Advanced UNIX Programming

Student Workbook

itcourseware
Training Materials For IT Professionals

ADVANCED UNIX PROGRAMMING

Jeff Howell

Published by ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO 80112

Contributing Authors: Channing Lovely and Danielle Waleri

Editor: Jan Waleri

Editorial Assistant: Ginny Jaranowski

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

Chapter 1 - Course Introduction 9

Course Objectives 10
Course Overview 12
Using the Workbook 13
Suggested References 14

Chapter 2 - UNIX Standards 17

Brief History of UNIX 18
AT&T and Berkeley UNIX Systems 20
Some Major Vendors 22
What is a Standard? 24
What is POSIX? 26
Other Industry Specifications and Standards 28
Library- vs. System-Level Functions 30
Labs 32

Chapter 3 - Files and Directories 35

Basic File Types 36
File Descriptors 38
The open() and creat() Functions 40
Keeping Track of Open Files 42
File Table Entries 44
The v-node Structure 46
The fcntl() Function 48
The fcntl() Function — with F_DUPFD Command 50
File Attributes 52
The access() Function 54
link(), unlink(), remove(), and rename() Functions 56
Functions to Create, Remove, and Read Directories 58
Labs 60

Chapter 4 - System I/O 63

Standard I/O vs System I/O 64
System I/O Calls — open() and close() 66
System I/O Calls — read() and write() 68
System I/O Calls — lseek() 70
File and Record Locking via fcntl() 72
File and Record Locking via fcntl() (cont'd) 74
Labs 76

Chapter 5 - Processes 79

What is a Process? 80
Process Creation and Termination 82
Process Memory Layout 84
Dynamic Memory Allocation 86
Accessing Environment Variables 88
Real and Effective User IDs 90
Labs 92

Chapter 6 - Process Management 95

The Difference Between Programs and Processes 96
The fork() System Function 98
Parent and Child 100
The exec() System Functions 102
Current Image and New Image 104
The wait() Functions 106
The waitpid() Function 108
Interpreter files and exec 110
Labs 112

Chapter 7 - Basic Interprocess Communication: Pipes 115

Interprocess Communication 116
Pipes 118
An Extended Example 120
FIFOs 122
More on FIFOs 124
Labs 126

Chapter 8 - Signals 129

What is a Signal? 130
Types of Signals 132
Signal Actions 134
Blocking Signals from Delivery 136
The sigaction() Function 138
Signal Sets and Operations 140
An Example 142
Sending a Signal to Another Process 144
Example 146
Blocking Signals with sigprocmask() 148
Scheduling and Waiting for Signals 150
Restarting System Calls (SVR4) 152
Signals and Reentrancy 154
Labs 156

Chapter 9 - Introduction to Pthreads 159

Processes and Threads 160
Creating Threads 162
Multi-tasking 164
Overview of Thread Archictures 166
Processes versus Threads 168
The Pthreads API 170
Thread Termination 172
Joining Threads 174
Detaching Threads 176
Passing Arguments to Threads 178
Labs 180

Chapter 10 - Pthreads Synchronization 183

The Sharing Problem 184
Mutexes 186
Creating and Initializing Mutexes 188
Using Mutexes 190
Additional Synchronization Requirement 192
Using Condition Variables 194
Labs 200

Chapter 11 - Overview of Client/Server Programming with Berkeley Sockets 203

Designing Applications for a Distributed Environment 204
Clients and Servers 206
Ports and Services 208
Connectionless vs. Connection-Oriented Servers 210
Stateless vs. Stateful Servers 212
Concurrency Issues 214
Labs 216

Chapter 12 - The Berkeley Sockets API 219

Berkeley Sockets 220
Data Structures of the Sockets API 222
Socket System Calls 224
Socket System Calls (cont'd) 226
Socket Utility Functions 228
Labs 230

Chapter 13 - TCP Client Design 233

Algorithms Instead of Details 234
Client Architecture 236
Generic Client/Server Model — TCP 238
The TCP Client Algorithm 240
Sample Socket-based Client 242
Sample Socket-based Client (cont'd) 244
Labs 246

Chapter 14 - TCP Server Design 249

General Concepts 250
Iterative Servers 252
Concurrent Servers 254
Performance Considerations 256
An Iterative Server Design 258
Iterative Server Example 260
A Concurrent Server Design 262
Labs 264

Chapter 15 - System V Interprocess Communication 267

System V IPC 268
Elements Common to msg, shm, and sem Facilities 270
The Three System V IPC Facilities 272
IPC via Message Queues — msgget() 274
IPC via Message Queues — msgctl() 276
IPC via Message Queues — msgsend() and msgrecv() 278
IPC via Shared Memory — shmget() 280
IPC via Shared Memory — shmctl() 282
IPC via Shared Memory — shmat() and shmdt() 284
Coordinating the Use of Shared Memory Segments 286
Semaphore Sets — semget() 288
Semaphore Sets — semctl() 290
Semaphore Sets — the semop() call 292
Shared Memory Coordination Using Semaphores 294
Commands for IPC Facility Handling - ipcs and ipcrm 300
Labs 302

Appendix A - Date and Time Functions 305

Overview 306
Time Representations 308
Decoding Calendar Time 310
Shorthand Functions — asctime() and ctime() 312
Formatting Date and Time Strings 314
Process Times 316
The Difference Between clock() and times() 318
Berkeley High Resolution Timer 320
Labs 322

Appendix B - Standard I/O 325

Standard I/O Calls to manipulate streams 326
Standard I/O Calls for Character I/O 328
Standard I/O Calls for String I/O 330
Standard I/O Calls for Formatted I/O 332
Standard I/O Calls for Binary I/O 334
Labs 336

Solutions 339

Index 407

Chapter 1 - Course Introduction

Course Objectives

- Develop the programming skills required to write applications that run on the UNIX operating system.

- Write portable applications using UNIX standards.

- Develop the basic skills required to write network programs using the Berkeley Sockets interface to the TCP/IP protocols.

This course is intended for experienced C programmers with user level-skills in the UNIX environment. Many programs will be written during the class. The lecture topics and lab exercises concentrate on UNIX system services, with less emphasis on application-specific subjects. The course is intended for application developers who will be using system services, as opposed to operating system "hackers" (like driver writers and other rare beasts), who create the services.

The particular applications that you will be requested to design, write, and work on during this class are intended to demonstrate the use of various system and library services provided in UNIX programming environments. The example programs and solutions to the exercises hopefully will provide some guidance when you get back to work and begin development on real projects.

A caveat: The examples and lab solutions in this course frequently neglect to check error returns from system calls and library calls, because the primary intention of the programs in this course is to teach the concepts and features available to UNIX programmers, and professional error checking code often reduces the clarity of the main point of an example. However, UNIX-specific error handling methods are explicitly discussed in the course.

We will spend some time discussing application source code portability and how standards support that goal.

