

*AIX 5L Application
Programming Environment*
(Course Code AU25)

Student Notebook

ERC 3.0

IBM Learning Services
Worldwide Certified Material

Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	IBM	Language Environment
MVS	OS/2	PowerPC
RISC System/6000	RS/6000	

AT&T and the AT&T and Globe design are registered trademarks of AT&T in the United States and other countries.

Notes is a trademark or registered trademarks of Lotus Development Corporation and/or IBM Corporation in the United States, other countries, or both.

Windows Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

SET and the SET Logo is a trademark owned by SET Secure Electronic Transaction LLC.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

August 2002 Edition

The information contained in this document has not been submitted to any formal IBM test and is distributed on an "as is" basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk. The original repository material for this course has been certified as being Year 2000 compliant.

© Copyright International Business Machines Corporation 1995, 2002. All rights reserved.

This document may not be reproduced in whole or in part without the prior written permission of IBM.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Trademarks	xiii
Course Description	xvii
Agenda	xix
Unit 1. Overview of AIX Application Programming	1-1
Unit Objectives	1-3
1.1 Overview of AIX Application Programming	1-5
The AIX Programming Environment	1-6
Program Creation	1-7
Program Compilation	1-9
Runtime Environment	1-11
Debugging	1-12
Kernel Interface	1-13
1.2 Overview of the Single UNIX Specification	1-15
The Single UNIX Specification	1-16
The Single UNIX Specification Documents	1-18
The Bottom Line	1-20
1.3 Finding Information	1-21
The man command	1-22
AIX 5L Online Documentation	1-24
Document Search and Scope	1-25
Document Search Results	1-27
Unit Summary	1-28
Unit 2. Compiling Programs	2-1
Unit Objectives	2-3
2.1 Overview of Compiling Programs	2-5
Overview	2-6
Traditional UNIX Compilation Process	2-8
AIX Compilation Process	2-10
The Extended Intermediate Language	2-12
Common Linkage Conventions	2-13
Other Compilers on AIX	2-14
Types of Licenses	2-16
Steps to Setup License for C for AIX	2-18
2.2 Compiler Commands and Options	2-21
The cc/xlc/c89 Commands	2-22
Compiler Configuration	2-24
cc/xlc/c89 Options (1 of 2)	2-26
cc/xlc/c89 Options (2 of 2)	2-28
64-bit Compilation	2-30
Compiler Listings (1 of 3)	2-32

Compiler Listings (2 of 3)	2-33
Compiler Listings (3 of 3)	2-34
Compiler Diagnostics	2-35
Compiler Modes	2-36
Preprocessor Directives	2-38
Preprocessor Options	2-40
Optimization	2-41
Code Inlining	2-43
2.3 The Executing Program	2-45
Program and Process Images	2-46
Virtual Address Space	2-48
64-bit Application Porting	2-49
Unit Summary	2-51
 Unit 3. The dbx and idebug Debuggers	3-1
Unit Objectives	3-2
3.1 The dbx Debugger	3-3
Introduction to dbx	3-4
Using the dbx Debugger	3-5
Compiling for the Debugger	3-6
Creating a Core Dump	3-7
Core File Naming Conventions	3-8
Contents of Core File	3-10
Invoking dbx	3-11
3.2 dbx Subcommands	3-13
dbx Subcommands	3-14
Debugging Options	3-15
Displaying the File	3-18
Moving Between Files and Procedures	3-19
The dbx Stack Trace and String Search	3-20
Displaying Variables (1 of 3)	3-21
Displaying Variables (2 of 3)	3-23
Displaying Variables (3 of 3)	3-24
Change Variables	3-25
Running a Program	3-26
Using Breakpoints (1 of 2)	3-27
Using Breakpoints (2 of 2)	3-29
Continuing Execution	3-30
Tracing	3-31
Other dbx Commands	3-32
3.3 The idebug Graphical Debugger	3-35
Introduction to idebug	3-36
The idebug Debugger	3-37
Segmentation Fault in a Library Routine	3-40
Moving Through the Function Call Traceback	3-41
Setting the Breakpoint	3-42
Listing the Currently Set Breakpoints	3-43

Viewing Modules in a Program	3-44
Stopped at a Breakpoint	3-45
Looking at a Local Variable (1 of 3)	3-46
Looking at a Local Variable (2 of 3)	3-47
Looking at a Local Variable (3 of 3)	3-48
Modifying a Local Variable (1 of 2)	3-49
Modifying a Local Variable (2 of 2)	3-50
End of the Trail	3-51
3.4 Other Debugging Tools	3-53
Other Debugging Tools	3-54
Unit Summary	3-58
Unit 4. The make and SCCS Facility	4-1
Unit Objectives	4-2
4.1 The make Facility	4-3
An Application Example	4-4
Targets and Dependencies	4-6
The makefile	4-7
Makefile Action Lines (1 of 2)	4-8
Makefile Action Lines (2 of 2)	4-9
4.2 Using make.	4-11
Using the make Command (1 of 2)	4-12
Using the make Command (2 of 2)	4-13
make with Command Line Targets	4-14
Pseudo Targets	4-15
Makefile Macros (1 of 2)	4-16
Makefile Macros (2 of 2)	4-17
Command Line Macros	4-18
Internal Macros	4-19
4.3 make Rules.	4-21
Inference Rules	4-22
Example Rules File	4-24
4.4 Source Code Control System (SCCS).	4-27
What is SCCS?	4-28
SCCS Terminology	4-29
Creating a SCCS file	4-30
get command	4-31
delta command	4-33
unget command	4-35
ID keywords	4-36
Other Helpful SCCS Commands	4-38
Unit Summary	4-40
Unit 5. AIX Dynamic Binding and Shared Libraries	5-1
Unit Objectives	5-2
5.1 AIX Dynamic Binding and Shared Libraries.	5-3
Overview	5-4

Static Binding Concepts	5-6
Dynamic Binding Concepts (1 of 2)	5-7
Dynamic Binding Concepts (2 of 2)	5-9
Import and Export Files	5-10
Creating a Loadable Object	5-11
Creating an Application	5-12
Execution - Time Binding	5-13
5.2 Shared Libraries	5-15
Library Terminologies	5-16
AIX Libraries	5-17
Linking to Libraries	5-19
Static Linking to Libraries	5-21
Nonshared Libraries	5-22
Creating a Nonshared Library	5-23
Shared Libraries	5-24
Creating a Shared Library	5-25
Linking to a Shared Library (1 of 2)	5-27
Linking to a Shared Library (2 of 2)	5-29
Using a Shared Library	5-30
Binding at Load Time	5-32
Run-Time Linking	5-34
Rebinding System Defined Symbols	5-37
Dynamic Binding and Shared Libraries (1 of 2)	5-39
Dynamic Binding and Shared Libraries (2 of 2)	5-40
5.3 Related Commands	5-41
AIX ld Options	5-42
The ar Command	5-44
Adding an Object to a Library	5-45
Replacing an Object in a Library	5-46
Some Useful Commands	5-47
Unit Summary	5-49
Unit 6. System Call Introduction	6-1
Unit Objectives	6-2
6.1 System Calls	6-3
Overview	6-4
System Call Overview	6-5
Executing System Calls	6-7
/usr/include/sys/errno.h	6-9
Major System Call Groups	6-10
Standard Headers Used with System Calls	6-11
6.2 General System Calls	6-13
Memory Management Subroutines	6-14
Time Management Subroutines	6-16
Unit Summary	6-18

Unit 7. AIX File and I/O System Calls	7-1
Unit Objectives	7-2
7.1 The AIX File System	7-3
Overview	7-4
File System Overview (1 of 2)	7-5
File System Overview (2 of 2)	7-7
Basic File Manipulation Calls	7-8
The open and creat Calls	7-9
The read and write System Calls	7-11
The lseek System Call	7-13
The close System Call	7-14
File I/O Example	7-15
File I/O Example Output	7-16
Writing to Disk Files (1 of 2)	7-17
Writing to Disk Files (2 of 2)	7-19
7.2 File Locking	7-21
File Locking	7-22
The lockf Call	7-23
File Locking Example (non-POSIX)	7-24
File Locking Example Output	7-25
The fcntl Call	7-26
flock Structure	7-27
File Locking Example (POSIX)	7-28
7.3 File Characteristics	7-29
Controlling File Characteristics	7-30
The access Call	7-31
File Access Example	7-32
The chmod Call	7-33
The chown Call	7-35
The chdir Call	7-36
The utime Call	7-37
The umask Call	7-39
File Characteristics Example	7-41
File Characteristics Example Result	7-42
7.4 More Files I/O Calls	7-43
Other File and Device I/O Calls	7-44
ioctl() subroutine	7-45
Device I/O Control Example (non-POSIX)	7-46
Changing Terminal Characteristics (POSIX)	7-47
7.5 Even More File and File System Calls	7-49
File System Calls	7-50
The stat Call	7-51
The statfs Call (1 of 2)	7-53
The statfs Call (2 of 2)	7-54
The mkdir Call	7-56
The link Call	7-57
The unlink Call	7-59

