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: Danielle North

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.

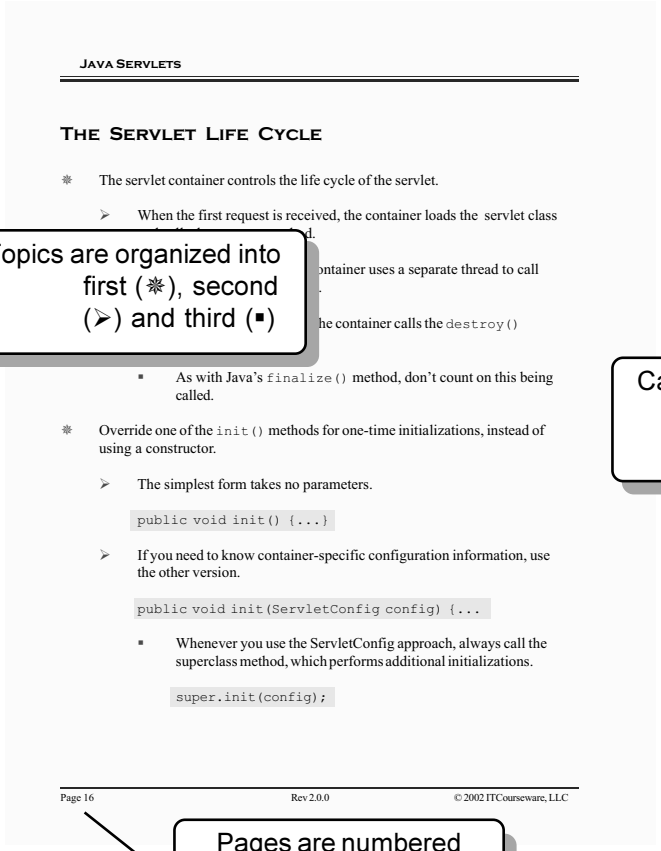
USING THE WORKBOOK

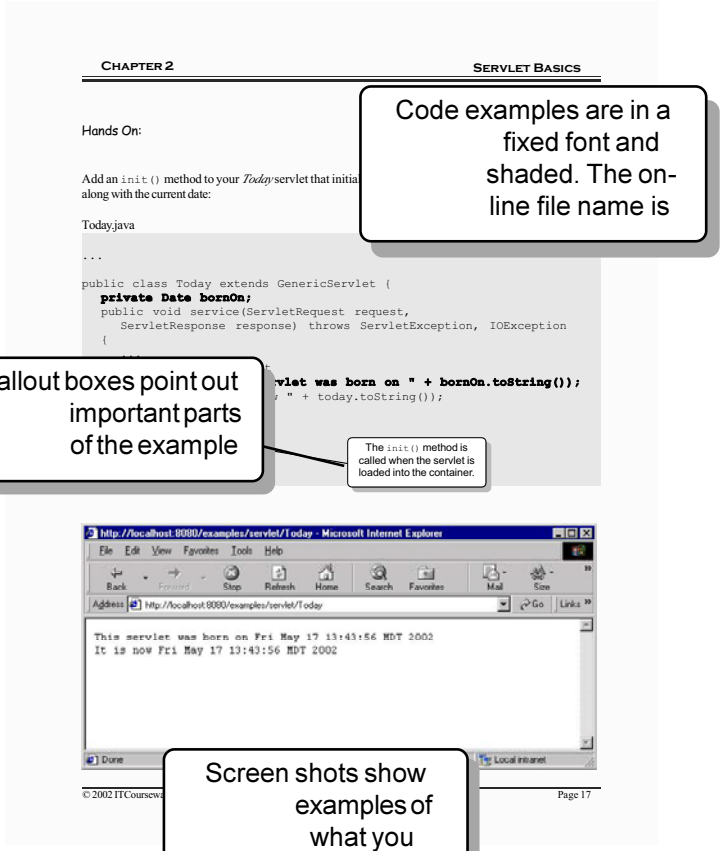
This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, 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

