UNIX SYSTEM PROGRAMMING USING C++

- Building RPC-based applications that run on heterogenous UNIX platforms
- Creating advanced multithreaded applications for multiprocessor systems
- Advanced ANSI, POSIX and UNIX programming techniques in C++
- Proven C++ classes and code examples to aid development of new applications

Terrence Chan

UNIX System Programming Using C++

Terrence Chan

**To join a Prentice Hall PTR internet mailing list:
point to http://www.prenhall.com**

**Prentice Hall PTR
Upper Saddle River, New Jersey 07458**

http://www.prenhall.com.

Library of Congress Cataloging-in-Publication Data
Chan, Terrence
UNIX system programming using C++ / Terrence Chan.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-331562-2
1. C++ (Computer program language) 2. UNIX (Computer file)
I. Title.
QA76.73.C153C46 1997
005.13'3--dc20 96-30559
CIP

Editorial/Production Supervisor: Nicholas Radhuber
Manufacturing Manager: Alexis Heydt
Acquisitions Editor: Greg Doench
Editorial Assistant: Leabe Berman
Cover Design: Lundgren Graphics, Ltd.
Cover Design Direction: Jerry Votta

Prentice-Hall, Inc.
A Simon & Schuster Company
Upper Saddle River, New Jersey 07458

The publisher offers discounts on this book when ordered in bulk quantities.
For more information, contact:
Corporate Sales Department
PTR Prentice Hall
1 Lake Street
Upper Saddle River, NJ 07458

Phone: 800-382-3419, Fax: 201-236-7141
E-mail: dan_rush@prenhall.com

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-331562-2

Prentice-Hall International (UK) Limited, London
Prentice-Hall of Australia Pty. Limited, Sydney
Prentice-Hall Canada Inc., Toronto
Prentice-Hall Hispanoamericana, S.A., Mexico
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Simon & Schuster Asia Pte. Ltd., Singapore
Editora Prentice-Hall do Brasil, Ltda., Rio de Janeiro

Table of Contents

Preface

The content of this book is derived from my several years of teaching Advanced UNIX Programming with C and C++ at two University of California Extensions (Berkeley and Santa Cruz). The objectives of the courses were to teach students advanced programming techniques using UNIX system calls and the ANSI C and C++ programming languages. Specifically, students who took the courses learned the following:

- Advanced ANSI C and C++ programming techniques, such as how to use function pointers and create functions that accept variable numbers of arguments
- The ANSI C library functions and C++ standard classes, and how to use them to reduce development time and to maximize portability of their applications
- Familiarity with the UNIX kernel structure and the system calls. These allow users to write sophisticated applications to manipulate system resources (e.g., files, processes, and system information), and to design new operating systems
- How to create network-based, multitasking, client/server applications which run on heterogenous UNIX platforms

The objective of this book is to convey to readers the techniques and concepts stated above. Furthermore, this book provides more detailed explanations and comprehensive examples on each topic than can be done in a course. Thus, readers can gain a better understanding of the subject matter and can learn at their own pace. This book also describes the latest advanced UNIX programming techniques on remote procedure calls and multithreaded programs. These techniques are important for the development of advanced distributed client/server applications in a symmetrical multiprocessing and network-based computing environment.

All the aforementioned information will be described in the C++ language. This is because in the last few years more and more advanced software developers are using C++ in applications development. This is due to the fact that the C++ language provides much stronger type-checking and includes object-oriented programming constructs than other procedural programming languages. These features are very useful in facilitating large-scale, complex UNIX system applications development and management.

This book covers the C++ programming language based on the draft version of the ANSI/ISO C++ standard [1, 2, 3]. Most of the latest C++ compilers provided by various computer vendors (e.g., Sun Microsystems Inc., Microsoft Corporation, Free Software Foundation, etc.) are compliant with this standard.

In addition to the C++ language, some significant C library functions, as defined by the ANSI C standard [4], are also described in this book. These functions are not covered by the C++ standard classes or by the UNIX application program interface. Thus, it is important that users be familiar with these to increase their knowledge base and choices of library functions.

The UNIX operating systems covered in this book include: UNIX System V.3, UNIX System V.4, BSD UNIX 4.3 and 4.4, Sun OS 4.1.3, and Solaris 2.4. The last two operating systems belong to SUN Microsystems, where Sun OS 4.1.3 is based on BSD 4.3 with UNIX System V.3 extensions, and Solaris 2.4 is based on the UNIX System V.4.

Although the primary focus of this book is on UNIX system programming, the IEEE (Institute of Electrical and Electronics Engineering) POSIX.1, POSIX.1b, and POSIX.1c standards are also covered in detail. This is to aid system programmers to develop applications that can be readily ported to different UNIX systems, as well as to POSIX-compliant systems (e.g., VMS and Windows-NT). This is important as most advanced commercial software products must run on heterogenous platforms by various computer vendors. Thus, the POSIX and ANSI standards can help users create highly platform-independent applications.

Target Audience

The book is targeted to benefit experienced software engineers and managers who are working on advanced system applications development in a UNIX environment. The products they develop may include advanced network-based client/server applications, distributed database systems, operating systems, compilers, or computer-aided design tools.

The readers should be familiar with the C++ language based on the AT&T version 3.0 (or the latest) and should have developed some C++ application programs on their own in the past. Moreover, the readers should be familiar with at least one version of UNIX system (e.g.,

UNIX System V). Specifically, the readers should know the UNIX file system architecture, user accounts assignment and management, file access control, and jobs control methods. Readers who need to brush up on UNIX system knowledge may consult any text book covering an introduction to the UNIX system.

Book Content

Although this book covers the ANSI C++ and C library functions and UNIX APIs extensively, the primary focus in describing these functions is to convey the following information to readers:

- Purposes of these functions
- Conformance of these functions to standard(s)
- How to use these functions
- Examples of their uses
- Where appropriate, how these functions are implemented in a UNIX system
- Any special considerations (e.g., conflict between the UNIX and POSIX standards) in using these functions

It is not the intention of the author to make this book a UNIX system programmer's reference manual. Thus, the function prototypes and header files required to use the ANSI library and UNIX API functions are described, but the detailed error codes that may be returned by these functions and the archive or shared libraries needed by users' programs will not be depicted. This type of information may be obtained via either the man pages of the functions or the programmer's reference manuals from the users' computer vendors.

The general organization of this book is:

- Chapter 1 describes the history of the C++ programming language and various UNIX systems. It also describes the ANSI/ISO C, ANSI/ISO C++, IEEE POSIX.1, POSIX.1b, and POSIX.1c standards
- Chapters 2 and 3 review the draft ANSI/ISO C++ programming language and object-oriented programming techniques. The C++ I/O stream classes, template functions, and exception handlings are also depicted in detail
- Chapter 4 describes the ANSI C library functions
- Chapter 5 gives an overview of the UNIX and POSIX APIs. Special header files and compile time options, as required by various standards, are depicted.
- Chapters 6 and 7 describe UNIX and POSIX.1 file APIs. These depict APIs that can be used to control various types of files in a system. They also describe file-locking techniques used to synchronize files in a multiprocessing environment

- Chapter 8 describes UNIX and POSIX.1 process creation and control methods. After reading this chapter, readers can write their own multiprocessing applications, such as a UNIX shell
- Chapter 9 describes UNIX and POSIX.1 signal handling methods
- Chapter 10 describes UNIX and POSIX.1b interprocess communication methods. These techniques are important in creating distributed client/server applications.
- Chapter 11 describes advanced network programming techniques using UNIX sockets and TLI
- Chapter 12 describes remote procedure call. This is important for development of network transport protocol-independent client/server application development on heterogenous UNIX platforms
- Chapter 13 describes multithreaded programming techniques. These techniques allow applications to make efficient use of multiprocessor resources available on any machines on which they run

Note that although this book is based on C++, the focus on this book is not object-oriented programming techniques. This is because some readers are expected to be new to UNIX system programming and/or C++ language, thus it may be difficult for these readers to learn both object-oriented and system programming techniques at the same time. However, this book includes many useful C++ classes for interprocess communication, sockets, TLI, remote procedure call, and multithreaded programming. These classes encapsulate the low-level programming interface to these advanced system functions, and can be easily extended and incorporated into user applications to reduce their development efforts, time, and costs.

Example Programs

Throughout the book extensive example programs are shown to illustrate uses of the C++ classes, library functions, and system APIs. All the examples have been compiled by a Sun Microsystems C++ (version 4.0) compiler and tested on a Sun SPARC-20 workstation running Solaris 2.4. These examples are also compiled and tested using the Free Software Foundation GNU g++ compiler (version 2.6.3) on a Sun SPARC-20 workstation. Since the GNU g++ compilers can be ported to various hardware platforms, the examples presented in this book should run on different platforms (e.g., Hewlett Packadd's HP-UX and International Business Machines's AIX) also.

Readers are encouraged to try out the example programs on their own systems to get more in-depth familiarity of this subject matter. Users may download an electronic copy of the example programs via anonymous ftp to *ftp.prenhall.com*. The directory that stores the example tar file is */pub/ptr/professional_computer_science.w-0.22/chan/unixsys*. There are README files in the tar file that describe the programs and their cross references to chapters in the book. Finally, readers are welcome to send Emails to the author at *twc@netcom.com*.

Acknowledgments

I would like to thank Peter Collinson, Jeff Gitlin, Chi Khuong, Frank Mitchell, and my wife Jessica Chan for their careful reviewing of the book manuscript. Much of their valuable input has been incorporated in the final version of this book. Furthermore, I would like to extend my appreciation to Greg Doench, Nick Radhuber, and Brat Bartow for their valuable assistance in helping me through the preparation and publication process of this book.