The tmpfile Call	7-60
The vmount and unmount Call	7-62
File System Example	7-63
7.6 Memory Mapped Files	7-65
Memory Mapped Files	7-66
Explicitly Mapped Files	7-68
The shmat Call for Mapping Files	7-71
Mapped File Example	7-72
The mmap Call	7-74
mmap Example	7-76
7.7 Working With Large Files	7-77
How Large Is Large?	7-78
What Can Go Wrong?	7-79
Open Protection	7-81
Working with Large Files	7-82
The File Size ulimit	7-84
Unit Summary	7-85
Unit 8. AIX Process Management Systems Calls	8-1
Unit Objectives	8-2
8.1 AIX Process Management System Calls	8-3
Properties of a Process	8-4
Process IDs	8-5
User and Group IDs (1 of 4)	8-6
User and Group IDs (2 of 4)	8-8
User and Group IDs (3 of 4)	8-10
User and Group IDs (4 of 4)	8-12
Process Groups (1 of 2)	8-13
Process Groups (2 of 2)	8-14
Process ID, Group ID, and User ID Example (1 of 2)	8-15
Process ID, Group ID, and User ID Example (2 of 2)	8-16
Process Scheduling	8-17
Scheduling Policies	8-19
CPU Timeslicing	8-21
Process Scheduling System Calls	8-22
Process Scheduling Example (1 of 2)	8-24
Process Scheduling Example (2 of 2)	8-25
8.2 Creating New Processes	8-27
Creating a Process	8-28
Process Life Cycle	8-30
Process Life Examples	8-32
The exec System Calls (1 of 4)	8-33
The exec System Calls (2 of 4)	8-35
The exec System Calls (3 of 4)	8-37
The exec System Calls (4 of 4)	8-39
The exit() System Call	8-41
The wait() System Call	8-43

Getting Status from Child	8-45
Effects of waitpid() and wait()	8-47
8.3 Examples of Process Life Cycle	8-49
Process Creation Example (1 of 2)	8-50
Process Creation Example (2 of 2)	8-51
Unit Summary	8-52
Unit 9. AIX Signal Management	9-1
Unit Objectives	9-2
9.1 AIX Signal Management	9-3
Signals (1 of 2)	9-4
Signals (2 of 2)	9-5
Examples of Signals	9-7
Sending Signals	9-9
9.2 Handling Signals	9-11
Signal Actions	9-12
Signal Handler	9-13
Signal Handler Example	9-15
The sigaction Call	9-16
sigaction Example	9-18
Signal Masks	9-19
Unit Summary	9-21
Unit 10. Interprocess Communication	10-1
Unit Objectives	10-2
10.1 Fundamentals of IPC	10-3
IPC Concepts	10-4
Unofficial IPC Techniques	10-5
10.2 Pipes	10-7
Pipes	10-8
Unnamed Pipes	10-10
Simple Unnamed Pipe Example	10-11
Named Pipes	10-14
Simple Named Pipe Example (1 of 2)	10-15
Simple Named Pipe Example (2 of 2)	10-16
10.3 System V IPCs	10-19
System V IPCS	10-20
IPC Keys	10-22
10.4 Using Shared Memory Segments	10-23
Setting Up Shared Memory	10-24
Shared Memory System Calls	10-26
Shared Memory Permissions	10-28
Using Shared Memory	10-30
Controlling Shared Memory Segments	10-31
Shared Memory Example (1 of 2)	10-33
Shared Memory Example (2 of 2)	10-34
10.5 Semaphores	10-35

Setting Up Semaphores	10-36
Creating Semaphores	10-37
Semaphore Permissions	10-38
Controlling Semaphores	10-39
Using Semaphores	10-40
Semaphores Example - Main Application	10-41
Semaphore Example - Called Functions	10-42
10.6 Message Queues	10-43
Message Queues	10-44
Setting Up Message Queues	10-45
Message Queue Permissions	10-46
Sending Messages	10-47
The msgbuf Structure	10-48
Receiving Messages	10-49
Message Queue Multiplexing	10-50
Controlling Message Queues	10-51
Message Queue Example (1 of 4)	10-52
Message Queue Example (2 of 4)	10-54
Message Queue Example (3 of 4)	10-55
Message Queue Example (4 of 4)	10-57
10.7 Introduction to Sockets	10-59
What Are Sockets?	10-60
IP Sockets - Addressed	10-61
IP Socket Parts	10-62
Socket Calls - Server	10-64
Socket Calls - Client	10-65
Socket Example - Server (1 of 2)	10-66
Socket Example - Server (2 of 2)	10-67
Socket Example - Client (1 of 2)	10-68
Socket Example - Client (2 of 2)	10-69
Unit Summary	10-70
Unit 11. Writing AIX Daemons	11-1
Unit Objectives	11-2
11.1 Just What Are Daemons?	11-3
What Is a Daemon?	11-4
Special Characteristics of a Daemon	11-6
11.2 Writing A Daemon	11-9
Steps to Writing a Daemon	11-10
Determine How The Daemon Started	11-11
Ignoring Signals	11-12
Closing stdin, stdout, and stderr	11-14
Relinquishing the Controlling Terminal	11-15
Environmental Concerns	11-17
11.3 The Syslog Facility	11-19
The Syslog Facility	11-20
Sending Messages to Syslog	11-21

Configuring the Syslog Daemon	11-23
11.4 Using The Daemon	11-25
Starting the Daemon	11-26
Daemon Example	11-27
Unit Summary	11-28
Unit 12. The Remote Procedure Call (RPC) Facility	12-1
Unit Objectives	12-2
12.1 RPC and Distributed Computing	12-3
Remote Procedure Call Flow	12-4
Advantages of Remote Procedure Calls	12-6
Where is RPC Used Today?	12-8
12.2 Writing An ONC RPC Application	12-11
Writing an ONC RPC Application	12-12
rpcgen	12-13
RPC Protocol Identifiers and Versions	12-15
The Protocol Description File	12-17
An Example Problem	12-19
The baseball.x File	12-20
baseball.h - The protocol Header File	12-21
bbfuncs.c - Server Side Implementation Routines	12-22
bbclient.c - A Simple Baseball Client Program	12-23
Running the Programs	12-25
Unit Summary	12-27
Unit 13. Threads: Getting Started	13-1
Unit Objectives	13-2
13.1 Thread Concepts	13-3
Threads versus Processes	13-4
Meeting POSIX Standards	13-6
Traditional Approach: Direct Manipulation	13-7
Global Approach: Objects	13-8
More Thread Terminology	13-9
Protecting Data Resources	13-11
Controlling Thread Execution	13-12
Thread Inheritance	13-13
Review - Match the Terms to Their Description	13-15
Match the Terms to Their Description - Answers	13-16
13.2 Producing a Simple Threads Program	13-17
Call Comparison	13-18
Thread Creation	13-19
Thread Deletion	13-21
Basic Thread Example	13-23
Thread Compilation	13-24
Getting the Thread Return_Data (1 of 2)	13-25
Getting the Thread Return_Data (2 of 2)	13-26
Getting the Thread Return_Data Example	13-27

Unit Summary	13-28
Unit 14. Threads: Worrying About Your Neighbor	14-1
Unit Objectives	14-2
14.1 Protecting Data Resources	14-3
Protecting Data Resources	14-4
Handling Global Data: Mutex	14-5
Handling Global Data: Example (1 of 2)	14-7
Handling Global Data: Example (2 of 2)	14-8
Thread Specific Data: Initialization	14-10
Thread-Specific Data: Usage and Destruction	14-12
Thread-Specific Data: Example (1 of 2)	14-13
Thread-Specific Data: Example (2 of 2)	14-14
Handling Local Static Data	14-15
14.2 Controlling Thread Execution	14-17
Controlling Execution	14-18
Condition Variable Logic	14-19
Condition Variable Calls	14-20
Condition Wait Implementation	14-22
Condition Wait Example (1 of 2)	14-23
Condition Wait Example (2 of 2)	14-24
Master/Slave Implementation	14-25
Master/Slave Example (1 of 2)	14-26
Master/Slave Example (2 of 2)	14-27
Signal Management	14-29
Signal Management Example (1 of 2)	14-30
Signal Management Example (2 of 2)	14-31
Cancellation Choices	14-32
Cancellation Points	14-33
Cancellation Cleanup	14-35
Cancellation Example (1 of 2)	14-37
Cancellation Example (2 of 2)	14-38
Scheduling	14-39
Scheduling Example (1 of 2)	14-40
Scheduling Example (2 of 2)	14-41
Once Only Initialization	14-42
Once Only Example (1 of 2)	14-43
Once Only Example (2 of 2)	14-44
Forking Considerations	14-45
Forking Example (1 of 2)	14-46
Forking Example (2 of 2)	14-47
Unit Summary	14-48
Appendix A. Handling Core Dumps.....	A-1
Glossary	X1

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both::

AIX®	IBM®	Language Environment®
MVS™	OS/2®	PowerPC®
RISC System/6000®	RS/6000®	

AT&T and the AT&T and Globe design are registered trademarks of AT&T in the United States and other countries.

Notes is a trademark or registered trademarks of Lotus Development Corporation and/or IBM Corporation in the United States, other countries, or both.

Windows Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

SET and the SET Logo is a trademark owned by SET Secure Electronic Transaction LLC.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Several of the exercises depend upon the successful completion of a preceding exercise or exercises.

Course Description

AIX 5L Application Programming Environment

Duration: 5 days

Purpose

This course is designed to teach C programmers how to use the base programming tools and common AIX system calls. It also provides an introduction to Open Network Computing Remote Procedure Call (ONC RPC) and POSIX threads programming.

Audience

This course is primarily geared for people who plan to program in C on the AIX platform. Although the course concentrates on C and AIX, many concepts can be used with other languages (for example, C++ or Fortran) and on other UNIX platforms.