Screen shots show examples of what you

Pages are numbered sequentially throughout the book, making

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

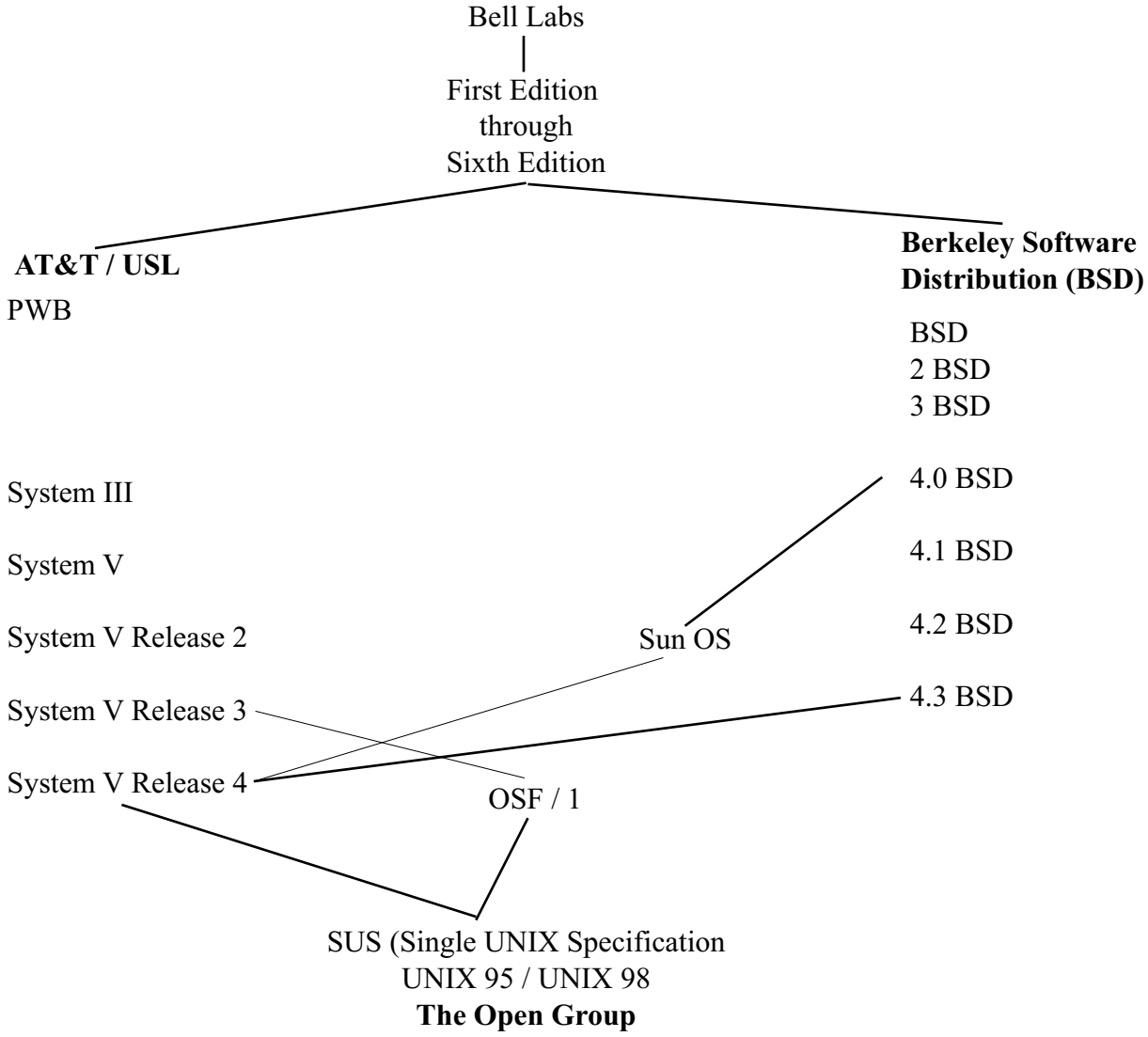
- Historically, UNIX was used for research in universities and government.
  - UNIX was distributed in source code format for many years, so many modifications were made by many different organizations.
- UNIX has been ported to many hardware platforms by vendors who provide "vendor added value" extensions 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 6 - PROCESS MANAGEMENT

