queue), the sender should automatically be awakened.

One feature that we don't show in Figure 15.25 is whether the IPC facility can automatically create a unique connection to a server for each client. We'll see in Chapter 17 that UNIX stream sockets provide this capability. The next three sections describe each of the three forms of XSI IPC in detail.

15.7 Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a *queue* and its identifier a *queue ID*.

> The Single UNIX Specification message-passing option includes an alternative IPC message queue interface derived from the POSIX real-time extensions. We do not discuss it in this text.

A new queue is created or an existing queue opened by `msgget`. New messages are added to the end of a queue by `msgsnd`. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to `msgsnd` when the message is added to a queue. Messages are fetched from a queue by `msgrcv`. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following `msqid_ds` structure associated with it:

```
struct msqid_ds {
  struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
  msgqnum_t        msg_qnum;     /* # of messages on queue */
  msglen_t         msg_qbytes;   /* max # of bytes on queue */
  pid_t            msg_lspid;    /* pid of last msgsnd() */
  pid_t            msg_lrpid;    /* pid of last msgrcv() */
  time_t           msg_stime;    /* last-msgsnd() time */
  time_t           msg_rtime;    /* last-msgrcv() time */
  time_t           msg_ctime;    /* last-change time */
  .
  .
  .
};
```

This structure defines the current status of the queue. The members shown are the ones defined by the Single UNIX Specification. Implementations include additional fields not covered by the standard.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
size in bytes of largest message we can send	16,384	8,192	16,384	derived
maximum size in bytes of a particular queue (i.e., the sum of all the sizes of messages on the queue)	2,048	16,384	2,048	65,536
maximum number of messages queues, systemwide	40	derived	40	128
maximum number of messages, systemwide	40	derived	40	8,192

Figure 15.26 System limits that affect message queues

Figure 15.26 lists the system limits that affect message queues. We show "derived" where a limit is derived from other limits. On Linux, for example, the maximum number of messages is based on the maximum number of queues and the maximum amount of data allowed on the queues. The maximum number of queues, in turn, is based on the amount of RAM installed in the system. Note that the queue maximum byte size limit further limits the maximum size of a message to be placed on a queue.

The first function normally called is `msgget` to either open an existing queue or create a new queue.

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

int msgget(key_t key, int flag);
```

Returns: message queue ID if OK, –1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and discussed whether a new queue is created or an existing queue is referenced. When a new queue is created, the following members of the `msqid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are all set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

On success, `msgget` returns the non-negative queue ID. This value is then used with the other three message queue functions.

The `msgctl` function performs various operations on a queue. This function and the related functions for semaphores and shared memory (`semctl` and `shmctl`) are the `ioctl`-like functions for XSI IPC (i.e., the garbage-can functions).

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

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Returns: 0 if OK, –1 on error

The *cmd* argument specifies the command to be performed on the queue specified by *msqid*.

`IPC_STAT` Fetch the `msqid_ds` structure for this queue, storing it in the structure pointed to by *buf*.

`IPC_SET` Copy the following fields from the structure pointed to by *buf* to the `msqid_ds` structure associated with this queue: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes`. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges. Only the superuser can increase the value of `msg_qbytes`.

`IPC_RMID` Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of `EIDRM` on its next attempted operation on the queue. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges.

We'll see that these three commands (IPC_STAT, IPC_SET, and IPC_RMID) are also provided for semaphores and shared memory.

Data is placed onto a message queue by calling msgsnd.

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

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Returns: 0 if OK, –1 on error

As we mentioned earlier, each message is composed of a positive long integer type field, a non-negative length (*nbytes*), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The *ptr* argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if *nbytes* is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg {
  long  mtype;      /* positive message type */
  char  mtext[512]; /* message data, of length nbytes */
};
```

The *ptr* argument is then a pointer to a mymesg structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.

> Some platforms support both 32-bit and 64-bit environments. This affects the size of long integers and pointers. For example, on 64-bit SPARC systems, Solaris allows both 32-bit and 64-bit applications to coexist. If a 32-bit application were to exchange this structure over a pipe or a socket with a 64-bit application, problems would arise, because the size of a long integer is 4 bytes in a 32-bit application, but 8 bytes in a 64-bit application. This means that a 32-bit application will expect that the mtext field will start 4 bytes after the start of the structure, whereas a 64-bit application will expect the mtext field to start 8 bytes after the start of the structure. In this situation, part of the 64-bit application's mtype field will appear as part of the mtext field to the 32-bit application, and the first 4 bytes in the 32-bit application's mtext field will be interpreted as a part of the mtype field by the 64-bit application.
>
> This problem doesn't happen with XSI message queues, however. Solaris implements the 32-bit version of the IPC system calls with different entry points than the 64-bit version of the IPC system calls. The system calls know how to deal with a 32-bit application communicating with a 64-bit application, and treat the type field specially to avoid it interfering with the data portion of the message. The only potential problem is a loss of information when a 64-bit application sends a message with a value in the 8-byte type field that is larger than will fit in a 32-bit application's 4-byte type field. In this case, the 32-bit application will see a truncated type value.

A *flag* value of IPC_NOWAIT can be specified. This is similar to the nonblocking I/O flag for file I/O (Section 14.2). If the message queue is full (either the total number of messages on the queue equals the system limit, or the total number of bytes on the queue equals the system limit), specifying IPC_NOWAIT causes msgsnd to return immediately with an error of EAGAIN. If IPC_NOWAIT is not specified, we are blocked until there is room for the message, the queue is removed from the system, or a signal is caught and the signal handler returns. In the second case, an error of EIDRM is returned ("identifier removed"); in the last case, the error returned is EINTR.

Note how ungracefully the removal of a message queue is handled. Since a reference count is not maintained with each message queue (as there is for open files), the removal of a queue simply generates errors on the next queue operation by processes still using the queue. Semaphores handle this removal in the same fashion. In contrast, when a file is removed, the file's contents are not deleted until the last open descriptor for the file is closed.

When `msgsnd` returns successfully, the `msqid_ds` structure associated with the message queue is updated to indicate the process ID that made the call (`msg_lspid`), the time that the call was made (`msg_stime`), and that one more message is on the queue (`msg_qnum`).

Messages are retrieved from a queue by `msgrcv`.

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

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Returns: size of data portion of message if OK, –1 on error

As with `msgsnd`, the *ptr* argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data. *nbytes* specifies the size of the data buffer. If the returned message is larger than *nbytes* and the `MSG_NOERROR` bit in *flag* is set, the message is truncated. (In this case, no notification is given to us that the message was truncated, and the remainder of the message is discarded.) If the message is too big and this *flag* value is not specified, an error of `E2BIG` is returned instead (and the message stays on the queue).

The *type* argument lets us specify which message we want.

type == 0 The first message on the queue is returned.

type > 0 The first message on the queue whose message type equals *type* is returned.

type < 0 The first message on the queue whose message type is the lowest value less than or equal to the absolute value of *type* is returned.

A nonzero *type* is used to read the messages in an order other than first in, first out. For example, the *type* could be a priority value if the application assigns priorities to the messages. Another use of this field is to contain the process ID of the client if a single message queue is being used by multiple clients and a single server (as long as a process ID fits in a long integer).

We can specify a *flag* value of `IPC_NOWAIT` to make the operation nonblocking, causing `msgrcv` to return –1 with `errno` set to `ENOMSG` if a message of the specified type is not available. If `IPC_NOWAIT` is not specified, the operation blocks until a message of the specified type is available, the queue is removed from the system (–1 is returned with `errno` set to `EIDRM`), or a signal is caught and the signal handler returns (causing `msgrcv` to return –1 with `errno` set to `EINTR`).

When `msgrcv` succeeds, the kernel updates the `msqid_ds` structure associated with the message queue to indicate the caller's process ID (`msg_lrpid`), the time of the call (`msg_rtime`), and that one less message is on the queue (`msg_qnum`).

Example — Timing Comparison of Message Queues and Full-Duplex Pipes

If we need a bidirectional flow of data between a client and a server, we can use either message queues or full-duplex pipes. (Recall from Figure 15.1 that full-duplex pipes are available through the UNIX domain sockets mechanism [Section 17.2], although some platforms provide a full-duplex pipe mechanism through the `pipe` function.)

Figure 15.27 shows a timing comparison of three of these techniques on Solaris: message queues, full-duplex (STREAMS) pipes, and UNIX domain sockets. The tests consisted of a program that created the IPC channel, called `fork`, and then sent about 200 megabytes of data from the parent to the child. The data was sent using 100,000 calls to `msgsnd`, with a message length of 2,000 bytes for the message queue, and 100,000 calls to `write`, with a length of 2,000 bytes for the full-duplex pipe and UNIX domain socket. The times are all in seconds.

Operation	User	System	Clock
message queue	0.58	4.16	5.09
full-duplex pipe	0.61	4.30	5.24
UNIX domain socket	0.59	5.58	7.49

Figure 15.27 Timing comparison of IPC alternatives on Solaris

These numbers show us that message queues, originally implemented to provide higher-than-normal-speed IPC, are no longer that much faster than other forms of IPC. (When message queues were implemented, the only other form of IPC available was half-duplex pipes.) When we consider the problems in using message queues (Section 15.6.4), we come to the conclusion that we shouldn't use them for new applications. □

15.8 Semaphores

A semaphore isn't a form of IPC similar to the others that we've described (pipes, FIFOs, and message queues). A semaphore is a counter used to provide access to a shared data object for multiple processes.

> The Single UNIX Specification includes an alternative set of semaphore interfaces that were originally part of its real-time extensions. We discuss these interfaces in Section 15.10.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.

A common form of semaphore is called a *binary semaphore*. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (`semget`) is independent of its initialization (`semctl`). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The *undo* feature that we describe later is supposed to handle this.

The kernel maintains a `semid_ds` structure for each semaphore set:

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
  unsigned short   sem_nsems; /* # of semaphores in set */
  time_t           sem_otime; /* last-semop() time */
  time_t           sem_ctime; /* last-change time */
  :
};
```

The Single UNIX Specification defines the fields shown, but implementations can define additional members in the `semid_ds` structure.

Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct {
  unsigned short  semval;   /* semaphore value, always >= 0 */
  pid_t           sempid;   /* pid for last operation */
  unsigned short  semncnt;  /* # processes awaiting semval>curval */
  unsigned short  semzcnt;  /* # processes awaiting semval==0 */
  :
};
```

Figure 15.28 lists the system limits that affect semaphore sets.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
maximum value of any semaphore	32,767	32,767	32,767	65,535
maximum value of any semaphore's adjust-on-exit value	16,384	32,767	16,384	32,767
maximum number of semaphore sets, systemwide	10	128	87,381	128
maximum number of semaphores, systemwide	60	32,000	87,381	derived
maximum number of semaphores per semaphore set	60	250	87,381	512
maximum number of undo structures, systemwide	30	32,000	87,381	derived
maximum number of undo entries per undo structures	10	unlimited	10	derived
maximum number of operations per `semop` call	100	32	5	512

Figure 15.28 System limits that affect semaphores

When we want to use XSI semaphores, we first need to obtain a semaphore ID by calling the `semget` function.

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

int semget(key_t key, int nsems, int flag);
```

Returns: semaphore ID if OK, –1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and discussed whether a new set is created or an existing set is referenced. When a new set is created, the following members of the `semid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `sem_otime` is set to 0.
- `sem_ctime` is set to the current time.
- `sem_nsems` is set to *nsems*.

The number of semaphores in the set is *nsems*. If a new set is being created (typically by the server), we must specify *nsems*. If we are referencing an existing set (a client), we can specify *nsems* as 0.

The `semctl` function is the catchall for various semaphore operations.

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

int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
```

Returns: (see following)

The fourth argument is optional, depending on the command requested, and if present, is of type `semun`, a union of various command-specific arguments:

```
union semun {
  int               val;   /* for SETVAL */
  struct semid_ds  *buf;   /* for IPC_STAT and IPC_SET */
  unsigned short   *array; /* for GETALL and SETALL */
};
```

Note that the optional argument is the actual union, not a pointer to the union.

> Usually our application must define the `semun` union. However, on FreeBSD 8.0, this is defined for us in `<sys/sem.h>`.

The *cmd* argument specifies one of the following ten commands to be performed on the set specified by *semid*. The five commands that refer to one particular semaphore value use *semnum* to specify one member of the set. The value of *semnum* is between 0 and *nsems – 1*, inclusive.

`IPC_STAT` Fetch the `semid_ds` structure for this set, storing it in the structure pointed to by *arg.buf*.

`IPC_SET` Set the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` fields from the structure pointed to by *arg.buf* in the `semid_ds` structure associated with this set. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.

`IPC_RMID` Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of `EIDRM` on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.

`GETVAL` Return the value of `semval` for the member *semnum*.

`SETVAL` Set the value of `semval` for the member *semnum*. The value is specified by *arg.val*.

`GETPID` Return the value of `sempid` for the member *semnum*.

`GETNCNT` Return the value of `semncnt` for the member *semnum*.

`GETZCNT` Return the value of `semzcnt` for the member *semnum*.

`GETALL` Fetch all the semaphore values in the set. These values are stored in the array pointed to by *arg.array*.

`SETALL` Set all the semaphore values in the set to the values pointed to by *arg.array*.

For all the `GET` commands other than `GETALL`, the function returns the corresponding value. For the remaining commands, the return value is 0 if the call succeeds. On error, the `semctl` function sets `errno` and returns –1.

The function `semop` atomically performs an array of operations on a semaphore set.