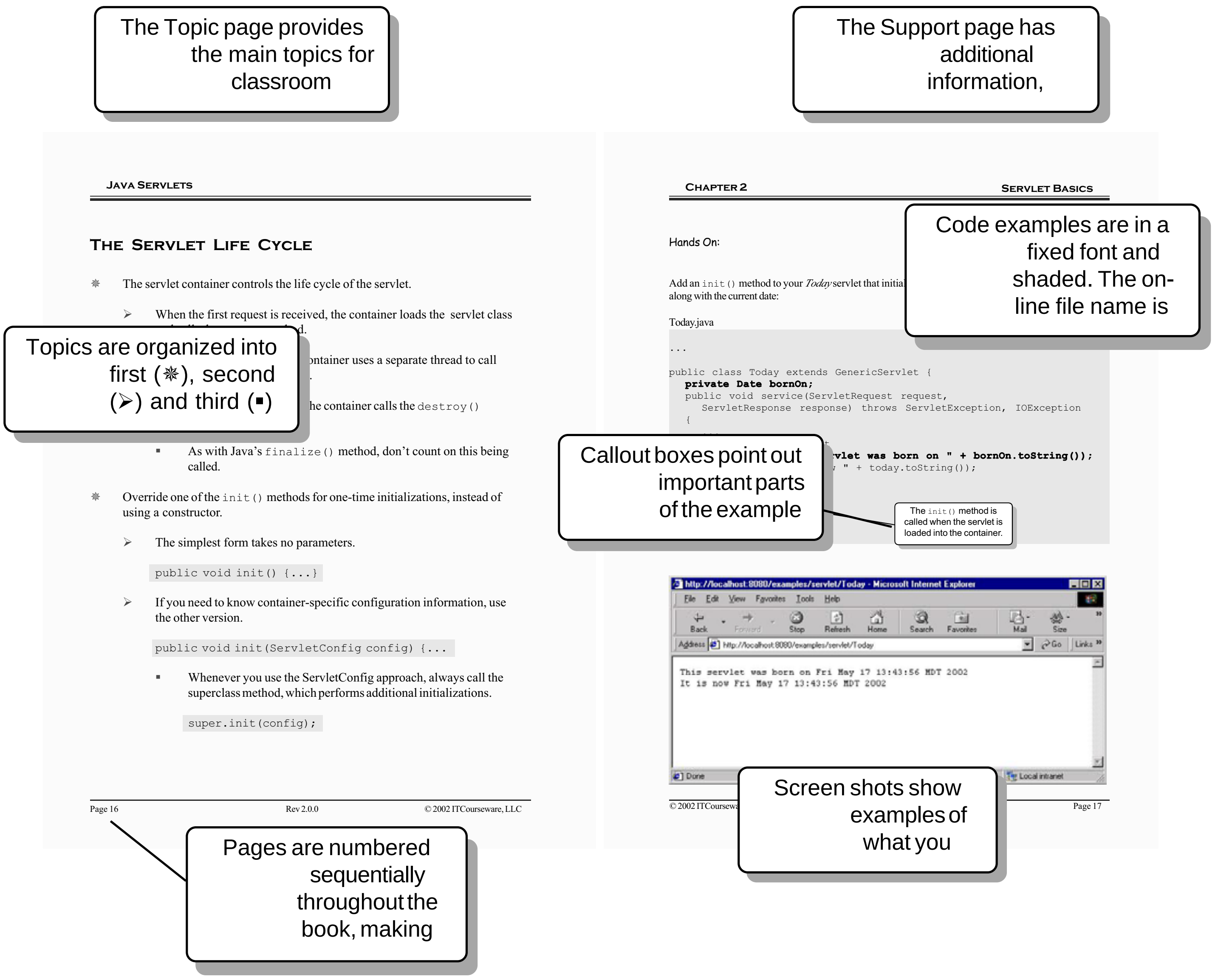
COURSE OVERVIEW

- **Audience**: This is a programming course designed for software development professionals.
- **Prerequisites**: C programming experience. User-level skills in the UNIX environment, such as file manipulation, editing, and use of utilities are also necessary.
- **Student Materials**:
  - Student workbook
- **Classroom Environment:**
  - UNIX software development system with one terminal per student.
  - UNIX and networking references.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. Hands On sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.


The Topic page provides the main topics for classroom
The Support page has additional information,
Topics are organized into first (❋), second (➢) and third (▪)
Code examples are in a fixed font and shaded. The on-line file name is
Callout boxes point out important parts of the example
The init() method is called when the servlet is loaded into the container.
Screen shots show examples of what you
Pages are numbered sequentially throughout the book, making
http://localhost:8080/examples/servlet/Today - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Mail Size
Address http://localhost:8080/examples/servlet/Today
Go Links
Done
Local intranet

JAVA SERVLETS

THE SERVLET LIFE CYCLE

❋ The servlet container controls the life cycle of the servlet.

➢ When the first request is received, the container loads the servlet class

ontainer uses a separate thread to call

he container calls the `destroy()`

▪ As with Java's `finalize()` method, don't count on this being called.

❋ Override one of the `init()` methods for one-time initializations, instead of using a constructor.

➢ The simplest form takes no parameters.

```
public void init() {...}
```

➢ If you need to know container-specific configuration information, use the other version.

```
public void init(ServletConfig config) {...
```

▪ Whenever you use the ServletConfig approach, always call the superclass method, which performs additional initializations.

```
super.init(config);
```

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC

CHAPTER 2 SERVLET BASICS

Hands On:

Add an `init()` method to your *Today* servlet that initial along with the current date:

Today.java

```
...

public class Today extends GenericServlet {
  private Date bornOn;
  public void service(ServletRequest request,
    ServletResponse response) throws ServletException, IOException
  {
```

```
vlet was born on " + bornOn.toString());
 " + today.toString());
```

```
This servlet was born on Fri May 17 13:43:56 MDT 2002
It is now Fri May 17 13:43:56 MDT 2002
```

© 2002 ITCoursewa Page 17

SUGGESTED REFERENCES

Bovett, Daniel P. and Marco Cesati. 2006. ***Understanding the Linux Kernel***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596005652.

Butenhof, David R. 1997. ***Programming with POSIX Threads***. Addison-Wesley, Reading, MA. ISBN 0201633922.

Comer, Douglas E. 2000. ***Internetworking with TCP/IP, Volume I***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0130183806.

Comer, Douglas E. and David L. Stevens. 1998. ***Internetworking with TCP/IP, Volume II***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0139738436.

Comer, Douglas E. and David L. Stevens. 2000. ***Internetworking with TCP/IP, Volumes III***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0130320714.

Gallmeister, Bill. 1995. ***POSIX.4 Programmers Guide : Programming for the Real World***. O'Reilly & Associates, Sebastopol, CA. ISBN 1565920740.

Goodheart, Berny and James Cox. 1994. ***The Magic Garden Explained: The Internals of UNIX System V Release 4***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0130981389.

Harbison, Samuel P. and Guy L. Steele, Jr. 2002. ***C: A Reference Manual***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0133262243.

Johnson, Michael K. and Erik W. Troan. 2004. ***Linux Application Development***. Addison-Wesley, Reading, MA. ISBN 0321219147.

Kernighan, Brian W. and Dennis M. Ritchie. 1988 ***The C Programming Language***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0131103628.

Lewine, Donald. 1991. ***POSIX Programmer's Guide: Writing Portable UNIX Programs***. O'Reilly & Associates, Sebastopol, CA. ISBN 0937175730.

Lewis, Bil and Daniel J. Berg. 1998. ***Multithreaded Programming with PThreads***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0136807291.

Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. 1998. ***Pthreads Programming***. O'Reilly & Associates, Sebastopol, CA. ISBN 1565921151.

Plauger, P.J. 1991. ***The Standard C Library***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0131315099.

Robbins, Kay A. and Steven Robbins. 1996. ***Practical UNIX Programming***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0134437063.

Rochkind, Marc J. 2004. ***Advanced UNIX Programming***. Addison-Wesley, Reading, MA. ISBN 0131411543.

Schimmel, Curt. 1994. ***UNIX Systems for Modern Architectures***. Addison-Wesley, Reading, MA. ISBN 0201633388.

Stevens, W. Richard and Stephen A. Rago. 2005. ***Advanced Programming in the UNIX Environment***. Addison-Wesley, Reading, MS. ISBN 0201433079.

Stevens, W. Richard, Bill Fenner, and Andrew M. Rudoff. 2003. ***UNIX Network Programming, Volume I***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0131411551.

Stevens, W. Richard. 1998. ***UNIX Network Programming, Volumes I, II***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0130810819.

X/Open Group. 1997. ***Go Solo 2: The Authorized Guide to Version 2 of the Single Unix Specification***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0135756898.

Zlotnick, F. 1991 ***The POSIX.1 Standard: A Programmer's Guide***. Benjamin Cummings, Redwood City, CA. ISBN 0805396055.

Chapter 2 - UNIX Standards

Objectives

- ❈ Write portable applications using industry standards.
- ❈ Explain the concepts of standards and open systems.
- ❈ Relate the history of the UNIX operating system to modern-day industry standards.
- ❈ Differentiate between library- and system-level functions, and when each are used.

