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 `typedefs` 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`.