Code	SUS mandatory	Symbolic constant	Description
ADV		_POSIX_ADVISORY_INFO	advisory information (real-time)
AIO		_POSIX_ASYNCHRONOUS_IO	asynchronous input and output (real-time)
BAR		_POSIX_BARRIERS	barriers (real-time)
CPT		_POSIX_CPUTIME	process CPU time clocks (real-time)
CS		_POSIX_CLOCK_SELECTION	clock selection (real-time)
CX	•		extension to ISO C standard
FSC	•	_POSIX_FSYNC	file synchronization
IP6		_POSIX_IPV6	IPv6 interfaces
MF	•	_POSIX_MAPPED_FILES	memory-mapped files
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)
MPR	•	_POSIX_MEMORY_PROTECTION	memory protection
MSG		_POSIX_MESSAGE_PASSING	message passing (real-time)
MX			IEC 60559 floating-point option
PIO		_POSIX_PRIORITIZED_IO	prioritized input and output
PS		_POSIX_PRIORITIZED_SCHEDULING	process scheduling (real-time)
RS		_POSIX_RAW_SOCKETS	raw sockets
RTS		_POSIX_REALTIME_SIGNALS	real-time signals extension
SEM		_POSIX_SEMAPHORES	semaphores (real-time)
SHM		_POSIX_SHARED_MEMORY_OBJECTS	shared memory objects (real-time)
SIO		_POSIX_SYNCHRONIZED_IO	synchronized input and output (real-time)
SPI		_POSIX_SPIN_LOCKS	spin locks (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)
TEF		_POSIX_TRACE_EVENT_FILTER	trace event filter
THR	•	_POSIX_THREADS	threads
TMO		_POSIX_TIMEOUTS	timeouts (real-time)
TMR		_POSIX_TIMERS	timers (real-time)
TPI		_POSIX_THREAD_PRIO_INHERIT	thread priority inheritance (real-time)
TPP		_POSIX_THREAD_PRIO_PROTECT	thread priority protection (real-time)
TPS		_POSIX_THREAD_PRIORITY_SCHEDULING	thread execution scheduling (real-time)
TRC		_POSIX_TRACE	trace
TRI		_POSIX_TRACE_INHERIT	trace inherit
TRL		_POSIX_TRACE_LOG	trace log
TSA	•	_POSIX_THREAD_ATTR_STACKADDR	thread stack address attribute
TSF	•	_POSIX_THREAD_SAFE_FUNCTIONS	thread-safe functions
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 address size
TYM		_POSIX_TYPED_MEMORY_OBJECTS	typed memory objects (real-time)
XSI	•	_XOPEN_UNIX	X/Open extended interfaces
XSR		_XOPEN_STREAMS	XSI STREAMS

Figure 2.5 POSIX.1 optional interface groups and codes

Some of the additional interfaces defined in the XSI are required, whereas others are optional. The interfaces 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 the `_XOPEN_REALTIME_THREADS` symbolic constant
- Advanced real-time threads
- Tracing
- XSI STREAMS: denoted by the `_XOPEN_STREAMS` symbolic constant
- Legacy: denoted by the `_XOPEN_LEGACY` symbolic constant

The Single UNIX Specification (SUS) 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 past 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 further 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. In addition, they investigated 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, for short) was published by The Open Group in 2001. The Base Specifications of SUSv3 are the same as the 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 this version 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 included more technical corrections edited in with the main text of the standard.

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 Std. 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 have been 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 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). 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. This was followed in 1991 with Release 2.0 of the BSD Networking Software, 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 any 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 was done by Bill Jolitz and was documented in a series of monthly articles in *Dr. Dobb's Journal* throughout 1991. Much of this code appears 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 5.2.1 operating system was one of the four 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, with an emphasis on portability between hardware platforms. The OpenBSD project (`http://www.openbsd.org`) is similar to FreeBSD but with an emphasis on security.

2.3.4 Linux

Linux is an operating system that provides a rich UNIX programming environment, and 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 Mandrake 9.2 distribution of Linux was one of the operating systems used to test the examples in this book. That distribution uses the 2.4.22 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]) and the FreeBSD operating system. Darwin is managed as an open source project, similar to FreeBSD and Linux.