OBJECTIVES

- Understand the difference between a program and a process.
- Begin programming with the multi-processing capabilities of UNIX.
- Use the UNIX process management system functions: **fork()**, **exec()**, **wait()**, **waitpid()**, **exit()**.
- Understand implications of attribute inheritance across **fork()** and **exec()**.

THE DIFFERENCE BETWEEN PROGRAMS AND PROCESSES

- A *process* is an environment in which a program executes, a set of attributes (maintained by the kernal) regarding a particular execution of a program.
- A program is stored in a file (with execute mode).
- A *program* consists of instruction segments and user-data segments.
- A program contains a *process image*, that is used to initialize a process from a disk file.
- Each process in UNIX has a unique integer id, called the *Process ID*.

We discussed processes and programs in the previous chapter. In this chapter, we will use the **fork()** and **exec()** system functions to create processes and load process images (aka programs) from files on disk into processes to run.

We will also look more closely at inheritance of attributes when a parent process creates a child (new) process.

THE FORK() SYSTEM FUNCTION

- The only way to create a new process in UNIX is for an existing process to call **fork().**

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

pid_t fork(void);
```

- When **fork()** is called, the system will create a new process that is almost an exact copy of the calling process.
- Instructions, user data, and system data of the calling process are used to initialize the new process.
- After **fork()** returns, both processes receive the return, and a child process is now executing.
- The calling process is now the parent of the new process, the child.
- Your interactive shell uses **fork()** as one step in running programs for you.

The UNIX mechanism of the fork to create new processes is sometimes tricky to understand at first. Think about it like this:

A process is running, and it has text and data segments. The process calls **fork()**, just like it would call **write()**, **printf()**, or any other function. The **fork()** function enters the operating system and, before returning, allocates space for a new process, then fills in the new process segments by copying all of the segments used by the calling process (the calling process is the existing process that called **fork()**). THERE ARE NOW TWO PROCESSES WHERE BEFORE THERE WAS ONLY ONE.

When the kernel is done creating the second (child) process, it must now return to the parent process who called **fork()** in the first place, but it must also somehow get the child process running. It does this not by starting the process up from the beginning of the code segment (where **main()** resides), but rather by starting the new process running as if it too had called **fork()** and was now returning from that function. In fact, the new process inherits the entire call stack from the original process, so it has to work this way.

The big difference between the returns from **fork()** in the parent and in the child (who never really called **fork()**), is the return value. Upon successful completion, **fork()** returns a value of **0** to the child process and returns the process ID of the child to the parent process.

fork1.c

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

void main(void) {
   if (fork() == 0) {  /* CHILD */
      printf("I am a child, my new PID is: %d\n", getpid());
      exit(0);
   }
   else {  /* PARENT */
      printf("I am a parent, my PID is: %d\n", getpid());
      exit(0);
   }
}
```

PARENT AND CHILD

- The child created by **fork()** is a copy of the parent.
- Following is a partial list of attributes inherited by the child from the parent across a **fork**:
  - Real user ID, real group ID, effective user ID, effective group ID
  - Environment
  - Close-on-exec flag [see **exec(2)**]
  - Signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, function address)
  - Set-user-ID mode bit
  - Set-group-ID mode bit
- Read the manual entry for **fork** for additional details of similarities and differences between the parent and child: **man fork**.