Prerequisites

Before taking this course, you should possess:

- Knowledge of the basic AIX commands (such as the vi editor, cd, ls, pwd, chmod, info, and kill)
- Basic understanding of UNIX concepts (such as processes, the shell, and the kernel).
- Working knowledge of C from some computer platform

These skills can be developed by taking:

- AU13 (or Q1313) AIX 5L Basics (three days)
- Q1070 Introduction to C Programming - AIX/UNIX (five days)
- Q1071 Advanced C Programming - AIX/UNIX (five days, optional but recommended)

Objectives

After completing this course, you should be able to:

- Use the base programming tools and utilities (including make and debugger) that are shipped with the AIX operating system.

- Compile programs using basic and advanced techniques (such as dynamic binding and shared libraries).
- Use the basic AIX file and I/O system calls.
- Manage processes (creation, status codes, and so on).
- Utilize AIX system calls to manage signals.
- Utilize many common Interprocess Communication (IPC) calls, such as pipes, shared memory, semaphores, message queues, and the basics of sockets.
- Write a simple daemon.
- Write a simple client and server using the Open Network Computing remote procedure call (ONC RPC) facility.
- Utilize faster parallel activity through the implementation of POSIX threads.

Recommended Reading

There are literally hundreds of books dealing with developing software in the UNIX environment. Here are just a few that seem to appear on a lot of reading lists:

- *Go Solo with the Single UNIX Specification*, published by X/Open Company Ltd., distributed by Prentice Hall, ISBN 0-13-439381-3
- *POSIX Programmer's Guide*, Donald Lewine, published by O'Reilly & Associates, Inc., ISBN 0-937175-73-0
- *Power Programming with RPC*, John Bloomer, published by O'Reilly & Associates, Inc., ISBN 0-937175-77-3
- *Pthreads Programming*, Nichols, Buttlar & Farrell, published by O'Reilly & Associates, Inc., ISBN 1-56592-115-1
- *Advanced Programming in the UNIX Environment*, W. Richard Stevens, published by Addison Wesley, ISBN 0-201-56317-7
- *UNIX Network Programming*, W. Richard Stevens, published by Prentice Hall, ISBN 0-13-949876-1

In addition, the AIX 5L for POWER V5.1 Documentation CD (or AIX online documentation) and the additional information provided on the C for AIX CD-ROM are excellent sources of relevant information.

Agenda

Day 1

Welcome
Overview of AIX Application Programming
Compiling Programs
Lab (Exercise 1)
The dbx and idebug Debuggers
Lab (Exercise 2)
The make and SCCS Facility
Lab (Exercise 3)

Day 2

AIX Dynamic Binding and Shared Libraries
Lab (Exercise 4)
System Call Introduction
AIX File and I/O System Calls
Lab (Exercise 5)

Day 3

AIX Process Management System Calls
Lab (Exercise 6)
AIX Signal Management
Lab (Exercise 7)
Interprocess Communication

Day 4

Lab (Exercise 8)
Writing AIX Daemons
Lab (Exercise 9)
The Remote Procedure Call (RPC) Facility
Lab (Exercise 10)

Day 5

Threads: Getting Started
Lab (Exercise 11)
Threads: Worrying About Your Neighbor
Lab (Exercise 12)
Review/Wrap-up

Unit 1. Overview of AIX Application Programming

What This Unit is About

This unit introduces the major themes of this course:

- The components involved in application programming in the AIX environment
- The Single Unix Specification and how it can be used to develop portable software
- Some of the ways of gaining access to information needed to support the programming effort

The first part of this unit presents the application programming components from a high level view in order to establish the scope of this course. Then each component is examined and the individual tools and subcomponents are listed for each component.

The second part of this unit is a briefing on the Single UNIX Specification, which forms the basis for programming portable software. The second part also explains the documents that comprise the Single UNIX Specification.

The third part of this unit presents methods of accessing documentation on the tools and system calls described throughout this course. This course emphasizes using the AIX 5L Version 5.1 online documentation to locate information. Some hardcopy manuals are also mentioned.

What You Should Be Able to Do

After completing this unit, you should be able to:

- List the steps involved in creating a program in the AIX environment
- Describe the components of program creation
- Describe the AIX run-time environment
- Explain the purpose of system calls
- Describe the role that the Single UNIX Specification plays in assisting in the development of portable software
- Use the **man** command to find information on AIX 5L Version 5.1 system calls and commands
- Use AIX 5L Version 5.1 online documentation as a reference for programming information

How You Will Check Your Progress

There is no exercise for this unit.

References

- Unit 2. System Management Tools and Documentation of ILS course AU14 for reference to AIX Online Documentation
- **<http://www.UNIX-systems.org/version3>** for information on the Single UNIX Specification

Unit Objectives

After completing this unit, you should be able to:

- List the steps involved in creating a program in the AIX environment
- Describe the components of program creation
- Describe the AIX run-time environment
- Explain the purpose of system calls
- Describe the role that the Single UNIX Specification plays in assisting in the development of portable software
- Use the **man** command to find information on AIX 5L Version 5.1 system calls and commands
- Use the AIX 5L Version 5.1 online documentation as a reference for programming information

Figure 1-1. Unit Objectives

AU253.0

Notes:

1.1 Overview of AIX Application Programming

The AIX Programming Environment

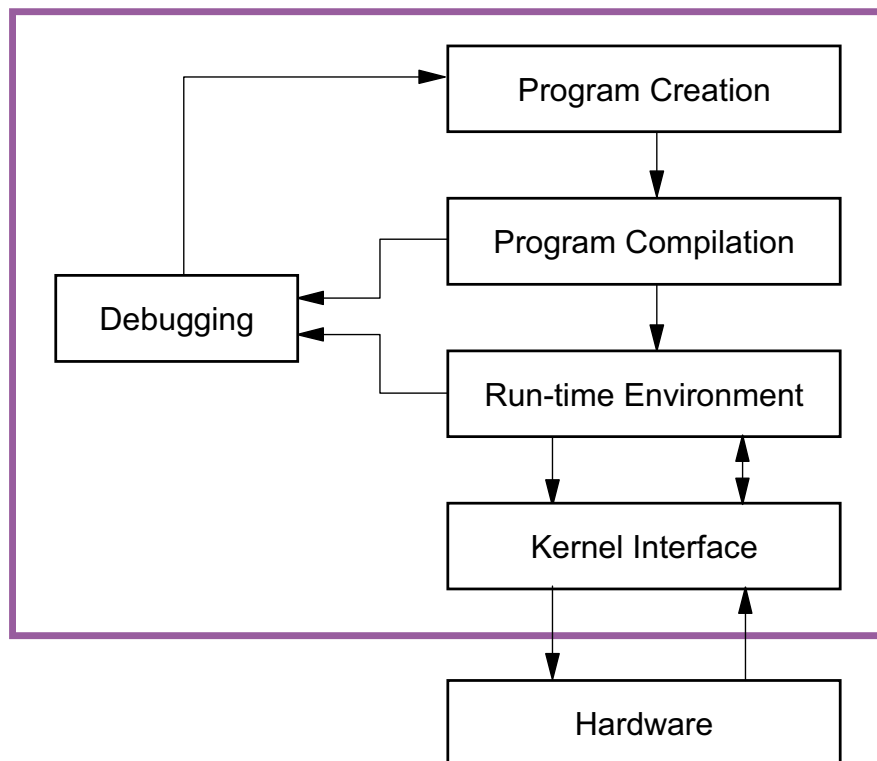


Figure 1-2. The AIX Programming Environment

AU253.0

Notes:

There are many aspects to programming on an UNIX system. This is certainly true in the case of the IBM AIX Operating System. Application designers, programmers, and those who maintain the applications need to be familiar with one or more major areas of the AIX programming environment.

Figure 1-2 illustrates the five major components, or aspects, of AIX application programming. This includes all areas considered to be within the scope of this course. The hardware component is generally more of a concern to systems programmers, but is included in this drawing to show its relationship to the AIX programming environment.

Each of these areas is detailed on the following pages, and discussed at length in the following units.