BRIEF HISTORY OF UNIX

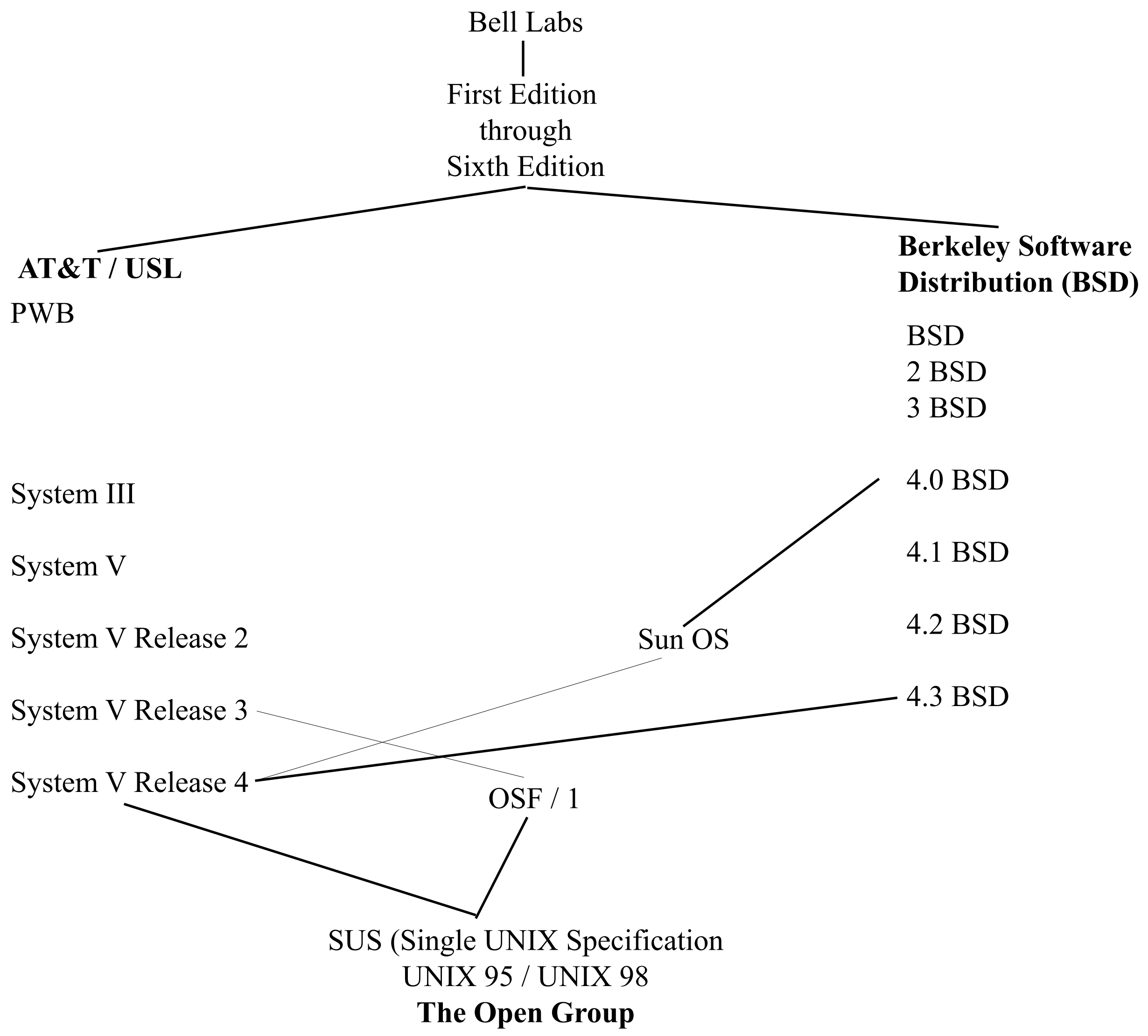
- Historically, UNIX was used for research in universities and government.
    - UNIX was distributed in source code format for many years, so many modifications were made by many different organizations.
- UNIX has been ported to many hardware platforms by vendors who provide "vendor added value" extensions or modifications.
- In the 1980s, UNIX became commercially popular for several reasons:
    - Customer demand for the benefits of open systems:
        - Application portability
        - Vendor independence
        - Connectivity/interoperability in multi-vendor environments
        - User portability
    - New workstation hardware could be brought to market more quickly with an existing operating system.
    - Major vendors (such as Sun, Digital, HP, IBM) implemented UNIX-based product lines.
    - UNIX provides excellent networking capabilities.

Many books go into detail on the history of UNIX and the reasons for its commercial popularity. For our purposes as application developers, we need to know the aspects of UNIX history that can affect application programming interfaces (API), such as the differences in system call parameters and function return codes in different versions of UNIX (i.e., Berkeley vs. SystemV).

AT&T AND BERKELEY UNIX SYSTEMS

- UNIX was originally written at Bell Laboratories in 1969. In the mid-1970s, the University of California at Berkeley began making additions and enhancements to UNIX. In the early 1980s, AT&T began offering support for AT&T System III UNIX.

Simplified UNIX Operating System History



UNIX was originally designed and written mostly by Ken Thompson, a computer science researcher, for the purpose of doing computer science research! AT&T provided UNIX source code at a low cost to many universities, including UCB. Berkeley UNIX built on the Sixth Edition, adding, over the years, many utilities such as **vi** and **csh**. Much research and development was done in the areas of file systems and networking. Again, the history is well documented in several books, such as Leffler, et al., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Reading, MA: Addison-Wesley Publishing Company, Inc., 1989.

AT&T, in 1982, merged several internal versions of UNIX and began licensing UNIX to vendors such as Hewlett-Packard. In 1985, AT&T began shipping UNIX System V, and committed to support it and maintain backward compatibility in future versions of UNIX.

Through the 1980s and early 1990s, as UNIX became critical to the strategies of more and more commercial computer companies, much activity involved controlling UNIX and attempting to use that control as a business advantage. However, influences such as the "threat" of NT and the continuing pressure for compatibly from the world of customers helped bring competing vendors together in several different inititatives.

The diagram on the facing page is oversimplified with respect to the number of actual versions and variants of UNIX and its relatives, and with respect to the cross-influencing that the various versions have had on each other.

SOME MAJOR VENDORS

- SunOS from Sun Microsystems was based on Berkeley UNIX.
  - SunOS merged with System V.3 to create UNIX System V Release 4 (SVR4 or System V.4).
  - Solaris 1.0 was based on SunOS; Solaris 2.0 and later are based on System V.4.
- HP-UX from Hewlett-Packard followed compliance with System V.3 and has all major BSD features.
  - HP-UX 10 was based on SVR4.
- AIX from IBM was based on System V.3 and incorporated many BSD features.
- UNIX System V Release 4 (SVR4) from UNIX System Laboratories is the merger of System V.3, SunOS, 4.3BSD, and XENIX.
- OSF/1 from Open Software Foundation was derived from Mach, an OS developed at Carnegie Mellon University, based on 4.2BSD.
  - OSF/1 was intended to be an "open" operating system: not controlled by any single vendor.
- Ultrix from DEC was BSD-based.
  - Later Digital UNIX was based on OSF/1.
- Almost all vendors now support versions of the Single UNIX Specification from The Open Group.

WHAT IS A STANDARD?

- A *specification* is a document that specifies a certain technological area.
    - It tells what a software system does and how to use it as an application programmer.
    - Specifications are produced by vendors, consortia, or users.
    - A vendor programming reference manual for a system is a specification.
- A *de facto standard* is a specification that is widely used.
- A *formal standard* is a specification that is produced through a formal process by a formal standards setting body, such as ANSI and IEEE.

A specification of an API is provided to programmers so that they can write applications. But if a company spends time and money to develop an application according to a vendor-dependent specification, then that application will run only on that vendor's system.

If a specification is made publicly available by a university or a government agency, or licensed by a vendor, and different system providers implement systems according to that specification, then it may be called a de facto standard and applications written to use that specification will run on more than one vendor's system. The specification is still controlled by the single provider.

Formal standards allow companies to "leverage their investment" in applications and programmers. This is because applications can be ported to different vendor platforms without rewriting code, and programmers can be productive immediately on new platforms without being retrained. Also, a formal standard may be modified through processes that solicit input from the people who are affected by evolution of the standard. MS-DOS may be a standard, but the evolution of revisions to MS-DOS are controlled by one company.

A standard that evolves through input from users of the standard (systems providers, application developers, end users) is called an *open standard*. System implementations based on open standards are *open systems*.

WHAT IS POSIX?