Let's look at one implication of parent/child relationships across **fork()**, and get a glimpse of the kinds of things you must be aware of when programming with multiple processes in UNIX. In this example, we will use the fact that the file descriptor table in a parent process is duplicated in the child, with open files still open, and look at some implications of I/O buffering.

fork2.c

```
#include <sys/types.h>
#include <unistd.h>
main() {
   printf("In parent before fork\n");  /* Line buffered if to tty otherwise fully
                                          buffered */

   if (fork() == 0) {  /* CHILD */
      printf("I am a child, my new PID is: %d\n", getpid());
      exit(0);
   }
   else {  /* PARENT */
      printf("I am a parent, my PID is and has been: %d\n", getpid());
      exit(0);
   }
}
```

1. Make and run **fork2.c**.

2. Now run it and redirect its standard output to a temporary file. Look at the contents of the file. The line **In parent before fork()** appears twice because the **printf()** is line-buffered if its output is going to an interactive device, but fully buffered if going to a non-interactive destination, like a file.

3. Add the header **#include <stdio.h>** to the program, then insert the line **setbuf(stdout, NULL)** before the first **printf()** in **main**. Rerun the program again, redirecting its standard output to a file. What's in the file this time?

4. Now remove the **setbuf** call, and change the **printf** as follows:

```
fprintf(stderr, "In parent before fork\n");
```

   Run the program once more, like this:

```
fork2  >  temp  2>&1
```

5. Finally, change the **fprintf()** back to a **printf()** as before, and insert the following line after the **printf()**, but before the **fork()**:

```
fflush(stdout);
```

THE EXEC() SYSTEM FUNCTIONS

- An **exec()** family function is the only way to execute programs in UNIX, and **fork()** is the only way to create processes for programs to execute in.
- Most of the time, the child process calls one of the **exec()** family functions after returning from **fork()** to execute a new program.
- When, for example, **execlp()** is called by the child, the newly-created process is re-initialized with the text and data segments loaded in from the specified program file on disk.
- If **execlp()** is successful, it will not return. The executed program will start execution at **main()**.
- Don't forget that the original parent is still running.
- Let's look at the manual entry for **exec**: **man exec**.

There is no function called **exec()**. Instead, you can choose between any of six calls to implement the exec mechanism, each with slightly different parameters. They all replace the current process image with a new process image. When a C program is executed as a result of this call, it is entered as if called by:

```
main(argc,argv)
```

The forms of **exec** are **execl()**, **execv()**, **execle()**, **execve()**, **execlp()**, **execvp()**. The letters **l**,**v**,**e**, and **p** stand for list, vector (array), environment, and path. You, the programmer, will choose one of the functions based on:

1. How you wish to pass the parameters to the new program — in a list or in an array.
2 Whether the environment variables passed to the new process are those of the calling process or are specified anew.
3. Whether the absolute path to the program file is specified or the **PATH** variable should be searched.

Read the manual entry for **exec** for more details about the six routines.

Example code fragment using **execl**:

```
...
if (fork() == 0) {
   execl("/home/joe/bin/dater", "dater", "jan", "14", "1992", (char *)0);
   perror("Couldn't run dater");
   exit(-1);  /* No need to check return code from exec.  If it returns it's
           a -1. It will only return if it couldn't exec. */
}
else
...
```

In this example, **/home/joe/bin/dater** is the pathname of the executable program to run, **dater** will be **argv[0]** when **dater** starts running, **jan** will be **argv[1]**, **14** will be **argv[2]**, **1992** will be **argv[3]**, and **argc** will be **4**.

When calling **execlp()**, if the first parameter is just a file name instead of a path (a path contains a slash), then the current **PATH** environment variable will be searched. The variable

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings inherited from the current process (the caller).

CURRENT IMAGE AND NEW IMAGE

- The image initialized by **exec()** inherits much from the calling process.