Program Creation

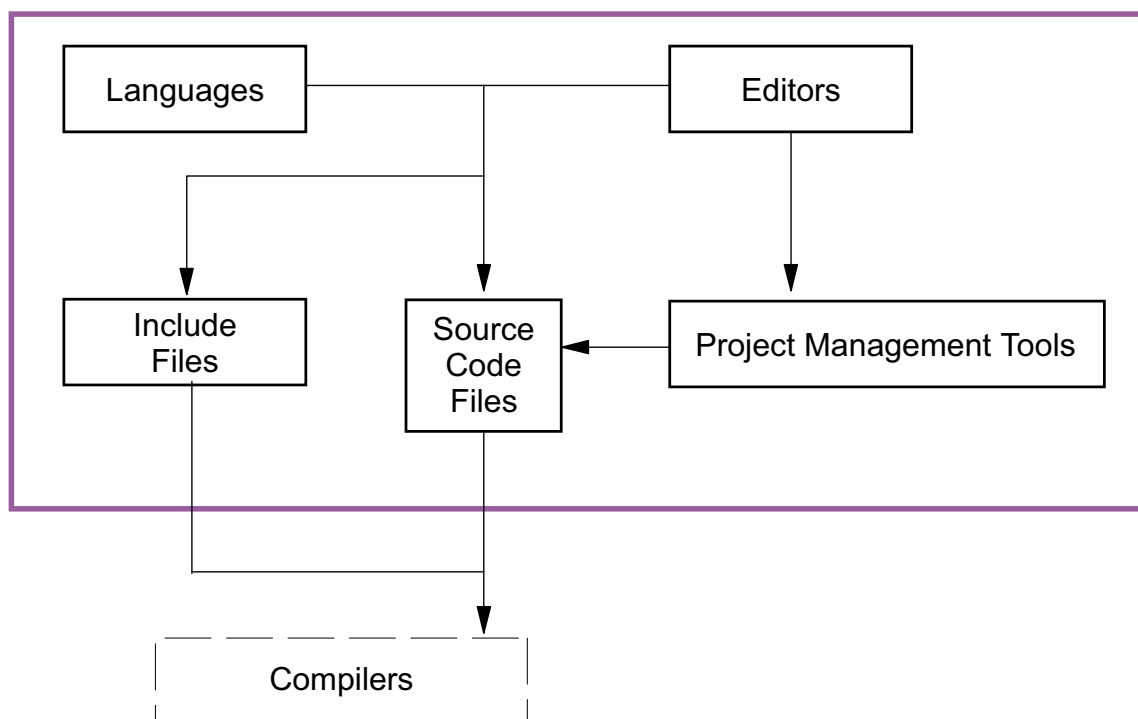


Figure 1-3. Program Creation

AU253.0

Notes:

The Program Creation component of the AIX programming environment contains the tools and disciplines needed to generate source code for a particular compiler. It also contains numerous tools for managing the software development project.

Languages are the basis of all software development. AIX supports C, C++, FORTRAN, Pascal, COBOL, Java, Perl, and ADA.

Packaging on AIX Version 4 and above is different from previous releases. To conserve on memory and disk space during installation, only the AIX run-time environment is installed, and little else. Additional bundles of software may then be installed, as desired, to customize the environment. The application development bundle is one of these and includes many of the programming tools. The compilers are now licensed and come as a complete set on CD-ROM.

AIX supports extended K & R (Kernigan & Ritchie) and ANSI C language models.

AIX includes the ed, vi, and INed editors. Other editors, such as emacs, cpedit (xedit for UNIX), and others are available from various sources. Try the Internet or, if you work for

IBM, try the internal UNIX forums. This course includes a few pointers on using the vi editor in a programming environment.

The editors are used to create source code and include files (header files) from the various languages. The source files and include files are used by the compiler to generate program objects. Include files can be incorporated (or included) into many source code files during the compilation process, and are generally used to define common data structures and macros.

Project management tools for the AIX programming environment include the make utility for automated compilation and linking, and many tools to aid in the creation, editing, and analysis of source code. The make utility is discussed later.

The Source Code Control System (SCCS) is another project development tool that allows users to control and track changes made to a file. The tool allows you to maintain several versions of the same file, which is very essential for a project.

Program Compilation

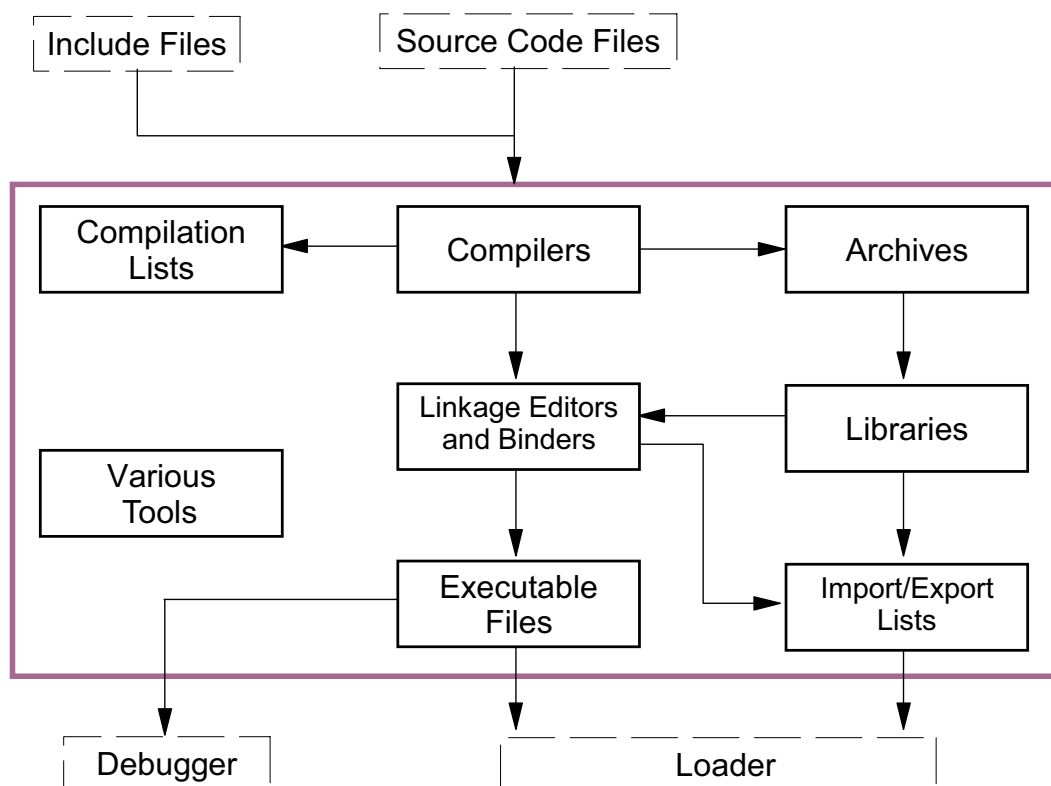


Figure 1-4. Program Compilation

AU253.0

Notes:

The source code and include files are used by the compiler to create program modules (or objects). Module names usually end in .o to signify an object. The object is then used by an archiver, for example, the UNIX **ar** command, to create a library or is passed to the linkage editor where it is bound to other objects to form an executable file (program).

The C, C++, FORTRAN, and Pascal languages, when compiled by their front-end compilers, share a common intermediate language (like assembler code) and a common back-end optimizer. This provides for interlanguage calling, which includes cross library calls.

The AIX compilers include options for generating various reports, called compiler listings. These reports can include information about the variables and functions used in a program, statistics on compile time, information on stack frames, or even a listing of the assembler code (the common intermediate back-end language). These reports are useful when debugging a program.

Libraries are collections of objects that contain functions that may be used by many different applications. These libraries may even be shared by many applications. This course includes a lecture on creating and using libraries.

The linkage editor phase links program objects (various .o files) and libraries into a final executable file. Binding is the process that combines multiple object modules into an executable program. Since AIX supports the concept of dynamic binding, modules can be imported and exported, requiring the linkage editor to call a binder that understands imported and exported symbols (functions and variables called from other modules and libraries). AIX includes a linkage editor and binder that understand dynamic binding. The compiler calls the linkage editor by default when compiling a program. The linkage editor, in turn, calls the binder. It is possible to have the compiler call a different linkage editor, or to have the linkage editor call a different binder.

Once again, AIX includes various tools to aid in the compilation phase of program creation.

Runtime Environment

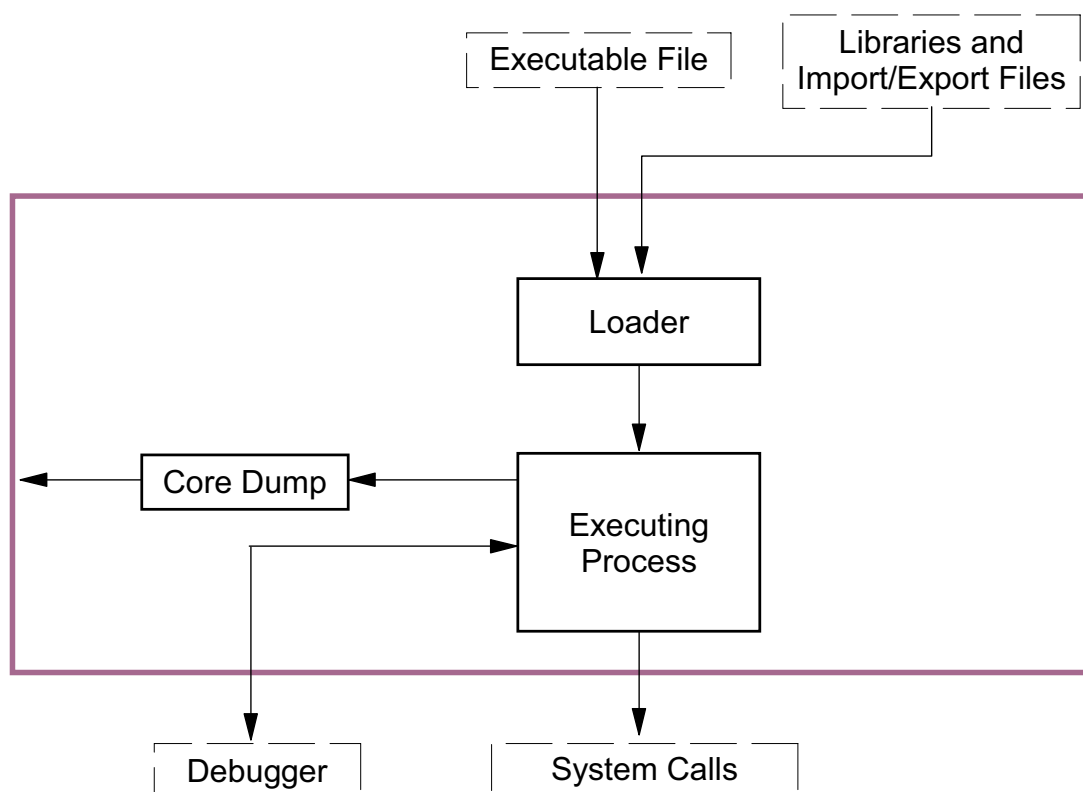


Figure 1-5. Runtime Environment

AU253.0

Notes:

The run-time environment includes everything needed to execute your program. When you enter the program name on the shell's command line, the shell creates a new process and uses it for program execution. The mechanism used to create a process will be discussed later.