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

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, –1 on error

The *semoparray* argument is a pointer to an array of semaphore operations, represented by `sembuf` structures:

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

The *nops* argument specifies the number of operations (elements) in the array.

The operation on each member of the set is specified by the corresponding `sem_op` value. This value can be negative, 0, or positive. (In the following discussion, we refer to the "undo" flag for a semaphore. This flag corresponds to the `SEM_UNDO` bit in the corresponding `sem_flg` member.)

1. The easiest case is when `sem_op` is positive. This case corresponds to the returning of resources by the process. The value of `sem_op` is added to the semaphore's value. If the undo flag is specified, `sem_op` is also subtracted from the semaphore's adjustment value for this process.
2. If `sem_op` is negative, we want to obtain resources that the semaphore controls.

   If the semaphore's value is greater than or equal to the absolute value of `sem_op` (the resources are available), the absolute value of `sem_op` is subtracted from the semaphore's value. This guarantees the resulting semaphore value is greater than or equal to 0. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.

   If the semaphore's value is less than the absolute value of `sem_op` (the resources are not available), the following conditions apply.

   a. If `IPC_NOWAIT` is specified, `semop` returns with an error of `EAGAIN`.
   b. If `IPC_NOWAIT` is not specified, the `semncnt` value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.
      i. The semaphore's value becomes greater than or equal to the absolute value of `sem_op` (i.e., some other process has released some resources). The value of `semncnt` for this semaphore is decremented (since the calling process is done waiting), and the absolute value of `sem_op` is subtracted from the semaphore's value. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.
      ii. The semaphore is removed from the system. In this case, the function returns an error of `EIDRM`.
      iii. A signal is caught by the process, and the signal handler returns. In this case, the value of `semncnt` for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of `EINTR`.
3. If `sem_op` is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.

If the semaphore's value is currently 0, the function returns immediately.

If the semaphore's value is nonzero, the following conditions apply.

a. If IPC_NOWAIT is specified, return is made with an error of EAGAIN.

b. If IPC_NOWAIT is not specified, the semzcnt value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.

   i. The semaphore's value becomes 0. The value of semzcnt for this semaphore is decremented (since the calling process is done waiting).

   ii. The semaphore is removed from the system. In this case, the function returns an error of EIDRM.

   iii. A signal is caught by the process, and the signal handler returns. In this case, the value of semzcnt for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of EINTR.

The semop function operates atomically; it does either all the operations in the array or none of them.

Semaphore Adjustment on exit

As we mentioned earlier, it is a problem if a process terminates while it has resources allocated through a semaphore. Whenever we specify the SEM_UNDO flag for a semaphore operation and we allocate resources (a sem_op value less than 0), the kernel remembers how many resources we allocated from that particular semaphore (the absolute value of sem_op). When the process terminates, either voluntarily or involuntarily, the kernel checks whether the process has any outstanding semaphore adjustments and, if so, applies the adjustment to the corresponding semaphore.

If we set the value of a semaphore using semctl, with either the SETVAL or SETALL commands, the adjustment value for that semaphore in all processes is set to 0.

Example — Timing Comparison of Semaphores, Record Locking, and Mutexes

If we are sharing a single resource among multiple processes, we can use one of three techniques to coordinate access. We can use a a semaphore, record locking, or a mutex that is mapped into the address spaces of both processes. It's interesting to compare the timing differences between the three techniques.

With a semaphore, we create a semaphore set consisting of a single member and initialize the semaphore's value to 1. To allocate the resource, we call semop with a sem_op of –1; to release the resource, we perform a sem_op of +1. We also specify SEM_UNDO with each operation, to handle the case of a process that terminates without releasing its resource.

With record locking, we create an empty file and use the first byte of the file (which need not exist) as the lock byte. To allocate the resource, we obtain a write lock on the

byte; to release it, we unlock the byte. The record locking properties guarantee that if a process terminates while holding a lock, the kernel automatically releases the lock.

To use a mutex, we need both processes to map the same file into their address spaces and initialize a mutex at the same offset in the file using the `PTHREAD_PROCESS_SHARED` mutex attribute. To allocate the resource, we lock the mutex; to release the resource, we unlock the mutex. If a process terminates without releasing the mutex, recovery is difficult unless we use a robust mutex (recall the `pthread_mutex_consistent` function discussed in Section 12.4.1).

Figure 15.29 shows the time required to perform these three locking techniques on Linux. In each case, the resource was allocated and then released 1,000,000 times. This was done simultaneously by three different processes. The times in Figure 15.29 are the totals in seconds for all three processes.

Operation	User	System	Clock
semaphores with undo	0.50	6.08	7.55
advisory record locking	0.51	9.06	4.38
mutex in shared memory	0.21	0.40	0.25

Figure 15.29 Timing comparison of locking alternatives on Linux

On Linux, record locking is faster than semaphores, but mutexes in shared memory outperform both semaphores and record locking. If we're locking a single resource and don't need all the fancy features of XSI semaphores, record locking is preferred over semaphores. The reasons are that it is much simpler to use, it is faster (on this platform), and the system takes care of any lingering locks when a process terminates. Even though using a mutex in shared memory is the fastest option on this platform, we still prefer to use record locking, unless performance is the primary concern. There are two reasons for this. First, recovery from process termination is more difficult using a mutex in memory shared among multiple processes. Second, the *process-shared* mutex attribute isn't universally supported yet. In older versions of the Single UNIX Specification, it was optional. Although it is still optional in SUSv4, it is now required by all XSI-conforming implementations.

> Of the four platforms covered in this text, only Linux 3.2.0 and Solaris 10 currently support the *process-shared* mutex attribute.
>
> □

15.9 Shared Memory

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server. The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access. (But as we saw at the end of the previous section, record locking or mutexes can also be used.)

The Single UNIX Specification shared memory objects option includes alternative interfaces, originally real-time extensions, to access shared memory. We don't discuss them in this text.

We've already seen one form of shared memory when multiple processes map the same file into their address spaces. The XSI shared memory differs from memory-mapped files in that there is no associated file. The XSI shared memory segments are anonymous segments of memory.

The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Section 15.6.2 */
  size_t           shm_segsz;   /* size of segment in bytes */
  pid_t            shm_lpid;    /* pid of last shmop() */
  pid_t            shm_cpid;    /* pid of creator */
  shmatt_t         shm_nattch;  /* number of current attaches */
  time_t           shm_atime;   /* last-attach time */
  time_t           shm_dtime;   /* last-detach time */
  time_t           shm_ctime;   /* last-change time */
  .
  .
  .
};
```

(Implementations add other structure members to support shared memory segments.)

The type `shmatt_t` is defined to be an unsigned integer at least as large as an `unsigned short`. Figure 15.30 lists the system limits that affect shared memory.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
maximum size in bytes of a shared memory segment	33,554,432	32,768	4,194,304	derived
minimum size in bytes of a shared memory segment	1	1	1	1
maximum number of shared memory segments, systemwide	192	4,096	32	128
maximum number of shared memory segments, per process	128	4,096	8	128

Figure 15.30 System limits that affect shared memory

The first function called is usually `shmget`, to obtain a shared memory identifier.

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

int shmget(key_t key, size_t size, int flag);
```

Returns: shared memory ID if OK, –1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and whether a new segment is created or an existing segment is referenced. When a new segment is created, the following members of the `shmid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.

- `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are all set to 0.
- `shm_ctime` is set to the current time.
- `shm_segsz` is set to the *size* requested.

The *size* parameter is the size of the shared memory segment in bytes. Implementations will usually round up this size to a multiple of the system's page size, but if an application specifies *size* as a value other than an integral multiple of the system's page size, the remainder of the last page will be unavailable for use. If a new segment is being created (typically by the server), we must specify its *size*. If we are referencing an existing segment (a client), we can specify *size* as 0. When a new segment is created, the contents of the segment are initialized with zeros.

The `shmctl` function is the catchall for various shared memory operations.

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Returns: 0 if OK, –1 on error

The *cmd* argument specifies one of the following five commands to be performed, on the segment specified by *shmid*.

`IPC_STAT` Fetch the `shmid_ds` structure for this segment, storing it in the structure pointed to by *buf*.

`IPC_SET` Set the following three fields from the structure pointed to by *buf* in the `shmid_ds` structure associated with this shared memory segment: `shm_perm.uid`, `shm_perm.gid`, and `shm_perm.mode`. This command can be executed only by a process whose effective user ID equals `shm_perm.cuid` or `shm_perm.uid` or by a process with superuser privileges.

`IPC_RMID` Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the `shm_nattch` field in the `shmid_ds` structure), the segment is not removed until the last process using the segment terminates or detaches it. Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that `shmat` can no longer attach the segment. This command can be executed only by a process whose effective user ID equals `shm_perm.cuid` or `shm_perm.uid` or by a process with superuser privileges.

Two additional commands are provided by Linux and Solaris, but are not part of the Single UNIX Specification.

`SHM_LOCK` Lock the shared memory segment in memory. This command can be executed only by the superuser.

`SHM_UNLOCK` Unlock the shared memory segment. This command can be executed only by the superuser.

Once a shared memory segment has been created, a process attaches it to its address space by calling `shmat`.

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

void *shmat(int shmid, const void *addr, int flag);
```

Returns: pointer to shared memory segment if OK, –1 on error

The address in the calling process at which the segment is attached depends on the *addr* argument and whether the `SHM_RND` bit is specified in *flag*.

- If *addr* is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.
- If *addr* is nonzero and `SHM_RND` is not specified, the segment is attached at the address given by *addr*.
- If *addr* is nonzero and `SHM_RND` is specified, the segment is attached at the address given by (*addr* – (*addr* modulus `SHMLBA`)). The `SHM_RND` command stands for "round." `SHMLBA` stands for "low boundary address multiple" and is always a power of 2. What the arithmetic does is round the address down to the next multiple of `SHMLBA`.

Unless we plan to run the application on only a single type of hardware (which is highly unlikely today), we should not specify the address where the segment is to be attached. Instead, we should specify an *addr* of 0 and let the system choose the address.

If the `SHM_RDONLY` bit is specified in *flag*, the segment is attached as read-only. Otherwise, the segment is attached as read–write.

The value returned by `shmat` is the address at which the segment is attached, or –1 if an error occurred. If `shmat` succeeds, the kernel will increment the `shm_nattch` counter in the `shmid_ds` structure associated with the shared memory segment.

When we're done with a shared memory segment, we call `shmdt` to detach it. Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process (often a server) specifically removes it by calling `shmctl` with a command of `IPC_RMID`.

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

int shmdt(const void *addr);
```

Returns: 0 if OK, –1 on error

The *addr* argument is the value that was returned by a previous call to `shmat`. If successful, `shmdt` will decrement the `shm_nattch` counter in the associated `shmid_ds` structure.

Example

Where a kernel places shared memory segments that are attached with an address of 0 is highly system dependent. Figure 15.31 shows a program that prints some information on where one particular system places various types of data.

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

#define ARRAY_SIZE  40000
#define MALLOC_SIZE 100000
#define SHM_SIZE    100000
#define SHM_MODE    0600    /* user read/write */

char    array[ARRAY_SIZE];  /* uninitialized data = bss */

int
main(void)
{
    int     shmid;
    char    *ptr, *shmptr;

    printf("array[] from %p to %p\n", (void *)&array[0],
      (void *)&array[ARRAY_SIZE]);
    printf("stack around %p\n", (void *)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %p to %p\n", (void *)ptr,
      (void *)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("shmat error");
    printf("shared memory attached from %p to %p\n", (void *)shmptr,
      (void *)shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

Figure 15.31 Print where various types of data are stored

Running this program on a 64-bit Intel-based Linux system gives us the following output:

```
$ ./a.out
array[] from 0x6020c0 to 0x60bd00
stack around 0x7fff957b146c
malloced from 0x9e3010 to 0x9fb6b0
shared memory attached from 0x7fba578ab000 to 0x7fba578c36a0
```

Figure 15.32 shows a picture of this, similar to what we said was a typical memory layout in Figure 7.6. Note that the shared memory segment is placed well below the stack. □

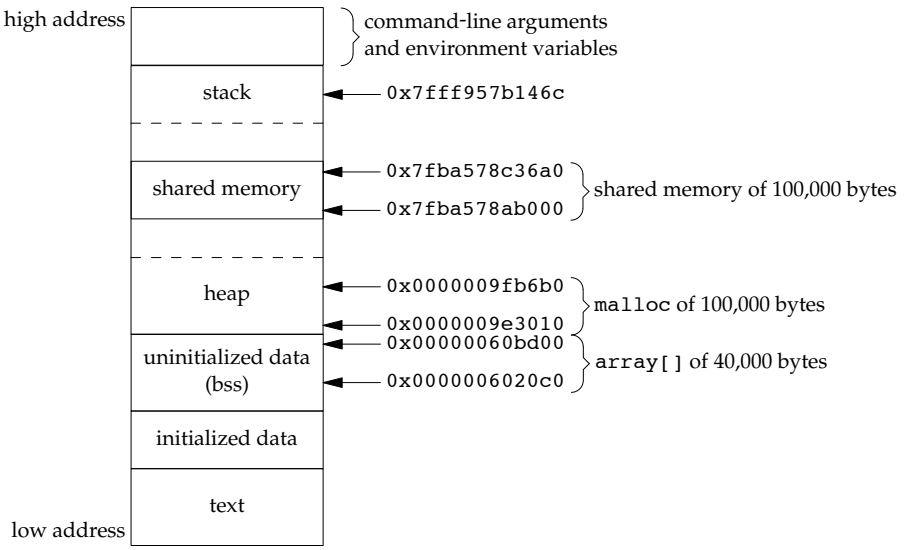

Figure 15.32 Memory layout on an Intel-based Linux system

Recall that the `mmap` function (Section 14.8) can be used to map portions of a file into the address space of a process. This is conceptually similar to attaching a shared memory segment using the `shmat` XSI IPC function. The main difference is that the memory segment mapped with `mmap` is backed by a file, whereas no file is associated with an XSI shared memory segment.