- Following is a partial list of attributes preserved in the new process image across an **exec** system call:

  - nice value [see **nice(2)**]
  - process ID
  - parent process ID
  - process group ID
  - **semadj** values [see **semop(2)**]
  - time left until an alarm clock signal [see **alarm(2)**]
  - current directory
  - file mode creation mask [see **umask(2)**]
  - **utime**, **stime**, **cutime**, and **cstime** [see **times(2)**]
  - file descriptor table (except those with **FD_CLOEXEC** set)
  - file-locks [see **fcntl(2)**]
  - process signal mask [see **sigprocmask(2)**]
  - pending signals [see **sigpending(2)**]

- Read the manual entry for **exec** for additional details of similarities and differences between the parent and child: **man exec**.

THE WAIT() FUNCTIONS

- The **wait()** function returns the process ID of a terminated child.

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

pid_t wait(int *stat_loc);
```

- **wait()** suspends the calling process until one of its immediate children terminates.

- **wait()** will return immediately if all child processes terminated prior to the call to **wait()**.

- **wait()** returns the process ID of the child (if the child terminates normally) and if **stat_loc** is non-zero, the status of the child process (set by a child call to **exit()**) will be stored in the location pointed to by **stat_loc**.

- See **wstat(5)** for macros that access the **stat_loc** data.

wait1.c

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
   int child_stat;

   printf("In parent before fork\n");  /* Line buffered if to tty
                                          otherwise fully buffered */
   fflush(stdout);
   if (fork() == 0) {  /* CHILD */
      printf("I am a child, my new PID is: %d\n", getpid());
      exit(97);
   }
   else {  /* PARENT */
      wait(&child_stat);
      if (WIFEXITED(child_stat))
         printf("Parent, child status: %d\n", WEXITSTATUS(child_stat));
      else
         printf("Child terminated abnormally - status unavailable\n");
      exit(0);
   }
   return 0;
}
```

THE WAITPID() FUNCTION

- POSIX added the **waitpid()** function to traditional UNIX.
- **waitpid()** allows better control than **wait()**.

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

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

- It allows a parent to:
  - Wait for any child (just like **wait()**).
  - Specify one or more specific children to wait for by modifying the value of **pid**.
- **options** are **WCONTINUED, WNOHANG, WNOWAIT** and **WUNTRACED** OR'd together, and can be used to wait without blocking if none of the specified children have terminated yet.

INTERPRETER FILES AND EXEC

- An interpreter file begins with a line of the form:

  ```
  #!pathname [arg]
  ```

- The **exec()** functions look into a file before they load it.

- If the file is not an **a.out** (see **a.out(4)**), and line 1 starts with **#!**, then **exec()** runs the interpreter (such as **/bin/sh**) specified by ***pathname***.

- If the file is not an **a.out**, and it doesn't start with **#!**, then **execlp()** and **execvp()** will run **sh(1)** with the file as its only argument, causing **sh** to read and execute that file as a script.

- See the **exec(2)** man page for further details.

An interpreter file begins with a line of the form

```
#!pathname [arg]
```

where ***pathname*** is the path of the interpreter, and ***arg*** is an optional argument. When an interpreter file is **exec**'d, the system execs the specified interpreter. The ***pathname*** specified in the interpreter file is passed as ***arg0*** to the interpreter. If ***arg*** was specified in the interpreter file, it is passed as ***arg1*** to the interpreter. The remaining arguments to the interpreter are ***arg0*** through ***argn*** of the originally-**exec**'d file. Good and confused now?

Let's see what this means with an example.