An executable is called a program, whereas a program under execution is called a process. If the process attempts to do something that the system will not allow, such as referencing a memory location not available to the process, an exception occurs. An exception is an error caused by the process that usually results in the termination of the process. Occasionally, the exception is so severe that a core dump is generated. A core dump is a file that is placed in the process' current directory. This file contains information about the process at the time of termination and is used with a debugger to determine the cause of the exception. Core files are very useful in determining the cause for the exception. The debugger and its usage is discussed in a lecture that follows.

Debugging

The features of a debugger include:

- Allows programmers to view source code at the time of execution
- Variables can be inspected
- Breakpoints control the execution of the program
- Stack traces show the flow of execution
- Allow debugging of multiple processes

AIX 5L Version 5.1 supports dbx, adb, and idebug debuggers.

Figure 1-6. Debugging

AU253.0

Notes:

Debuggers are an important tool that programmers use to debug application programs. They provide the ability to look through source code, examine variables, control the flow of execution, and several other tasks.

AIX includes the popular UNIX debugger, dbx. The dbx debugger can be used with C, C++, FORTRAN, or Pascal programs. It requires source code for the specified program and can use a core file (core dump), if one exists.

Usually, the program must be compiled in a special way to make it usable by dbx. The program thus compiled is normally called the debug version of the executable. With the help of dbx, we can step into the program execution process and take a look at that line in the source code that is currently being executed.

AIX includes other debuggers, such as idebug, which is basically an X-Windows implementation similar to dbx in function.

dbx and idebug are covered in a later lecture.

Kernel Interface

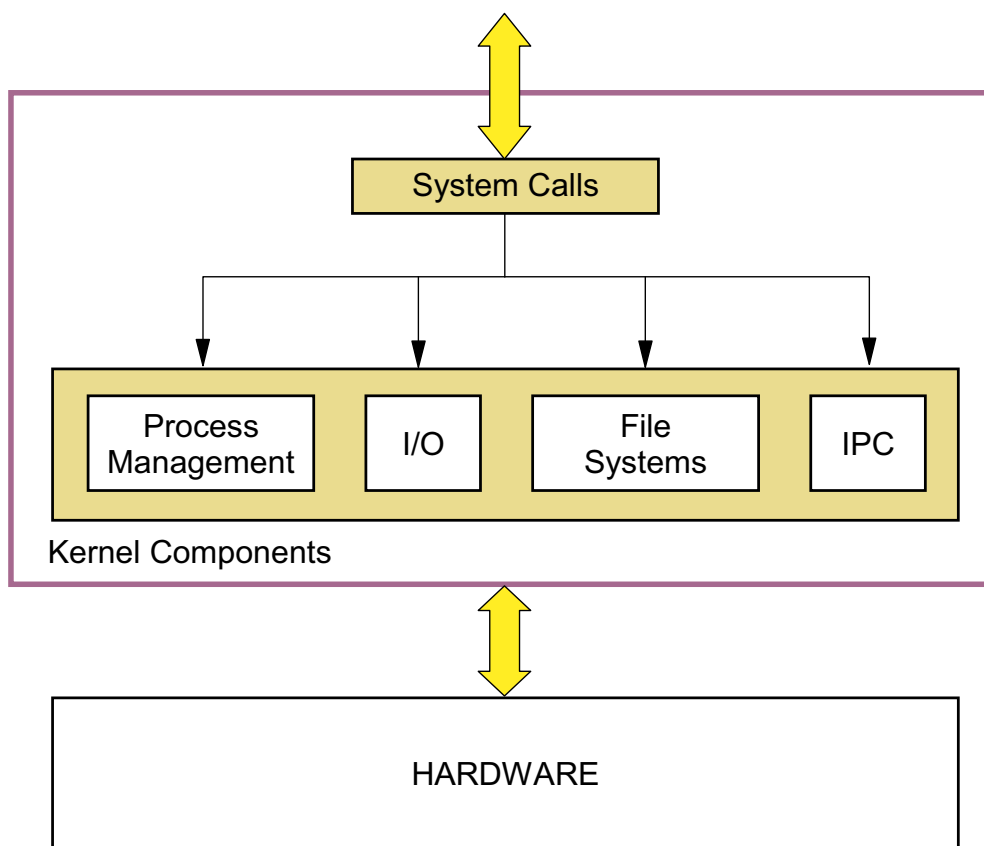


Figure 1-7. Kernel Interface

AU253.0

Notes:

The AIX kernel is the heart of the operating system. It contains various tables used to manage processes and system I/O. The area of memory used by the kernel is "off limits" to user applications. However, sometimes a user application may need information that is only available from the kernel. Since the kernel manages system I/O, user applications must also request I/O services from the kernel. This is how applications communicate with the system hardware.

The user application's interface to the kernel is through system calls. A system call is actually code in the kernel that can be executed by the kernel on behalf of a calling user application. A good example of this situation is when an application requests data from a disk file. The application performs a read system call, which causes the kernel to execute the code it has to perform a read. The data is then accessed and copied from the disk file into the application's memory space.

System calls exist for managing processes, working with the file system, requesting I/O, and communicating with other processes (Interprocess Communication, or IPC).

1.2 Overview of the Single UNIX Specification

The Single UNIX Specification

- Novell acquired the rights to the UNIX source code and trademark when it purchased UNIX Software Laboratories from AT&T in 1993.
- Later in 1993, the Common Open Software Environment project was started with the goal of defining what UNIX is (the results of the project were known as Spec 1170).
- The Spec 1170 work and the UNIX trademark were transferred to X/Open.
- Spec 1170 evolved and was published as the Single UNIX Specification in 1995 by X/Open (conformance with the Single UNIX Specification becomes the key factor in determining if a product may use the UNIX brand name).

Figure 1-8. The Single UNIX Specification

AU253.0

Notes:

Early in 1993, Novell purchased UNIX Software Laboratories from AT&T. As a result of the purchase, Novell acquired the rights to the UNIX trademark and source code.

At about the same time, a group of major UNIX vendors (Hewlett-Packard, IBM, Novell/USL, the Open Software Foundation, and Sun Microsystems) started the Common Open Software Environment project, with the mandate to develop a definition of UNIX that would be adopted by the project participants. This project specified roughly 1,170 interfaces and the resulting document was referred to as Spec 1170.

As it became clear that the COSE project was going to be successful, Novell transferred the UNIX trademark to X/Open, which also was given the Spec 1170 work. X/Open soon publishes the Single UNIX Specification, a set of documents that is to define what UNIX is. Today, a vendor wishing to use the UNIX brand name on their product must demonstrate that their product conforms with the Single UNIX Specification.

One source of considerable information on the Single UNIX Specification is the book *Go Solo with the Single UNIX Specification* by Stephen R Walli, published by X/Open and Prentice-Hall, ISBN 0-13-439381-3 (this book includes a CD-ROM that contains a

searchable copy of the Single UNIX Specification). Also, the X/Open WWW home page is at **<http://www.opengroup.org/>**.

The Single UNIX Specification Documents

The Single UNIX Specification consists of the following documents:

- *System Interface Definitions, Issue 5, Version 2 (XBD)*
- *System Interfaces and Headers, Issue 5, Version 2 (XSH)*
- *Commands and Utilities, Issue 5, Version 2 (XCU)*
- *Networking Services, Issue 5*
- *X/Open Curses, Issue 4*

Figure 1-9. The Single UNIX Specification Documents

AU253.0

Notes:

The five documents making up the Single UNIX Specification are:

- System Interface Definitions, Issue 5, Version 2 (often abbreviated XBD)
Defines key terms used by XSH and XCU as well as containing a large glossary of terms and concepts.
- System Interfaces and Headers, Issue 5, Version 2 (often abbreviated XSH)
Defines all of the programming interfaces and header files in the Single UNIX Specification except for the networking and terminal programming interfaces defined separately (see below). One of the aspects of XSH which we will be referring to throughout this course is the POSIX-1 standard. POSIX-1 defines the C language callable routines and interfaces, which are at the core of any implementation of UNIX.
- Commands and Utilities, Issue 5, Version 2 (XCU)
Defines the software development tools (including the C language and compiler) included in the Single UNIX Specification.

- Networking Services, Issue 5

Defines the networking services included in the Single UNIX Specification, including an interface to the OSI networking model, a Berkeley-style sockets interface, and IP address resolution.

- X/Open Curses, Issue 4

Defines a terminal-independent set of interfaces for writing character-based applications.

The Bottom Line

By adhering to the Single UNIX Specification, a software developer can:

- Have confidence that the resulting application will port cleanly to all major UNIX platforms
- Minimize the long term costs of maintaining an application that runs on a variety of platforms
- Reduce the risk of unexpected costs associated with as-yet unplanned ports of the application to other platforms

Figure 1-10. The Bottom Line

AU253.0

Notes:

Software that must run on a variety of platforms is rarely developed simultaneously on all of the target platforms (the results of such an approach are often quite chaotic). Committing to standards like the Single UNIX Specification can greatly simplify the issues surrounding the development of portable software by clearly indicating which interfaces are widely available.