Example—Memory Mapping of `/dev/zero`

Shared memory can be used between unrelated processes. But if the processes are related, some implementations provide a different technique.

> The following technique works on FreeBSD 8.0, Linux 3.2.0, and Solaris 10. Mac OS X 10.6.8 currently doesn't support the mapping of character devices into the address space of a process.

The device `/dev/zero` is an infinite source of 0 bytes when read. This device also accepts any data that is written to it, ignoring the data. Our interest in this device for IPC arises from its special properties when it is memory mapped.

- An unnamed memory region is created whose size is the second argument to `mmap`, rounded up to the nearest page size on the system.
- The memory region is initialized to 0.
- Multiple processes can share this region if a common ancestor specifies the `MAP_SHARED` flag to `mmap`.

The program in Figure 15.33 is an example that uses this special device.

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS      1000
#define SIZE        sizeof(long)    /* size of shared memory area */

static int
update(long *ptr)
{
    return((*ptr)++);   /* return value before increment */
}

int
main(void)
{
    int     fd, i, counter;
    pid_t   pid;
    void    *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("open error");
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0)) == MAP_FAILED)
        err_sys("mmap error");
    close(fd);      /* can close /dev/zero now that it's mapped */

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {           /* parent */
        for (i = 0; i < NLOOPS; i += 2) {
            if ((counter = update((long *)area)) != i)
                err_quit("parent: expected %d, got %d", i, counter);

            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else {                        /* child */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("child: expected %d, got %d", i, counter);

            TELL_PARENT(getppid());
        }
    }

    exit(0);
}
```

Figure 15.33 IPC between parent and child using memory mapped I/O of `/dev/zero`

The program opens the /dev/zero device and calls mmap, specifying a size of a long integer. Note that once the region is mapped, we can close the device. The process then creates a child. Since MAP_SHARED was specified in the call to mmap, writes to the memory-mapped region by one process are seen by the other process. (If we had specified MAP_PRIVATE instead, this example wouldn't work.)

The parent and the child then alternate running, incrementing a long integer in the shared memory-mapped region, using the synchronization functions from Section 8.9. The memory-mapped region is initialized to 0 by mmap. The parent increments it to 1, then the child increments it to 2, then the parent increments it to 3, and so on. Note that we have to use parentheses when we increment the value of the long integer in the update function, since we are incrementing the value and not the pointer.

The advantage of using /dev/zero in the manner that we've shown is that an actual file need not exist before we call mmap to create the mapped region. Mapping /dev/zero automatically creates a mapped region of the specified size. The disadvantage of this technique is that it works only between related processes. With related processes, however, it is probably simpler and more efficient to use threads (Chapters 11 and 12). Note that no matter which technique is used, we still need to synchronize access to the shared data. □

Example — Anonymous Memory Mapping

Many implementations provide anonymous memory mapping, a facility similar to the /dev/zero feature. To use this facility, we specify the MAP_ANON flag to mmap and specify the file descriptor as –1. The resulting region is anonymous (since it's not associated with a pathname through a file descriptor) and creates a memory region that can be shared with descendant processes.

> The anonymous memory-mapping facility is supported by all four platforms discussed in this text. Note, however, that Linux defines the MAP_ANONYMOUS flag for this facility, but defines the MAP_ANON flag to be the same value for improved application portability.

To modify the program in Figure 15.33 to use this facility, we make three changes: (a) remove the open of /dev/zero, (b) remove the close of fd, and (c) change the call to mmap to the following:

```
if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                 MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED)
```

In this call, we specify the MAP_ANON flag and set the file descriptor to –1. The rest of the program from Figure 15.33 is unchanged. □

The last two examples illustrate sharing memory among multiple related processes. If shared memory is required between unrelated processes, there are two alternatives. Applications can use the XSI shared memory functions, or they can use mmap to map the same file into their address spaces using the MAP_SHARED flag.

15.10 POSIX Semaphores

The POSIX semaphore mechanism is one of three IPC mechanisms that originated with the real-time extensions to POSIX.1. The Single UNIX Specification placed the three mechanisms (message queues, semaphores, and shared memory) in option classes. Prior to SUSv4, the POSIX semaphore interfaces were included in the semaphores option. In SUSv4, these interfaces were moved to the base specification, but the message queue and shared memory interfaces remained optional.

The POSIX semaphore interfaces were meant to address several deficiencies with the XSI semaphore interfaces:

- The POSIX semaphore interfaces allow for higher-performance implementations compared to XSI semaphores.
- The POSIX semaphore interfaces are simpler to use: there are no semaphore sets, and several of the interfaces are patterned after familiar file system operations. Although there is no requirement that they be implemented in the file system, some systems do take this approach.
- The POSIX semaphores behave more gracefully when removed. Recall that when an XSI semaphore is removed, operations using the same semaphore identifier fail with `errno` set to `EIDRM`. With POSIX semaphores, operations continue to work normally until the last reference to the semaphore is released.

POSIX semaphores are available in two flavors: named and unnamed. They differ in how they are created and destroyed, but otherwise work the same. Unnamed semaphores exist in memory only and require that processes have access to the memory to be able to use the semaphores. This means they can be used only by threads in the same process or threads in different processes that have mapped the same memory extent into their address spaces. Named semaphores, in contrast, are accessed by name and can be used by threads in any processes that know their names.

To create a new named semaphore or use an existing one, we call the `sem_open` function.

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ... /* mode_t mode,
                unsigned int value */ );
```

Returns: Pointer to semaphore if OK, `SEM_FAILED` on error

When using an existing named semaphore, we specify only two arguments: the name of the semaphore and a zero value for the *oflag* argument. When the *oflag* argument has the `O_CREAT` flag set, we create a new named semaphore if it does not yet exist. If it already exists, it is opened for use, but no additional initialization takes place.

When we specify the `O_CREAT` flag, we need to provide two additional arguments. The *mode* argument specifies who can access the semaphore. It can take on the same values as the permission bits for opening a file: user-read, user-write, user-execute, group-read, group-write, group-execute, other-read, other-write, and other-execute. The resulting permissions assigned to the semaphore are modified by the caller's file

creation mask (Sections 4.5 and 4.8). Note, however, that only read and write access matter, but the interfaces don't allow us to specify the mode when we open an existing semaphore. Implementations usually open semaphores for both reading and writing.

The *value* argument is used to specify the initial value for the semaphore when we create it. It can take on any value from 0 to `SEM_VALUE_MAX` (Figure 2.9).

If we want to ensure that we are creating the semaphore, we can set the *oflag* argument to `O_CREAT|O_EXCL`. This will cause `sem_open` to fail if the semaphore already exists.

To promote portability, we must follow certain conventions when selecting a semaphore name.

- The first character in the name should be a slash (/). Although there is no requirement that an implementation of POSIX semaphores uses the file system, if the file system *is* used, we want to remove any ambiguity as to the starting point from which the name is interpreted.
- The name should contain no other slashes to avoid implementation-defined behavior. For example, if the file system is used, the names `/mysem` and `//mysem` would evaluate to the same filename, but if the implementation doesn't use the file system, the two names could be treated as different (consider what would happen if the implementation hashed the name to an integer value used to identify the semaphore).
- The maximum length of the semaphore name is implementation defined. The name should be no longer than `_POSIX_NAME_MAX` (Figure 2.8) characters, because this is the minimum acceptable limit to the maximum name length if the implementation does use the file system.

The `sem_open` function returns a semaphore pointer that we can pass to other semaphore functions when we want to operate on the semaphore. When we are done with the semaphore, we can call the `sem_close` function to release any resources associated with the semaphore.

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Returns: 0 if OK, –1 on error

If our process exits without having first called `sem_close`, the kernel will close any open semaphores automatically. Note that this doesn't affect the state of the semaphore value—if we have incremented its value, this doesn't change just because we exit. Similarly, if we call `sem_close`, the semaphore value is unaffected. There is no mechanism equivalent to the `SEM_UNDO` flag found with XSI semaphores.

To destroy a named semaphore, we can use the `sem_unlink` function.

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Returns: 0 if OK, –1 on error

The sem_unlink function removes the name of the semaphore. If there are no open references to the semaphore, then it is destroyed. Otherwise, destruction is deferred until the last open reference is closed.

Unlike with XSI semaphores, we can only adjust the value of a POSIX semaphore by one with a single function call. Decrementing the count is analogous to locking a binary semaphore or acquiring a resource associated with a counting semaphore.

> Note that there is no distinction of semaphore type with POSIX semaphores. Whether we use a binary semaphore or a counting semaphore depends on how we initialize and use the semaphore. If a semaphore only ever has a value of 0 or 1, then it is a binary semaphore. When a binary semaphore has a value of 1, we say that it is "unlocked;" when it has a value of 0, we say that it is "locked."

To decrement the value of a semaphore, we can use either the sem_wait or sem_trywait function.

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);

int sem_wait(sem_t *sem);
```

Both return: 0 if OK, –1 on error

With the sem_wait function, we will block if the semaphore count is 0. We won't return until we have successfully decremented the semaphore count or are interrupted by a signal. We can use the sem_trywait function to avoid blocking. If the semaphore count is zero when we call sem_trywait, it will return –1 with errno set to EAGAIN instead of blocking.

A third alternative is to block for a bounded amount of time. We can use the sem_timedwait function for this purpose.

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict tsptr);
```

Returns: 0 if OK, –1 on error

The *tsptr* argument specifies the absolute time when we want to give up waiting for the semaphore. The timeout is based on the CLOCK_REALTIME clock (recall Figure 6.8). If the semaphore can be decremented immediately, then the value of the timeout doesn't matter—even though it might specify some time in the past, the attempt to decrement the semaphore will still succeed. If the timeout expires without being able to decrement the semaphore count, then sem_timedwait will return –1 with errno set to ETIMEDOUT.

To increment the value of a semaphore, we call the sem_post function. This is analogous to unlocking a binary semaphore or releasing a resource associated with a counting semaphore.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Returns: 0 if OK, –1 on error

If a process is blocked in a call to `sem_wait` (or `sem_timedwait`) when we call `sem_post`, the process is awakened and the semaphore count that was just incremented by `sem_post` is decremented by `sem_wait` (or `sem_timedwait`).

When we want to use POSIX semaphores within a single process, it is easier to use unnamed semaphores. This only changes the way we create and destroy the semaphore. To create an unnamed semaphore, we call the `sem_init` function.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Returns: 0 if OK, –1 on error

The *pshared* argument indicates if we plan to use the semaphore with multiple processes. If so, we set it to a nonzero value. The *value* argument specifies the initial value of the semaphore.

Instead of returning a pointer to the semaphore like `sem_open` does, we need to declare a variable of type `sem_t` and pass its address to `sem_init` for initialization. If we plan to use the semaphore between two processes, we need to ensure that the *sem* argument points into the memory extent that is shared between the processes.

When we are done using the unnamed semaphore, we can discard it by calling the `sem_destroy` function.

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Returns: 0 if OK, –1 on error

After calling `sem_destroy`, we can't use any of the semaphore functions with *sem* unless we reinitialize it by calling `sem_init` again.

One other function is available to allow us to retrieve the value of a semaphore. We call the `sem_getvalue` function for this purpose.

```
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict valp);
```

Returns: 0 if OK, –1 on error

On success, the integer pointed to by the *valp* argument will contain the value of the semaphore. Be aware, however, that the value of the semaphore can change by the time that we try to use the value we just read. Unless we use additional synchronization mechanisms to avoid this race, the `sem_getvalue` function is useful only for debugging.

The `sem_getvalue` function is not supported by Mac OS X 10.6.8.

Example

One of the motivations for introducing the POSIX semaphore interfaces was that they can be made to perform significantly better than the existing XSI semaphore interfaces. It is instructive to see if this goal was reached in existing systems, even though these systems were not designed to support real-time applications.

In Figure 15.34, we compare the performance of using XSI semaphores (without `SEM_UNDO`) and POSIX semaphores when 3 processes compete to allocate and release the semaphore 1,000,000 times on two platforms (Linux 3.2.0 and Solaris 10).

Operation	Solaris 10			Linux 3.2.0		
	User	System	Clock	User	System	Clock
XSI semaphores	11.85	15.85	27.91	0.33	5.93	7.33
POSIX semaphores	13.72	10.52	24.44	0.26	0.75	0.41

Figure 15.34 Timing comparison of semaphore implementations

In Figure 15.34, we can see that POSIX semaphores provide only a 12% improvement over XSI semaphores on Solaris, but on Linux the improvement is 94% (almost 18 times faster)! If we trace the programs, we find that the Linux implementation of POSIX semaphores maps the file into the process address space and performs individual semaphore operations without using system calls. □

Example

Recall from Figure 12.5 that the Single UNIX Specification doesn't define what happens when one thread locks a normal mutex and a different thread tries to unlock it, but that error-checking mutexes and recursive mutexes generate errors in this case. Because a binary semaphore can be used like a mutex, we can use a semaphore to create our own locking primitive to provide mutual exclusion.