Finally, I am grateful to my former students at the University of California Santa Cruz Extension who took my Advanced UNIX System Calls course and gave me valuable feedback in the refinement of the course material, much of which is used throughout this book.

References

[1]. Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual,* Addison-Wesley, 1990.

[2]. Andrew Koenig, *Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++ (Committees: WG21/N0414, X3J16/94-0025*), January 1994.

[3]. Bjarne Stroustrup, *Standardizing C++. The C++ Report. Vol. 1. No. 1*, 1989.

[4]. American National Standard Institute, *American National Standard for Information Systems - Programming Language C, X3.159 - 1989*, 1989.

CHAPTER 1

UNIX and ANSI Standards

Since the invention of UNIX in the late 1960s, there has been a proliferation of different versions of UNIX on different computer systems. Recent UNIX systems have developed from AT&T System V and BSD 4.x UNIX. However, most computer vendors often add their own extensions to either the AT&T or BSD UNIX on their systems, thus creating the different versions of UNIX. In late 1980, AT&T and Sun Microsystems worked together to create the UNIX System V release 4, which is an attempt to set a UNIX system standard for the computer industry. This attempt was not totally successful, as only a few computer vendors today adopt the UNIX System V.4.

However, in the late 1980s, a few organizations proposed several standards for a UNIX-like operating system and the C language programming environment. These standards are based primarily on UNIX, and they do not impose dramatic changes in vendors' systems; thus, they are easily adopted by vendors. Furthermore, two of these standards, ANSI C and POSIX (which stands for Portable Operating System Interface), are defined by the American National Standard Institute (ANSI) and by the Institute of Electrical and Electronics Engineers (IEEE). They are very influential in setting standards in the industry; thus, most computer vendors today provide UNIX systems that conform to the ANSI C and POSIX.1 (a subset of the POSIX standards) standards.

Most of the standards define an operating system environment for C-based applications. Applications that adhere to the standards should be easily ported to other systems that conform to the same standards. This is especially important for advanced system programmers who make extensive use of system-level application program interface (API) functions (which include library functions and system calls). This is because not all UNIX systems pro-

vide a uniform set of system APIs. Furthermore, even some common APIs may be implemented differently on different UNIX systems (e.g., the *fcntl* API on UNIX System V can be used to lock and unlock files, something that the BSD UNIX version of *fcntl* API does not support). The ANSI C and POSIX standards require all conforming systems to provide a uniform set of standard libraries and system APIs, respectively; the standards also define the signatures (the data type, number of arguments, and return value) and behaviors of these functions on all systems. In this way, programs that use these functions can be ported to different systems that are compliant with the standards.

Most of the functions defined by the standards are a subset of those available on most UNIX systems. The ANSI C and POSIX committees did create a few new functions on their own, but the purpose of these functions is to supplement ambiguity or deficiency of some related constructs in existing UNIX and C. Thus, the standards are easily learned by experienced UNIX and C developers, and easily supported by computer vendors.

The objective of this book is to help familiarize users with advanced UNIX system programming techniques, including teaching users how to write portable and easily maintainable codes. This later objective can be achieved by making users familiar with the functions defined by the various standards and with those available from UNIX so that users can make an intelligent choice of which functions or APIs to use.

The rest of this chapter gives an overview of the ANSI C, draft ANSI/ISO C++, and the POSIX standards. The subsequent chapters describe the functions and APIs defined by these standards and others available from UNIX in more detail.

1.1 The ANSI C Standard

In 1989, the American National Standard Institute (ANSI) proposed C programming language standard X3.159-1989 to standardize the C programming language constructs and libraries. This standard is commonly known as the *ANSI C standard,* and it attempts to unify the implementation of the C language supported on all computer systems. Most computer vendors today still support the C language constructs and libraries as proposed by Brian Kernighan and Dennis Ritchie (commonly known as *K&R C*) as default, but users may install the ANSI C development package as an option (for an extra fee).

The major differences between ANSI C and K&R C are as follows:

- Function prototyping
- Support of the *const* and *volatile* data type qualifiers
- Support wide characters and internationalization
- Permit function pointers to be used without dereferencing

Although this book focuses on the C++ programming technique, readers still need to be familiar with the ANSI C standard because many standard C library functions are not covered by the C++ standard classes, thus almost all C++ programs call one or more standard C library functions (e.g., get time of day, or use the *strlen* function, etc.). Furthermore, for some readers who may be in the process of porting their C applications to C++, this section describes some similarities and differences between ANSI C and C++, so as to make it easy for those users to transit from ANSI C to C++.

ANSI C adopts C++ function prototype technique where function definition and declaration include function names, arguments' data types, and return value data types. Function prototypes enable ANSI C compilers to check for function calls in user programs that pass invalid numbers of arguments or incompatible argument data types. These fix a major weakness of the K&R C compilers: Invalid function calls in user programs often pass compilation but cause programs to crash when they are executed.

The following example declares a function *foo* and requires that *foo* take two arguments: the first argument *fmt* is of *char** data type, and the second argument is of *double* data type. The function *foo* returns an *unsigned long* value:

```
unsigned long foo ( char* fmt, double data )
{
    /* body of foo */
}
```

To create a declaration of the above function, a user simply takes the above function definition, strips off the body section, and replaces it with a semicolon character. Thus, the external declaration of the above function *foo* is:

```
unsigned long foo ( char* fmt, double data );
```

For functions that take a variable number of arguments, their definitions and declarations should have "..." specified as the last argument to each function:

```
int printf( const char* fmt, ...);

int printf( const char* fmt, ... )
{
    /* body of printf */
}
```

The *const* key word declares that some data cannot be changed. For example, the above function prototype declares a *fmt* argument that is of a *const char** data type, meaning that the

function *printf* cannot modify data in any character array that is passed as an actual argument value to *fmt*.

The *volatile* key word specifies that the values of some variables may change asynchronously, giving a hint to the compiler's optimization algorithm not to remove any "redundant" statements that involve "volatile" objects. For example, the following statements define an *io_Port* variable that contains an address of an I/O port of a system. The two statements that follow the definition are to wait for two bytes of data to arrive from the I/O port and retain only the second byte of data:

```
char get_io()
{
        volatile char* io_Port = 0x7777;
        char ch = *io_Port;                    /* read first byte of data */
        ch = *io_Port;                         /* read second byte of data */
}
```

In the above example, if the *io_Port* variable is not declared to be "volatile," when the program is compiled, the compiler may eliminate the second *ch = *io_Port* statement, as it is considered redundant with respect to the previous statement.

The *const* and *volatile* data type qualifiers are also supported in C++.

ANSI C supports internationalization by allowing C programs to use wide characters. Wide characters use more than one byte of storage per character. These are used in countries where the ASCII character set is not the standard. For example, the Korean character set requires two bytes per character. Furthermore, ANSI C also defines the *setlocale* function, which allows users to specify the format of date, monetary, and real number representations. For example, most countries display the date in <day>/<month>/<year> format, whereas the US displays the date in <month>/<day>/<year> format.

The function prototype of the *setlocale* function is:

```
#include <locale.h>

char setlocale ( int category, const char* locale );
```

The *setlocale* function prototype and possible values of the *category* argument are declared in the <locale.h> header. The *category* values specify what format class(es) is to be changed. Some possible values of the *category* argument are:

category value	Effect on standard C functions/macros
LC_CTYPE	Affects the behaviors of the <ctype.h> macros
LC_TIME	Affects the date and time format as returned by the *strftime, ascftime* functions, etc.
LC_NUMERIC	Affects the number representation formats via the *printf* and *scanf* functions
LC_MONETARY	Affects the monetary value format returned by the *localeconv* function
LC_ALL	Combines the effects of all the above

The *locale* argument value is a character string that defines which locale to use. Possible values may be C, POSIX, en_US, etc. The C, POSIX, en_US locales refer to the UNIX, POSIX, and US locales. By default, all processes on an ANSI C or POSIX compliant system execute the equivalent of the following call at their process start-up time:

```
setlocale( LC_ALL, "C" );
```

Thus, all processes start up have a known locale. If a *locale* value is NULL, the *setlocale* function returns the current *locale* value of a calling process. If a *locale* value is "" (a null string), the *setlocale* function looks for an environment variable LC_ALL, an environment variable with the same name as the *category* argument value, and, finally, the LANG environment variable - in that order - for the value of the *locale* argument.

The *setlocale* function is an ANSI C standard that is also adopted by POSIX.1.

ANSI C specifies that a function pointer may be used like a function name. No dereference is needed when calling a function whose address is contained in the pointer. For example, the following statements define a function pointer *funcptr,* which contains the address of the function *foo*:

```
extern void foo ( double xyz, const int* Iptr );
void (*funcptr)(double, const int*) = foo;
```

The function *foo* may be invoked by either directly calling *foo* or via the *funcptr.* The following two statements are functionally equivalent:

```
foo (12.78, "Hello world");
funcptr (12.78, "Hello world");
```

The K&R C requires *funcptr* be dereferenced to call *foo*. Thus, an equivalent statement to the above, using K&R C syntax, is:

```
(*funcptr)(12.78, "Hello world");
```

Both the ANSI C and K&R C function pointer uses are supported in C++.