Even if an organization is currently developing software that is not expected to run on other platforms, the pressures of the real world (that is, customers, changing price-performance ratios, newly opening markets, and the diversity of the global marketplace) may eventually result in the application being ported to other platforms. Software that adheres to standards like the Single UNIX Specification is typically dramatically easier to port to other platforms.

1.3 Finding Information

The man Command

```
$ man read

read Command

Purpose

Reads one line from standard input.

Syntax
    read [ -r ] VariableName...
Description

    ...
```

Figure 1-11. The man command

AU253.0

Notes:

The **man** command is a common tool on UNIX systems. It is used to access online manual pages for commands. The syntax for using the **man** command is:

```
man commandname or systemcallname
```

In AIX, the man pages also include information on system calls. It can be accessed by issuing the **man** command followed by the system call's name. If a given system call has the same name as an AIX command (one given from the shell's command line), you will see the man page for the command. After paging through that output, the system call version of the command is displayed.

An example might be the **read** command. **read** is a command and a system call. (Actually, the **man** command displays about a half dozen different implementations of the **read** command!) An easier way to go directly to the pages for system calls and libraries is to use a **2** with the **man** command before the system call name.

Example:

```
man 2 read
```

You can use a number of key combinations along with the display of the **man** command. Some of the keys used are as follows:

- <Ctrl>F - To view the next page.
- <Ctrl>B - To view the previous page.
- <Ctrl>C - To cancel the man display.
- <Enter> - To view the next line.
- Any number n - To view the next n lines of the man display.

AIX 5L Online Documentation

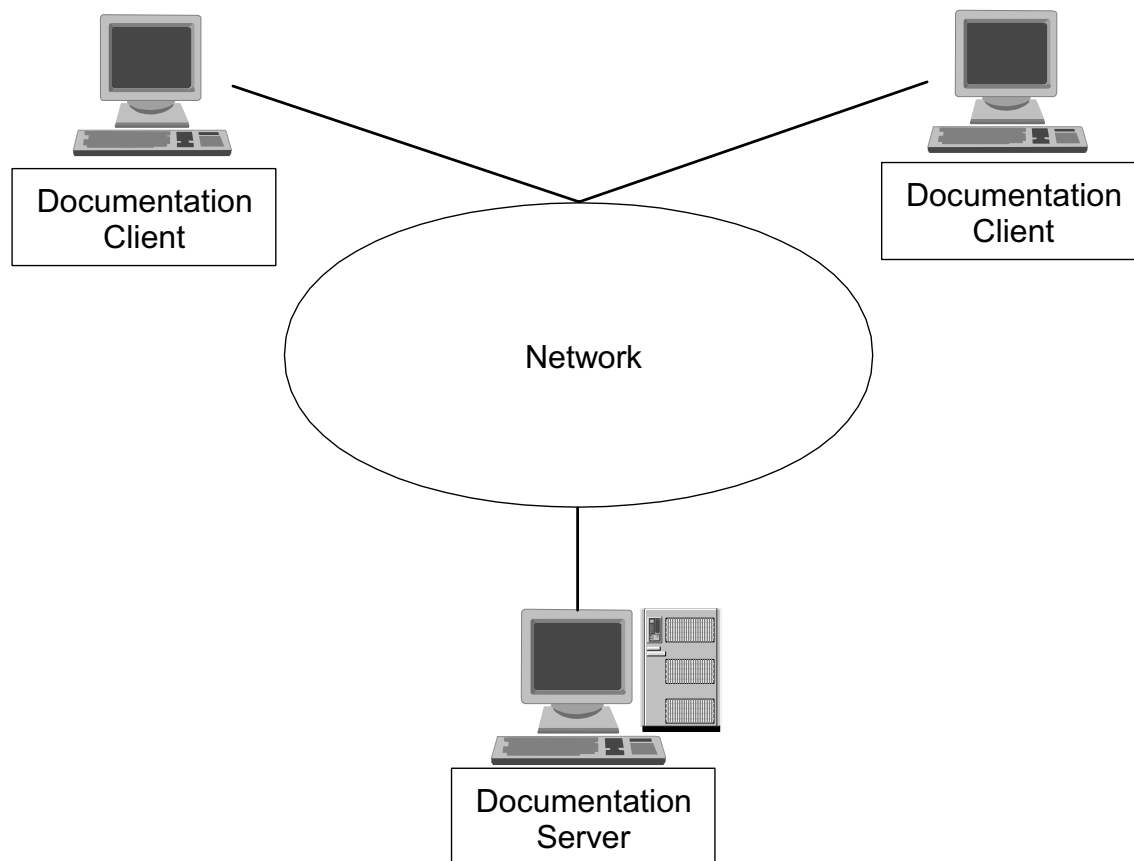


Figure 1-12. AIX 5L Online Documentation

AU253.0

Notes:

In addition to providing the **man** command for searching for information on commands and system calls, AIX also gives you the ability to look for documentation online. This ability makes searching even easier and simpler, because it has a Web interface to it.

The system administrator is required to set up the documentation server. This involves completing a series of installations. The documentation, in HTML format, is loaded on the documentation server. Any other computer on the network with an appropriate browser (for example, Netscape Navigator) can access the server as a documentation client.

A user on a client computer sends an AIX document request. The request is sent to the Web server, which in turn uses the documentation server. The requested document is searched for on the server and the results are sent back to the client.

Document Search and Scope

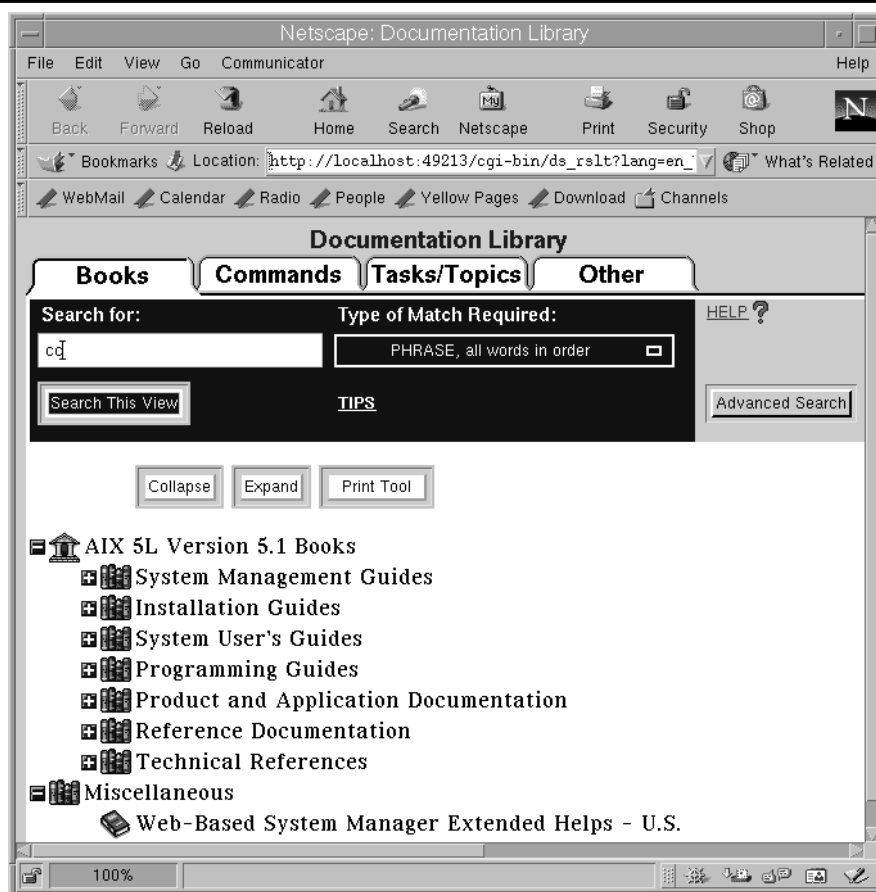


Figure 1-13. Document Search and Scope

AU253.0

Notes:

The documentation can be accessed in several ways:

- Using your Web browser to go to the URL **`http://<hostname>/cgi-bin/ds_form`**
- Using The “Search” function of the *Documentation Library* icon using CDE (Common Desktop Environment)
- The **`docsearch`** command

The host name could be localhost if the client is the same as the server, as is the case Figure 1-13. Otherwise, the host name will be that of the remote computer where the documentation server is installed. AIX 5L Version 5.1 online documentation is also available at the IBM Web site **`http://www.ibm.com/servers/aix/library/index.html`** (this URL can be changed). On this Web page, click the link to *AIX 5L Manuals*, which will take you to the Document library Web page.

The library home page provides options for simple and advanced search. A simple search looks for a single word; an advanced search looks for a maximum of three words.

The documents are categorized as Books, Commands, Tasks, and Topics. You can choose either of the views. The search is done accordingly.