Assuming we were to create our own lock that could be locked by one thread and unlocked by another, our lock structure might look like

```
struct slock {
  sem_t *semp;
  char   name[_POSIX_NAME_MAX];
};
```

The program in Figure 15.35 shows an implementation of a semaphore-based mutual exclusion primitive.

```
#include "slock.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

struct slock *
```

```
s_alloc()
{
    struct slock *sp;
    static int cnt;

    if ((sp = malloc(sizeof(struct slock))) == NULL)
        return(NULL);
    do {
        snprintf(sp->name, sizeof(sp->name), "/%ld.%d", (long)getpid(),
          cnt++);
        sp->semp = sem_open(sp->name, O_CREAT|O_EXCL, S_IRWXU, 1);
    } while ((sp->semp == SEM_FAILED) && (errno == EEXIST));
    if (sp->semp == SEM_FAILED) {
        free(sp);
        return(NULL);
    }
    sem_unlink(sp->name);
    return(sp);
}

void
s_free(struct slock *sp)
{
    sem_close(sp->semp);
    free(sp);
}

int
s_lock(struct slock *sp)
{
    return(sem_wait(sp->semp));
}

int
s_trylock(struct slock *sp)
{
    return(sem_trywait(sp->semp));
}

int
s_unlock(struct slock *sp)
{
    return(sem_post(sp->semp));
}
```

Figure 15.35 Mutual exclusion using a POSIX semaphore

We create a name based on the process ID and a counter. We don't bother to protect the counter with a mutex, because if two racing threads call `s_alloc` at the same time and end up with the same name, using the `O_EXCL` flag in the call to `sem_open` will cause one to succeed and one to fail with `errno` set to `EEXIST`, so we just retry if this happens. Note that we unlink the semaphore after opening it. This destroys the name so that no other process can access it and simplifies cleanup when the process ends. □

15.11 Client–Server Properties

Let's detail some of the properties of clients and servers that are affected by the various types of IPC used between them. The simplest type of relationship is to have the client `fork` and `exec` the desired server. Two half-duplex pipes can be created before the `fork` to allow data to be transferred in both directions. Figure 15.16 is an example of this arrangement. The server that is executed can be a set-user-ID program, giving it special privileges. Also, the server can determine the real identity of the client by looking at its real user ID. (Recall from Section 8.10 that the real user ID and real group ID don't change across an `exec`.)

With this arrangement, we can build an *open server*. (We show an implementation of this client–server mechanism in Section 17.5.) It opens files for the client instead of the client calling the `open` function. This way, additional permission checking can be added, above and beyond the normal UNIX system user/group/other permissions. We assume that the server is a set-user-ID program, giving it additional permissions (root permission, perhaps). The server uses the real user ID of the client to determine whether to give it access to the requested file. This way, we can build a server that allows certain users permissions that they don't normally have.

In this example, since the server is a child of the parent, all the server can do is pass back the contents of the file to the parent. Although this works fine for regular files, it can't be used for special device files, for example. We would like to be able to have the server open the requested file and pass back the file descriptor. Whereas a parent can pass a child an open descriptor, a child cannot pass a descriptor back to the parent (unless special programming techniques are used, which we cover in Chapter 17).

We showed the next type of server in Figure 15.23. The server is a daemon process that is contacted using some form of IPC by all clients. We can't use pipes for this type of client–server arrangement. A form of named IPC is required, such as FIFOs or message queues. With FIFOs, we saw that an individual per-client FIFO is also required if the server is to send data back to the client. If the client–server application sends data only from the client to the server, a single well-known FIFO suffices. (The System V line printer spooler used this form of client–server arrangement. The client was the `lp`(1) command, and the server was the `lpsched` daemon process. A single FIFO was used, since the flow of data was only from the client to the server. Nothing was sent back to the client.)

Multiple possibilities exist with message queues.

1. A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient. For example, the clients can send their requests with a type field of 1. Included in the request must be the client's process ID. The server then sends the response with the type field set to the client's process ID. The server receives only the messages with a type field of 1 (the fourth argument for `msgrcv`), and the clients receive only the messages with a type field equal to their process IDs.

2. Alternatively, an individual message queue can be used for each client. Before sending the first request to a server, each client creates its own message queue

with a key of IPC_PRIVATE. The server also has its own queue, with a key or identifier known to all clients. The client sends its first request to the server's well-known queue, and this request must contain the message queue ID of the client's queue. The server sends its first response to the client's queue, and all future requests and responses are exchanged on this queue.

One problem with this technique is that each client-specific queue usually has only a single message on it: a request for the server or a response for a client. This seems wasteful of a limited systemwide resource (a message queue), and a FIFO can be used instead. Another problem is that the server has to read messages from multiple queues. Neither select nor poll works with message queues.

Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method (a semaphore or record locking).

The problem with this type of client–server relationship (the client and the server being unrelated processes) is for the server to identify the client accurately. Unless the server is performing a nonprivileged operation, it is essential that the server know who the client is. This is required, for example, if the server is a set-user-ID program. Although all these forms of IPC go through the kernel, there is no facility provided by them to have the kernel identify the sender.

With message queues, if a single queue is used between the client and the server (so that only a single message is on the queue at a time, for example), the msg_lspid of the queue contains the process ID of the other process. But when writing the server, we want the effective user ID of the client, not its process ID. There is no portable way to obtain the effective user ID, given the process ID. (Naturally, the kernel maintains both values in the process table entry, but other than rummaging around through the kernel's memory, we can't obtain one, given the other.)

We'll use the following technique in Section 17.2 to allow the server to identify the client. The same technique can be used with FIFOs, message queues, semaphores, and shared memory. For the following description, assume that FIFOs are being used, as in Figure 15.23. The client must create its own FIFO and set the file access permissions of the FIFO so that only user-read and user-write are on. We assume that the server has superuser privileges (or else it probably wouldn't care about the client's true identity), so the server can still read and write to this FIFO. When the server receives the client's first request on the server's well-known FIFO (which must contain the identity of the client-specific FIFO), the server calls either stat or fstat on the client-specific FIFO. The server assumes that the effective user ID of the client is the owner of the FIFO (the st_uid field of the stat structure). The server verifies that only the user-read and user-write permissions are enabled. As another check, the server should look at the three times associated with the FIFO (the st_atime, st_mtime, and st_ctime fields of the stat structure) to verify that they are recent (no older than 15 or 30 seconds, for example). If a malicious client can create a FIFO with someone else as the owner and set the file's permission bits to user-read and user-write only, then the system has other fundamental security problems.

To use this technique with XSI IPC, recall that the `ipc_perm` structure associated with each message queue, semaphore, and shared memory segment identifies the creator of the IPC structure (the `cuid` and `cgid` fields). As with the example using FIFOs, the server should require the client to create the IPC structure and have the client set the access permissions to user-read and user-write only. The times associated with the IPC structure should also be verified by the server to be recent (since these IPC structures hang around until explicitly deleted).

We'll see in Section 17.3 that a far better way of doing this authentication is for the kernel to provide the effective user ID and effective group ID of the client. This is done by the socket subsystem when file descriptors are passed between processes.

15.12 Summary

We've detailed numerous forms of interprocess communication: pipes, named pipes (FIFOs), the three forms of IPC commonly called XSI IPC (message queues, semaphores, and shared memory), and an alternative semaphore mechanism provided by POSIX. Semaphores are really a synchronization primitive, not true IPC, and are often used to synchronize access to a shared resource, such as a shared memory segment. With pipes, we looked at the implementation of the `popen` function, at coprocesses, and at the pitfalls that can be encountered with the standard I/O library's buffering.

After comparing the timing of message queues versus full-duplex pipes, and semaphores versus record locking, we can make the following recommendations: learn pipes and FIFOs, since these two basic techniques can still be used effectively in numerous applications. Avoid using message queues and semaphores in any new applications. Full-duplex pipes and record locking should be considered instead, as they are far simpler. Shared memory still has its use, although the same functionality can be provided through the use of the `mmap` function (Section 14.8).

In the next chapter, we will look at network IPC, which can allow processes to communicate across machine boundaries.

Exercises

15.1 In the program shown in Figure 15.6, remove the `close` right before the `waitpid` at the end of the parent code. Explain what happens.

15.2 In the program in Figure 15.6, remove the `waitpid` at the end of the parent code. Explain what happens.

15.3 What happens if the argument to `popen` is a nonexistent command? Write a small program to test this.

15.4 In the program shown in Figure 15.18, remove the signal handler, execute the program, and then terminate the child. After entering a line of input, how can you tell that the parent was terminated by `SIGPIPE`?

15.5 In the program in Figure 15.18, use the standard I/O library for reading and writing the pipes instead of `read` and `write`.

17

Advanced IPC

17.1 Introduction

In the previous two chapters, we discussed various forms of IPC, including pipes and sockets. In this chapter, we look at an advanced form of IPC—the UNIX domain socket mechanism—and see what we can do with it. With this form of IPC, we can pass open file descriptors between processes running on the same computer system, server processes can associate names with their file descriptors, and client processes running on the same system can use these names to rendezvous with the servers. We'll also see how the operating system provides a unique IPC channel per client.

17.2 UNIX Domain Sockets

UNIX domain sockets are used to communicate with processes running on the same machine. Although Internet domain sockets can be used for this same purpose, UNIX domain sockets are more efficient. UNIX domain sockets only copy data; they have no protocol processing to perform, no network headers to add or remove, no checksums to calculate, no sequence numbers to generate, and no acknowledgements to send.

UNIX domain sockets provide both stream and datagram interfaces. The UNIX domain datagram service is reliable, however. Messages are neither lost nor delivered out of order. UNIX domain sockets are like a cross between sockets and pipes. You can use the network-oriented socket interfaces with them, or you can use the `socketpair` function to create a pair of unnamed, connected, UNIX domain sockets.

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

int socketpair(int domain, int type, int protocol, int sockfd[2]);
```

Returns: 0 if OK, –1 on error

Although the interface is sufficiently general to allow `socketpair` to be used with other domains, operating systems typically provide support only for the UNIX domain.

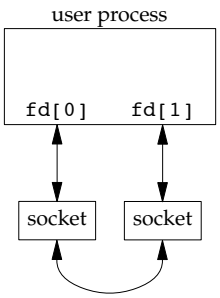

Figure 17.1 A socket pair

A pair of connected UNIX domain sockets acts like a full-duplex pipe: both ends are open for reading and writing (see Figure 17.1). We'll refer to these as "fd-pipes" to distinguish them from normal, half-duplex pipes.

Example — `fd_pipe` Function

Figure 17.2 shows the `fd_pipe` function, which uses the `socketpair` function to create a pair of connected UNIX domain stream sockets.

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

/*
 * Returns a full-duplex pipe (a UNIX domain socket) with
 * the two file descriptors returned in fd[0] and fd[1].
 */
int
fd_pipe(int fd[2])
{
    return(socketpair(AF_UNIX, SOCK_STREAM, 0, fd));
}
```

Figure 17.2 Creating a full-duplex pipe

Some BSD-based systems use UNIX domain sockets to implement pipes. But when `pipe` is called, the write end of the first descriptor and the read end of the second descriptor are both closed. To get a full-duplex pipe, we must call `socketpair` directly. □

Example — Polling XSI Message Queues with the Help of UNIX Domain Sockets

In Section 15.6.4, we said one of the problems with using XSI message queues is that we can't use `poll` or `select` with them, because they aren't associated with file descriptors. However, sockets *are* associated with file descriptors, and we can use them to notify us when messages arrive. We'll use one thread per message queue. Each thread will block in a call to `msgrcv`. When a message arrives, the thread will write it down one end of a UNIX domain socket. Our application will use the other end of the socket to receive the message when `poll` indicates data can be read from the socket.

The program in Figure 17.3 illustrates this technique. The `main` function creates the message queues and UNIX domain sockets and starts one thread to service each queue. Then it uses an infinite loop to poll one end of the sockets. When a socket is readable, it reads from the socket and writes the message on the standard output.

```
#include "apue.h"
#include <poll.h>
#include <pthread.h>
#include <sys/msg.h>
#include <sys/socket.h>

#define NQ      3       /* number of queues */
#define MAXMSZ  512     /* maximum message size */
#define KEY     0x123   /* key for first message queue */

struct threadinfo {
    int qid;
    int fd;
};

struct mymesg {
    long mtype;
    char mtext[MAXMSZ];
};

void *
helper(void *arg)
{
    int                 n;
    struct mymesg       m;
    struct threadinfo   *tip = arg;

    for(;;) {
        memset(&m, 0, sizeof(m));
        if ((n = msgrcv(tip->qid, &m, MAXMSZ, 0, MSG_NOERROR)) < 0)
            err_sys("msgrcv error");
        if (write(tip->fd, m.mtext, n) < 0)
            err_sys("write error");
    }
}

int
main()
```

```
{
    int                 i, n, err;
    int                 fd[2];
    int                 qid[NQ];
    struct pollfd       pfd[NQ];
    struct threadinfo   ti[NQ];
    pthread_t           tid[NQ];
    char                buf[MAXMSZ];

    for (i = 0; i < NQ; i++) {
        if ((qid[i] = msgget((KEY+i), IPC_CREAT|0666)) < 0)
            err_sys("msgget error");

        printf("queue ID %d is %d\n", i, qid[i]);

        if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
            err_sys("socketpair error");
        pfd[i].fd = fd[0];
        pfd[i].events = POLLIN;
        ti[i].qid = qid[i];
        ti[i].fd = fd[1];
        if ((err = pthread_create(&tid[i], NULL, helper, &ti[i])) != 0)
            err_exit(err, "pthread_create error");
    }

    for (;;) {
        if (poll(pfd, NQ, -1) < 0)
            err_sys("poll error");
        for (i = 0; i < NQ; i++) {
            if (pfd[i].revents & POLLIN) {
                if ((n = read(pfd[i].fd, buf, sizeof(buf))) < 0)
                    err_sys("read error");
                buf[n] = 0;
                printf("queue id %d, message %s\n", qid[i], buf);
            }
        }
    }

    exit(0);
}
```

Figure 17.3 Poll for XSI messages using UNIX domain sockets

Note that we use datagram (`SOCK_DGRAM`) sockets instead of stream sockets. This allows us to retain message boundaries so when we read from the socket, we read only one message at a time.