- POSIX.1 defines the interface between application programs and the services provided by the operating system.
- POSIX is an API for basic operating system functions.
- POSIX is based on historical implementations of UNIX System V and Berkeley UNIX, but it is not an operating system: POSIX is a specification.
- Some of the main goals of POSIX are:
    - Source code application portability — the ability to port programs from system to system.
    - Contract specification — the interface contract between the application and the operating system.
        - No implementation details are specified for either the system or the application.
    - Keep the standard as small as possible.
    - Keep to a minimum the changes required for existing UNIX programs to meet the standard.

POSIX.1 specifies only a subset of the features available in real UNIX systems.

POSIX library routines are combined with other system libraries. Careful study of vendor documentation reveals non-standard extensions, features, and incompatibilities with standard specifications. You can certainly use non-standard vendor features, but be aware that you are doing so and design your programs for the best chances of portability (layers, wrappers, preprocessor logic, etc.).

OTHER INDUSTRY SPECIFICATIONS AND STANDARDS

❋ System V Interface Definition (SVID)

   ➢ The description of UNIX System V, originally produced by AT&T, but now owned by The Open Group.

❋ X/Open Portability Guide (XPG)

   ➢ Specifies a Common Applications Environment (CAE) intended to ensure application portability and connectivity. The CAE is now known as "Open Group Technical Standards."

❋ POSIX

   ➢ A collection of IEEE standards that specify interfaces between programs (or users) and the operating system.

❋ Standard C

   ➢ The definition of the standardized C language, defined by ANSI; sometimes called "ANSI C."

❋ The Open Group (***www.opengroup.org***) offers product branding.

   ➢ Products that have been tested and guaranteed to conform to industry-standard specifications (such as X/Open and POSIX) can receive the Open Brand.

   ➢ Vendor products that have been registered with The Open Group for branding are listed on the website.

   ➢ Conformance to the Single UNIX Specification is required for the UNIX 95 and UNIX 98 brands.

The SVID was written by AT&T in the 1980s as the definitive specification of the interfaces between applications and the UNIX System V operating system. Companies that licensed UNIX from AT&T to resell (such as HP and IBM) could claim that their major-vendor-enhanced version of UNIX was SVID-compliant by running the System V Verification Suite (SVVS). The SVID has had a strong influence on the POSIX specifications. The SVID went with USL when it was sold to Novell, and was subsequently transferred to X/Open (see below).

> X/Open Company, Ltd. was founded by several European companies in 1984. Member companies of X/Open provide input to the XPG CAE specifications. Software developed by systems vendors, independent software developers, or end user organizations will be more likely to be portable and interoperable if it complies with X/Open guidelines. X/Open is not a standards-setting organization. "It is a joint initiative by members of the business comunity to integrate evolving standards into a common, beneficial and continuing strategy. — X/Open Portability Guide (December 1988)

POSIX is language independent — it does not require the use of Standard C, but efforts were made to ensure that the runtime library routines specified by POSIX and ANSI are compatibile.

PASC is the IEEE's Portable Application Standards Committee. It is chartered with defining standard application service interfaces — most notably those in the POSIX family. PASC was formerly known as the Technical Committee on Operating Systems.

X/Open and OSF became The Open Group in 1996.

LIBRARY- VS. SYSTEM-LEVEL FUNCTIONS

- Library-level functions are used to create portable C applications.
  - These functions are usually documented in Section 3 of the online manual pages.
  - **malloc()**, **fopen()**, and **printf()** are examples of standard or library-level functions.
  - Library-level functions call system-level functions to do their work.
- System-level functions provide low-level services, such as file operations, memory manipulation, and process management.
  - These functions are usually documented in Section 2 of the online manual pages.
  - System calls are direct entry points into the operating system.
- Both library calls and system calls are specified by standards, such as POSIX and SUS.

errno.c

```
/* errno.c
 * This program demonstrates the use of malloc and errno.
 */
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void main(void) {
   char *buf;     /* Pointer to be used for malloc */
   int   fd;      /* File descriptor */

/* Use malloc to dynamically allocate 80 characters */
   buf = (char *) malloc (80 * (sizeof (char)));

   strcpy(buf, "This is in the malloc’d buffer\n");
   printf("%s", buf);

/* Use free to release the memory back to the system */
   free(buf);

/* Attempt to open a non-existent file to demo errno */
   fd = open("NOT-HERE", O_RDONLY);
   if (fd == -1) {
      switch (errno) {
      case ENOENT:
         printf("File NOT-HERE does not exist\n");
         break;
      default:
         perror("open");
         break;
      }
   }
   else
      close(fd); /* Just in case it DOES exist! */
}
```

LABS

❶ Write a program that calls the **getlogin()** function to determine your login name, then calls **getpwnam()** to get a pointer to a **passwd** structure. From the **passwd** structure, display your initial working directory and your initial shell.
(Solution: *getlogin.c*)

❷ Under some conditions **getlogin()** will return null. This will happen if the calling process is not attached to a terminal that a user logged into (such as a daemon).

Write a program to use the **getuid()** then the **getpwuid()** functions to retrieve the password structure for the user ID of the calling process.
(Solution: *getlogin2.c*)

❸ Write a function to use **getcwd()** to get the current working directory of the process and display it. The **getcwd()** function wants a pointer to a character buffer to hold the string that identifies the current working directory, and the size of the buffer. If the size passed to **getcwd()** is less than the number of characters in the directory pathname as determined by **getcwd()**, then **getcwd()** fails. The problem here is that the maximum path length allowable is implementation-dependent, so the purpose of this exercise is to demonstrate what you must go through sometimes for portability.

Suggested algorithm: Use **malloc()** to obtain a pointer to a buffer whose size is an initial guess. Call **getcwd()** with the buffer and its size, and check the return from **getcwd()**. If it fails, do a switch on the global **errno** variable to make sure that the reason for the failure was **ERANGE**, which means that the length of the pathname found by **getcwd()** is beyond the range of the size of the buffer. If that is the case, increase the size of the buffer, then try **getcwd()** again. Use **realloc()** to increase the size of the buffer.
(Solution: *getcwd.c*)

❹ Read the manual entries for **getpwent(), setpwent(),** and **endpwent()**. How do these functions work? Discuss re-entrancy issues, such as in a threaded application.

The online reference manual is separated into several sections that cover both UNIX commands and C functions. Topics are often found in more than one section of the manual. It may be that the topic relates to one or more areas of UNIX or C, or there may be UNIX commands and C functions that have the same name.

To look up a topic in a particular section, enter the section number before the topic:

```
man 3 printf
```

To search for a topic in the manual use the **-k** option. **-k** means "keyword."

```
man -k printf
```

Although vendors vary on this, the UNIX manual typically is divided into topical sections as follows:

Section 1	Commands	Section 5	Miscellaneous Facilities
Section 2	System Calls	Section 6	Games
Section 3	Library Calls	Section 7	Files and Devices
Section 4	File Formats	Section 8	System Administration

CHAPTER 8 - SIGNALS

OBJECTIVES

- ❈ Understand the concepts and uses of signals in UNIX.
- ❈ Write programs to handle signals.
- ❈ Know the effects of signals on system calls.

WHAT IS A SIGNAL?

❈ The POSIX.1 standard says a signal is, "a mechanism by which a process may be notified of, or affected by, an event occurring in the system."

❈ An event occurs and *generates* a signal.

❈ The signal is *delivered*, and the appropriate action is taken by the process in response to the signal.

❈ Between the generation and the delivery, the signal is *pending*.

❈ Typical events that generate signals are:

  ➢ The user presses the interrupt key on the keyboard.

  ➢ An alarm timer has expired and the system needs to notify the process that started the timer.

  ➢ A user types the **kill(1)** command, identifying a process to be killed.

Signals are sometimes compared to hardware interrupts because, from the receiving process' point of view, a signal is an asynchronous interruption. Some signals may come at any random time (such as when a user causes the event), or they may come in response to something the process itself does, such as a floating point error caused by attempted division by zero.

In this chapter, we will study signals as specified by POSIX.1, with brief looks at typical vendor extensions to POSIX signals.

Some examples of what signals are used for:

- Cleaning up if a process is told to terminate.
- Self-imposed timeouts on a process so that it doesn't wait forever on something that might not happen, such as waiting for input.
- Synchronizing processes by sending signals back and forth.

TYPES OF SIGNALS

- Each signal has a name, defined in *signal.h*.

- The two general types of events that cause signals are either errors or asynchronous events.