Document Search Results

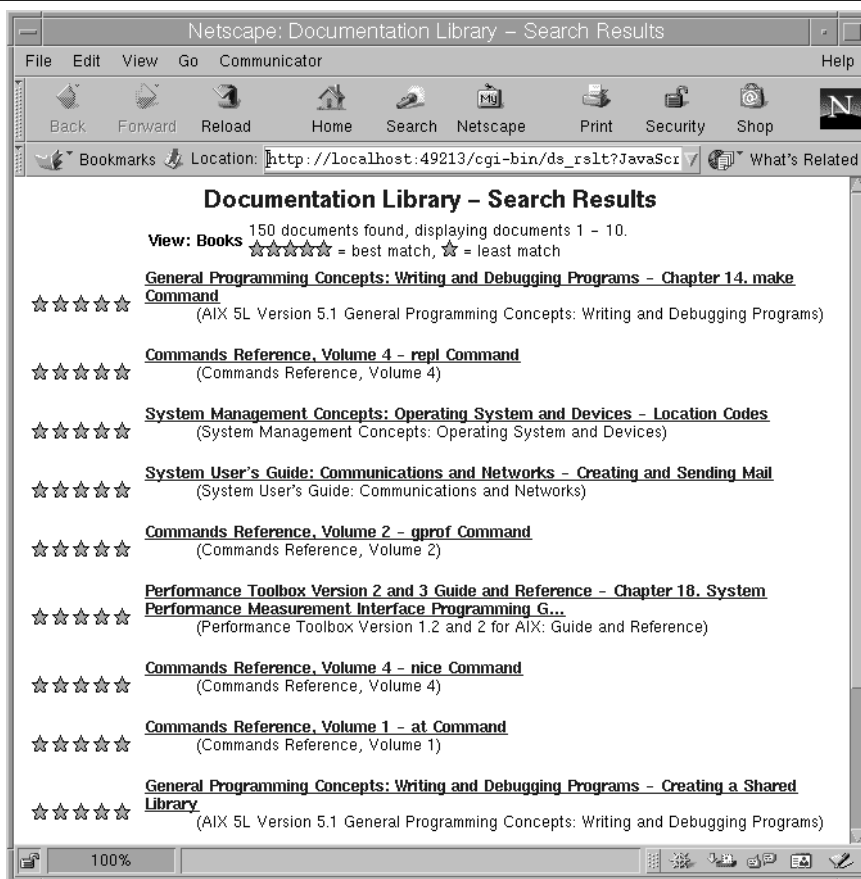


Figure 1-14. Document Search Results

AU253.0

Notes:

Use the Search window on the library home page to search for documentation on any command.

Figure 1-14 shows the result of the search for the **cc** command documentation. The star system is used to indicate the documents that best match your keywords. The more the number of stars, the better the match. Five stars is the best match.

Click on any item to view the document.

Unit Summary

- The AIX Programming Environment
- Program Creation and Compilation
- The AIX Run-time Environment
- Debugging
- System Calls
- The Single UNIX Specification
- The **man** Command
- AIX 5L Version 5.1 Online Documentation

Figure 1-15. Unit Summary

AU253.0

Notes:

Unit 2. Compiling Programs

What This Unit is About

There are many important differences between compiling C programs on most UNIX systems and compiling C programs in the AIX environment. This unit identifies the similarities and differences.

Options for the C for AIX compiler are defined, as well as options for the linkage editor. The AIX Extended Common Object File Format (XCOFF) is introduced, setting the stage for many other discussions throughout this course.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Describe how the AIX compiler process differs from a traditional UNIX compilation
- List the benefits of the AIX Extended Intermediate Language and the Common Linkage Conventions
- Use the **cc**, **xlc**, and **c89** commands to compile C programs
- Generate and read compiler listings
- Add a customized compiler configuration to the vac.cfg file
- Use the AIX preprocessor directives
- List and describe various optimization techniques used by the AIX compilers
- Identify components of the program and process images
- Describe the 64-bit programming model

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 1

References

- AIX 5L Version 5.1 documentation
- C for AIX documentation

- XL Fortran for AIX User's Guide
- Appendix E of ILS course AU14 for LUM (IBM License Use Management)

Unit Objectives

After completing this unit, you should be able to:

- Describe how the AIX compiler process differs from a traditional UNIX compilation
- List the benefits of the AIX Extended Intermediate Language and the Common Linkage Conventions
- Use the **cc**, **xlc**, and **c89** commands to compile C programs
- Generate and read compiler listings
- Add a customized compiler configuration to the vac.cfg file
- Use the AIX preprocessor directives
- List and describe various optimization techniques used by the AIX compilers
- Identify components of the program and process images
- Describe the 64-bit programming model

Figure 2-1. Unit Objectives

AU253.0

Notes:

2.1 Overview of Compiling Programs

Overview

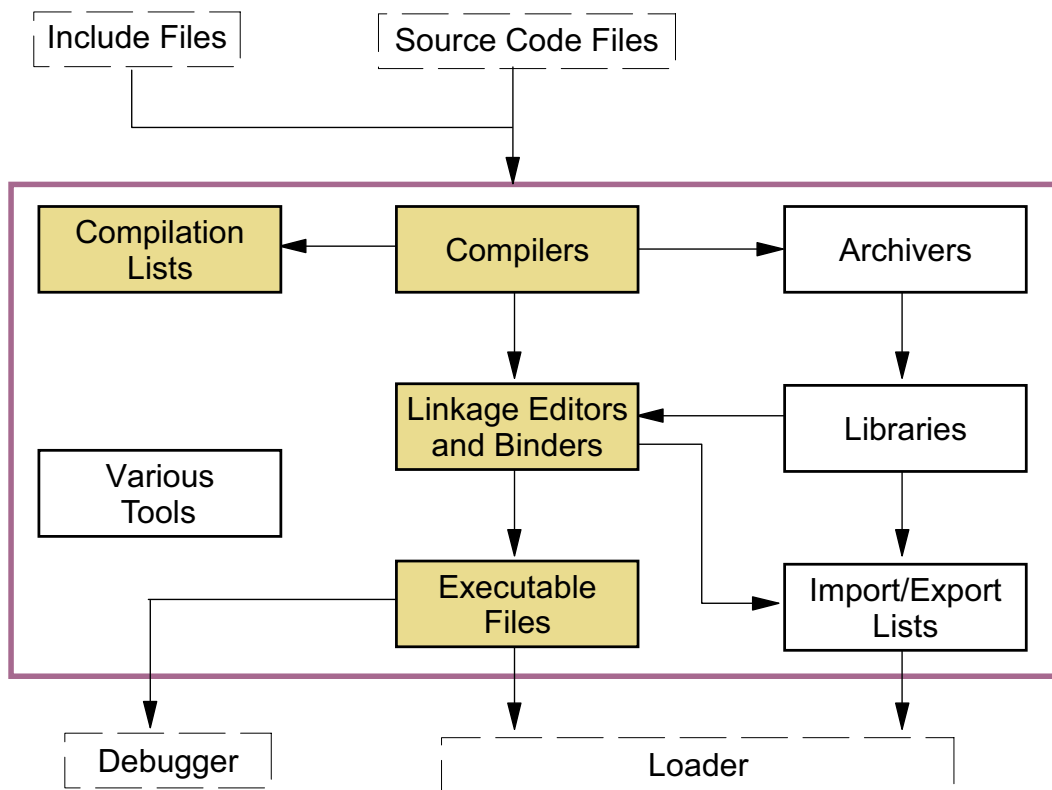


Figure 2-2. Overview

AU253.0

Notes:

This lecture discusses how to turn source code and header files into executable programs. The procedures and concepts of compiling programs are pretty much the same, regardless of the language or system. There are subtle differences in compiler behavior and options. This lecture introduces general compiling procedures as well as specific issues related to the AIX compilers, with an emphasis on the C compiler. The C compiler is no longer part of the operating system. The C for AIX compiler is a part of the XL family.

AIX provides a special relationship between the C, C++, FORTRAN, and Pascal compilers. Each of these languages has a compiler front-end that creates a common intermediate language. This intermediate language is then processed by a common optimizing back-end to produce object code.

The AIX compilers can generate various output reports, called listings, which provide specific compilation information.

Compiled programs are not automatically executable. The object files must first be linked with other object files or objects in one or more libraries. This linking process is performed

by the linkage editor. This lecture describes how the compiler and linkage editor work together to create the executable program.

Libraries are not discussed in this lecture. Libraries and special AIX linkage capabilities are presented in another lecture.

Traditional UNIX Compilation Process

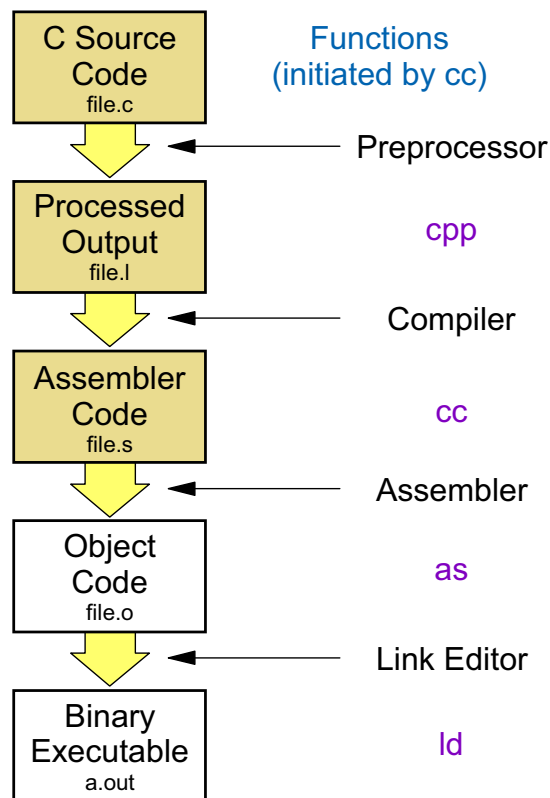


Figure 2-3. Traditional UNIX Compilation Process

AU253.0

Notes:

Figure 2-3 illustrates the steps involved in compiling a C program on most UNIX systems.