Mac OS X version 10.3 (Darwin 7.4.0) 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. It is based on System V Release 4, with more than ten years of enhancements from the engineers at Sun Microsystems. It is the only commercially successful SVR4 descendant, and is formally certified to be a UNIX system. (For more information on UNIX certification, see `http://www.opengroup.org/certification/idx/unix.html`.)

The Solaris 9 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 and currently 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 5.2.1, Linux 2.4.22, Mac OS X 10.3, and Solaris 9. Although only Solaris can call itself a UNIX system, all four provide a UNIX programming environment. Because all four are POSIX compliant to varying degrees, we will also concentrate on the features that are 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. As SUSv3 is a superset of POSIX.1, we'll also note any features that are part of SUSv3 but not part of POSIX.1.

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 aiding the portability of our software.

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 characters 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 value of the limit.

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 characters in a filename. Before SVR4, System V historically allowed only 14 characters 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-character limit, whereas a filename in another file system could have a 255-character limit.

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

1. Compile-time limits (headers)
2. Runtime limits that are 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_MIN`	min value of `short`	−32,767	−32,768
`SHRT_MAX`	max value of `short`	32,767	32,767
`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	16

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

2.5.1 ISO C Limits

All the limits defined by ISO C are compile-time limits. Figure 2.6 shows the limits from the C standard that are defined in the file `<limits.h>`. These constants are always defined in the header and don't change in a given system. The third column 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.

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 value is 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.

In Figure 2.7, we show the values of `FOPEN_MAX` and `TMP_MAX` on the four platforms we discuss in this book.

> ISO C also defines the constant `FILENAME_MAX`, but we avoid using it, because some operating system implementations historically have defined it to be too small to be of use.

Limit	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
FOPEN_MAX	20	16	20	20
TMP_MAX	308,915,776	238,328	308,915,776	17,576

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 only concern ourselves with the ones that affect the base POSIX.1 interfaces. These limits and constants are divided into the following five categories:

1. Invariant minimum values: the 19 constants in Figure 2.8
2. Invariant value: SSIZE_MAX
3. Runtime increasable values: CHARCLASS_NAME_MAX, COLL_WEIGHTS_MAX, LINE_MAX, NGROUPS_MAX, and RE_DUP_MAX
4. Runtime invariant values, possibly indeterminate: ARG_MAX, CHILD_MAX, HOST_NAME_MAX, LOGIN_NAME_MAX, OPEN_MAX, PAGESIZE, RE_DUP_MAX, STREAM_MAX, SYMLOOP_MAX, TTY_NAME_MAX, and TZNAME_MAX
5. Pathname variable values, possibly indeterminate: FILESIZEBITS, LINK_MAX, MAX_CANON, MAX_INPUT, NAME_MAX, PATH_MAX, PIPE_BUF, and SYMLINK_MAX

Of these 44 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 19 invariant minimum values are shown in Figure 2.8.

Name	Description: minimum acceptable value for	Value
_POSIX_ARG_MAX	length of arguments to exec functions	4,096
_POSIX_CHILD_MAX	number of child processes per real user ID	25
_POSIX_HOST_NAME_MAX	maximum length of a host name as returned by gethostname	255
_POSIX_LINK_MAX	number of links to a file	8
_POSIX_LOGIN_NAME_MAX	maximum 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	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_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_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 invariant minimum values from <limits.h>

These values are invariant; they 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 1003.1-2001. A strictly-conforming application is a POSIX-conforming application that does not rely on any undefined behavior, does not use any obsolescent interfaces, and does not require values of constants larger than the minimums shown in Figure 2.8.

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 255 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 19 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 19 implementation values are items 2–5 from our list earlier in this section: the invariant value, the runtime increasable value, the runtime invariant values, and the pathname variable values.) The problem is that not all of the 19 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. So, POSIX.1 decided to provide three runtime functions for us to call—`sysconf`, `pathconf`, and `fpathconf`—to determine the actual implementation value at runtime. 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 Linux, for example, the number of `iovec` structures you can use with `readv` or `writev` is limited only by the amount of memory on the system. Thus, `IOV_MAX` is considered indeterminate on Linux. We'll return to this problem of indeterminate runtime limits in Section 2.5.5.