This technique allows us to use either `poll` or `select` (indirectly) with message queues. As long as the costs of one thread per queue and copying each message two extra times (once to write it to the socket and once to read it from the socket) are acceptable, this technique will make it easier to use XSI message queues.

Well use the program shown in Figure 17.4 to send messages to our test program from Figure 17.3.

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

#define MAXMSZ 512

struct mymesg {
    long mtype;
    char mtext[MAXMSZ];
};

int
main(int argc, char *argv[])
{
    key_t key;
    long qid;
    size_t nbytes;
    struct mymesg m;

    if (argc != 3) {
        fprintf(stderr, "usage: sendmsg KEY message\n");
        exit(1);
    }
    key = strtol(argv[1], NULL, 0);
    if ((qid = msgget(key, 0)) < 0)
        err_sys("can't open queue key %s", argv[1]);
    memset(&m, 0, sizeof(m));
    strncpy(m.mtext, argv[2], MAXMSZ-1);
    nbytes = strlen(m.mtext);
    m.mtype = 1;
    if (msgsnd(qid, &m, nbytes, 0) < 0)
        err_sys("can't send message");
    exit(0);
}
```

Figure 17.4 Post a message to an XSI message queue

This program takes two arguments: the key associated with the queue and a string to be sent as the body of the message. When we send messages to the server, it prints them as shown below.

```
$ ./pollmsg &                          run the server in the background
[1] 12814
$ queue ID 0 is 196608
queue ID 1 is 196609
queue ID 2 is 196610
$ ./sendmsg 0x123 "hello, world"       send a message to the first queue
queue id 196608, message hello, world
$ ./sendmsg 0x124 "just a test"        send a message to the second queue
queue id 196609, message just a test
$ ./sendmsg 0x125 "bye"                send a message to the third queue
queue id 196610, message bye
```

□

17.2.1 Naming UNIX Domain Sockets

Although the socketpair function creates sockets that are connected to each other, the individual sockets don't have names. This means that they can't be addressed by unrelated processes.

In Section 16.3.4, we learned how to bind an address to an Internet domain socket. Just as with Internet domain sockets, UNIX domain sockets can be named and used to advertise services. The address format used with UNIX domain sockets differs from that used with Internet domain sockets, however.

Recall from Section 16.3 that socket address formats differ from one implementation to the next. An address for a UNIX domain socket is represented by a sockaddr_un structure. On Linux 3.2.0 and Solaris 10, the sockaddr_un structure is defined in the header <sys/un.h> as

```
struct sockaddr_un {
    sa_family_t  sun_family;     /* AF_UNIX */
    char         sun_path[108];  /* pathname */
};
```

On FreeBSD 8.0 and Mac OS X 10.6.8, however, the sockaddr_un structure is defined as

```
struct sockaddr_un {
    unsigned char  sun_len;        /* sockaddr length */
    sa_family_t    sun_family;     /* AF_UNIX */
    char           sun_path[104];  /* pathname */
};
```

The sun_path member of the sockaddr_un structure contains a pathname. When we bind an address to a UNIX domain socket, the system creates a file of type S_IFSOCK with the same name.

This file exists only as a means of advertising the socket name to clients. The file can't be opened or otherwise used for communication by applications.

If the file already exists when we try to bind the same address, the bind request will fail. When we close the socket, this file is not automatically removed, so we need to make sure that we unlink it before our application exits.

Example

The program in Figure 17.5 shows an example of binding an address to a UNIX domain socket.

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

int
main(void)
{
    int fd, size;
```

```
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("socket failed");
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        err_sys("bind failed");
    printf("UNIX domain socket bound\n");
    exit(0);
}
```

Figure 17.5 Binding an address to a UNIX domain socket

When we run this program, the `bind` request succeeds. If we run the program a second time, however, we get an error, because the file already exists. The program won't succeed again until we remove the file.

```
$ ./a.out                                          run the program
UNIX domain socket bound
$ ls -l foo.socket                                 look at the socket file
srwxr-xr-x  1 sar       0 May 18 00:44 foo.socket
$ ./a.out                                          try to run the program again
bind failed: Address already in use
$ rm foo.socket                                    remove the socket file
$ ./a.out                                          run the program a third time
UNIX domain socket bound                           now it succeeds
```

The way we determine the size of the address to bind is to calculate the offset of the `sun_path` member in the `sockaddr_un` structure and add to it the length of the pathname, not including the terminating null byte. Since implementations vary in which members precede `sun_path` in the `sockaddr_un` structure, we use the `offsetof` macro from `<stddef.h>` (included by `apue.h`) to calculate the offset of the `sun_path` member from the start of the structure. If you look in `<stddef.h>`, you'll see a definition similar to the following:

```
#define offsetof(TYPE, MEMBER)   ((int)&((TYPE *)0)->MEMBER)
```

The expression evaluates to an integer, which is the starting address of the member, assuming that the structure begins at address 0. □

17.3 Unique Connections

A server can arrange for unique UNIX domain connections to clients using the standard `bind`, `listen`, and `accept` functions. Clients use `connect` to contact the server; after the connect request is accepted by the server, a unique connection exists between the client and the server. This style of operation is the same that we illustrated with Internet domain sockets in Figures 16.16 and 16.17.

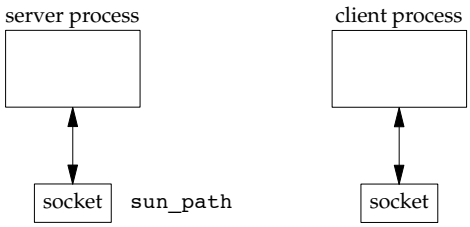

Figure 17.6 Client and server sockets before a `connect`

Figure 17.6 shows a client process and a server process before a connection exists between the two. The server has bound its socket to a `sockaddr_un` address and is listening for connection requests. Figure 17.7 shows the unique connection between the client and server after the server has accepted the client's connection request.

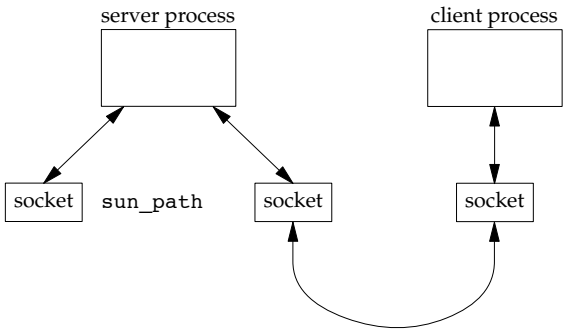

Figure 17.7 Client and server sockets after a `connect`

We will now develop three functions that can be used to create unique connections between unrelated processes running on the same machine. These functions mimic the connection-oriented socket functions discussed in Section 16.4. We use UNIX domain sockets for the underlying communication mechanism here.

```
#include "apue.h"

int serv_listen(const char *name);
```

Returns: file descriptor to listen on if OK, negative value on error

```
int serv_accept(int listenfd, uid_t *uidptr);
```

Returns: new file descriptor if OK, negative value on error

```
int cli_conn(const char *name);
```

Returns: file descriptor if OK, negative value on error

The `serv_listen` function (Figure 17.8) can be used by a server to announce its willingness to listen for client connect requests on a well-known name (some pathname in the file system). Clients will use this name when they want to connect to the server. The return value is the server's UNIX domain socket used to receive client connection requests.

The `serv_accept` function (Figure 17.9) is used by a server to wait for a client's connect request to arrive. When one arrives, the system automatically creates a new UNIX domain socket, connects it to the client's socket, and returns the new socket to the server. Additionally, the effective user ID of the client is stored in the memory to which *uidptr* points.

A client calls `cli_conn` (Figure 17.10) to connect to a server. The *name* argument specified by the client must be the same name that was advertised by the server's call to `serv_listen`. On return, the client gets a file descriptor connected to the server.

Figure 17.8 shows the `serv_listen` function.

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define QLEN    10

/*
 * Create a server endpoint of a connection.
 * Returns fd if all OK, <0 on error.
 */
int
serv_listen(const char *name)
{
    int                 fd, len, err, rval;
    struct sockaddr_un  un;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        return(-1);
    }

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-2);

    unlink(name);   /* in case it already exists */

    /* fill in socket address structure */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    /* bind the name to the descriptor */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -3;
```

```
        goto errout;
    }

    if (listen(fd, QLEN) < 0) { /* tell kernel we're a server */
        rval = -4;
        goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}
```

Figure 17.8 The `serv_listen` function

First, we create a single UNIX domain socket by calling `socket`. We then fill in a `sockaddr_un` structure with the well-known pathname to be assigned to the socket. This structure is the argument to `bind`. Note that we don't need to set the `sun_len` field present on some platforms, because the operating system sets this for us, deriving it from the address length we pass to the `bind` function.

Finally, we call `listen` (Section 16.4) to tell the kernel that the process will be acting as a server awaiting connections from clients. When a connect request from a client arrives, the server calls the `serv_accept` function (Figure 17.9).

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <time.h>
#include <errno.h>

#define STALE   30  /* client's name can't be older than this (sec) */

/*
 * Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID from the pathname
 * that it must bind before calling us.
 * Returns new fd if all OK, <0 on error
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    int                 clifd, err, rval;
    socklen_t           len;
    time_t              staletime;
    struct sockaddr_un  un;
    struct stat         statbuf;
    char                *name;

    /* allocate enough space for longest name plus terminating null */
```

```
    if ((name = malloc(sizeof(un.sun_path + 1))) == NULL)
        return(-1);
    len = sizeof(un);
    if ((clifd = accept(listenfd, (struct sockaddr *)&un, &len)) < 0) {
        free(name);
        return(-2);     /* often errno=EINTR, if signal caught */
    }

    /* obtain the client's uid from its calling address */
    len -= offsetof(struct sockaddr_un, sun_path); /* len of pathname */
    memcpy(name, un.sun_path, len);
    name[len] = 0;          /* null terminate */
    if (stat(name, &statbuf) < 0) {
        rval = -3;
        goto errout;
    }

#ifdef  S_ISSOCK    /* not defined for SVR4 */
    if (S_ISSOCK(statbuf.st_mode) == 0) {
        rval = -4;      /* not a socket */
        goto errout;
    }
#endif

    if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
        (statbuf.st_mode & S_IRWXU) != S_IRWXU) {
          rval = -5;    /* is not rwx------ */
          goto errout;
    }

    staletime = time(NULL) - STALE;
    if (statbuf.st_atime < staletime ||
        statbuf.st_ctime < staletime ||
        statbuf.st_mtime < staletime) {
          rval = -6;    /* i-node is too old */
          goto errout;
    }

    if (uidptr != NULL)
        *uidptr = statbuf.st_uid;   /* return uid of caller */
    unlink(name);       /* we're done with pathname now */
    free(name);
    return(clifd);

errout:
    err = errno;
    close(clifd);
    free(name);
    errno = err;
    return(rval);
}
```

Figure 17.9 The `serv_accept` function

The server blocks in the call to `accept`, waiting for a client to call `cli_conn`. When `accept` returns, its return value is a brand-new descriptor that is connected to the client. Additionally, the pathname that the client assigned to its socket (the name that contained the client's process ID) is returned by `accept`, through the second argument (the pointer to the `sockaddr_un` structure). We copy this pathname and ensure that it is null terminated (if the pathname takes up all available space in the `sun_path` member of the `sockaddr_un` structure, there won't be room for the terminating null byte). Then we call `stat` to verify that the pathname is indeed a socket and that the permissions allow only user-read, user-write, and user-execute. We also verify that the three times associated with the socket are no older than 30 seconds. (Recall from Section 6.10 that the `time` function returns the current time and date in seconds past the Epoch.)

If all these checks are OK, we assume that the identity of the client (its effective user ID) is the owner of the socket. Although this check isn't perfect, it's the best we can do with current systems. (It would be better if the kernel returned the effective user ID to us through a parameter to `accept`.)

The client initiates the connection to the server by calling the `cli_conn` function (Figure 17.10).