In addition to the above, ANSI C also defines a set of *cpp* (C preprocessor) symbols which may be used in user programs. These symbols are assigned actual values at compile time:

***cpp* symbol**	**Use**
__STDC__	Feature test macro. Value is 1 if a compiler is ANSI C conforming, 0 otherwise
__LINE__	Evaluated to the physical line number of a source file for which this symbol is reference
__FILE__	Value is the file name of a module that contains this symbol
__DATE__	Value is the date that a module containing this symbol is compiled
__TIME__	Value is the time that a module containing this symbol is compiled

The following *test_ansi_c.c* program illustrates uses of these symbols:

```
#include <stdio.h>
int main()
{
#if __STDC__ == 0
      printf("cc is not ANSI C compliant\n");
#else
      printf(" %s compiled at %s:%s. This statement is at line %d\n",
                  __FILE__, __DATE__, __TIME__, __LINE__);
#endif
      return 0;
}
```

Note that C++ supports the __LINE__, __FILE__, __DATE__, and __TIME__ symbols, but not __STDC__.

Finally, ANSI C defines a set of standard library functions and associated headers. These headers are the subset of the C libraries available on most systems that implement K&R C. The ANSI C standard libraries are described in Chapter 4.

1.2 The ANSI/ISO C++ Standard

In early 1980s, Bjarne Stroustrup at AT&T Bell Laboratories developed the C++ programming language. C++ was derived from C and incorporated object-oriented constructs, such as classes, derived classes, and virtual functions, from simula67 [1].The objective of developing C++ is "to make writing good programs earlier and more pleasant for individual programmer" [2]. The name C++ signifies the evolution of the language from C and was coined by Rick Mascitti in 1983.

Since its invention, C++ has gained wide acceptance by software professionals. In 1989, Bjarne Stroustrup published *The Annotated C++ Reference Manual* [3]. This manual became the base for the draft ANSI C++ standard, as developed by the X3J16 committee of ANSI. In early 1990s, the WG21 committee of the International Standard Organization (ISO) joined the ANSI X3J16 committee to develop a unify ANSI/ISO C++ standard. A draft version of such a ANSI/ISO standard was published in 1994 [4]. However, the ANSI/ISO standard is still in the development stage, and it should become an official standard in the near future.

Most latest commercial C++ compilers, which are based on the AT&T C++ language version 3.0 or later, are compliant with the draft ANSI/ISO standard. Specifically, these compilers should support C++ classes, derived classes, virtual functions, operator overloading. Furthermore, they should also support template classes, template functions, exception handling, and the iostream library classes.

This book will describe the C++ language features as defined by the draft ANSI/ISO C++ standard.

1.3 Differences Between ANSI C and C++

C++ requires that all functions must be declared or defined before they can be referenced. ANSI C uses the K&R C default function declaration for any functions that are referenced before their declaration and definition in a user program.

Another difference between ANSI C and C++ is given the following function declaration:

```
int foo ();
```

ANSI C treats the above function as an old C function declaration and interprets it as declared in the following manner:

```
int foo (...);
```

which means *foo* may be called with any number of actual arguments. However, for C++, the same declaration is treated as the following declaration:

```
int foo ( void );
```

which means *foo* may not accept any argument when it is called.

Finally, C++ encrypts external function names for type-safe linkage. This ensures that an external function which is incorrectly declared and referenced in a module will cause the link editor (*/bin/ld*) to report an undefined function name. ANSI C does not employ the type-safe linkage technique and, thus, does not catch these types of user errors.

There are many other differences between ANSI C and C++, but the above items are the more common ones run into by users (For a detailed documentation of the ANSI C standard, please see [5]).

The next section describes the POSIX standards, which are more elaborate and comprehensive than are the ANSI C standard for UNIX system developers.

1.4 The POSIX Standards

Because many versions of UNIX exist today and each of them provides its own set of application programming interface (API) functions, it is difficult for system developers to create applications that can be easily ported to different versions of UNIX. To overcome this problem, the IEEE society formed a special task force called POSIX in the 1980s to create a set of standards for operating system interfacing. Several subgroups of the POSIX such as POSIX.1, POSIX.1b and POSIX.1c are concerned with the development of a set of standards for system developers.

Specifically, the POSIX.1 committee proposes a standard for a base operating system application programming interface; this standard specifies APIs for the manipulation of files and processes. It is formally known as the IEEE standard 1003.1-1990 [6], and it was also adopted by the ISO as the international standard ISO/IEC 9945:1:1990. The POSIX.1b committee proposes a set of standard APIs for a real-time operating system interface; these include interprocess communication. This standard is formally known as the IEEE standard

1003.4-1993 [7]. Lastly, the POSIX.1c standard [8] specifies multithreaded programming interface. This is the newest POSIX standard and its details are described in the last chapter of this book.

Although much of the work of the POSIX committees is based on UNIX, the standards they proposed are for a generic operating system that is not necessarily a UNIX system. For example, VMS from the Digital Equipment Corporation, OS/2 from International Business Machines, and Windows-NT from the Microsoft Corporation are POSIX-compliant, yet they are not UNIX systems. Most current UNIX systems, like UNIX System V release 4, BSD UNIX 4.4, and computer vendor-specific operating systems (e.g., Sun Microsystem's Solaris 2.x, Hewlett Packard's HP-UX 9.05 and 10.x, and IBM's AIX 4.1.x, etc.) are all POSIX.1-compliant but they still maintain their system-specific APIs.

This book will discuss the POSIX.1, POSIX.1b and POSIX.1c APIs, and also UNIX system-specific APIs. Furthermore, in the rest of the book, unless stated otherwise, when the word *POSIX* is mentioned alone, it refers to both the POSIX.1 and POSIX.1b standards.

To ensure a user program conforms to the POSIX.1 standard, the user should either define the manifested constant _POSIX_SOURCE at the beginning of each source module of the program (before the inclusion of any headers) as:

```
#define _POSIX_SOURCE
```

or specify the -D_POSIX_SOURCE option to a C++ compiler (CC) in a compilation:

```
%    CC -D_POSIX_SOURCE *.C
```

This manifested constant is used by *cpp* to filter out all non-POSIX.1 and non-ANSI C standard codes (e.g., functions, data types, and manifested constants) from headers used by the user program. Thus, a user program that is compiled and run successfully with this switch defined is POSIX.1-conforming.

POSIX.1b defines a different manifested constant to check conformance of user programs to that standard. The new macro is _POSIX_C_SOURCE, and its value is a time-stamp indicating the POSIX version to which a user program conforms. The possible values of the _POSIX_C_SOURCE macro are:

_POSIX_C_SOURCE value	**Meaning**
198808L	First version of POSIX.1 compliance
199009L	Second version of POSIX.1 compliance
199309L	POSIX.1 and POSIX.1b compliance

Each _POSIX_C_SOURCE value consists of the year and month that a POSIX standard was approved by IEEE as a standard. The *L* suffix in a value indicates that the value's data type is a long integer.

The _POSIX_C_SOURCE may be used in place of the _POSIX_SOURCE. However, some systems that support POSIX.1 only may not accept the _POSIX_C_SOURCE definition. Thus, readers should browse the *unistd.h* header file on their systems and see which constants, or both, are used in the file.

There is also a _POSIX_VERSION constant that may be defined in the <unistd.h> header. This constant contains the POSIX version to which the system conforms. The following sample program checks and displays the _POSIX_VERSION constant of the system on which it is run:

```
/* show_posix_ver.C */
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE          199309L
#include <iostream.h>
#include <unistd.h>
int main()
{
#ifdef _POSIX_VERSION
      cout << "System conforms to POSIX: "
            << _POSIX_VERSION << endl;
#else
      cout << "_POSIX_VERSION is undefined\n";
#endif
      return 0;
}
```

In general, a user program that must be strictly POSIX.1- and POSIX.1b-compliant may be written as follows:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE   199309L
#include <unistd.h>
/* include other headers here */
int main()
{
      ...
}
```

1.4.1 The POSIX Environment

Although POSIX was developed based on UNIX, a POSIX-compliant system is not necessarily a UNIX system. A few UNIX conventions have different meanings, according to the POSIX standards. Specifically, most standard C and C++ header files are stored under the */usr/include* directory in any UNIX system, and each of them is referenced by the:

```
#include <header_file_name>
```

This method of referencing header files is adopted in POSIX. However, for each name specified in a *#included* statement, there need not be a physical file of that name existing on a POSIX-conforming system. In fact the data that should be contained in that named object may be builtin to a compiler, or stored by some other means on a given system. Thus, in a POSIX environment, included files are called simply *headers* instead of *header files*. This "headers" naming convention will be used in the rest of the book. Furthermore, in a POSIX-compliant system, the */usr/include* directory does not have to exist. If users are working on a non-UNIX but POSIX-compliant system, please consult the C or C++ programmer's manual to determine the standard location, if any, of the headers on the system.

Another difference between POSIX and UNIX is the concept of *superuser*. In UNIX, a superuser has privilege to access all system resources and functions. The superuser user ID is always zero. However, the POSIX standards do not mandate that all POSIX-conforming systems support the concept of a superuser, nor does the user ID of zero require any special privileges. Furthermore, although some POSIX.1 and POSIX.1b APIs require the functions to be executed in "special privilege," it is up to an individual conforming system to define how a "special privilege" is to be assigned to a process.

1.4.2 The POSIX Feature Test Macros

Some UNIX features are optional to be implemented on a POSIX-conforming system. Thus, POSIX.1 defines a set of feature test macros, which, if defined on a system, means that the system has implemented the corresponding features.

These feature test macros, if defined, can be found in the <unistd.h> header. Their names and uses are:

Feature test macro	**Effects if defined on a system**
_POSIX_JOB_CONTROL	The system supports the BSD-style job control
_POSIX_SAVED_IDS	Each process running on the system keeps the saved set-UID and set-GID, so that it can change its effective user ID and group ID to those values via the *seteuid* and *setegid* APIs, respectively

Feature test macro	Effects if defined on a system
_POSIX_CHOWN_RESTRICTED	If the defined value is -1, users may change ownership of files owned by them. Otherwise, only users with special privilege may change ownership of any files on a system. If this constant is undefined in <unistd.h> header, users must use the *pathconf* or *fpathconf* function (described in the next section) to check the permission for changing ownership on a per-file basis
_POSIX_NO_TRUNC	If the defined value is -1, any long path name passed to an API is silently truncated to NAME_MAX bytes; otherwise, an error is generated. If this constant is undefined in the <unistd.h> header, users must use the *pathconf* or *fpathconf* function to check the path name truncation option on a per-directory basis
_POSIX_VDISABLE	If the defined value is -1, there is no disabling character for special characters for all terminal device files; otherwise, the value is the disabling character value. If this constant is undefined in the <unistd.h> header, users must use the *pathconf* or *fpathconf* function to check the disabling character option on a per-terminal device file basis

The following sample program prints the POSIX-defined configuration options supported on any given system:

```
/* show_test_macros.C */
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE    199309L
#include <iostream.h>
#include <unistd.h>
int main()
{
#ifdef _POSIX_JOB_CONTROL
      cout << "System supports job control\n";
#else
      cout << "System does not support job control\n";
#endif
```

```
#ifdef _POSIX_SAVED_IDS
      cout << "System supports saved set-UID and saved set-GID\n";
#else
      cout << "System does not support saved set-UID and "
           << " saved set-GID\n";
#endif

#ifdef _POSIX_CHOWN_RESTRICTED
      cout << "chown_restricted option is: " <<
                 _POSIX_CHOWN_RESTRICTED << endl;
#else
      cout << "System does not support chown_restricted option\n";
#endif

#ifdef _POSIX_NO_TRUNC
      cout << "Pathname trunc option is: " << _POSIX_NO_TRUNC
           << endl;
#else
      cout << "System does not support system-wide pathname"
           << " trunc option\n";
#endif

#ifdef _POSIX_VDISABLE
      cout << "Disable char. for terminal files is: "
           << _POSIX_VDISABLE << endl;
#else
      cout << "System does not support _POSIX_VDISABLE\n";
#endif
      return 0;
}
```

1.4.3 Limits Checking at Compile Time and at Run Time

POSIX.1 and POSIX.1b define a set of system configuration limits in the form of manifested constants in the <limits.h> header. Many of these limits are derived from the UNIX systems and they have the same manifested constant names as their UNIX counterparts, plus the _POSIX_ prefix. For example, UNIX systems define the constant CHILD_MAX, which specifies the maximum number of child processes a process may create at any one time. The corresponding POSIX.1 constant is _POSIX_CHILD_MAX. The reason for defining these

constants is that although most UNIX systems define a similar set of constants, their values vary substantially from one UNIX system to another. The POSIX-defined constants specify the minimum values for these constants for all POSIX-conforming systems; thus, it facilitates application programmers to develop programs that use these system configuration limits.

The following is a list of POSIX.1-defined constants in the <limits.h> header:

Compile time limit	**Min. value**	**Meaning**
_POSIX_CHILD_MAX	6	Maximum number of child processes that may be created at any one time by a process
_POSIX_OPEN_MAX	16	Maximum number of files that may be opened simultaneously by a process
_POSIX_STREAM_MAX	8	Maximum number of I/O streams that may be opened simultaneously by a process
_POSIX_ARG_MAX	4096	Maximum size, in bytes, of arguments that may be passed to an *exec* function call
_POSIX_NGROUP_MAX	0	Maximum number of supplemental groups to which a process may belong
_POSIX_PATH_MAX	255	Maximum number of characters allowed in a path name
_POSIX_NAME_MAX	14	Maximum number of characters allowed in a file name
_POSIX_LINK_MAX	8	Maximum number of links a file may have
_POSIX_PIPE_BUF	512	Maximum size of a block of data that may be atomically read from or written to a pipe file
_POSIX_MAX_INPUT	255	Maximum capacity, in bytes, of a terminal's input queue
_POSIX_MAX_CANON	255	Maximum size, in bytes, of a terminal's canonical input queue
_POSIX_SSIZE_MAX	32767	Maximum value that can be stored in a *ssize_t*-typed object
_POSIX_TZNAME_MAX	3	Maximum number of characters in a time zone name

The following is a list of POSIX.1b-defined constants:

Compile time limit	Min. value	Meaning
_POSIX_AIO_MAX	1	Number of simultaneous asynchronous I/O
_POSIX_AIO_LISTIO_MAX	2	Maximum number of operations in one listio
_POSIX_TIMER_MAX	32	Maximum number of timers that can be used simultaneously by a process
_POSIX_DELAYTIMER_MAX	32	Maximum number of overruns allowed per timer
_POSIX_MQ_OPEN_MAX	2	Maximum number of message queues that may be accessed simultaneously per process
_POSIX_MQ_PRIO_MAX	2	Maximum number of message priorities that can be assigned to messages
_POSIX_RTSIG_MAX	8	Maximum number of real-time signals
_POSIX_SIGQUEUE_MAX	32	Maximum number of real time signals that a process may queue at any one time
_POSIX_SEM_NSEMS_MAX	256	Maximum number of semaphores that may be used simultaneously per process
_POSIX_SEM_VALUE_MAX	32767	Maximum value that may be assigned to a semaphore

Note that the POSIX-defined constants specify only the minimum values for some system configuration limits. A POSIX-conforming system may be configured with higher values for these limits. Furthermore, not all these constants must be specified in the <limits.h> header, as some of these limits may be indeterminate or may vary for individual files.

To find out the actual implemented configuration limits system-wide or on individual objects, one can use the *sysconf, pathconf*, and *fpathconf* functions to query these limits' values at run time. These functions are defined by POSIX.1; the *sysconf* is used to query general system-wide configuration limits that are implemented on a given system; *pathconf* and *fpathconf* are used to query file-related configuration limits. The two functions do the same thing; the only difference is that *pathconf* takes a file's path name as argument, whereas *fpathconf* takes a file descriptor as argument. The prototypes of these functions are:

```
#include <unistd.h>

long sysconf ( const int limit_name );
long pathconf ( const char* pathname, int flimit_name );
long fpathconf ( const int fdesc, int flimit_name );
```

The *limit_name* argument value is a manifested constant as defined in the <unistd.h> header. The possible values and the corresponding data returned by the *sysconf* function are:

Limit value	***sysconf* return data**
_SC_ARG_MAX	Maximum size, in bytes, of argument values that may be passed to an *exec* API call
_SC_CHILD_MAX	Maximum number of child processes that may be owned by a process simultaneously
_SC_OPEN_MAX	Maximum number of opened files per process
_SC_NGROUPS_MAX	Maximum number of supplemental groups per process
_SC_CLK_TCK	The number of clock ticks per second.
_SC_JOB_CONTROL	The _POSIX_JOB_CONTROL value
_SC_SAVED_IDS	The _POSIX_SAVED_IDS value
_SC_VERSION	The _POSIX_VERSION value
_SC_TIMERS	The _POSIX_TIMERS value
_SC_DELAYTIMER_MAX	Maximum number of overruns allowed per timer
_SC_RTSIG_MAX	Maximum number of real time signals
_SC_MQ_OPEN_MAX	Maximum number of message queues per process
_SC_MQ_PRIO_MAX	Maximum priority value assignable to a message
_SC_SEM_MSEMS_MAX	Maximum number of semaphores per process
_SC_SEM_VALUE_MAX	Maximum value assignable to a semaphore
_SC_SIGQUEUE_MAX	Maximum number of real time signals that a process may queue at any one time
_SC_AIO_LISTIO_MAX	Maximum number of operations in one listio
_SC_AIO_MAX	Number of simultaneous asynchronous I/O

As can be seen in the above, all constants used as a *sysconf* argument value have the _SC_ prefix. Similarly, the *flimit_name* argument value is a manifested constant defined in the <unistd.h> header. These constants all have the _PC_ prefix. The following lists some of these constants and their corresponding return values from either *pathconf* or *fpathconf* for a named file object:

Limit value	***pathconf* return data**
_PC_CHOWN_RESTRICTED	The _POSIX_CHOWN_RESTRICTED value
_PC_NO_TRUNC	Return the _POSIX_NO_TRUNC value
_PC_VDISABLE	Return the _POSIX_VDISABLE value
_PC_PATH_MAX	Maximum length, in bytes, of a path name
_PC_LINK_MAX	Maximum number of links a file may have
_PC_NAME_MAX	Maximum length, in bytes, of a file name
_PC_PIPE_BUF	Maximum size of a block of data that may be auto-

	matically read from or written to a pipe file
_PC_MAX_CANON	Maximum size, in bytes, of a terminal's canonical input queue
_PC_MAX_INPUT	Maximum capacity, in bytes, of a terminal's input queue

These variables parallel their corresponding variables as defined on most UNIX systems (the UNIX variable names are the same as those of POSIX, but without the _POSIX_ prefix). These variables may be used at compile time, such as the following:

```
char pathname [ _POSIX_PATH_MAX + 1];
for (int i=0; i < _POSIX_OPEN_MAX; i++)
   close (i);                                   // close all file descriptors
```

The following *test_config.C* program illustrates the use of *sysconf, pathconf,* and *fpathconf:*

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>
int main()
{
      int res;
      if ((res=sysconf(_SC_OPEN_MAX))==-1)
         perror("sysconf");
      else cout << "OPEN_MAX: " << res << endl;

      if ((res=pathconf("/",_PC_PATH_MAX))==-1)
         perror("pathconf");
      else cout << "Max path name: " << (res+1) << endl;

      if ((res=fpathconf(0,_PC_CHOWN_RESTRICTED))==-1)
         perror("fpathconf");
      else
         cout << "chown_restricted for stdin: " << res << endl;
      return 0;
}
```