2.5.3 XSI Limits

The XSI also defines constants that deal with implementation limits. They include:

1. Invariant minimum values: the ten constants in Figure 2.9
2. Numerical limits: `LONG_BIT` and `WORD_BIT`
3. Runtime invariant values, possibly indeterminate: `ATEXIT_MAX`, `IOV_MAX`, and `PAGE_SIZE`

The invariant minimum values are listed in Figure 2.9. Many of these values deal with message catalogs. The last two illustrate the situation in which the POSIX.1 minimums were too small—presumably to allow for embedded POSIX.1 implementations—so the

Single UNIX Specification added symbols with larger minimum values for XSI-conforming systems.

Name	Description	Minimum acceptable value	Typical value
NL_ARGMAX	maximum value of digit in calls to `printf` and `scanf`	9	9
NL_LANGMAX	maximum number of bytes in LANG environment variable	14	14
NL_MSGMAX	maximum message number	32,767	32,767
NL_NMAX	maximum number of bytes in *N*-to-1 mapping characters	(none specified)	1
NL_SETMAX	maximum set number	255	255
NL_TEXTMAX	maximum number of bytes in a message string	_POSIX2_LINE_MAX	2,048
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 2.9 XSI invariant 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 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 filedes, 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 2.10 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 2.11 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.

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
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	maximum number of open files per process	_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
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
TTY_NAME_MAX	length of a terminal device name, including the terminating null	_SC_TTY_NAME_MAX
TZNAME_MAX	maximum number of bytes for the name of a time zone	_SC_TZNAME_MAX

Figure 2.10 Limits and *name* arguments to sysconf

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 2.10 and 2.11 lists the limit constants we'll deal with throughout the rest of this book.
2. Some *names* 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 8.16).

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`
`SYMLINK_MAX`	number of bytes in a symbolic link	`_PC_SYMLINK_MAX`

Figure 2.11 Limits and *name* arguments to `pathconf` and `fpathconf`

There are some restrictions for the *pathname* argument to `pathconf` and the *filedes* argument to `fpathconf`. 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` 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.

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.)

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 2.12 builds a C program that prints the value of each pathconf and sysconf symbol.

```
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 %%d\\n\", %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 %%d\\n\", %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 2.12 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 2.13, 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 %d\n", 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 %d\n", 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 2.13 Print all possible sysconf and pathconf values

Figure 2.14 summarizes results from Figure 2.13 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.

Limit	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	
				UFS file system	PCFS file system
ARG_MAX	65,536	131,072	262,144	1,048,320	1,048,320
ATEXIT_MAX	32	2,147,483,647	no symbol	no limit	no limit
CHARCLASS_NAME_MAX	no symbol	2,048	no symbol	14	14
CHILD_MAX	867	999	100	7,877	7,877
clock ticks/second	128	100	100	100	100
COLL_WEIGHTS_MAX	0	255	2	10	10
FILESIZEBITS	unsupported	64	no symbol	41	unsupported
HOST_NAME_MAX	255	unsupported	no symbol	no symbol	no symbol
IOV_MAX	1,024	no limit	no symbol	16	16
LINE_MAX	2,048	2,048	2,048	2,048	2,048
LINK_MAX	32,767	32,000	32,767	32,767	1
LOGIN_NAME_MAX	17	256	no symbol	9	9
MAX_CANON	255	255	255	256	256
MAX_INPUT	255	255	255	512	512
NAME_MAX	255	255	765	255	8
NGROUPS_MAX	16	32	16	16	16
OPEN_MAX	1,735	1,024	256	256	256
PAGESIZE	4,096	4,096	4,096	8,192	8,192
PAGE_SIZE	4,096	4,096	no symbol	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	1,735	16	20	256	256
SYMLINK_MAX	unsupported	no limit	no symbol	no symbol	no symbol
SYMLOOP_MAX	32	no limit	no symbol	no symbol	no symbol
TTY_NAME_MAX	255	32	no symbol	128	128
TZNAME_MAX	255	6	255	no limit	no limit

Figure 2.14 Examples of configuration limits

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. □

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 2.15 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, 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.

Standards prior to SUSv3 were unclear as to whether or not PATH_MAX included a null byte at the end of the pathname. If the operating system implementation conforms to one of these prior versions, 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 were allocating space for a call to getcwd, for example—to return the absolute pathname of the current working directory; see Section 4.22—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 int  pathmax = PATH_MAX;
#else
static int  pathmax = 0;
#endif

#define SUSV3   200112L

static long posix_version = 0;

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

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

    if (posix_version == 0)
        posix_version = sysconf(_SC_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 */
        }
    }
    if (posix_version < SUSV3)
        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 2.15 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.

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 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. As with our pathname example, this is not guaranteed to work for all cases, but it's the best we can do. We show this technique in Figure 2.16.