The standard signals supported by every POSIX.1 system are:

Name	Event
SIGABRT	Abnormal termination; see **abort()**
SIGALRM	Timeout; see **alarm()**
SIGFPE	Arithmetic Exception
SIGHUP	Hangup [see **termio(7)**]
SIGILL	Illegal Instruction
SIGINT	Interrupt [see **termio(7)**]
SIGKILL	Killed
SIGPIPE	Broken Pipe
SIGQUIT	Quit [see **termio(7)**]
SIGSEGV	Segmentation Fault
SIGTERM	Terminated
SIGUSR1	User Signal 1
SIGUSR2	User Signal 2

The default action for all signals in the above table is termination of the process.

POSIX.1 also supports job control signals on systems that can do job control:

Name	Event
SIGCHLD	Child process stopped or terminated
SIGCONT	Continue stopped process
SIGSTOP	Stop
SIGTSTP	User stop requested from tty
SIGTIN	Background tty read attempted
SIGTTOU	Background tty write attempted

Vendor implementations provide other signals provided by Berkeley and SVR4.

SIGNAL ACTIONS

- A process has three choices on how to handle signals that are delivered to it.
- The type of action chosen is on a per signal basis. The actions are:
  - Take the default action for the signal, which for most signals is to terminate the process.
  - Ignore the signal.
  - Catch the signal.
    - This means to tell the system (ahead of time) to execute a process-supplied, signal-handling function upon receipt of the signal.
- Note: **SIGKILL** and **SIGSTOP** cannot be ignored or caught.

If a process does not advise the system how it wants to handle a specific signal, then the default action associated with that signal will be taken if and when that signal is generated for the process. The default actions for signals are listed in the reference manual, **signal(5)**.

If the process advises the system that it wishes to ignore the signal, then the signal will be discarded if it gets generated for that process.

Catching signals is where all the action is. In this case, when a signal is delivered to a process, a function of the process' choosing is called. Since signals are asynchronous, they can come between any two instructions being executed by the process. The flow of control is interrupted, and the signal-handling function is called by the system, just like any other C function. An integer parameter containing the signal number is passed to the function. If the function returns after it executes, then the flow of control in the process will pick up where it left off.

BLOCKING SIGNALS FROM DELIVERY

- A process can choose to temporarily block signals from delivery.
- Every process has a *signal mask*, the set of signals that are currently blocked.
- If an event causes a blocked signal to be generated, the signal is called a *pending* signal.
  - A pending signal will be delivered after it is unblocked.
- Each process starts out with a signal mask inherited from its parents.
- Blocking a signal and ignoring a signal are not the same.

There may be times when a process does not wish to be interrupted by one or more specific signals. Such a time is often known as a *critical section*. The process can block signals from being delivered during critical sections, then unblock and receive signals that may have arrived during that time.

A blocked signal is one that the process has notified the system not to deliver, if such a signal was to occur. If such a signal does actually occur, then it will be added to the process' set of pending signals.

Note that a blocked signal is not an ignored signal. An ignored signal will never be delivered.

THE SIGACTION() FUNCTION

❋ The **sigaction()** function allows the calling process to examine and/or specify the action to be associated with a specific signal.

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act,
                struct sigaction *oact);
```

❋ The **sig** argument is the signal for which an action is being specified.

❋ The **act** argument is the address of a **sigaction** structure that describes the actions to be taken for **sig**.

❋ The **oact** (old action) argument is the address of a **sigaction** structure that will be filled with the previous action for **sig**.

The **sigaction** structure looks like this:

```
struct sigaction {
   void (*sa_handler)(int);   /* Signal handler function     */
   sigset_t sa_mask;          /* Extra signals to be blocked */
   int sa_flags;              /* Flags to modify delivery    */
      /* Possibly additional implementation-dependent members */
};
```

sa_handler is one of:

1. **SIG_DFL** for the default action.
2. **SIG_IGN** to ignore this signal.
3. A pointer to a signal-handling function to catch the signal.

sa_mask is a set of signals to be blocked during execution of the signal-handling function. This set is added to the process' current signal mask, the set of signals that are currently blocked, for the duration of the signal-handler execution. Also, the first parameter to the **sigaction()** function call, **sig**, is added to the signal mask for the duration of the signal-handler.

sa_flags is a set of flags that can modify how the signal is delivered. One such non-POSIX flag, **SA_RESTART**, affects whether interrupted system calls are restarted.

In the **sigaction()** function call, if **oact** is **NULL** then it is ignored. If **act** is **NULL**, then the current action for the signal is returned in **oact**.

SIGNAL SETS AND OPERATIONS

- The two primary data structures used in POSIX.1 signal handling are the **sigaction** structure and signal sets of type **sigset_t**.
- The **sa_mask** member of the **sigaction** structure is a signal set.
- Signal sets are manipulated with the following five functions:

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
int sigismember(const sigset_t *set, int sig);
```

- These five functions do not affect the process signal mask or any actions on any signals.
  - They simply manipulate signal set data structures.
- NOTE: A **sigset_t** data structure must be initialized with either **sigemptyset()** or **sigfillset()** before applying any other operation to it.

Given a signal **set** declaration in a program:

```
sigset_t set;
```

The call **sigemptyset(*&set*)** initializes the set so that all signals are excluded.

The call **sigfillset(*&set*)** initializes the set so that all signals are included.

The call **sigaddset(*&set,* SIGQUIT)** adds the **SIGQUIT** signal to the set.

The call **sigdelset(*&set,* SIGQUIT)** deletes the **SIGQUIT** signal from the set.

The call **sigismember(*&set,* SIGQUIT)** returns **1** if **SIGQUIT** is in the set, else **0**.

These library functions that manipulate the bits in a **sigset_t** structure are called ***sigsetops***, and are typically documented in **sigsetops(3)**.

AN EXAMPLE

- We have reached the point where we can show an example program that uses POSIX signals.

- In this example, we call the **alarm()** function to send a **SIGALRM** signal to the process after a few seconds.

- From the **alarm()** manual page:

```
#include <unistd.h>
unsigned alarm(unsigned sec);
```

  - **alarm()** instructs the alarm clock of the calling process to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by ***sec*** have elapsed.

  - Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

  - If **sec** is **0**, any previously-made alarm request is canceled. **fork()** sets the alarm clock of a new process to **0**.

    - The **exec** family of routines leaves the process' current alarm value unchanged, so the new program will inherit the prior alarm clock.

- **pause()** is used to wait for a signal:

```
#include <unistd.h>
int pause(void);
```

  - **pause()** will return when a signal handler executes and returns.

alarmer.c

```
#include  <unistd.h>
#include <signal.h>

void alarm_handler(int);

main(int argc, char *argv[]) {
   int seconds;
   struct sigaction sigact;

   if (argc >= 2) seconds = atoi(argv[1]);
   else seconds = 3;

   sigact.sa_handler = alarm_handler;  /* Point to the handler */
   sigemptyset(&sigact.sa_mask);       /* No signals will be
                                       blocked */
   sigact.sa_flags = 0;                /* No additional flags */
   sigaction(SIGALRM, &sigact, NULL); /* Set the action for
                      SIGALRM */
   alarm(seconds);                     /* Start the process alarm clock */
   pause();                            /* Wait for a signal */

}

void alarm_handler(int signo) {
   printf("Alarm went off! (Signal #%d)\n", signo);
}
```

SENDING A SIGNAL TO ANOTHER PROCESS

- A process can send a signal to another process or group of processes with the **kill()** function.