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define CLI_PATH    "/var/tmp/"
#define CLI_PERM    S_IRWXU         /* rwx for user only */

/*
 * Create a client endpoint and connect to a server.
 * Returns fd if all OK, <0 on error.
 */
int
cli_conn(const char *name)
{
    int                 fd, len, err, rval;
    struct sockaddr_un  un, sun;
    int                 do_unlink = 0;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        return(-1);
    }

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    /* fill socket address structure with our address */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    sprintf(un.sun_path, "%s%05ld", CLI_PATH, (long)getpid());
```

```
    len = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);

    unlink(un.sun_path);        /* in case it already exists */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -2;
        goto errout;
    }
    if (chmod(un.sun_path, CLI_PERM) < 0) {
        rval = -3;
        do_unlink = 1;
        goto errout;
    }

    /* fill socket address structure with server's address */
    memset(&sun, 0, sizeof(sun));
    sun.sun_family = AF_UNIX;
    strcpy(sun.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);
    if (connect(fd, (struct sockaddr *)&sun, len) < 0) {
        rval = -4;
        do_unlink = 1;
        goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    if (do_unlink)
        unlink(un.sun_path);
    errno = err;
    return(rval);
}
```

Figure 17.10 The `cli_conn` function

We call `socket` to create the client's end of a UNIX domain socket. We then fill in a `sockaddr_un` structure with a client-specific name.

We don't let the system choose a default address for us, because the server would be unable to distinguish one client from another (if we don't explicitly bind a name to a UNIX domain socket, the kernel implicitly binds an address to it on our behalf and no file is created in the file system to represent the socket). Instead, we bind our own address—a step we usually don't take when developing a client program that uses sockets.

The last five characters of the pathname we bind are made from the process ID of the client. We call `unlink`, just in case the pathname already exists. We then call `bind` to assign a name to the client's socket. This creates a socket file in the file system with the same name as the bound pathname. We call `chmod` to turn off all permissions other than user-read, user-write, and user-execute. In `serv_accept`, the server checks these permissions and the user ID of the socket to verify the client's identity.

We then have to fill in another `sockaddr_un` structure, this time with the well-known pathname of the server. Finally, we call the `connect` function to initiate the connection with the server.

17.4 Passing File Descriptors

Passing an open file descriptor between processes is a powerful technique. It can lead to different ways of designing client–server applications. It allows one process (typically a server) to do everything that is required to open a file (involving such details as translating a network name to a network address, dialing a modem, and negotiating locks for the file) and simply pass back to the calling process a descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are hidden from the client.

We must be more specific about what we mean by "passing an open file descriptor" from one process to another. Recall Figure 3.8, which showed two processes that have opened the same file. Although they share the same v-node, each process has its own file table entry.

When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to share the same file table entry. Figure 17.11 shows the desired arrangement.

Technically, we are passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn't true.) Having two processes share an open file table is exactly what happens after a `fork` (recall Figure 8.2).

What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor. Closing the descriptor by the sender doesn't really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't specifically received the descriptor yet).

We define the following three functions that we use in this chapter to send and receive file descriptors. Later in this section, we'll show the code for these three functions.

```
#include "apue.h"

int send_fd(int fd, int fd_to_send);

int send_err(int fd, int status, const char *errmsg);
```

Both return: 0 if OK, –1 on error

```
int recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t));
```

Returns: file descriptor if OK, negative value on error

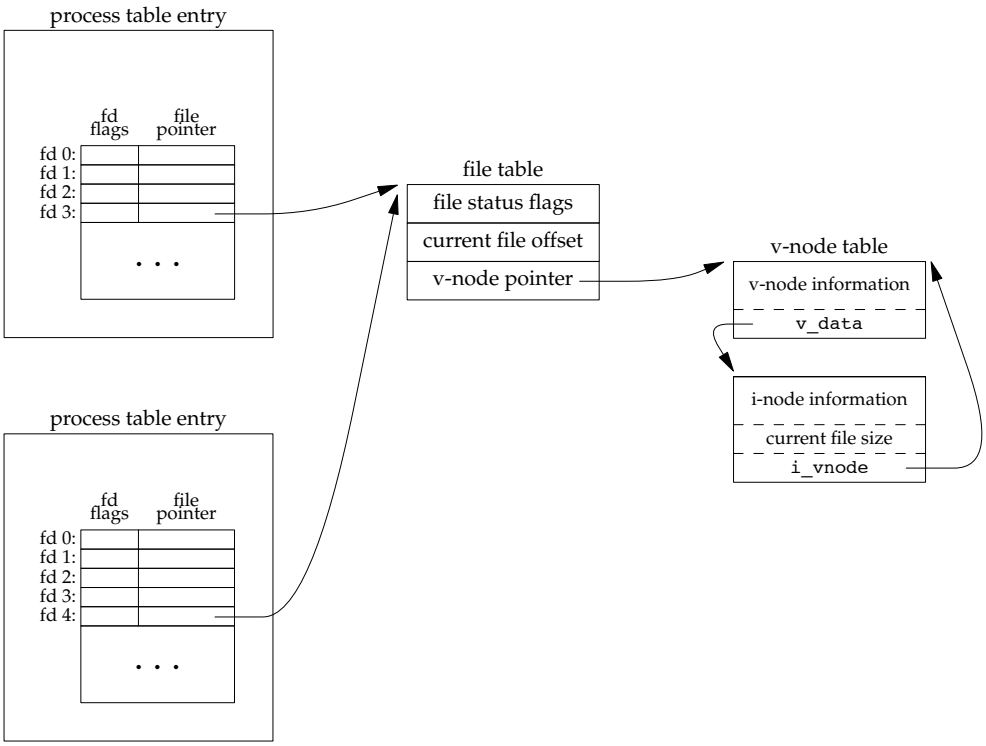

Figure 17.11 Passing an open file from the top process to the bottom process

A process (normally a server) that wants to pass a descriptor to another process calls either `send_fd` or `send_err`. The process waiting to receive the descriptor (the client) calls `recv_fd`.

The `send_fd` function sends the descriptor *fd_to_send* across using the UNIX domain socket represented by *fd*. The `send_err` function sends the *errmsg* using *fd*, followed by the *status* byte. The value of *status* must be in the range –1 through –255.

Clients call `recv_fd` to receive a descriptor. If all is OK (the sender called `send_fd`), the non-negative descriptor is returned as the value of the function. Otherwise, the value returned is the *status* that was sent by `send_err` (a negative value in the range –1 through –255). Additionally, if an error message was sent by the server, the client's *userfunc* is called to process the message. The first argument to *userfunc* is the constant `STDERR_FILENO`, followed by a pointer to the error message and its length. The return value from *userfunc* is the number of bytes written or a negative number on error. Often, the client specifies the normal `write` function as the *userfunc*.

We implement our own protocol that is used by these three functions. To send a descriptor, `send_fd` sends two bytes of 0, followed by the actual descriptor. To send an error, `send_err` sends the *errmsg*, followed by a byte of 0, followed by the absolute value of the *status* byte (1 through 255). The `recv_fd` function reads everything on the

socket until it encounters a null byte. Any characters read up to this point are passed to the caller's *userfunc*. The next byte read by `recv_fd` is the status byte. If the status byte is 0, a descriptor was passed; otherwise, there is no descriptor to receive.

The function `send_err` calls the `send_fd` function after writing the error message to the socket. This is shown in Figure 17.12.

```
#include "apue.h"

/*
 * Used when we had planned to send an fd using send_fd(),
 * but encountered an error instead.  We send the error back
 * using the send_fd()/recv_fd() protocol.
 */
int
send_err(int fd, int errcode, const char *msg)
{
    int     n;

    if ((n = strlen(msg)) > 0)
        if (writen(fd, msg, n) != n)    /* send the error message */
            return(-1);

    if (errcode >= 0)
        errcode = -1;   /* must be negative */

    if (send_fd(fd, errcode) < 0)
        return(-1);

    return(0);
}
```

Figure 17.12 The `send_err` function

To exchange file descriptors using UNIX domain sockets, we call the `sendmsg`(2) and `recvmsg`(2) functions (Section 16.5). Both functions take a pointer to a `msghdr` structure that contains all the information on what to send or receive. The structure on your system might look similar to the following:

```
struct msghdr {
    void         *msg_name;        /* optional address */
    socklen_t     msg_namelen;     /* address size in bytes */
    struct iovec *msg_iov;         /* array of I/O buffers */
    int           msg_iovlen;      /* number of elements in array */
    void         *msg_control;     /* ancillary data */
    socklen_t     msg_controllen;  /* number of ancillary bytes */
    int           msg_flags;       /* flags for received message */
};
```

The first two elements are normally used for sending datagrams on a network connection, where the destination address can be specified with each datagram. The next two elements allow us to specify an array of buffers (scatter read or gather write), as we described for the `readv` and `writev` functions (Section 14.6). The `msg_flags` field contains flags describing the message received, as summarized in Figure 16.15.

Two elements deal with the passing or receiving of control information. The `msg_control` field points to a `cmsghdr` (control message header) structure, and the `msg_controllen` field contains the number of bytes of control information.

```
struct cmsghdr {
    socklen_t  cmsg_len;    /* data byte count, including header */
    int        cmsg_level;  /* originating protocol */
    int        cmsg_type;   /* protocol-specific type */
    /* followed by the actual control message data */
};
```

To send a file descriptor, we set `cmsg_len` to the size of the `cmsghdr` structure, plus the size of an integer (the descriptor). The `cmsg_level` field is set to `SOL_SOCKET`, and `cmsg_type` is set to `SCM_RIGHTS`, to indicate that we are passing access rights. (`SCM` stands for *socket-level control message*.) Access rights can be passed only across a UNIX domain socket. The descriptor is stored right after the `cmsg_type` field, using the macro `CMSG_DATA` to obtain the pointer to this integer.

Three macros are used to access the control data, and one macro is used to help calculate the value to be used for `cmsg_len`.

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

unsigned char *CMSG_DATA(struct cmsghdr *cp);
```

Returns: pointer to data associated with `cmsghdr` structure

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);
```

Returns: pointer to first `cmsghdr` structure associated with the `msghdr` structure, or `NULL` if none exists

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mp,
                            struct cmsghdr *cp);
```

Returns: pointer to next `cmsghdr` structure associated with the `msghdr` structure given the current `cmsghdr` structure, or `NULL` if we're at the last one

```
unsigned int CMSG_LEN(unsigned int nbytes);
```

Returns: size to allocate for data object *nbytes* large

The Single UNIX Specification defines the first three macros, but omits `CMSG_LEN`.

The `CMSG_LEN` macro returns the number of bytes needed to store a data object of size *nbytes*, after adding the size of the `cmsghdr` structure, adjusting for any alignment constraints required by the processor architecture, and rounding up.

The program in Figure 17.13 is the `send_fd` function, which passes a file descriptor over a UNIX domain socket. In the `sendmsg` call, we send both the protocol data (the null and the status byte) and the descriptor.

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

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;

    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd_to_send;      /* the fd to pass */
        buf[1] = 0;     /* zero status means OK */
    }

    buf[0] = 0;         /* null byte flag to recv_fd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Figure 17.13 Sending a file descriptor over a UNIX domain socket

To receive a descriptor (Figure 17.14), we allocate enough room for a `cmsghdr` structure and a descriptor, set `msg_control` to point to the allocated area, and call `recvmsg`. We use the `CMSG_LEN` macro to calculate the amount of space needed.

We read from the socket until we read the null byte that precedes the final status byte. Everything up to this null byte is an error message from the sender.

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int             newfd, nr, status;
    char            *ptr;
    char            buf[MAXLINE];
    struct iovec    iov[1];
    struct msghdr   msg;

    status = -1;
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_ret("recvmsg error");
            return(-1);
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
```

```
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;   /* prevent sign extension */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");
                    newfd = *(int *)CMSG_DATA(cmptr);
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

Figure 17.14 Receiving a file descriptor over a UNIX domain socket

Note that we are always prepared to receive a descriptor (we set `msg_control` and `msg_controllen` before each call to `recvmsg`), but only if `msg_controllen` is nonzero on return did we actually receive a descriptor.

Recall the hoops we needed to jump through to determine the identity of the caller in the `serv_accept` function (Figure 17.9). It would have been better for the kernel to pass us the credentials of the caller on return from the call to `accept`. Some UNIX domain socket implementations provide similar functionality when exchanging messages, but their interfaces differ.

> FreeBSD 8.0 and Linux 3.2.0 provide support for sending credentials over UNIX domain sockets, but they do it differently. Mac OS X 10.6.8 is derived in part from FreeBSD, but has credential passing disabled. Solaris 10 doesn't support sending credentials over UNIX domain sockets. However, it supports the ability to obtain the credentials of a process passing a file descriptor over a STREAMS pipe, although we do not discuss the details here.

With FreeBSD, credentials are transmitted as a `cmsgcred` structure:

```
#define CMGROUP_MAX 16

struct cmsgcred {
    pid_t  cmcred_pid;                  /* sender's process ID */
    uid_t  cmcred_uid;                  /* sender's real UID */
    uid_t  cmcred_euid;                 /* sender's effective UID */
    gid_t  cmcred_gid;                  /* sender's real GID */
    short  cmcred_ngroups;              /* number of groups */
    gid_t  cmcred_groups[CMGROUP_MAX];  /* groups */
};
```

When we transmit credentials, we need to reserve space only for the `cmsgcred` structure. The kernel will fill in this structure for us to prevent an application from pretending to have a different identity.

On Linux, credentials are transmitted as a `ucred` structure:

```
struct ucred {
    pid_t  pid;  /* sender's process ID */
    uid_t  uid;  /* sender's user ID */
    gid_t  gid;  /* sender's group ID */
};
```

Unlike FreeBSD, Linux requires that we initialize this structure before transmission. The kernel will ensure that applications either use values that correspond to the caller or have the appropriate privilege to use other values.

Figure 17.15 shows the `send_fd` function updated to include the credentials of the sending process.

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

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
#define CREDSTRUCT      ucred
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct CREDSTRUCT   *credp;
    struct cmsghdr      *cmp;
    struct iovec        iov[1];
    struct msghdr       msg;
    char                buf[2]; /* send_fd/recv_ufd 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
```

```
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    msg.msg_flags = 0;
    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        cmp = cmptr;
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_RIGHTS;
        cmp->cmsg_len    = RIGHTSLEN;
        *(int *)CMSG_DATA(cmp) = fd_to_send;    /* the fd to pass */
        cmp = CMSG_NXTHDR(&msg, cmp);
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_CREDTYPE;
        cmp->cmsg_len    = CREDSLEN;
        credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
#if defined(SCM_CREDENTIALS)
        credp->uid = geteuid();
        credp->gid = getegid();
        credp->pid = getpid();
#endif
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_ufd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Figure 17.15 Sending credentials over UNIX domain sockets

Note that we need to initialize the credentials structure only on Linux.