```
#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, we're not
 * guaranteed that this is adequate.
 */
#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 2.16 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 limits values that are effectively unlimited. Such is the case with the Linux limit for ATEXIT_MAX (see Figure 2.14). 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 2.16 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 extensions 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 extensions, 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, and Korn shells with the ulimit command.) If our system supports this functionality, we could change the function in Figure 2.16 to call sysconf every time it is called, not only 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), the Single UNIX Specification 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 2.17 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_THREADS is undefined, we can call sysconf with the *name* argument set to _SC_THREADS to determine whether the platform supports the POSIX threads 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 extensions.

If the symbolic constant is defined by the platform, we have three possibilities.

1. If the symbolic constant is defined to have the value −1, then the corresponding option is unsupported by the platform.
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.

Figure 2.17 summarizes the options and their symbolic constants that can be used with `sysconf`, in addition to those listed in Figure 2.5.

Name of option	Description	*name* argument
`_POSIX_JOB_CONTROL`	indicates whether the implementation supports job control	`_SC_JOB_CONTROL`
`_POSIX_READER_WRITER_LOCKS`	indicates whether the implementation supports reader–writer locks	`_SC_READER_WRITER_LOCKS`
`_POSIX_SAVED_IDS`	indicates whether the implementation supports the saved set-user-ID and the saved set-group-ID	`_SC_SAVED_IDS`
`_POSIX_SHELL`	indicates whether the implementation supports the POSIX shell	`_SC_SHELL`
`_POSIX_VERSION`	indicates the POSIX.1 version	`_SC_VERSION`
`_XOPEN_CRYPT`	indicates whether the implementation supports the XSI encryption option group	`_SC_XOPEN_CRYPT`
`_XOPEN_LEGACY`	indicates whether the implementation supports the XSI legacy option group	`_SC_XOPEN_LEGACY`
`_XOPEN_REALTIME`	indicates whether the implementation supports the XSI real-time option group	`_SC_XOPEN_REALTIME`
`_XOPEN_REALTIME_THREADS`	indicates whether the implementation supports the XSI real-time threads option group	`_SC_XOPEN_REALTIME_THREADS`
`_XOPEN_VERSION`	indicates the XSI version	`_SC_XOPEN_VERSION`

Figure 2.17 Options and *name* arguments to `sysconf`

The symbolic constants used with `pathconf` and `fpathconf` are summarized in Figure 2.18. 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.

Name of option	Description	*name* argument
`_POSIX_CHOWN_RESTRICTED`	indicates whether use of chown is restricted	`_PC_CHOWN_RESTRICTED`
`_POSIX_NO_TRUNC`	indicates whether pathnames 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`	indicates whether asynchronous I/O can be used with the associated file	`_PC_ASYNC_IO`
`_POSIX_PRIO_IO`	indicates whether prioritized I/O can be used with the associated file	`_PC_PRIO_IO`
`_POSIX_SYNC_IO`	indicates whether synchronized I/O can be used with the associated file	`_PC_SYNC_IO`

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

2. The value returned for `_SC_XOPEN_VERSION` indicates the version of the XSI that the system complies with. The value associated with Version 3 of the Single UNIX Specification is 600.
3. The values `_SC_JOB_CONTROL`, `_SC_SAVED_IDS`, and `_PC_VDISABLE` no longer represent optional features. As of Version 3 of the Single UNIX Specification, these features are now required, although these symbols are retained for backward compatibility.
4. `_PC_CHOWN_RESTRICTED` and `_PC_NO_TRUNC` return −1 without changing `errno` if the feature is not supported for the specified *pathname* or *filedes*.
5. 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.
6. The referenced file for `_PC_NO_TRUNC` must be a directory. The return value applies to filenames within the directory.
7. The referenced file for `_PC_VDISABLE` must be a terminal file.

In Figure 2.19 we show several configuration options and their corresponding values on the four sample systems we discuss in this text. Note that several of the systems haven't yet caught up to the latest version of the Single UNIX Specification. For example, Mac OS X 10.3 supports POSIX threads but defines `_POSIX_THREADS` as

```
#define _POSIX_THREADS
```

without specifying a value. To conform to Version 3 of the Single UNIX Specification, the symbol, if defined, should be set to −1, 0, or 200112.