1.5 The POSIX.1 FIPS Standard

FIPS stands for Federal Information Processing Standard. The POSIX.1 FIPS standard was developed by the National Institute of Standards and Technology (NIST, formerly, the National Bureau of Standards), a department within the US Department of Commerce. The latest version of this standard, FIPS 151-1, is based on the POSIX.1-1988 standard. The POSIX.1 FIPS standard is a guideline for federal agencies acquiring computer systems. Specifically, the FIPS standard is a restriction of the POSIX.1-1988 standard, and it requires the following features to be implemented in all FIPS-conforming systems:

- Job control; the _POSIX_JOB_CONTROL symbol must be defined
- Saved set-UID and saved set-GID; the _POSIX_SAVED_IDS symbol must be defined
- Long path name is not supported; the _POSIX_NO_TRUNC should be defined - its value is not -1
- The _POSIX_CHOWN_RESTRICTED must be defined - its value is not -1. This means only an authorized user may change ownership of files, system-wide
- The _POSIX_VDISABLE symbol must be defined - its value is not equal to -1
- The NGROUP_MAX symbol's value must be at least 8
- The read and write API should return the number of bytes that have been transferred after the APIs have been interrupted by signals
- The group ID of a newly created file must inherit the group ID of its containing directory

The FIPS standard is a more restrictive version of the POSIX.1 standard. Thus, a FIPS 151-1 conforming system is also POSIX.1-1988 conforming, but not vice versa. The FIPS standard is outdated with respect to the latest version of the POSIX.1, and it is used primarily by US federal agencies. This book will, therefore, focus more on the POSIX.1 standard than on FIPS.

1.6 The X/Open Standards

The X/Open organization was formed by a group of European companies to propose a common operating system interface for their computer systems. The organization published the *X/Open Portability Guide,* issue 3 (XPG3) in 1989, and issue 4 (XPG4) in 1994. The portability guides specify a set of common facilities and C application program interface functions to be provided on all UNIX-based "open systems." The XPG3 [9] and XPG4 [10] are based on ANSI-C, POSIX.1, and POSIX.2 standards, with additional constructs invented by the X/Open organization.

In addition to the above, in 1993 a group of computer vendors (e.g., Hewlett-Packard, International Business Machines, Novell, Open Software Foundation, and Sun Microsystems, Inc.) initiated a project called *Common Open Software Environment* (COSE). The goal of the project was to define a single UNIX programming interface specification that would be supported by all the vendors. This specification is known as *Spec 1170* and has been incorporated into XPG4 as part of the X/Open Common Application Environment (CAE) specifications.

The X/Open CAE specifications have a much broader scope than do the POSIX and ANSI-C standards. This means applications that conform to ANSI-C and POSIX also conform to the X/Open standards, but not necessarily vice versa. Furthermore, though most computer vendors and independent software vendors (ISVs) adopted POSIX and ANSI-C, some of them have yet to conform to the X/Open standards. Thus, this book will focus primarily on the common UNIX system programming interface and the ANSI-C and POSIX standards. Readers may consult more detailed publications [4,5] for further information on the X/Open CAE specifications.

1.7 Summary

This chapter gave an overview of the various standards that are applicable to UNIX system programmers. The objective is to familiarize readers with these standards and to help readers understand the benefits they provide. The details of these standards and their corresponding functions and APIs, as provided on most UNIX systems, are described in the rest of the book.

1.8 References

[1]. O-J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA Common Base Language,* 1970.

[2]. Bjarne Stroustrup, *The C++ Programming Language,* Second Edition, 1991.

[3]. Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual,* Addison-Wesley, 1990.

[4]. Andrew Koenig, *Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++ (Committees: WG21/N0414, X3J16/94-0025*), 1994.

[5]. American National Standard Institute, *American National Standard for Information Systems - Programming Language C, X3.159 - 1989*, 1989.

[6]. Institute of Electrical and Electronics Engineers, *Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language], IEEE 1003.1.* 1990.

[7]. Institute of Electrical and Electronics Engineers, *Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language] - Amendment: Real-Time Extension, IEEE 1003.1b.* 1993.

[8]. Institute of Electrical and Electronics Engineers, *Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language] - Amendment: Thread Extension, IEEE 1003.1c.* 1995.

[9]. X/Open, *X/Open Portability Guide,* Prentice Hall, 1989.

[10]. X/Open, *X/Open CAE Specification, Issue 4*, Prentice Hall, 1994.

CHAPTER 6

UNIX Files

Files are the building blocks of any operating system, as most operations in a system invariably deal with files. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory, and creates a process to execute the command on your behalf. Furthermore, in the course of execution, a process may read from or write to files. All these operations involve files. Thus, the design of an operating system always begins with an efficient file management system.

Files in UNIX and POSIX systems cover a wide range of file types. These include text files, binary files, directory files, and device files. Furthermore, UNIX and POSIX systems provide a set of common system interfaces to files, such that they can be handled in a consistent manner by application programs. This, in turn, simplifies the task of developing application programs on those systems.

This chapter will explore the various file types in UNIX and POSIX systems and will show how they are created and used. Moreover, there is a set of common file attributes that an operating system keeps for each file in the system -- these attributes and their uses are explained in detail. Finally, the UNIX System V kernel and process-specific data structures used to support file manipulation are described to tie in the system call interface for files. The UNIX and POSIX.1 system calls for file handling are discussed in the next chapter.

6.1 File Types

A file in a UNIX or POSIX system may be one of the following types:

- Regular file
- Directory file
- FIFO file
- Character device file
- Block device file

A ***regular file*** may be either a text file or a binary file. UNIX and POSIX systems do not make any distinction between these two file types, and both may be "executable", provided that the execution rights of these files are set and that these files may be read or written to by users with the appropriate access permission.

Regular files may be created, browsed through, and modified by various means such as text editors or compilers, and they can be removed by specific system commands (e.g., *rm* in UNIX).

A ***directory file*** is like a file folder that contains other files, including subdirectory files. It provides a means for users to organize their files into some hierarchical structure based on file relationship or uses. For example, the UNIX */bin* directory contains all system executable-programs, such as *cat, rm, sort*, etc.

A directory may be created in UNIX by the *mkdir* command. The following UNIX command will create the */usr/foo/xyz* directory if it does not exist:

```
mkdir   /usr/foo/xyz
```

A UNIX directory is considered empty if it contains no other files except the "." and ".." files, and it may be removed via the *rmdir* command. The following UNIX command removes the */usr/foo/xyz* directory if it exists:

```
rmdir   /usr/foo/xyz
```

The content of a directory file may be displayed in UNIX by the *ls* command.

A ***block device file*** represents a physical device that transmits data a block at a time. Examples of block devices are hard disk drives and floppy disk drives. A ***character device file,*** on the other hand, represents a physical device that transmits data in a character-based manner. Examples of character devices are line printers, modems, and consoles. A physical device may have both block and character device files representing it for different access

methods. For example, a character device file for a hard disk is used to do raw (nonblocking) data transfer between a process and the disk.

An application program may perform read and write operations on a device file in the same manner as on a regular file, and the operating system will automatically invoke an appropriate device driver function to perform the actual data transfer between the physical device and the application.

Note that a physical device may have both a character and a block device file refer to it, so that an application program may choose to transfer data with that device by either a character-based (via the character device file) or block-based (via the block device file) method.

A device file is created in UNIX via the *mknod* command. The following UNIX command creates a character device file with the name */dev/cdsk0*, and the major and minor numbers of the device file are 115 and 5, respectively. The argument *c* specifies that the file to be created is a character device file:

```
mknod    /dev/cdsk    c    115    5
```

A major device number is an index to a kernel table that contains the addresses of all device driver functions known to the system. Whenever a process reads data from or writes data to a device file, the kernel uses the device file's major number to select and invoke a device driver function to carry out the actual data transfer with a physical device. A minor device number is an integer value to be passed as an argument to a device driver function when it is called. The minor device number tells the device driver function what actual physical device it is talking to (a driver function may serve multiple physical device types), and the I/O buffering scheme to be used for data transfer.

Device driver functions are supplied either by physical device vendors or by operating system vendors. Whenever a device driver function is installed to a system, the operating system kernel will require reconfiguration. This scheme allows an operating system to be extended at any customer site to handle any new device type preferred by users.

A block device file is also created in UNIX by the *mknod* command, except that the second argument to the *mknod* command will be *b* instead of *c*. The *b* argument specifies that the file to be created is a block device file. The following command creates a */dev/bdsk* block device file with the major and minor device numbers of 287 and 101, respectively:

```
mknod    /dev/bdsk    b    287    101
```

In UNIX, *mknod* must be invoked through superuser privileges. Furthermore, it is conventional in UNIX to put all device files in either the */dev* directory or a subdirectory beneath it.

A ***FIFO file*** is a special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer. Unlike regular files, however, the size of the buffer associated with a FIFO file is fixed to PIPE_BUF. (PIPE_BUF and its POSIX.1 minimum value, _POSIX_PIPE_BUF, are defined in the <limits.h> header). A process may write more than PIPE_BUF bytes of data to a FIFO file, but it may be blocked when the file buffer is filled. In this case the process must wait for a reader process to read data from the pipe and make room for the write operation to complete. Finally, data in the buffer is accessed in a first-in-first-out manner, hence the file is called a FIFO.

The buffer associated with a FIFO file is allocated when the first process opens the FIFO file for read or write. The buffer is discarded when all processes which are connected to the FIFO close their references (e.g., stream pointers) to the FIFO file. Thus the data stored in a FIFO buffer is temporary; they last as long as there is one process which has a direct connection to the FIFO file for data access.