The function in Figure 17.16 is a modified version of `recv_fd`, called `recv_ufd`, that returns the user ID of the sender through a reference parameter.

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */
#include <sys/un.h>

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define CR_UID          cmcred_uid
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
```

```
#define CREDSTRUCT      ucred
#define CR_UID          uid
#define CREDOPT         SO_PASSCRED
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_ufd(int fd, uid_t *uidptr,
         ssize_t (*userfunc)(int, const void *, size_t))
{
    struct cmsghdr      *cmp;
    struct CREDSTRUCT   *credp;
    char                *ptr;
    char                buf[MAXLINE];
    struct iovec        iov[1];
    struct msghdr       msg;
    int                 nr;
    int                 newfd = -1;
    int                 status = -1;
#if defined(CREDOPT)
    const int           on = 1;

    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("setsockopt error");
        return(-1);
    }
#endif
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
```

```
            err_ret("recvmsg error");
            return(-1);
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;   /* prevent sign extension */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");

                    /* process the control data */
                    for (cmp = CMSG_FIRSTHDR(&msg);
                      cmp != NULL; cmp = CMSG_NXTHDR(&msg, cmp)) {
                        if (cmp->cmsg_level != SOL_SOCKET)
                            continue;
                        switch (cmp->cmsg_type) {
                        case SCM_RIGHTS:
                            newfd = *(int *)CMSG_DATA(cmp);
                            break;
                        case SCM_CREDTYPE:
                            credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
                            *uidptr = credp->CR_UID;
                        }
                    }
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

Figure 17.16 Receiving credentials over UNIX domain sockets

On FreeBSD, we specify `SCM_CREDS` to transmit credentials; on Linux, we use `SCM_CREDENTIALS`.

17.5 An Open Server, Version 1

Using file descriptor passing, we now develop an open server—a program that is executed by a process to open one or more files. Instead of sending the contents of the file back to the calling process, however, this server sends back an open file descriptor. As a result, the open server can work with any type of file (such as a device or a socket) and not simply regular files. The client and server exchange a minimum amount of information using IPC: the filename and open mode sent by the client, and the descriptor returned by the server. The contents of the file are not exchanged using IPC.

There are several advantages in designing the server to be a separate executable program (either one that is executed by the client, as we develop in this section, or a daemon server, which we develop in the next section).

- The server can easily be contacted by any client, similar to the client calling a library function. We are not hard-coding a particular service into the application, but designing a general facility that others can reuse.
- If we need to change the server, only a single program is affected. Conversely, updating a library function can require that all programs that call the function be updated (i.e., relinked with the link editor). Shared libraries can simplify this updating (Section 7.7).
- The server can be a set-user-ID program, providing it with additional permissions that the client does not have. Note that a library function (or shared library function) can't provide this capability.

The client process creates an fd-pipe and then calls `fork` and `exec` to invoke the server. The client sends requests across the fd-pipe using one end, and the server sends back responses over the fd-pipe using the other end.

We define the following application protocol between the client and the server.

1. The client sends a request of the form "`open` <*pathname*> <*openmode*>\0" across the fd-pipe to the server. The <*openmode*> is the numeric value, in ASCII decimal, of the second argument to the `open` function. This request string is terminated by a null byte.
2. The server sends back an open descriptor or an error by calling either `send_fd` or `send_err`.

This is an example of a process sending an open descriptor to its parent. In Section 17.6, we'll modify this example to use a single daemon server, where the server sends a descriptor to a completely unrelated process.

We first have the header, `open.h` (Figure 17.17), which includes the standard headers and defines the function prototypes.

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

#define CL_OPEN "open"          /* client's request for server */

int     csopen(char *, int);
```

Figure 17.17 The `open.h` header

The `main` function (Figure 17.18) is a loop that reads a pathname from standard input and copies the file to standard output. The function calls `csopen` to contact the open server and return an open descriptor.

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

#define BUFFSIZE    8192

int
main(int argc, char *argv[])
{
    int     n, fd;
    char    buf[BUFFSIZE];
    char    line[MAXLINE];

    /* read filename to cat from stdin */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (line[strlen(line) - 1] == '\n')
            line[strlen(line) - 1] = 0; /* replace newline with null */

        /* open the file */
        if ((fd = csopen(line, O_RDONLY)) < 0)
            continue;   /* csopen() prints error from server */

        /* and cat to stdout */
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
        if (n < 0)
            err_sys("read error");
        close(fd);
    }

    exit(0);
}
```

Figure 17.18 The client `main` function, version 1

The function `csopen` (Figure 17.19) does the `fork` and `exec` of the server, after creating the fd-pipe.

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    pid_t           pid;
    int             len;
```

```
    char            buf[10];
    struct iovec    iov[3];
    static int      fd[2] = { -1, -1 };

    if (fd[0] < 0) {    /* fork/exec our open server first time */
        if (fd_pipe(fd) < 0) {
            err_ret("fd_pipe error");
            return(-1);
        }
        if ((pid = fork()) < 0) {
            err_ret("fork error");
            return(-1);
        } else if (pid == 0) {      /* child */
            close(fd[0]);
            if (fd[1] != STDIN_FILENO &&
              dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            if (fd[1] != STDOUT_FILENO &&
              dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            if (execl("./opend", "opend", (char *)0) < 0)
                err_sys("execl error");
        }
        close(fd[1]);               /* parent */
    }
    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";      /* string concatenation */
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* +1 for null at end of buf */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(fd[0], &iov[0], 3) != len) {
        err_ret("writev error");
        return(-1);
    }

    /* read descriptor, returned errors handled by write() */
    return(recv_fd(fd[0], write));
}
```

Figure 17.19 The csopen function, version 1

The child closes one end of the fd-pipe, and the parent closes the other. For the server that it executes, the child also duplicates its end of the fd-pipe onto its standard input and standard output. (Another option would have been to pass the ASCII representation of the descriptor `fd[1]` as an argument to the server.)

The parent sends to the server the request containing the pathname and open mode. Finally, the parent calls `recv_fd` to return either the descriptor or an error. If an error is returned by the server, `write` is called to output the message to standard error.

Now let's look at the open server. It is the program `opend` that is executed by the client in Figure 17.19. First, we have the `opend.h` header (Figure 17.20), which includes the standard headers and declares the global variables and function prototypes.

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

#define CL_OPEN "open"          /* client's request for server */

extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open() flag: O_xxx ... */
extern char *pathname;  /* of file to open() for client */

int      cli_args(int, char **);
void     handle_request(char *, int, int);
```

Figure 17.20 The `opend.h` header, version 1

The `main` function (Figure 17.21) reads the requests from the client on the fd-pipe (its standard input) and calls the function `handle_request`.

```
#include    "opend.h"

char     errmsg[MAXLINE];
int      oflag;
char    *pathname;

int
main(void)
{
    int     nread;
    char    buf[MAXLINE];

    for ( ; ; ) {   /* read arg buffer from client, process request */
        if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("read error on stream pipe");
        else if (nread == 0)
            break;      /* client has closed the stream pipe */
        handle_request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

Figure 17.21 The server `main` function, version 1

The function `handle_request` in Figure 17.22 does all the work. It calls the function `buf_args` to break up the client's request into a standard `argv`-style argument list and calls the function `cli_args` to process the client's arguments. If all is OK, `open` is called to open the file, and then `send_fd` sends the descriptor back to the client across the fd-pipe (its standard output). If an error is encountered, `send_err` is called to send back an error message, using the client–server protocol that we described earlier.

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

void
handle_request(char *buf, int nread, int fd)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        snprintf(errmsg, MAXLINE-1,
          "request not null terminated: %*.*s\n", nread, nread, buf);
        send_err(fd, -1, errmsg);
        return;
    }
    if (buf_args(buf, cli_args) < 0) {  /* parse args & set options */
        send_err(fd, -1, errmsg);
        return;
    }
    if ((newfd = open(pathname, oflag)) < 0) {
        snprintf(errmsg, MAXLINE-1, "can't open %s: %s\n", pathname,
          strerror(errno));
        send_err(fd, -1, errmsg);
        return;
    }
    if (send_fd(fd, newfd) < 0)     /* send the descriptor */
        err_sys("send_fd error");
    close(newfd);       /* we're done with descriptor */
}
```

Figure 17.22 The `handle_request` function, version 1

The client's request is a null-terminated string of white-space-separated arguments. The function `buf_args` in Figure 17.23 breaks this string into a standard `argv`-style argument list and calls a user function to process the arguments. We use the ISO C function `strtok` to tokenize the string into separate arguments.

```
#include "apue.h"

#define MAXARGC     50  /* max number of arguments in buf */
#define WHITE   " \t\n" /* white space for tokenizing arguments */

/*
 * buf[] contains white-space-separated arguments.  We convert it to an
 * argv-style array of pointers, and call the user's function (optfunc)
 * to process the array.  We return -1 if there's a problem parsing buf,
 * else we return whatever optfunc() returns.  Note that user's buf[]
 * array is modified (nulls placed after each token).
 */
int
buf_args(char *buf, int (*optfunc)(int, char **))
{
```

```
    char    *ptr, *argv[MAXARGC];
    int     argc;

    if (strtok(buf, WHITE) == NULL)     /* an argv[0] is required */
        return(-1);
    argv[argc = 0] = buf;
    while ((ptr = strtok(NULL, WHITE)) != NULL) {
        if (++argc >= MAXARGC-1)    /* -1 for room for NULL at end */
            return(-1);
        argv[argc] = ptr;
    }
    argv[++argc] = NULL;

    /*
     * Since argv[] pointers point into the user's buf[],
     * user's function can just copy the pointers, even
     * though argv[] array will disappear on return.
     */
    return((*optfunc)(argc, argv));
}
```

Figure 17.23 The `buf_args` function

The server's function that is called by `buf_args` is `cli_args` (Figure 17.24). It verifies that the client sent the right number of arguments and stores the pathname and open mode in global variables.

```
#include    "opend.h"

/*
 * This function is called by buf_args(), which is called by
 * handle_request().  buf_args() has broken up the client's
 * buffer into an argv[]-style array, which we now process.
 */
int
cli_args(int argc, char **argv)
{
    if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
        strcpy(errmsg, "usage: <pathname> <oflag>\n");
        return(-1);
    }
    pathname = argv[1];     /* save ptr to pathname to open */
    oflag = atoi(argv[2]);
    return(0);
}
```

Figure 17.24 The `cli_args` function

This completes the open server that is invoked by a `fork` and `exec` from the client. A single fd-pipe is created before the `fork` and is used to communicate between the client and the server. With this arrangement, we have one server per client.

17.6 An Open Server, Version 2

In the previous section, we developed an open server that was invoked by a `fork` and `exec` by the client, demonstrating how we can pass file descriptors from a child to a parent. In this section, we develop an open server as a daemon process. One server handles all clients. We expect this design to be more efficient, since a `fork` and an `exec` are avoided. We use a UNIX domain socket connection between the client and the server and demonstrate passing file descriptors between unrelated processes. We'll use the three functions `serv_listen`, `serv_accept`, and `cli_conn` introduced in Section 17.3. This server also demonstrates how a single server can handle multiple clients, using both the `select` and `poll` functions from Section 14.4.

This version of the client is similar to the client from Section 17.5. Indeed, the file `main.c` is identical (Figure 17.18). We add the following line to the `open.h` header (Figure 17.17):

```
#define CS_OPEN "/tmp/opend.socket"  /* server's well-known name */
```

The file `open.c` does change from Figure 17.19, since we now call `cli_conn` instead of doing the `fork` and `exec`. This is shown in Figure 17.25.

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    int             len;
    char            buf[12];
    struct iovec    iov[3];
    static int      csfd = -1;

    if (csfd < 0) {     /* open connection to conn server */
        if ((csfd = cli_conn(CS_OPEN)) < 0) {
            err_ret("cli_conn error");
            return(-1);
        }
    }

    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";  /* string concatenation */
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* null always sent */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
```

```
    if (writev(csfd, &iov[0], 3) != len) {
        err_ret("writev error");
        return(-1);
    }

    /* read back descriptor; returned errors handled by write() */
    return(recv_fd(csfd, write));
}
```

Figure 17.25 The csopen function, version 2

The protocol from the client to the server remains the same.

Next, we'll look at the server. The header `opend.h` (Figure 17.26) includes the standard headers and declares the global variables and the function prototypes.

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

#define CS_OPEN "/tmp/opend.socket" /* well-known name */
#define CL_OPEN "open"              /* client's request for server */

extern int   debug;     /* nonzero if interactive (not daemon) */
extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open flag: O_xxx ... */
extern char *pathname;  /* of file to open for client */

typedef struct {    /* one Client struct per connected client */
  int   fd;         /* fd, or -1 if available */
  uid_t uid;
} Client;

extern Client   *client;        /* ptr to malloc'ed array */
extern int       client_size;   /* # entries in client[] array */

int      cli_args(int, char **);
int      client_add(int, uid_t);
void     client_del(int);
void     loop(void);
void     handle_request(char *, int, int, uid_t);
```

Figure 17.26 The opend.h header, version 2

Since this server handles all clients, it must maintain the state of each client connection. This is done with the `client` array declared in the `opend.h` header. Figure 17.27 defines three functions that manipulate this array.

```
#include    "opend.h"

#define NALLOC  10      /* # client structs to alloc/realloc for */

static void
client_alloc(void)      /* alloc more entries in the client[] array */
```

```
{
    int     i;

    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size+NALLOC)*sizeof(Client));
    if (client == NULL)
        err_sys("can't alloc for client array");

    /* initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1;  /* fd of -1 means entry available */

    client_size += NALLOC;
}

/*
 * Called by loop() when connection request from a new client arrives.
 */
int
client_add(int fd, uid_t uid)
{
    int     i;

    if (client == NULL)     /* first time we're called */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) {   /* find an available entry */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i);  /* return index in client[] array */
        }
    }

    /* client array full, time to realloc for more */
    client_alloc();
    goto again;     /* and search again (will work this time) */
}

/*
 * Called by loop() when we're done with a client.
 */
void
client_del(int fd)
{
    int     i;

    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
```

```
            return;
        }
    }
    log_quit("can't find client entry for fd %d", fd);
}
```

Figure 17.27 Functions to manipulate `client` array

The first time `client_add` is called, it calls `client_alloc`, which in turn calls `malloc` to allocate space for ten entries in the array. After these ten entries are all in use, a later call to `client_add` causes `realloc` to allocate additional space. By dynamically allocating space this way, we have not limited the size of the `client` array at compile time to some value that we guessed and put into a header. These functions call the `log_` functions (Appendix B) if an error occurs, since we assume that the server is a daemon.