An entry is marked as "undefined" if the feature is not defined, i.e., the system doesn't define the symbolic constant or its corresponding `_PC` or `_SC` name. In contrast, the "defined" entry means that the symbolic constant is defined, but no value is specified, as in the preceding `_POSIX_THREADS` example. An entry is "unsupported" if the system defines the symbolic constant, but it has a value of −1, or it has a value of 0 but the corresponding `sysconf` or `pathconf` call returned −1.

Limit	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	
				UFS file system	PCFS file system
`_POSIX_CHOWN_RESTRICTED`	1	1	1	1	1
`_POSIX_JOB_CONTROL`	1	1	1	1	1
`_POSIX_NO_TRUNC`	1	1	1	1	unsupported
`_POSIX_SAVED_IDS`	unsupported	1	unsupported	1	1
`_POSIX_THREADS`	200112	200112	defined	1	1
`_POSIX_VDISABLE`	255	0	255	0	0
`_POSIX_VERSION`	200112	200112	198808	199506	199506
`_XOPEN_UNIX`	unsupported	1	undefined	1	1
`_XOPEN_VERSION`	unsupported	500	undefined	3	3

Figure 2.19 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. But 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 use any implementation-defined limits, 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.

Previous versions of the POSIX.1 standard defined the `_POSIX_SOURCE` constant. This has been 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=200112 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  200112
```

To make the functionality of Version 3 of the Single UNIX Specification available to applications, we need to define the constant `_XOPEN_SOURCE` to be 600. This has the same effect as defining `_POSIX_C_SOURCE` to be 200112L 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=600 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=600 -std=c99 file.c -o file
```

Another feature test macro is __STDC__, which is automatically defined by the C compiler if the compiler conforms to the ISO C standard. This allows us to write C programs that compile under both ISO C compilers and non-ISO C compilers. For example, to take advantage of the ISO C prototype feature, if supported, a header could contain

```
#ifdef __STDC__
void  *myfunc(const char *, int);
#else
void  *myfunc();
#endif
```

Although most C compilers these days support the ISO C standard, this use of the __STDC__ feature test macro can still be found in many header files.

2.8 Primitive System Data Types

Historically, certain C data types have been associated with certain UNIX system variables. For example, the 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, 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 also. These data types are defined in the headers with the C typedef facility. Most end in _t. Figure 2.20 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 Conflicts 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 SUSv3 is a superset of POSIX.1. There are some differences.

Type	Description
caddr_t	core address (Section 14.9)
clock_t	counter of clock ticks (process time) (Section 1.10)
comp_t	compressed clock ticks (Section 8.14)
dev_t	device numbers (major and minor) (Section 4.23)
fd_set	file descriptor sets (Section 14.5.1)
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)
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 2.20 Some common primitive system data types

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. To 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 term, clock ticks per second, defined differently by ISO C and POSIX.1. Both standards also use the same data type (clock_t) to hold these different values. The difference can be seen in Solaris, where clock returns microseconds (hence CLOCKS_PER_SEC is 1 million), whereas sysyconf returns the value 100 for clock ticks per second.

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, many 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'll 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 over the past two decades with the standardization of the UNIX programming environment. We've described the dominant standards—ISO C, POSIX, and the Single UNIX Specification—and their effect on the four implementations 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 one can obtain copies of the standards that we've discussed.

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, on FreeBSD 5.2.1, `size_t` is defined in 26 different headers. Because all 26 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 2.16 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 the kernel data structures 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.

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`. These constants are defined in the `<unistd.h>` header.

File descriptors range from 0 through `OPEN_MAX`. (Recall Figure 2.10.) 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 increased this limit to 63.

> With FreeBSD 5.2.1, Mac OS X 10.3, and Solaris 9, 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. Linux 2.4.22 places a hard limit of 1,048,576 on the number of file descriptors per process.

3.3 open Function

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

```
#include <fcntl.h>

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

                                   Returns: file descriptor if OK, −1 on error
```

We show the third argument as `. . .`, which is the ISO C way to specify that the number and types of the remaining arguments may vary. For this function, the third 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 *pathname* 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.