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int sig);
```

- The process must have permission to send the signal to the process identified by **pid**.
- Rules for permissions are:
  - The superuser can send a signal to any process.
  - If the real or effective user ID of the sender is equal to the real or effective ID of the receiver, then the signal can be sent.

Example

sync_child.c

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void handler(int);

main(int argc, char *argv[]) {
  struct sigaction sigact;
  sigset_t suspend;

  /*— Set up to call handler on SIGUSR1 —*/
  sigact.sa_handler = handler;       /* Point to the handler */
  sigemptyset(&sigact.sa_mask);      /* Block no add’l signals in handler */
  sigact.sa_flags = 0;               /* No flags */
  sigaction(SIGUSR1, &sigact, NULL);  /* Set the action for SIGUSR1 */

  sigemptyset(&suspend);             /* Set up to wait for SIGUSR1 */

  /*— Assume the child did some real work here, then wanted to let
        the parent know it was done with phase 1. —*/

  fprintf(stderr, "Child sending SIGUSR1 once ...\n");
  if (kill(getppid(), SIGUSR1) == -1) {
    perror("first kill to parent failed");
    exit(-1);
  }
  fprintf(stderr,"Child sent SIGUSR1 once.\n");

  fprintf(stderr, "Child waiting for parent\n");
  sigsuspend(&suspend);

  /*— Now the child does phase 2 then alerts the parent —*/

  fprintf(stderr,"Child sending SIGUSR1 twice.\n");
  if (kill(getppid(), SIGUSR1) == -1) {
    perror("second kill to parent failed");
    exit(-1);
  }
  fprintf(stderr,"Child sent SIGUSR1 twice.\n");
}
void handler(int signo) {
  fprintf(stderr, "Child received %d\n", signo);
}
```

sync_parent.c

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void handler(int);
int count_usr1 = 0;

main(int argc, char *argv[]) {
   struct sigaction sigact;
   sigset_t hold_off, suspend;
   int cpid;
   int child_sig_sent;

   child_sig_sent = 0;                   /* We have not sent sig. to child */

   /*— Set up to call handler on SIGUSR1 —*/
   sigact.sa_handler = handler;          /* Point to the handler */
   sigemptyset(&sigact.sa_mask);         /* No add'l signals will be blocked */
   sigact.sa_flags = 0;                  /* No additional flags */
   sigaction(SIGUSR1, &sigact, NULL);    /* Set the action for SIGUSR1 */

   /*— Set up and block SIGUSR1 from child till we're ready —*/
   sigemptyset(&hold_off);               /* Empty the hold_off mask */
   sigaddset(&hold_off, SIGUSR1);        /* Add to mask to block SIGUSR1 */
   sigprocmask(SIG_BLOCK, &hold_off, NULL); /* Block SIGUSR1 till we're ready (at
                                              sigsuspend call) */

   sigemptyset(&suspend);                /* Set up to wait for SIGUSR1 */

   if ((cpid = fork()) == -1) {
      perror("child fork failed");
      exit(-1);
   }
   if (cpid == 0) {
      execl("sync_child", "sync_child", (char *)0);
      perror("exec of sync_child failed");
      exit(-1);
   }
   while(count_usr1 < 2)
      if (count_usr1 == 1 && child_sig_sent++ == 0)
         kill(cpid, SIGUSR1);            /* Tell child to start phase 2 */
      else
         sigsuspend(&suspend);           /* Wait for SIGUSR1 from child */
}
void handler(int signo) {
   count_usr1++;
   fprintf(stderr, "Parent received %d, count_usr1 %d\n", signo, count_usr1);
}
```

BLOCKING SIGNALS WITH SIGPROCMASK()

❅ The **sigprocmask()** function is used to change or examine the signal mask of the calling process.

```
#include <signal.h>

int sigprocmask(int value, const sigset_t *set,
sigset_t *oset);
```

❅ Recall that the signal mask is the set of signals that are currently blocked from being delivered to the process.

```
#include <signal.h>
int   sigprocmask(int value, const sigset_t *set, sigset_t *oset);
```

If value == **SIG_BLOCK**

The set pointed to by **set** is added to the current signal mask.

If value == **SIG_UNBLOCK**

The set pointed to by **set** is removed from the current signal mask.

If value == **SIG_SETMASK**

The current signal mask is replaced by the set pointed to by **set**.

If oset is not **NULL**, the previous mask is stored in the space pointed to by **oset**.

If set is **NULL**, then value is not significant and the process' signal mask is unchanged; thus, the call can be used to inquire about currently-blocked signals.

SCHEDULING AND WAITING FOR SIGNALS

- The **sigsuspend()** function is used to atomically unblock one or more signals, then wait for a signal.

```
#include <signal.h>
int sigsuspend(const sigset_t *set);
```

- The power of **sigsuspend()** is that it removes race condition gaps that might result in suspending a process forever.

```
#include <signal.h>
int sigsuspend(const sigset_t *set);
```

sigsuspend() replaces the process' signal mask with the set of signals pointed to by **set** and then suspends the process until delivery of a signal whose action is either to execute a signal catching function or to terminate the process. If the action is to terminate the process, **sigsuspend()** does not return. If the action is to execute a signal catching function, **sigsuspend()** returns after the signal catching function returns. On return, the signal mask is restored to the set that existed before the call to **sigsuspend()**.

The **sigsuspend()** function is used to:

1. Unblock one or more blocked signals and then
2. Pause to wait for a signal to arrive.

It does this in one atomic step so that blocked signals can't arrive between the time they get unblocked and the time the process pauses.

Consider this situation:

... code being executed with blocked signals ...

```
/* Unblock signals */
sigprocmask(SIG_UNBLOCK, &set_with_signals_to_unblock, NULL);

/* → SIGNAL IS DELIVERED HERE !!! */
/*     (either it was pending or it */
/*      got generated right now)    */

pause();    /* pause and wait forever for
        a signal that already came
        and has been handled */
```

If we replace the **sigprocmask()** and the **pause()** with a single **sigsuspend()**, then the gap is closed:

... code being executed with blocked signals ...

```
/* Unblock and pause for signals */
sigsuspend(&temporary_sigmask);
```

Restarting System Calls (SVR4)

❈ When a signal arrives, your code is interrupted after an instruction.

❈ If the action is **SIG_DFL**, your process usually terminates.

❈ If the action is to catch it, your handler executes.

❈ After your handler finishes, your code resumes at the next instruction.

❈ What happens if a signal arrives during execution of a system call, such as a lengthy I/O operation?

Originally UNIX systems would terminate system calls that were interrupted by a signal, and return a **-1** to the process with **errno == EINTR**. This was designed to let programs be interrupted from blocked I/O calls. If a program wished to restart the system call, it had to test for **EINTR** and restart it.

Following is an example code fragment that manually restarts a system call:

```
sigact.sa_handler = handler;
sigemptyset(&sigact.sa_mask);
sigact.sa_flags = 0;
      ...
start_read:
   if (read (device, buf, count) == -1)  /* Assume read is blocked */
      if (errno == EINTR)
         goto start_read;
```

Some versions of UNIX provide a mechanism that allows system calls to be restarted after a caught signal arrives and the handler returns. POSIX.1 doesn't provide or require this mechanism, but it allows it. On SVR4, the following code fragment is equivalent to the one above:

```
sigact.sa_handler = handler;
sigemptyset(&sigact.sa_mask);
sigact.sa_flags = SA_RESTART; /* Set sys call restart flag*/
      ...
if (read (device, buf, count) == -1)
   /* Assume read is blocked */
```

SIGNALS AND REENTRANCY

- Do not call non-reentrant functions in a signal handler.
- What happens if a non-reentrant function in your program is interrupted by a signal, and then you call the same function in your handler?
- Some reasons why a function may be non-reentrant include:
  - It uses static data structures.
  - It calls **malloc()** or **free()**.
  - It uses standard I/O.
- Also be aware that any functions called in a signal handler might overwrite the value in **errno**.
  - It is advisable to save the value of **errno** at the beginning of the handler and replace it at the end.

LABS

❶ Write a program that prints "Go ahead, interrupt me" once a second, and terminates after the keyboard **SIGINT** generator key is pressed twice.
(Solution: *sigint.c*)

❷ Modify the program in ❶ above so that it changes its message to "OK, once more" after the first interrupt, but still terminates after the second interrupt.
(Solution: *sigint2.c*)

❸ Write a parent program that sets up a signal handler to catch **SIGUSR1**, then starts a child program. Have the child send **SIGUSR1** twice in a row to the parent. In the parent's signal handler, print a message when a signal arrives. Does the parent catch both signals?

Change the second signal sent by the child to be **SIGUSR2**, and update the parent to catch both **SIGUSR1** and **SIGUSR2** with the same handler. Also, in the parent's handler, print out the signal number. Are both signals caught this time? What's the difference?
(Solutions: *parentsigs.c, childsigs.c*)

❹ Modify the examples *sync_parent.c* and *sync_child.c* so that the parent forks two children who will communicate with the parent similar to the example with only one child. One child uses **SIGUSR1**, the other **SIGUSR2**. Have each program loop to keep things going for a while. Use some sleeps to slow it down so you can see the action, then remove the sleeps to speed it up to see if it works at speed.

This is a form of simple IPC; although no data is being sent between processes, they are communicating and controlling each other.
(Solutions: *syncparent.c, syncchild1.c, syncchild2.c*)

❺ Write a program that catches **SIGINT** and returns from the signal handler (as opposed to exiting the program within the handler). After setting up the handler with **sigaction()**, use **read()** to read from the standard input keyboard. In your code, check **errno** on an error return from **read()** to see if it is **EINTR**, and if so use **perror()** to print a message before dying.

Modify the program to restart the read if it is interrupted by **SIGINT**. Modify the program to use the **SA_RESTART** flag to automatically restart the read.

Test each of these versions by pressing the keyboard **SIGINT** generation key.
(Solution: *reader.c*)

CHAPTER 9 - INTRODUCTION TO PTHREADS

OBJECTIVES

- ❅ Explain the differences between processes and threads.
- ❅ Describe user-space threads versus kernel threads.
- ❅ Decide when to thread an application, and whether to use a process model or a thread model to do so.
- ❅ Write programs that create, manage, and terminate Pthreads.

PROCESSES AND THREADS

- A *process* is an environment, or context, in which a program executes.
  - A *program* consists of a sequential flow of execution within a process.
  - **fork()** creates a new child process, and the program in the new process begins execution on return from the fork call.
  - The parent and child share nothing, though the child inherits much.
- A thread is also an execution context for code instructions, and multiple threads may exist within a single process.
  - A new thread is created with **pthread_create**; execution starts with a call to the function specified as a **pthread_create** parameter.
  - Threads within a process share process resources such as global variables, open files, current directory, etc.
  - Each thread has its own program counter and stack.
  - All threads in a process are peers, not parent/child.
- This course covers POSIX Threads, specified by the POSIX 1.c standard.

We emphasize that a process is an environment, or context, within which a set of instructions, a program, executes. Strictly speaking, a *thread* is also a context for instructions. One or more threads can exist within a process, each consisting of: 1) A program counter containing the address of the next instruction to execute, and 2) A stack containing local variables, function return addresses, and function return values.

The main program runs in a thread created automatically at program start. It is referred to as the *main* or *initial* thread.

When people say "thread," often they are referring to the flow of instruction execution and the application logic realized by that flow, rather than the context provided by the thread. This book uses the term for both the context and the executing code.

CREATING THREADS

- Create a thread with **pthread_create**.

```
#include <pthread.h>
int pthread_create(
    pthread_t *thread_id,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void *arg
);
```

  - **thread_id** is the new thread's ID, which can be used for thread management.
  - **attr** specifies attributes to be applied to the new thread.
  - **start_routine** is the function where execution begins.
  - **arg** is passed to the function.

- The new thread runs concurrently with the calling thread.
  - Upon return from **pthread_create**, the calling thread will continue execution concurrently with the new thread.
  - Which thread executes first is indeterminate, just like with **fork()**.

p1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void * funcA(void *);   /* Prototype funcA */
int n=1, limit=15;      /* Global variables */

int main(int argc, char **argv)
{
  pthread_t thrA;

  if (argc > 1) { limit = atoi(argv[1]); }

  pthread_create(&thrA, NULL, &funcA, NULL);
  while (n <= limit) {
    fprintf(stderr, "In main: %d\n", n++);  // Global n is unprotected.
    sleep(1);
  }
  pthread_exit(0);
}

void * funcA(void *p)
{
  while (n <= limit) {
    fprintf(stderr, "In thread A: %d\n", n++);  // Global n is
unprotected.
    sleep(1);
  }
}
```

Hands On:

Make and run *p1.c*, and study the output. Compile thus: **cc p1.c -o p1 -lpthread**

MULTI-TASKING

- Actual behavior of multiple execution contexts, whether at the process level or at the thread level, depends on many factors.
- If the host machine has only one processor (CPU) then only one context can be running at any time; some call this *concurrent processing*.
  - OS scheduling algorithms determine how to share the single CPU amongst multiple contexts (time sharing, round robin, FIFO, priority schemes, etc.)
  - Stopping a context and starting a different one is called *context switching*, which takes time.
- If the host has multiple CPUs, simultaneous processing is possible; some call this *parallel processing*.
  - The OS still allocates processor time based on scheduling algorithms, but multiple sequences of code execute simultaneously.
  - Context switching per CPU still must occur.
  - The OS (kernel), uses a CPU when it needs to run.
  - *Symmetric multi-processing* (SMP) means the kernel and user applications can all run on any CPU.
- Partitioning an application into processes or threads can improve performance of single-CPU concurrency.
  - When an application task running in a process or a thread blocks, that context stops executing.
  - Tasks that are independent of the blocked task can keep working if they run in a separate context.

The POSIX standard provides for portability, so that programs using Pthreads will run on uniprocessor or multiprocessor systems. Some threaded programs should run faster on a multiprocessor system because logically independent tasks can be executed simultaneously, however they will still run successfully on a uniprocessor system.

OVERVIEW OF THREAD ARCHICTURES

- The POSIX Pthreads standard allows for several different underlying implementations.
- User-space threads run within a process such that the kernel doesn't know about the multiple threads in the process.
  - Threads within the process are scheduled and managed by a thread library linked with the process.
  - The process is the only execution context the kernel sees, thus simultaneous thread execution is not possible.
  - If a thread blocks, the entire process blocks and loses the CPU.
- Kernel threads are OS entities in which a single thread can execute.
  - The kernel schedules and runs threads as independent execution contexts within a process.
  - Threads in a process may run in parallel on multiple processors.
  - If a thread in a process blocks, other threads in the process may still run.
- Current Unix and Linux versions implement kernel threads.
  - Solaris, HP-UX (as of 10.30), Linux (as of 2.2).

Pthreads on Linux

Up until version 2.6, Linux threads (known as LinuxThreads) were created using the Linux **clone** intrinsic, which is also called by **fork()**. Both **fork()** and **pthread_create** were wrappers around **clone** (**fork()** still is). **clone** creates new processes, so threads were not actually threads-within-a-process, rather each thread ran in a genuine Linux process — they all actually show up in ps, including a separate manager thread. This caused many compatibility problems with POSIX threads, and it made large-scale Pthreaded applications problematic on Linux.

In the Linux development version 2.5, LinuxThreads were replaced by the Native POSIX Threads Library, NPTL, which is now part of version 2.6. The NPTL creates threads as thread contexts within a process. NPTL threads are faster and more efficient than LinuxThreads, and POSIX compatibility problems have been resolved.

PROCESSES VERSUS THREADS

- Before deciding whether to design concurrency into your application with processes or threads, first determine if the application will benefit from concurrency, i.e., will it perform faster or be easier to design and support.
    - Are there independent compute-intensive tasks that can be organized to run in parallel?
        - Tasks are independent if they can run in any order, with possible interleaved execution through time-sharing.
    - Are there asynchronous requirements such as I/O requests that block, or network interrupts that occur randomly and must be processed?
    - If an application consists of sequential tasks that each depend on the completion of the previous task, then concurrency doesn't make sense.
- Benefits of programming an application using Pthreads instead of creating multiple processes:
    - Creation — Less system overhead is required to create a thread than a process, because fork duplication requirements are avoided.
    - Running — Switching contexts between threads is faster, because the thread context is small (P-counter, stack, minimal other stuff) and the enclosing process doesn't change.
    - Sharing data — Sharing data between threads does not require IPC mechanisms, which use time consuming kernel calls.
        - However, to avoid data corruption and logic errors, threads must coordinate data access twith Pthreads synchronization mechanisms, which must be programmed very carefully.

Recall that the **fork()** operation creates a new process that is a duplicate of the parent. In addition to allocating kernel structures for the new process, the kernel copies the parent's address space, environment, file descriptor table, and several other attributes that are inherited by the child. Mechanisms exist to reduce the overhead of forking, such as copy-on-write and the Linux **clone** implementation, however it is still faster to create a new thread context within a process than it is to create an entirely new process.

Switching contexts between threads also takes less overhead than switching process contexts because most of the attributes of a process are shared between the threads in the process, so a smaller number of changes must occur to stop one thread and start another.