A FIFO file may be created in UNIX via the *mkfifo* command. The following command creates a FIFO file called */usr/prog/fifo_pipe* if it does not exist:

```
mkfifo      /usr/prog/fifo_pipe
```

In some early versions of UNIX (e.g., UNIX System V.3), FIFO files were created via the *mknod* command. The following UNIX command creates the */usr/prog/fifo_pipe* FIFO file if it does not exist:

```
mknod      /usr/prog/fifo_pipe      p
```

The UNIX System V.4 supports both the *mknod* and *mkfifo* commands, whereas BSD UNIX supports only the *mkfifo* command to create FIFO files.

A FIFO file may be removed like any regular file. Thus FIFO files can be removed in UNIX via the *rm* command.

Beside the above file types, BSD UNIX and UNIX System V.4 also define a ***symbolic link file*** type. A symbolic link file contains a path name which references another file in either the local or a remote file system. POSIX.1 does not yet support symbolic link file type, although it has been proposed to be added to the standard in a future revision.

A symbolic link may be created in UNIX via the *ln* command. The following command creates a symbolic link */usr/mary/slink* which references the file */usr/jose/original*. The *cat* command which follows will print the content of the */usr/jose/original* file:

```
ln   -s   /usr/jose/original     /usr/mary/slink
cat  -n   /usr/mary/slink
```

The path name referenced by a symbolic link may be depicted in UNIX via the *ls -l* command on the symbolic link file. The following command will show that */usr/mary/slink* is a symbolic link to the */usr/jose/original* file:

```
%   ls   -l   /usr/mary/slink
sr--r--r--   1   terry      20 Aug 20, 1994    slink -> /usr/jose/original
%
```

It is possible to create a symbolic link to reference another symbolic link. When symbolic links are supplied as arguments to the UNIX commands *vi, cat, more, head, tail*, etc., these commands will dereference the symbolic links to access the actual files that the links reference. However, the UNIX commands *rm, mv*, and *chmod* will operate only on the symbolic link arguments directly and not on the files that they reference.

6.2 The UNIX and POSIX File Systems

Files in UNIX or POSIX systems are stored in a tree-like hierarchical file system. The root of a file system is the root directory, denoted by the "/" character. Each intermediate node in a file system tree is a directory file. The leaf nodes of a file system tree are either empty directory files or other types of files.

The absolute path name of a file consists of the names of all the directories, specified in the descending order of the directory hierarchy, starting from "/," that are ancestors of the file. Directory names are delimited by the "/" characters in a path name. For example, if the path name of a file is */usr/xyz/a.out*, it means that the file *a.out* is located in a directory called *xyz*, and the *xyz* directory is, in turn, stored in the *usr* directory. Furthermore, the *usr* directory is in the "/" directory.

A relative path name may consist of the "." and ".." characters. These are references to the current and parent directories, respectively. For example, the path name *../../.login* denotes a file called *.login,* which may be found in a directory two levels up from the current directory. Although POSIX.1 does not require a directory file to contain "." and ".." files, it does specify that relative path names with "." and ".." characters be interpreted in the same manner as in UNIX.

A file name may not exceed NAME_MAX characters, and the total number of characters of a path name may not exceed PATH_MAX. The POSIX.1-defined minimum values for

NAME_MAX and PATH_MAX are _POSIX_NAME_MAX and _POSIX_PATH_MAX, respectively. These are all defined in the <limits.h> header.

Furthermore, POSIX.1 specifies the following character set is to be supported by all POSIX.1-compliant operating systems as legal file name characters. This means application programs that are to be ported to POSIX.1 and UNIX systems should manipulate files with names in the following character set only:

```
A to Z    a to z    0 to 9    _    -    .
```

The path name of a file is called a *hard link*. A file may be referenced by more than one path name if a user creates one or more hard links to the file using the UNIX *ln* command. For example, the following UNIX command creates a new hard link */usr/prog/new/n1* for the file */usr/foo/path1*. After the *ln* command, the file can be referenced by either path name.

```
ln    /usr/foo/path1    /usr/prog/new/n1
```

Note that if the *-s* option is specified in the above command, the /usr/prog/n1 will be a symbolic link instead of a hard link. The differences between hard and symbolic links will be explained in Chapter 7.

The following files are commonly defined in most UNIX systems, although they are not mandated by POSIX.1:

File	Use
/etc	Stores system administrative files and programs
/etc/passwd	Stores all user information
/etc/shadow	Stores user passwords (For UNIX System V only)
/etc/group	Stores all group information
/bin	Stores all the system programs like cat, rm, cp, etc.
/dev	Stores all character and block device files
/usr/include	Stores standard header files
/usr/lib	Stores standard libraries
/tmp	Stores temporary files created by programs

6.3 The UNIX and POSIX File Attributes

Both UNIX and POSIX.1 maintain a set of common attributes for each file in a file system. These attributes and the data they specify are:

Attribute	Value meaning
file type	Type of file
access permission	The file access permission for owner, group, and others
Hard link count	Number of hard links of a file
UID	The file owner user ID
GID	The file group ID
file size	The file size in bytes
last access time	The time the file was last accessed
last modify time	The time the file was last modified
last change time	The time the file access permission, UID, GID, or hard link count was last changed
inode number	The system inode number of the file
file system ID	The file system ID where the file is stored

Most of the above information can be depicted in UNIX by the *ls -l* command on any files.

The above attributes are essential for the kernel to manage files. For example, when a user attempts to access a file, the kernel matches the user's UID and GID against those of the file to determine which category (user, group, or others) of access permission should be used for the access privileges of the user. Furthermore, the last modification time of files is used by the UNIX *make* utility to determine which source files are newer than their corresponding executable files and require recompilation.

Although the above information is stored for all file types, not all file types make use of the information. For example, the file size attribute has no meaning for character and block device files.

In addition to the above attributes, UNIX systems also store the major and minor device numbers for each device file. In POSIX.1, the support of device files is implementation-dependent; thus, it does not specify major and minor device numbers as standard attributes for device files.

All the above attributes are assigned by the kernel to a file when it is created. Some of these attributes will stay unchanged for the entire life of the file, whereas others may change as the file is being used. The attributes that are constant for any file are:

- File type
- File inode number

- File system ID
- Major and minor device number (for device files on UNIX systems only)

The other attributes are changed by the following UNIX commands or system calls:

UNIX command	System call	Attributes changed
chmod	chmod	Changes access permission, last change time
chown	chown	Changes UID, last change time
chgrp	chown	Changes GID, last change time
touch	utime	Changes last access time, modification time
ln	link	Increases hard link count
rm	unlink	Decreases hard link count. If the hard link count is zero, the file will be removed from the file system
vi, emac	-	Changes file size, last access time, last modification time

6.4 Inodes in UNIX System V

Two of the file attributes which were mentioned but not explained in the above are the inode number and the file system ID. One may also notice that file names are not part of the attributes which an operating system keeps for files. This section will use UNIX System V as the context to give answers to all these puzzles.

In UNIX System V, a file system has an inode table which keeps tracks of all files. Each entry of the inode table is an inode record which contains all the attributes of a file, including an unique inode number and the physical disk address where the data of the file is stored. Thus if a kernel needs to access information of a file with an inode number of, say 15, it will scan the inode table to find an entry which contains an inode number of 15, in order to access the necessary data. Since an operating system may have access to multiple file systems at one time (they are connected to the operating system via the *mount* system command, and each is assigned an unique file system ID), and an inode number is unique within a file system only, a file inode record is identified by a file system ID and an inode number.

An operating system does not keep the name of a file in its inode record, because the mapping of file names to inode numbers is done via directory files. Specifically, a directory file contains a list of names and their respective inode numbers for all files stored in that directory. For example, if a directory *foo* contains files *xyz, a.out*, and *xyz_ln1*, where *xyz_ln1* is a hard link of *xyz*, the content of the directory *foo* is shown in Figure 6.1 (most implementation-dependent data is omitted).

To access a file, for example /usr/joe, the UNIX kernel always knows the "/" directory inode number of any process (it is kept in a process U-area and may be changed via the *chdir* system call). It will scan the "/" directory file (via the "/" inode record) to find the inode number of the *usr* file. Once it gets the *usr* file inode number, it checks that the calling process has permission to search the *usr* directory and accesses the content of the *usr* file. It then looks for the inode number of the *joe* file.

Whenever a new file is created in a directory, the UNIX kernel allocates a new entry in the inode table to store the information of the new file. Moreover, it will assign a unique inode number to the file and add the new file name and inode number to the directory file that contains it.

inode number	*file name*
115	.
89	..
201	*xyz*
346	*a.out*
201	*xyz_ln1*

Figure 6.1 A sample directory file content

Inode numbers and file system IDs are defined in POSIX.1, but the uses of these attributes are implementation-dependent. Inode tables are kept in their file systems on disk, but the UNIX kernel maintains an in-memory inode table to contain a copy of the recently accessed inode records.