One and only one of these three 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_CREAT	Create the file if it doesn't exist. This option requires a third argument to the open 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_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_TRUNC	If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0.
O_NOCTTY	If the *pathname* 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_NONBLOCK	If the *pathname* 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 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.

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

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.

Solaris 9 supports all three flags. FreeBSD 5.2.1 and Mac OS X 10.3 have a separate flag (`O_FSYNC`) that does the same thing as `O_SYNC`. Because the two flags are equivalent, FreeBSD 5.2.1 defines them to have the same value (but curiously, Mac OS X 10.3 doesn't define `O_SYNC`). FreeBSD 5.2.1 and Mac OS X 10.3 don't support the `O_DSYNC` or `O_RSYNC` flags. Linux 2.4.22 treats both flags the same as `O_SYNC`.

The file descriptor returned by `open` 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 with the `dup2` function.

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 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 pathnames are truncated or whether an error is returned. As we saw in Chapter 2, this value can vary based on the type of the file system.

Whether or not 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.

As another example, see Figure 2.19. 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 the entire pathname exceeds `PATH_MAX` or any filename component of the pathname exceeds `NAME_MAX`.

3.4 `creat` Function

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

```
#include <fcntl.h>

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

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

Note that this function is equivalent to

```
open(pathname, 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(pathname, 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 filedes);
```

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 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 filedes, 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 current offset), or 2 (relative to 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/motd
seek OK
$ cat < /etc/motd | ./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 and not test if it's 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 2.20), 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 the 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've 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_V6_ILP32_OFF32`	`int`, `long`, pointer, and `off_t` types are 32 bits.	`_SC_V6_ILP32_OFF32`
`_POSIX_V6_ILP32_OFFBIG`	`int`, `long`, and pointer types are 32 bits; `off_t` types are at least 64 bits.	`_SC_V6_ILP32_OFFBIG`
`_POSIX_V6_LP64_OFF64`	`int` types are 32 bits; `long`, pointer, and `off_t` types are 64 bits.	`_SC_V6_LP64_OFF64`
`_POSIX_V6_LP64_OFFBIG`	`int` types are 32 bits; `long`, pointer, and `off_t` types are at least 64 bits.	`_SC_V6_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. Confusing things further is the name changes that were made between Version 2 and Version 3 of the Single UNIX Specification.

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 by setting the _FILE_OFFSET_BITS constant to the desired value, this is not guaranteed to be portable.

Note that even though you might enable 64-bit file offsets, your ability to create a file larger than 2 TB ($2^{31}-1$ 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 filedes, 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 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 filedes, char *buf, unsigned nbytes);
```