Normally the server will run as a daemon, but we want to provide an option that allows it to be run in the foreground, with diagnostic messages sent to the standard error. This should make the server easier to test and debug, especially if we don't have permission to read the log file where the diagnostic messages are normally written. We'll use a command-line option to control whether the server runs in the foreground or as a daemon in the background.

It is important that all commands on a system follow the same conventions, because this makes them easier to use. If someone is familiar with the way command-line options are formed with one command, it would create more chances for mistakes if another command followed different conventions.

This problem is sometimes visible when dealing with white space on the command line. Some commands require that an option be separated from its argument by white space, but other commands require the argument to follow immediately after its option, without any intervening spaces. Without a consistent set of rules to follow, users either have to memorize the syntax of all commands or resort to a trial-and-error process when invoking them.

The Single UNIX Specification includes a set of conventions and guidelines that promote consistent command-line syntax. They include such suggestions as "Restrict each command-line option to a single alphanumeric character" and "All options should be preceded by a – character."

Luckily, the `getopt` function exists to help command developers process command-line options in a consistent manner.

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *options);

extern int optind, opterr, optopt;
extern char *optarg;
```

Returns: the next option character, or
–1 when all options have been processed

The *argc* and *argv* arguments are the same ones passed to the `main` function of the program. The *options* argument is a string containing the option characters supported by the command. If an option character is followed by a colon, then the option takes an argument. Otherwise, the option exists by itself. For example, if the usage statement for a command was

```
command [-i] [-u username] [-z] filename
```

we would pass `"iu:z"` as the *options* string to `getopt`.

The `getopt` function is normally used in a loop that terminates when `getopt` returns –1. During each iteration of the loop, `getopt` will return the next option processed. It is up to the application to sort out any conflict in options, however; `getopt` simply parses the options and enforces a standard format.

When it encounters an invalid option, `getopt` returns a question mark instead of the character. If an option's argument is missing, `getopt` will also return a question mark, but if the first character in the options string is a colon, `getopt` returns a colon instead. The special pattern `--` will cause `getopt` to stop processing options and return –1. This allows users to provide command arguments that start with a minus sign but aren't options. For example, if you have a file named `-bar`, you can't remove it by typing

```
rm -bar
```

because `rm` will try to interpret `-bar` as options. The way to remove the file is to type

```
rm -- -bar
```

The `getopt` function supports four external variables.

`optarg` If an option takes an argument, `getopt` sets `optarg` to point to the option's argument string when an option is processed.

`opterr` If an option error is encountered, `getopt` will print an error message by default. To disable this behavior, applications can set `opterr` to 0.

`optind` The index in the `argv` array of the next string to be processed. It starts at 1 and is incremented for each argument processed by `getopt`.

`optopt` If an error is encountered during options processing, `getopt` will set `optopt` to point to the option string that caused the error.

The open server's `main` function (Figure 17.28) defines the global variables, processes the command-line options, and calls the function `loop`. If we invoke the server with the `-d` option, the server runs interactively instead of as a daemon. This option is used when testing the server.

```
#include    "opend.h"
#include    <syslog.h>

int      debug, oflag, client_size, log_to_stderr;
char     errmsg[MAXLINE];
char    *pathname;
```

```
Client  *client = NULL;

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

    log_open("open.serv", LOG_PID, LOG_USER);

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ((c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
        case 'd':       /* debug */
            debug = log_to_stderr = 1;
            break;

        case '?':
            err_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemonize("opend");

    loop();     /* never returns */
}
```

Figure 17.28 The server `main` function, version 2

The function `loop` is the server's infinite loop. We'll show two versions of this function. Figure 17.29 shows one version that uses `select`; Figure 17.30 shows another version that uses `poll`.

```
#include    "opend.h"
#include    <sys/select.h>

void
loop(void)
{
    int     i, n, maxfd, maxi, listenfd, clifd, nread;
    char    buf[MAXLINE];
    uid_t   uid;
    fd_set  rset, allset;

    FD_ZERO(&allset);

    /* obtain fd to listen for client requests on */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    FD_SET(listenfd, &allset);
    maxfd = listenfd;
    maxi = -1;
```

```
    for ( ; ; ) {
        rset = allset;  /* rset gets modified each time around */
        if ((n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
            log_sys("select error");

        if (FD_ISSET(listenfd, &rset)) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            i = client_add(clifd, uid);
            FD_SET(clifd, &allset);
            if (clifd > maxfd)
                maxfd = clifd;  /* max fd for select() */
            if (i > maxi)
                maxi = i;   /* max index in client[] array */
            log_msg("new connection: uid %d, fd %d", uid, clifd);
            continue;
        }

        for (i = 0; i <= maxi; i++) {   /* go through client[] array */
            if ((clifd = client[i].fd) < 0)
                continue;
            if (FD_ISSET(clifd, &rset)) {
                /* read argument buffer from client */
                if ((nread = read(clifd, buf, MAXLINE)) < 0) {
                    log_sys("read error on fd %d", clifd);
                } else if (nread == 0) {
                    log_msg("closed: uid %d, fd %d",
                      client[i].uid, clifd);
                    client_del(clifd);  /* client has closed cxn */
                    FD_CLR(clifd, &allset);
                    close(clifd);
                } else {    /* process client's request */
                    handle_request(buf, nread, clifd, client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.29 The `loop` function using `select`

This function calls `serv_listen` (Figure 17.8) to create the server's endpoint for the client connections. The remainder of the function is a loop that starts with a call to `select`. Two conditions can be true after `select` returns.

1. The descriptor `listenfd` can be ready for reading, which means that a new client has called `cli_conn`. To handle this, we call `serv_accept` (Figure 17.9) and then update the `client` array and associated bookkeeping information for the new client. (We keep track of the highest descriptor number for the first

argument to `select`. We also keep track of the highest index in use in the `client` array.)

2. An existing client's connection can be ready for reading. This means that the client has either terminated or sent a new request. We find out about a client termination by `read` returning 0 (end of file). If `read` returns a value greater than 0, there is a new request to process, which we handle by calling `handle_request`.

We keep track of which descriptors are currently in use in the `allset` descriptor set. As new clients connect to the server, the appropriate bit is turned on in this descriptor set. The appropriate bit is turned off when the client terminates.

We always know when a client terminates, whether the termination is voluntary or not, since all the client's descriptors (including the connection to the server) are automatically closed by the kernel. This differs from the XSI IPC mechanisms.

The `loop` function that uses `poll` is shown in Figure 17.30.

```
#include    "opend.h"
#include    <poll.h>

#define NALLOC  10  /* # pollfd structs to alloc/realloc */

static struct pollfd *
grow_pollfd(struct pollfd *pfd, int *maxfd)
{
    int             i;
    int             oldmax = *maxfd;
    int             newmax = oldmax + NALLOC;

    if ((pfd = realloc(pfd, newmax * sizeof(struct pollfd))) == NULL)
        err_sys("realloc error");
    for (i = oldmax; i < newmax; i++) {
        pfd[i].fd = -1;
        pfd[i].events = POLLIN;
        pfd[i].revents = 0;
    }
    *maxfd = newmax;
    return(pfd);
}

void
loop(void)
{
    int             i, listenfd, clifd, nread;
    char            buf[MAXLINE];
    uid_t           uid;
    struct pollfd   *pollfd;
    int             numfd = 1;
    int             maxfd = NALLOC;

    if ((pollfd = malloc(NALLOC * sizeof(struct pollfd))) == NULL)
        err_sys("malloc error");
```

```
    for (i = 0; i < NALLOC; i++) {
        pollfd[i].fd = -1;
        pollfd[i].events = POLLIN;
        pollfd[i].revents = 0;
    }

    /* obtain fd to listen for client requests on */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    client_add(listenfd, 0);    /* we use [0] for listenfd */
    pollfd[0].fd = listenfd;

    for ( ; ; ) {
        if (poll(pollfd, numfd, -1) < 0)
            log_sys("poll error");

        if (pollfd[0].revents & POLLIN) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            client_add(clifd, uid);

            /* possibly increase the size of the pollfd array */
            if (numfd == maxfd)
                pollfd = grow_pollfd(pollfd, &maxfd);
            pollfd[numfd].fd = clifd;
            pollfd[numfd].events = POLLIN;
            pollfd[numfd].revents = 0;
            numfd++;
            log_msg("new connection: uid %d, fd %d", uid, clifd);
        }

        for (i = 1; i < numfd; i++) {
            if (pollfd[i].revents & POLLHUP) {
                goto hungup;
            } else if (pollfd[i].revents & POLLIN) {
                /* read argument buffer from client */
                if ((nread = read(pollfd[i].fd, buf, MAXLINE)) < 0) {
                    log_sys("read error on fd %d", pollfd[i].fd);
                } else if (nread == 0) {
hungup:
                    /* the client closed the connection */
                    log_msg("closed: uid %d, fd %d",
                      client[i].uid, pollfd[i].fd);
                    client_del(pollfd[i].fd);
                    close(pollfd[i].fd);
                    if (i < (numfd-1)) {
                        /* pack the array */
                        pollfd[i].fd = pollfd[numfd-1].fd;
                        pollfd[i].events = pollfd[numfd-1].events;
                        pollfd[i].revents = pollfd[numfd-1].revents;
                        i--;    /* recheck this entry */
```

```
                    }
                    numfd--;
                } else {        /* process client's request */
                    handle_request(buf, nread, pollfd[i].fd,
                      client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.30 The `loop` function using `poll`

To allow for as many clients as there are possible open descriptors, we dynamically allocate space for the array of `pollfd` structures using the same strategy as used in the `client_alloc` function for the `client` array (see Figure 17.27).

We use the first entry (index 0) of the `pollfd` array for the `listenfd` descriptor. The arrival of a new client connection is indicated by a `POLLIN` on the `listenfd` descriptor. As before, we call `serv_accept` to accept the connection.

For an existing client, we have to handle two different events from `poll`: a client termination is indicated by `POLLHUP`, and a new request from an existing client is indicated by `POLLIN`. The client can close its end of the connection while there is still data to be read from the server's end of the connection. Even though the endpoint is marked as hung up, the server can read all the data queued on its end. But with this server, when we receive the hangup from the client, we can `close` the connection to the client, effectively throwing away any queued data. There is no reason to process any requests still remaining, since we can't send any responses back.

As with the `select` version of this function, new requests from a client are handled by calling the `handle_request` function (Figure 17.31). This function is similar to the earlier version (Figure 17.22). It calls the same function, `buf_args` (Figure 17.23), that calls `cli_args` (Figure 17.24), but since it runs from a daemon process, it logs error messages instead of printing them on the standard error.

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

void
handle_request(char *buf, int nread, int clifd, uid_t uid)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        snprintf(errmsg, MAXLINE-1,
          "request from uid %d not null terminated: %*.*s\n",
          uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
    log_msg("request: %s, from uid %d", buf, uid);
```

```
    /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    if ((newfd = open(pathname, oflag)) < 0) {
        snprintf(errmsg, MAXLINE-1, "can't open %s: %s\n",
          pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    /* send the descriptor */
    if (send_fd(clifd, newfd) < 0)
        log_sys("send_fd error");
    log_msg("sent fd %d over fd %d for %s", newfd, clifd, pathname);
    close(newfd);       /* we're done with descriptor */
}
```

Figure 17.31 The `request` function, version 2

This completes the second version of the open server, which uses a single daemon to handle all the client requests.

17.7 Summary

The key points in this chapter are the ability to pass file descriptors between processes and the ability of a server to accept unique connections from clients. Although all platforms provide support for UNIX domain sockets (refer back to Figure 15.1), we've seen that there are differences in each implementation, which makes it more difficult for us to develop portable applications.

We used UNIX domain sockets throughout this chapter. We saw how to use them to implement a full-duplex pipe and how they can be used to adapt the I/O multiplexing functions from Section 14.4 to work indirectly with XSI message queues.

We presented two versions of an open server. One version was invoked directly by the client, using `fork` and `exec`. The second was a daemon server that handled all client requests. Both versions used the file descriptor passing and receiving functions.

We also saw how to use the `getopt` function to enforce consistent command-line processing for our programs. The final version of the open server used the `getopt` function, the client–server connection functions introduced in Section 17.3, and the I/O multiplexing functions from Section 14.4.