Say you have a shell script named **myscript** whose contents are:

```
#!/bin/sh -v apple banana
echo $1 $2 $3 $4
```

If you run it on the command line as

```
$ myscript peaches pears
```

the shell does this

```
execlp("myscript", "myscript", "peaches", "pears",
   (char *0));
```

In **myscript**, the interpreter is specified to be **/bin/sh**. So, when the interpreter **/bin/sh** is run, the string **/bin/sh** will be passed to it as ***arg0***. The **-v** is called ***arg*** in the above paragraph. It is optional, but it is specified in this example, so it will be passed to **/bin/sh** as ***arg1***. The remaining arguments ARE NOT **apple** and **banana**. They are extraneous and will never be used. The remaining arguments passed to **/bin/sh** as ***arg0*** through ***argn*** (in a shell script they will be called **$1 - $n**) are: ***arg0***=**myscript**, ***arg1***=**peaches**, ***arg2***=**pears**. So when the interactive shell runs your command, **execlp** really runs this:

```
/bin/sh -v myscript peaches pears
```

Note that the shell reads and ignores the first line of the file since it begins with **#**. Any interpreter will similarly need to ignore that first line, so it is very common for interpreters to allow **#** to start a comment.

If the new process file is not an executable object file, and the first line does not start with **#!** as described above, then **execlp()** and **execvp()** will start **sh(1)**, and pass the contents of the file as standard input.

LABS

❶ Write a program that opens a file and then **fork**s. Have the parent write the string **ABC\n** to the file several times in a loop, and have the child write the string **abc\n** to the file several times in a loop. Look at the file contents when the programs terminate. What is the relationship between the parent and child file descriptors, and the file table in this case? Have the program print the file offset before each write in the parent and child. Use **offset = lseek(fd, 0, SEEK_CUR);** to get the current offset. Look up **lseek()** to make sure you understand this.
(Solution: *abc.c*)

❷ Modify the program in ❶ above to open the same file in both the parent and child after the **fork()**. Be sure to remove the file after lab ❶ or use a different file. Have each write to the file several times in a loop; inspect the file. What is the relationship between the parent and child file descriptors and the file table in this case?
(Solution: *abc2.c*)

❸ Write a program (to be a child) whose exit status is paramaterized by **atoi(argv[1])**. Write a parent program that **fork**s and **exec**s two copies of the child. Have the parent wait for one of the children but not the other, and print the child's exit status so you are sure you waited for the right one.
(Solutions: *parenta.c, childa.c*)

❹ Write a program that sleeps for several seconds. Have a parent **fork** and **exec** the program, print to **stdout** the child's PID, then wait for the child, printing either the child's exit status or a report of abnormal child termination. Run the parent in the background from your shell, and when it prints the child's PID, kill the child from the keyboard with **kill(1)**.
(Solutions: *parentb.c, childb.c*)

❺ Write a program that loops, repeatedly **fork**ing a child that sleeps. How many child processes can you start up? This number is a tunable operating system parameter, **MAXUP**. All the children should go away when their naps are over.
(Solutions: *loopfork.c, childb.c*)

CHAPTER 14 - TCP SERVER DESIGN

OBJECTIVES

- Describe the algorithms of server software design.
- Construct a simple TCP server.

General Concepts

- A server program implements a basic algorithm:
  - It creates a socket and binds the socket to the well-known port which it will use to receive client requests.
  - It then enters an infinite loop in which it:
    - accepts a request.
    - processes the request.
    - replies to the client.
- The algorithm above is valid, but not sophisticated enough to handle real-world situations.
  - More details must be considered in the design phase of a server program.
    - Will the server program be required to handle multiple, simultaneous requests?
    - Are the resources utilized by the server program "scarce?"
- The combination of iterative vs. concurrent and connection-oriented versus connectionless yields four basic server program design algorithms:
  - Iterative, connection-oriented servers.
  - Concurrent, connection-oriented servers.
  - Iterative, connectionless servers.
  - Concurrent, connectionless servers.
- This chapter will only deal with connection-oriented (TCP) servers.

The facing page is highly euphemistic. Many issues must be considered when designing a server program. Top-down design is not only a good idea from a purist point of view, but for server design it is vital to the initial implementation of the server program, and crucial to subsequent modifications of the program. Failure to adequately design a server program is conducive to hair loss (i.e., pulling it out yourself) of the programmers responsible for the implementation and maintenance of the program.