- First, the second argument was changed from a `char *` to a `void *` to be consistent with ISO C: the type `void *` is used for generic pointers.
- Next, the return value must 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 filedes, 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 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 3.4 copies a file, using only the `read` and `write` functions. 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.

```
#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 3.4 Copy standard input to standard output

- Many applications assume that standard input is file descriptor 0 and that standard output is file descriptor 1. In this example, we use the two defined names, `STDIN_FILENO` and `STDOUT_FILENO`, from `<unistd.h>`.
- 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 3.5 shows the results for reading a 103,316,352-byte file, using 20 different buffer sizes.

The file was read using the program shown in Figure 3.4, with standard output redirected to `/dev/null`. The file system used for this test was the Linux `ext2` 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 a `BUFFSIZE` of 4,096. Increasing the buffer size beyond this 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. From the last few entries in Figure 3.5, it appears that read-ahead in `ext2` stops having an effect after 128 KB.

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.

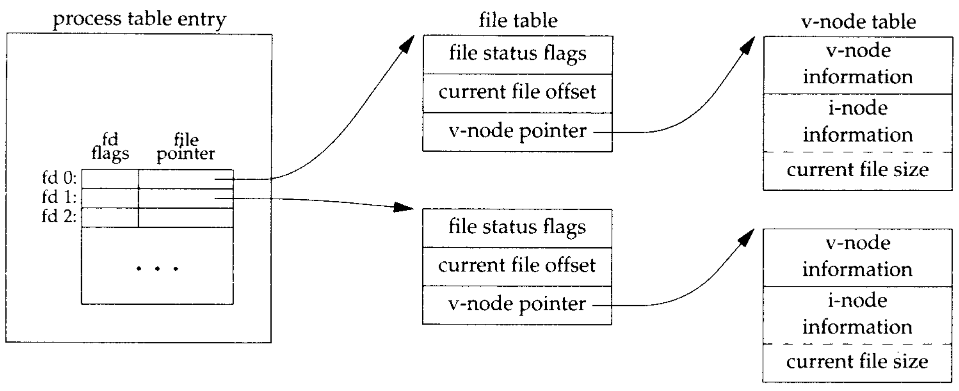

Figure 3.6 Kernel data structures for open files

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 thus 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 3.7. 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.)

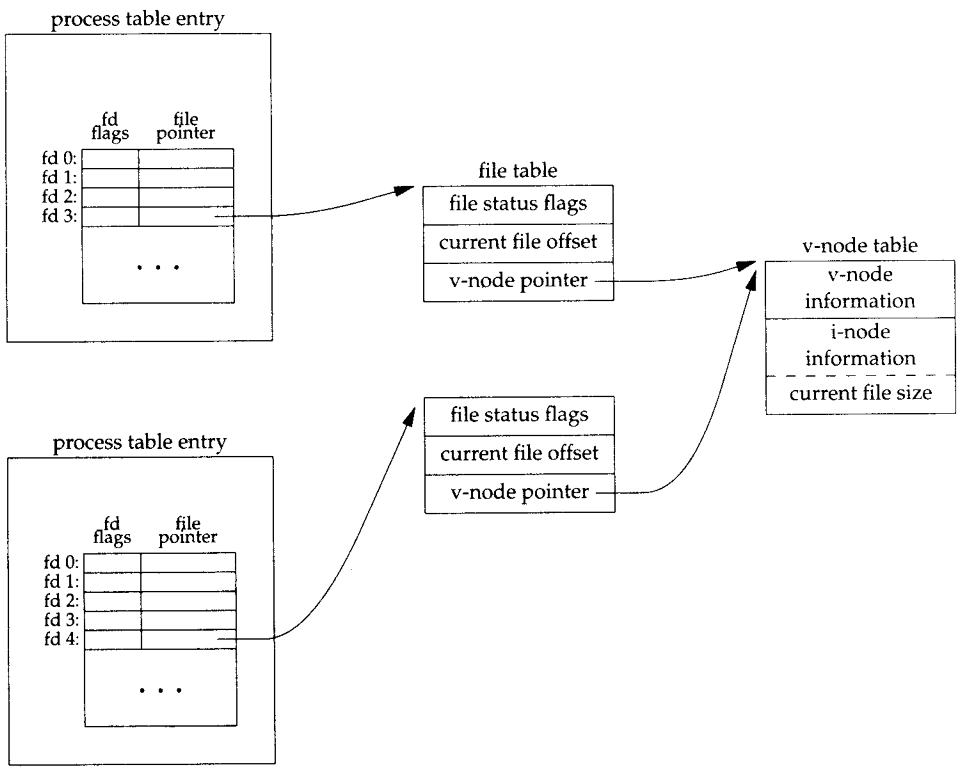

Figure 3.7 Two independent processes with the same file open

- 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 3.7. 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 can 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 XSI extensions that allow applications to seek and perform I/O atomically. These extensions are pread and pwrite.

```
#include <unistd.h>

ssize_t pread(int filedes, 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 filedes, 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 using `pread`.
- The file pointer 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(pathname, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(pathname, 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, or none are performed. It must not be possible for 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).

The function `fsync` refers only to a single file, specified by the file descriptor *filedes*, and waits for the disk writes to complete before returning. The intended use of `fsync` is for an application, such as a database, that 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 5.2.1 and Mac OS X 10.3 do 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 filedes, 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. But when we describe record locking in Section 14.3, 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`)
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 seven of these ten *cmd* values. (We'll wait until Section 14.3 to describe the last three, which deal with record locking.) Refer to Figure 3.6, since we'll be referring to 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 *filedes*. The new file descriptor is returned as the value of the function. It is the lowest-numbered descriptor that is not already open, that is greater than or equal to the third argument (taken as an integer). The new descriptor shares the same file table entry as *filedes*. (Refer to Figure 3.8.) 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_GETFD`	Return the file descriptor flags for *filedes* as the value of the function. Currently, only one file descriptor flag is defined: the `FD_CLOEXEC` flag.