6.5 Application Program Interface to Files

Both UNIX and POSIX systems provide an application interface similar to files, as follows:

- Files are identified by path names
- Files must be created before they can be used. The UNIX commands and corresponding system calls to create various types of files are:

File type	UNIX command	UNIX and POSIX.1 system call
Regular files	vi, ex, etc.	open, creat
Directory files	mkdir	mkdir, mknod
FIFO files	mkfifo	mkfifo, mknod
Device files	mknod	mknod
Symbolic links	ln -s	symlink

- Files must be opened before they can be accessed by application programs. UNIX and POSIX.1 define the *open* API, which can be used to open any files. The *open* function returns an integer file descriptor, which is a file handle to be used in other system calls to manipulate the open file
- A process may open at most OPEN_MAX files of any types at any one time. The OPEN_MAX and its POSIX.1-defined minimum value _POSIX_OPEN_MAX are defined in the <limits.h> header
- The *read* and *write* system calls can be used to read data from and write data to opened files
- File attributes can be queried by the *stat* or *fstat* system call
- File attributes can be changed by the *chmod*, *chown*, *utime*, and *link* system calls
- File hard links can be removed by the *unlink* system call

To facilitate the query of file attributes by application programs, UNIX and POSIX.1 define a *struct stat* data type in the <sys/stat.h> header. A *struct stat* record contains all the user-visible attributes of any file being queried, and it is assigned and returned by the *stat* or *fstat* function. The POSIX.1 declaration of the *struct stat* type is:

```
struct stat
{
        dev_t     st_dev;     /* file system ID */
        ino_t     st_ino;     /* File inode number */
        mode_t    st_mode;    /* Contains file type and access flags */
        nlink_t   st_nlink;   /* Hard link count */
        uid_t     st_uid;     /* File user ID */
        gid_t     st_gid;     /* File group ID */
        dev_t     st_rdev;    /* Contains major and minor device numbers */
        off_t     st_size;    /* File size in number of bytes */
        time_t    st_atime;   /* Last access time */
        time_t    st_mtime;   /* Last modification time */
        time_t    st_ctime;   /* Last status change time */
};
```

If the path name (or file descriptor) of a symbolic link file is passed as an argument to a *stat* (or *fstat*) system call, the function will resolve the link reference and show the attributes of the actual file to which the link refers. To query the attributes of a symbolic link file itself, one can use the *lstat* system call instead. Because symbolic link files are not yet supported by POSIX.1, the *lstat* system call is also not a POSIX.1 standard.

6.6 UNIX Kernel Support for Files

In UNIX System V.3, the kernel has a file table that keeps track of all opened files in the system. There is also an inode table that contains a copy of the file inodes most recently accessed.

When a user executes a command, a process is created by the kernel to carry out the command execution. The process has its own data structure which, among other things, is a file descriptor table. The file descriptor table has OPEN_MAX entries, and it records all files opened by the process. Whenever the process calls the *open* function to open a file for read and/or write, the kernel will resolve the path name to the file inode. If the file inode is not found or the process lacks appropriate permissions to access the inode data, the *open* call fails and returns a -1 to the process. If, however, the file inode is accessible to the process, the kernel will proceed to establish a path from an entry in the file descriptor table, through a file table, onto the inode for the file being opened. The process for that is as follows:

1. The kernel will search the process file descriptor table and look for the first unused entry. If an entry is found, that entry will be designated to reference the file. The index to the entry will be returned to the process (via the return value of the *open* function) as the file descriptor of the opened file.

2. The kernel will scan the file table in its kernel space to find an unused entry that can be assigned to reference the file.

   If an unused entry is found, the following events will occur:

   a. The process's file descriptor table entry will be set to point to this file table entry.
   b. The file table entry will be set to point to the inode table entry where the inode record of the file is stored.
   c. The file table entry will contain the current file pointer of the open file. This is an offset from the beginning of the file where the next read or write operation will occur .
   d. The file table entry will contain an open mode that specifies that the file is opened for read-only, write-only, or read and write, etc. The open mode is specified from the calling process as an argument to the *open* function call.
   e. The reference count in the file table entry is set to 1. The reference count keeps

track of how many file descriptors from any process are referencing the entry.

f. The reference count of the in-memory inode of the file is increased by 1. This count specifies how many file table entries are pointing to that inode.

If either (1) or (2) fails, the *open* function will return with a -1 failure status, and no file descriptor table or file table entry will be allocated.

Figure 6.2 shows a process's file descriptor table, the kernel file table, and the inode table after the process has opened three files: *xyz* for read-only, *abc* for read-write, and *abc* again for write-only.

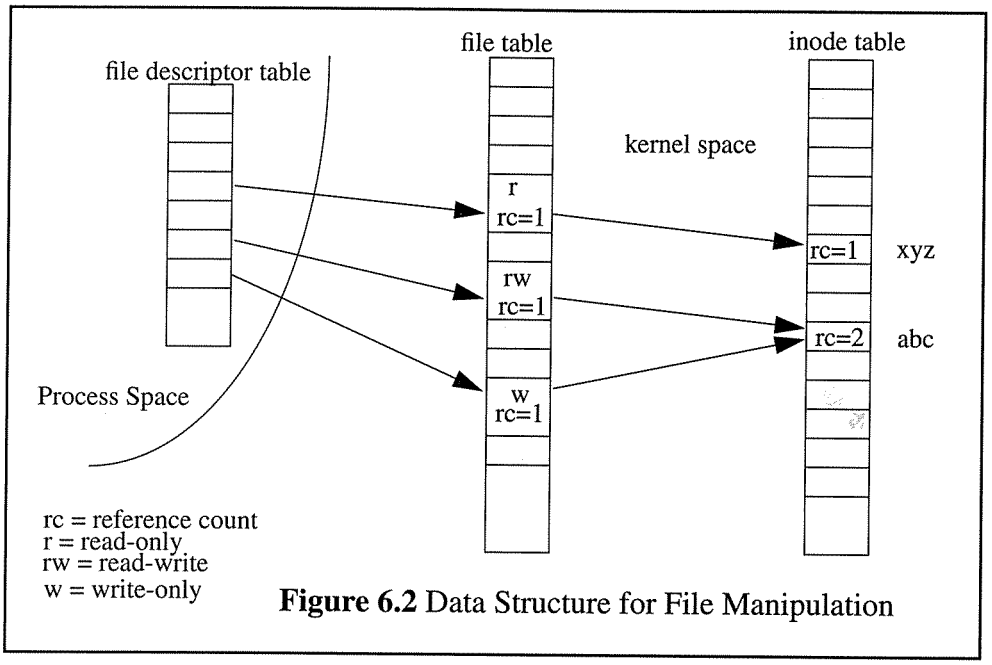


Figure 6.2 Data Structure for File Manipulation

Note that the reference count of an allocated file table entry is usually 1, but a process may use the *dup* (or *dup2*) function to make multiple file descriptor table entries point to the same file table entry. Alternatively, the process may call the *fork* function to create a child process, such that the child and parent process file table entries are pointing to corresponding file table entries at the same time. All these will cause a file table entry reference count to be larger than 1. The *dup, dup2*, and *fork* functions and their uses will be explained in more detail in Chapter 8.

The reference count in a file inode record specifies how many file table entries are pointing to the file inode record. If the count is not zero, it means that one or more processes are currently opening the file for access.

Once an *open* call succeeds, the process can use the returned file descriptor for future reference. Specifically, when the process attempts to read (or write) data from the file, it will use the file descriptor as the first argument to the *read* (or *write*) system call. The kernel will use the file descriptor to index the process's file descriptor table to find the file table entry of the opened file. It then checks the file table entry data to make sure that the file is opened with the appropriate mode to allow the requested read (or write) operation.

If the read (or write) operation is found compatible with the file's open mode, the kernel will use the pointer specified in the file table entry to access the file's inode record (as stored in the inode table). Furthermore, it will use the file pointer stored in the file table entry to determine where the read (or write) operation should occur in the file. Finally, the kernel checks the file's file type in the inode record and invokes an appropriate driver function to initiate the actual data transfer with a physical device.

If a process calls the *lseek* system call to change the file pointer to a different offset for the next read (or write) operation, the kernel will use the file descriptor to index the process file descriptor table to find the pointer to the file table entry. The kernel then accesses the file table entry to get the pointer to the file's inode record. It then checks that the file is not a character device file, a FIFO file, or a symbolic link file, as these files allow only sequential read and write operations. If the file type is compatible with *lseek*, the kernel will change the file pointer in the file table entry according to the value specified in the *lseek* arguments.

When a process calls the *close* function to close an opened file, the sequence of events are as follows:

1. The kernel sets the corresponding file descriptor table entry to be unused.

2. It decrements the reference count in the corresponding file table entry by 1. If the reference count is still non-zero, go to 6.

3. The file table entry is marked as unused.

4. The reference count in the corresponding file inode table entry is decrement by one. If the count is still nonzero, go to 6.

5. If the hard-link count of the inode is not zero, it returns to the caller with a success status. Otherwise it marks the inode table entry as unused and deallocates all the physical disk storage of the file, as all the file path names have been removed by some process.

6. It returns to the process with a 0 (success) status.

6.7 Relationship of C Stream Pointers and File Descriptors

C stream pointers (FILE*) are allocated via the *fopen* C function call. A stream pointer is more efficient to use for applications doing extensive sequential read from or write to files, as the C library functions perform I/O buffering with streams. On the other hand, a file descriptor, allocated by an *open* system call, is more efficient for applications that do frequent random access of file data, and I/O buffering is not desired. Another difference between the two is stream pointers is supported on all operating systems, such as VMS, CMS, DOS, and UNIX, that provide C compilers. File descriptors are used only in UNIX and POSIX.1-compliant systems; thus, programs that use stream pointers are more portable than are those using file descriptors.