ITERATIVE SERVERS

- Iterative server programs are the easiest to design, build, and maintain.
- Iterative server programs handle one request at a time.
- Iterative server designs are best for providing simple services, or for providing services based on scarce resources.

A simple service is one that can quickly receive, process, and respond to client requests. Since the server iterates, finishing each client request before starting the next, the service and response should be quick so queued up clients don't wait long.

Just what are “scarce” resources? What is meant in the context of this course by the term “scarce resources" are resources that may have limitations on the number of processes that can access them (ie: only one process at a time may have access to the disk controller), or resources that need protection from multiple process access (ie: a home-grown database that does not have built-in record or file locking).

CONCURRENT SERVERS

- Concurrent servers are more complex to design, build and maintain than iterative servers.

- Concurrent servers can handle multiple, simultaneous requests.

  - They do this by executing multiple processes or threads, one for each client request.

  - This means **fork()** in UNIX.

- A concurrent server can offer performance up to the limitations of the hardware it is run on.

The underlying hardware can enhance or limit the performance of a concurrent server program. On machines with multiple processors, fast (multiple) I/O channels, and/or other "neat stuff" installed, concurrent server programs will perform well. On machines with limited resources, or operating systems that do not handle multiple processes well, concurrent server programs are probably not the best choice.

PERFORMANCE CONSIDERATIONS

- A concurrent server program will provide runtime performance up to the limitations of the hardware it is running on.

- The choice of connection-oriented vs. connectionless can have serious performance implications.

  - A connectionless server can provide visibly higher runtime performance in a small development environment, and then perform miserably when deployed in the real world.

  - A connection-oriented server is easier to design, build, and maintain.

- A stateful server program can offer impressive performance gains at the cost of higher maintenance and decreased reliability.

"Performance" means different things to different people. An iterative server will go from initial design to implementation quicker than a concurrent server providing the same services. An iterative server will also be easier to maintain in operation.

The decision to include state information in a server must be considered very carefully. A stateful server is vulnerable to client failures, network instability, and a host of other threats to "sane" operation. Tread cautiously.

AN ITERATIVE SERVER DESIGN

- Iterative servers can be connection-oriented or connectionless.
- The basic algorithm of an iterative, connection-oriented server is:
  - Create a socket and bind to the well-known port of the service being offered.
  - Listen at the passive socket.
  - Accept the connection from the client, creating a new temporary socket to handle the communication.
  - Using the new socket, repeatedly read the next request from the client, process it and reply back to the client.
  - When communication with a client is complete, close the temporary socket, and go back to listening at the well-known port.

ITERATIVE SERVER EXAMPLE

```
int wait_and_serve(int s)
{
   int ns;
   struct sockaddr_in csin;
   int clen;
   char buf[100];

   for(;;) {
      clen = sizeof(struct sockaddr_in);
      ns = accept(s, (struct sockaddr *) &csin, &clen);

      if (ns < 0) {
         perror("accept call failed");
         exit(-1);
      }

      rewind(stdin);
      while (fgets(buf, 100, stdin)) {
         if ( (write(ns, buf, strlen(buf))) == -1)
       system("echo write error >/tmp/w");
      }
      close(ns);
   }
}

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

   s = get_server_socket();
   if (s < 0)
      exit(1);
   wait_and_serve(s);
}
```



```
/* fdayserv.c   Iterative "file of the day" server program. */
#include <stdio.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>

int get_server_socket()
{
   int s;
   struct protoent *proto;
   struct servent *service;
   struct sockaddr_in sin;

   if ( (proto = getprotobyname("tcp")) < (struct protoent *)0 ) {
      perror("getprotobyname call failed");
      return(-1);
   }

   if ( (s = socket(A_INET, SOCK_STREAM, proto->p_proto)) < -1 ) {
      perror("socket call failed");
      return(-1);
   }

   if ( (service = getservbyname("fday", "tcp")) < (struct servent *)0 )
{
      perror("getservbyname call failed");
      return(-1);
   }

   memset(&sin, 0, sizeof(sin));   /* Zero out the structure first */
   sin.sin_family = AF_INET;
   sin.sin_addr.s_addr = htonl(INADDR_ANY);
   sin.sin_port = service->s_port;

   if (bind (s, (struct sockaddr *)&sin, sizeof(sin)) < 0 ) {
      perror("bind call failed");
      return(-1);
   }

   if ( (listen(s,5) < 0)) {
      perror("listen call failed");
      return(-1);
   }

   return(s);
}
```