THE PTHREADS API

- The Pthreads API is a large library of C language functions.
- These functions can be grouped into several categories:
  - Creating, destroying, and managing execution of threads.
  - Creating, initializing, and managing thread attribute objects.
  - Synchronizing threads with mutexes and condition variables.
  - Signal handling.
- Not all implementations support every function, and some implementations provide non-standard thread facilities, so be aware of portability issues.
- As in introductory treatment of Pthreads in this course, we will concentrate on creating, managing, and synchronizing threads.
  - We will not cover scheduling policies, execution priorities, or signal handling.
  - Because of complexity, debugging challenges, and the difficulty of proving correct execution, many threads experts believe threaded programs should be designed to accept scheduling and priority defaults, and not use signals.

Here are all the functions in the standard Posix Pthreads library. We cover several of them in in this course.

```
pthread_atfork()
pthread_cancel()
pthread_cleanup_pop()
pthread_cleanup_push()
pthread_create()
pthread_detach()
pthread_equal()
pthread_exit()
pthread_getschedparam()
pthread_getspecific()
pthread_join()
pthread_key_create()
pthread_key_delete()
pthread_kill()
pthread_once()
pthread_self()
pthread_setcancelstate()
pthread_setcanceltype()
pthread_setschedparam()
pthread_setspecific()
pthread_sigmask()
pthread_testcancel()

pthread_attr_destroy()
pthread_attr_getdetachstate()
pthread_attr_getschedparam()
pthread_attr_getstackaddr()
pthread_attr_getstacksize()
pthread_attr_init()
pthread_attr_setdetachstate()
pthread_attr_setschedparam()
pthread_attr_setstackaddr()
pthread_attr_setstacksize()

pthread_cond_broadcast()
pthread_cond_destroy()
pthread_cond_init()
pthread_cond_signal()
pthread_cond_timedwait()
pthread_cond_wait()

pthread_condattr_destroy()
pthread_condattr_getpshared()
pthread_condattr_init()
pthread_condattr_setpshared()

pthread_mutex_destroy()
pthread_mutex_init()
pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()

pthread_mutexattr_destroy()
pthread_mutexattr_getpshared()
pthread_mutexattr_init()
pthread_mutexattr_setpshared()
```

THREAD TERMINATION

- There are several different ways for a thread to terminate:

  1. It can call **pthread_exit**.

     ```
     pthread_exit(void *status)
     ```

  2. It can return from its start routine.

  3. Another thread can kill it with **pthread_cancel**.

  4. All threads in a process terminate if the process terminates.

- In most applications, threads that explicitly terminate themselves should do so by calling **pthread_exit**.

  - An optional status can be passed to **pthread_exit**, which is retrievable by threads that join the terminated thread.

  - If main calls **pthread_exit**, then the main process thread will terminate but other threads will keep running.

  - If any thread calls **exit** (or if main runs into its closing brace) then the process terminates.

Joining Threads

- A thread can wait for another thread to complete by joining it.

```
pthread_join(pthread_t id, **void status);
```

- The thread that calls **pthread_join** will be suspended until the thread identified by **id** (the joined thread) terminates or gets cancelled.
- **status** will contain the value that the joined thread passed to **pthread_exit**.
- If the joined thread has already terminated, then **pthread_join** will return immediately.

Why would a thread join another thread, that is, suspend itself until another thread finishes? Isn't parallelism the point of threads?

One example is an application that must, at startup, establish connections with several servers before the application can proceed. The application might be designed so the main thread starts a thread per server to establish the connection to that server, then waits on each thread, one after another, with **pthread_join**. This scheme will establish the server connections in parallel, which should be faster than having a single thread establish them all sequentially.

If you think about it, you will see that essentially, the main thread chills until the slowest connection completes, during which time all the faster connections will have happened in parallel so time will be saved.

Note:
Remember, **pthread_join** will return immediately if the thread has already terminated.

DETACHING THREADS

❈ Based on creation attributes, a thread is created as either joinable or detached.

```
pthread_attr_t attr1;      /* Create attribute object. */
pthread_attr_init(&attr1);  /* Initialize object to defaults */
pthread_attr_setdetachstate(&attr1, PTHREAD_CREATE_DETACHED);
pthread_create(&thr, &attr1, func, NULL);
```

❈ A detached thread cannot be joined.

❈ You create a thread as *detached* so the system will recover resources when the thread terminates.

➢ When a joinable thread terminates, the system does not release its thread context until it is joined, which can limit the number of threads a process can have.

➢ Unless the application design requires a thread to be joined, then it should be created as detached.

❈ A thread can detach another thread while the other thread is still running, or after it has terminated.

```
int pthread_detach(pthread_t id);
```

❈ A thread can detach itself with **pthread_detach** and **pthread_self**.

```
pthread_detach(pthread_self());
```

detach.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void * funcA(void *);  /* Prototype funcA */
int n=0, limit=20000;    /* Global variables */

int main(int argc, char **argv)
{
  pthread_t thrA;
  pthread_attr_t attr;
  int r;

  if (argc > 1) { limit = atoi(argv[1]); }

  pthread_attr_init(&attr);  /* Initialize object to defaults */
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

  while (n <= limit) {
    if ( (r=pthread_create(&thrA, &attr, &funcA, NULL)) != 0 ) {
      fprintf(stderr, "%d %s\n", n, strerror(r));
      exit(0);
    }
  //  pthread_detach(thrA);  // Alternate way of detaching a thread
    printf("In main: %d\n", n++);
  }
  pthread_exit(0);
}

void * funcA(void *p)
{
  printf("In thread: %d\n", n);
  sleep(1);
}
```

Hands On:

1. Examine *detach.c*, then make and run it, passing in the number of threads to create as a command line argument. Run it repeatedly to create increasingly larger numbers of threads. How many simultaneous threads can you create?

2. On Linux, the **ulimit** parameter for stack size (which is shown with **ulimit -s** and modified with **ulimit -s size**) affects the number of simultaneous threads a process can create, because each thread is allocated a stack. The **ulimit** stack size value is shown in 1024-byte units. Try reducing it a little at a time and see how many threads you can create. Note that you cannot increase a **ulimit** value other than by logging out and back in (unless you're the superuser). Also be aware that if you make the stack size too small you may have problems running utilities such as **ls** and **vi**.

PASSING ARGUMENTS TO THREADS

❈ The fourth parameter to **pthread_create** is an optional argument passed to the new thread.

```
int pthread_create(
    pthread_t *thread_id,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void *arg
);
```

❈ The argument is a pointer, cast to **(void *)**.

❈ Be careful not to inadvertently modify "pointed to" values after passing the pointer to a new thread.

❈ You can effectively pass multiple arguments by passing a pointer to a structure.

The following example program shows the syntactical mechanics of passing an argument in the **pthread_create** call, and the retrieval in the function. However, the program has a serious problem because the main argument might return and continue executing before **thrA** starts up.

badarg.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void * funcA(void *);
void * funcB(void *);

int main(int argc, char **argv)
{
  pthread_t thrA, thrB;
  int arg;

  arg=1;
  pthread_create(&thrA, NULL, &funcA, (void *) &arg);
  arg=2;  /* UH OH!  thrA might get a 2
  pthread_create(&thrB, NULL, &funcB, (void *) &arg);
  pthread_exit(0);
}

void * funcA(void *p)
{
  int *v = p;
  printf("In thread A: %d\n", *v);
  pthread_exit(0);
}

void * funcB(void *p)
{
  int *v = p;
  printf("In thread B: %d\n", *v);
  pthread_exit(0);
}
```

Hands On:

Examine then run the shell script *badarg.sh*.

LABS

❶ Add a second function named **funcB** to *p1.c*, and have **main** create two threads. **funcB** should be just like **funcA** except for the **fprintf**. Make and run the new program.
(Solution: *p2.c*)

❷ Remove the **sleep** calls in your program from lab ❶. Test it several times, changing the limit value each time to larger numbers, even up to 5000. Redirect the output (**stderr**) to a file each time, if you wish, for easy perusal. Look at the sequence in which the threads run.
(Solution: *p3.c; Run: p3 2>outfile*)

❸ Modify *detach.c* so that the threads are joinable, then see how many you can create. On Linux, manipulate **ulimit -s**.
(Solution: *joinable.c*)

❹ Correct *badarg.c* so that passed values are not susceptible to erroneous modification.
(Solution: *badargfix.c*)