To support stream pointers, each UNIX process has a fixed-size stream table with OPEN_MAX entries. Each entry is of type FILE and contains working data from an open file. Data stored in a FILE record includes a buffer for I/O data buffering, the file I/O error status, and an end-of-file flag, etc. When *fopen* is called, it scans the calling process FILE table to find an unused entry, then assigns this entry to reference the file and returns the address of this entry (FILE*) as the stream pointer for the file. Furthermore, in UNIX, the *fopen* function calls the *open* function to perform the actual file opening, and a FILE record contains a file descriptor for the open file. One can extract the file descriptor associated with a stream pointer via the *fileno* macro, which is declared in the <stdio.h> header:

```
int fileno ( FILE* stream_pointer );
```

Thus, if a process calls *fopen* to open a file for access, there will be an entry in the process FILE table and an entry in the process's file descriptor table being used to reference the file. If the process calls *open* to open the file, only an entry in the process's file descriptor table is assigned to reference the file. However, one can convert a file descriptor to a stream pointer via the *fdopen* C library function:

```
FILE* fdopen ( int file_descriptor, char * open_mode );
```

The *fdopen* function has an action similar to the *fopen* function, namely, it assign a process FILE table entry to reference the file, records the file descriptor value in the entry, and returns the address of the entry to the caller.

After either the *fileno* or *fdopen* call, the process may reference the file via either the stream pointer or the file descriptor. Other C library functions for files also rely on the operat-

ing system APIs to perform the actual functions. The following lists some C library functions and the underlying UNIX APIs they use to perform their functions:

C Library function	UNIX system call used
fopen	open
fread, fgetc, fscanf, fgets	read
fwrite, fputc, fprintf, fputs	write
fseek, ftell, frewind	lseek
fclose	close

6.8 Directory Files

A directory is a record-oriented file. Each record contains the information of a file residing in that directory. The record data type is *struct dirent* in UNIX System V and POSIX.1, and *struct direct* in BSD UNIX. The record content is implementation-dependent, but in UNIX and POSIX systems they all contain two essential member fields: a file name and an inode number. The usage of directory files is to map file names to their corresponding inode numbers so that an operating system can resolve any file path name to locate its inode record.

Although an application can use the *open, read, write, lseek*, and *close* system calls to manipulate directory files, UNIX and POSIX.1 define a set of portable functions to open, browse, and close directory files. They are built on top of the *open, read, write,* and *close* system calls and are defined in <dirent.h> for UNIX System V and POSIX.1-compliant systems or in <sys/dir.h> for BSD UNIX:

Directory function	Purpose
opendir	Opens a directory file
readdir	Reads the next record from file
closedir	Closes a directory file
rewinddir	Sets file pointer to beginning of file

The *opendir* function returns a handle of type DIR*. It is analogous to the FILE* handle for a C stream file. The handle is used in the *readdir, rewinddir*, and *closedir* function calls to specify which directory file to manipulate.

Besides the above functions, UNIX systems also define the *telldir* and *seekdir* functions for random access of different records in a directory file. These functions are not POSIX.1 standard, and they are analogous to the *ftell* and *fseek* C library functions, respectively.

If a process adds or deletes a file in a directory file while another process has opened the file via the *opendir*, it is implementation-dependent as to whether the latter process will see the new changes via the *readdir* function. However, if the latter process does a *rewinddir* and then reads the directory via the *readdir*, according to POSIX.1, it should read the latest content of the directory file.

6.9 Hard and Symbolic Links

A hard link is a UNIX path name for a file. Most UNIX files have only one hard link. However, users may create additional hard links for files via the *ln* command. For example. the following command creates a new link call */usr/joe/book.new* for the file */usr/mary/fun.doc*:

```
ln    /usr/mary/fun.doc    /usr/joe/book.new
```

After the above command, users may refer to the same file by either */usr/joe/book.new* or */usr/mary/fun.doc*.

Symbolic links can be created in the same manner as hard links, except that you must specify the *-s* option to the *ln* command. Thus, using the above example, you can create */usr/joe/book.new* as a symbolic link instead of a hard link with the following command:

```
ln    -s  /usr/mary/fun.doc    /usr/joe/book.new
```

Symbolic links or hard links are used to provide alternative means of referencing files. For example, you are at the */usr/jose/proj/doc* directory, and you are constantly browsing the file */usr/include/sys/unistd.h*. Thus, rather than specifying the full path name */usr/include/sys/unistd.h* every time you reference it, you could define a link to that file as follows:

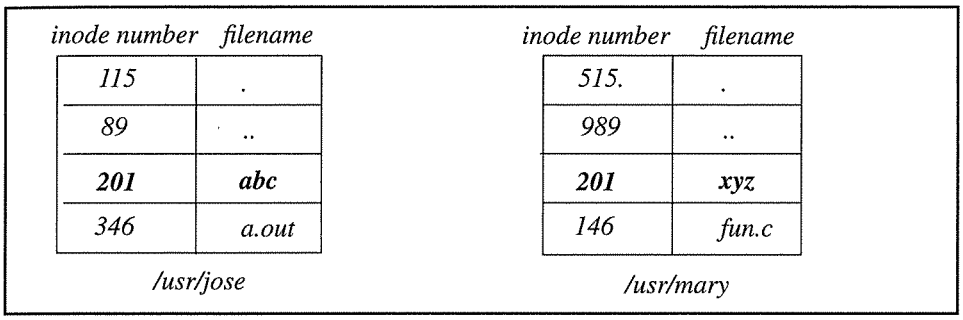
```
ln    /usr/include/sys/unistd.h    uniref
```

From now on, you can refer to that file as *uniref*. Thus links facilitate users in referencing files.

ln differs from the *cp* command in that *cp* creates a duplicated copy of a file to another file with a different path name, whereas *ln* primarily creates a new directory entry to reference a file. For example, given the following command:

```
ln    /usr/jose/abc    /usr/mary/xyz
```

the directory files */usr/jose* and */usr/mary* will contain:



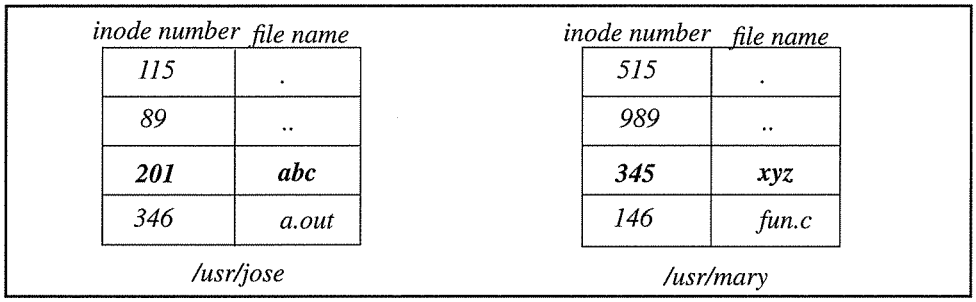
inode number	*filename*
115	.
89	..
201	***abc***
346	*a.out*

/usr/jose

inode number	*filename*
515.	.
989	..
201	***xyz***
146	*fun.c*

/usr/mary

Note that both the */usr/jose/abc* and */usr/mary/xyz* refer to the same inode number, 201. Thus there is no new file created. If, however, we use the *ln -s* or the *cp* command to create the */usr/mary/xyz* file, a new inode will be created, and the directory files of */usr/jose* and */usr/mary* will look like the following:



inode number	*file name*
115	.
89	..
201	***abc***
346	*a.out*

/usr/jose

inode number	*file name*
515	.
989	..
345	***xyz***
146	*fun.c*

/usr/mary

If the */usr/mary/xyz* file was created by the *cp* command, its data content will be identical to that of */usr/jose/abc*, and the two files will be separate objects in the file system. However, if the */usr/mary/xyz* file was created by the *ln -s* command, then the file data will consist only of the path name */usr/mary/abc*.

Thus, *ln* helps save disk space over *cp* by not creating duplicated copies of files. Moreover, whenever a user makes changes to a link (hard or symbolic) of a file, the changes are visible from all the other links of the file. This is not true for files created by *cp*, as the duplicated file is a separate object from the original.

Hard links are used in all versions of UNIX. The limitations of hard links are:

* Users cannot create hard links for directories, unless they have superuser (root) privileges. This is to prevent users from creating cyclic links in a file system. An example of a cyclic link is like the following command:

```
ln    /usr/jose/text/unix_link    /usr/jose
```

If this command succeeds, then whenever a user does a *ls -R /usr/jose*, the *ls* command will run into an infinite loop in displaying, recursively, the subdirectory tree of */usr/jose*. UNIX allows the superuser to create hard links on directories with the assumption that a supervisor will not make this kind of mistake

* Users cannot create hard links on a file system that references files on a different system. This is because a hard link is just a directory entry (in a directory file that stores the new link) to reference the same inode number as the original link, but inode numbers are unique only within a file system (hard links cannot be used to reference files on remote file systems)

To overcome the above limitations, BSD UNIX invented the symbolic link concept. A symbolic link can reference a file on any file system because its data is a path name, and an operating system kernel can resolve path names to locate files in either local or remote file systems. Furthermore, users are allowed to create symbolic links to directories, as the kernel can detect cyclic directories caused by symbolic links. Thus, there will be no infinite loops in directory traversal. Symbolic links are supported in the UNIX System V.4, but not by POSIX.1.

The following table summarizes the differences between symbolic and hard links:

Hard Link	Symbolic Link
Does not create a new inode	Create a new inode
Can not link directories, unless this is done by root	Can link directories
Can not link files across file systems	Can link files across file systems
Increase the hard link count of the linked inode	Does not change the hard link count of the linked inode

6.10 Summary

This chapter describes the UNIX and POSIX file systems and the different file types in the systems. It also depicts how these various files are created and used. Furthermore, the UNIX System V system-wide and per-process data structures that are used to support file manipulation and the application program interfaces for files are covered. The objective of this chapter is to familiarize readers with the UNIX file structures so that they can understand why the UNIX and POSIX system calls were created, how they work, and what their applications for users are.

The next chapter will describe the UNIX and POSIX file APIs in more detail.