A CONCURRENT SERVER DESIGN

- Concurrent servers can be connection-oriented or connectionless.
- Concurrent, connection-oriented servers are very common. The algorithm is shown below:
- Parent Server (runs first)
  - Create a socket and bind to the well-known port of the service being offered.
  - Listen at the passive socket.
  - Block on an **accept()** call until the next client request, then spawn a child to handle the communication using the new socket returned.
- Spawned Child Server
  - Inherit the client request, as well as the new socket.
  - Handle the communication with the client, processing and replying to all the client requests.
  - Close the new socket and die.
- An example concurrent, connection-oriented server is shown on the facing page.


```
/* echos.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <fcntl.h>
#include <netdb.h>
#define LINELEN 128

int main( int argc, char *argv[] )
{
    char *service = "echox";      /* default service */
    char buf[LINELEN+1];          /* buffer for one line of text */
    int  s, n;            /* socket descriptor, read count */
    int  ns;              /* temporary socket descriptor */
    struct servent *service_num;     /* service entity */
    struct sockaddr_in sin, cin;      /* A socket address structure */
    int szcin=sizeof(cin);       /* sizeof client address structure */

    /* server info */ /* get the well-known port for the service */
      if ( (service_num = getservbyname(service, "tcp")) == (struct servent *)0 ) {
             perror("getservbyname call failed");
              return(-1);
       }
    if ( (s = socket(AF_INET, SOCK_STREAM, 0)) == -1 ) {
        perror("socket call failed");
        return(-1);
    }
    memset(&sin, 0, sizeof(sin));  /* Zero out the structure first */
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = service_num->s_port;
    if( bind( s, (struct sockaddr *)&sin, sizeof(sin)) < 0 ){
        perror("bind call failed");
        return(-1);
    }
    if ( (listen(s,5) == -1)) {
        perror("listen call failed");
        return(-1);
    }

    while(1) { /* block on accept call for client request */
        if( (ns = accept(s, (struct sockaddr *)&cin, &szcin)) != -1 ){
            switch( fork() ) { /* spawn a child to handle client */
                case 0: /* Child Server */
                    while( n = read( ns, buf, LINELEN ) )
                        write( ns, buf, n );
                    close(ns);
                    exit(0);
                    break;  /* should never get here */
                case -1: /* fork failure */
                    close(ns);
                    perror("fork failed");
                    exit(-1);
                    break;  /* the switch */
                default: /* Parent Server */
                    close(ns);
                    break;  /* the switch */
            }
        }
    }
}
```

LABS

❶ Compile and run *echos.c* and *echoc.c* using bigfile as an argument to *echoc.c*. What happens if more than one of you starts an *echos.c* server? What modifications must be made? What are the tradeoffs? Examine what happens when there is only one server running and several requests are posted simultaneously.

❷ The *echos.c* program is a concurrent, connection-oriented server. Convert it to an iterative, connection-oriented server and repeat ❶.
(Solution: *echoic.c*)

❸ Review the design of the client you built for ❸ of the previous chapter. Design and build a server that initially just acknowledges the communication, and then add a service that the server will provide. What kind of server did you choose? Why?
(Solution: *server.c*)