The UNIX C compiler, traditionally named `cc`, acts as a front-end to many other UNIX programs. When `cc` is invoked, it automatically calls a preprocessor called `cpp`. The preprocessor reads the source code files (source code file names must end in `.c`), looking for lines that begin with the `#` symbol. These lines represent preprocessor directives, which cause the `cpp` program to take certain actions. Two commonly found preprocessor directives are `#include` and `#define`. The `#include` preprocessor directive instructs the preprocessor to locate and read in code found in another file. The `#define` directive is used to globally define a symbolic constant. An example might be:

```
#define MAX 1000
```

where the preprocessor will replace all occurrences of the string `MAX` with the value `1000`.

To expand `#include <xyz.h>`, the compiler searches for the header file `xyz.h` in the standard directory (`/usr/include`) and the directories specified using the `-I` compiler option until it finds the specified file.

To expand **#include** “**xyz.h**”, the compiler searches for the *xyz.h* header file in the directory that contains the source code and then the standard or specified sequence of places until it finds the specified file.

The `cpp` program creates a file with the same base name as the source code file, but ending with `.i`. You normally will not see this file, as it is automatically removed by the compiler after compilation.

On most UNIX systems, the compiler compiles the `.i` file into assembler source code and stores the results in a file with the same base name as the original source file, but ending with `.s`.

The `cc` program then calls the assembler, called `as`. The assembler generates an object file with the same base name as the original source file, but ending with `.o`. The `.s` file is normally removed.

Finally, the `cc` program calls the linkage editor, `ld`, to link all object files and libraries needed to complete the executable program. The `ld` program also sets a flag within a header section of the resulting program to mark the file as executable. By default, the executable file's name is `a.out`. Users will generally override this default name by specifying a different desired name for the executable file to the linkage editor.

AIX Compilation Process

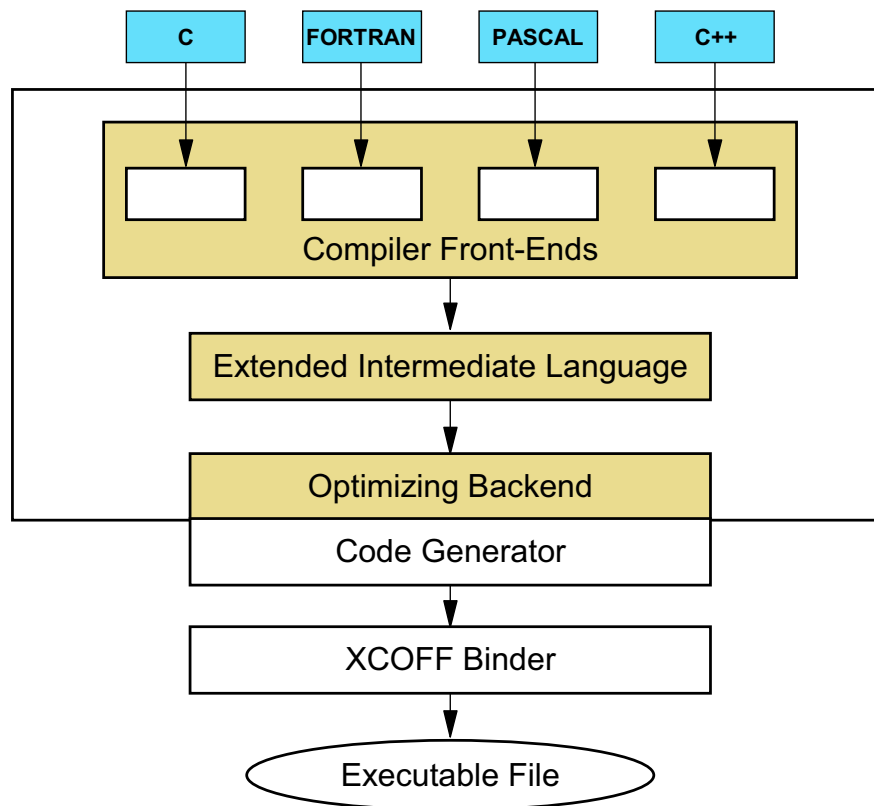


Figure 2-4. AIX Compilation Process

AU253.0

Notes:

There are many similarities between the traditional UNIX C compilation process and the AIX compilation process. However, there are some interesting differences, as described below:

As mentioned earlier, AIX employs a family of languages that uses separate compiler front-ends, but share a common intermediate language and optimizing back-end. This feature enables the use of a common linkage editor, which provides useful services such as interlanguage calling.

Each of these compiler front-ends generates a common output language, called the Extended Intermediate Language (XIL). A common compiler back-end is used to finish the process.

The AIX code generator is similar to the traditional UNIX assembler. It creates the code used by the linkage editor.

The AIX linkage editor calls the XCOFF binder, a utility that creates the executable program. The binder provides the dynamic binding feature discussed in later lectures.

System V UNIX introduced the concept of COFF (Common Object File Format) to standardize the structure of executable files. AIX expands this concept to include the IBM Table of Contents (TOC) section for fast address relocation. IBM COFF also includes a loader section to handle dynamic binding resolutions. For these reasons, IBM refers to their executable image as XCOFF (Extended Common Object File Format). The XCOFF structure is described in detail later in this lecture.

Like a traditional UNIX compilation, the linkage editor program is called `ld` and can be invoked separately.

The Extended Intermediate Language

- Common output from C, C++, FORTRAN, and Pascal compilers
- Intermediate language between language front-ends and optimizing back-ends
- Supports multiple architectures
- Tuned for optimization techniques

Figure 2-5. The Extended Intermediate Language

AU253.0

Notes:

As mentioned earlier, AIX provides compilers for C, C++, FORTRAN, and Pascal that generate a common intermediate language. This language is designed to be optimized by the common back-end of the compilers.

The resulting output supports the common linkage conventions described on the following page.

AIX also supports COBOL and ADA compilers; however, these compilers do not generate the Extended Intermediate Language that is generated by the C, C++, FORTRAN, and Pascal compilers, but C can call COBOL and vice versa.

Common Linkage Conventions

The compiler back-end and code generator provide code with common linkage conventions:

- Interlanguage calling support
- Listing and cross reference capabilities
- Common external symbol naming convention
- Common register and subroutine linkage
- Symbolic debugging support
- Exception handling and trace-back support

Figure 2-6. Common Linkage Conventions

AU253.0

Notes:

A goal of the extended language is to allow for a common linkage convention. In other words, programs compiled by the C, C++, FORTRAN, and Pascal compilers all produce the same intermediate language that can be used by a single linkage editor.

This common linkage convention allows interlanguage calling. For instance, a FORTRAN program module can make calls to a Pascal program module, or a C program module can call routines in a FORTRAN library.

Another feature of the common linkage convention is that it supports symbolic debugging for the four languages.

Exception handling refers to how the operating system reacts to illegal program instructions. Here, a common set of exception handlers supports the three languages.

Other Compilers on AIX

- XL Fortran for AIX Version 7
 - Invoked by **xlf**, **xlf90**, and **xlf95**
 - Source files use the .f suffix
 - /etc/xlf.cfg configuration file
- Pascal Compiler:
 - Invoked with **xlp**
 - Source files use the .pas suffix
 - /etc/xlp.cfg configuration file
- IBM VisualAge C++ Professional for AIX Version 5
 - Invoked with **xlc**
 - Source files use the .C suffix
 - /etc/vacpp.cfg configuration file
 - C compiler component provided by IBM C for AIX Version 5
 - Integrated Development Environment (IDE)

Figure 2-7. Other Compilers on AIX

AU253.0

Notes:

Figure 2-7 briefly mentions some details about other compilers on AIX.

The XL Fortran for AIX can additionally be invoked with **xlf90_r**, **xlf90_r7**, **xlf95_r**, **xlf95_r7**, **xlf_r**, **xlf_r7**, or **f77**. To compile FORTRAN source code with a suffix other than .f, use the -qsuffix option while invoking the compiler.

The VisualAge C++ Professional for AIX Version 5 is an Integrated Development Environment that operates with the incremental compiler when used in the AIX Common Desktop Environment (CDE). The batch compiler is run from the command line and is suitable for use in a development environment that uses makefiles.

Additional compilers include:

- COBOL Compiler:
 - Invoked with **cob**
 - Source files use .cbl suffix
 - Not part of the Extended Compiler Family

- ADA Compiler:
 - Invoked with **ada**
 - Source files use .ada suffix

Types of Licenses

IBM License Use Management (LUM)

- Nodelock license
 - Simple nodelocked license
 - Concurrent nodelocked license
 - Use-once nodelocked license
 - Per-server license
 - Trial period license
- Network license
 - Concurrent license
 - Reservable license
 - Use-once license
 - Per-seat license

Figure 2-8. Types of Licenses

AU253.0

Notes:

The types of licenses available are nodelocked licenses and network licenses.

Nodelock is a licensing mechanism that requires each workstation on which the licensed product operates to have its own unique key. Some of the types are described below:

- A simple nodelock license authorizes an unlimited number of simultaneous uses of the licensed application on the local machine.
- A concurrent nodelock license authorizes a fixed number of concurrent users for this software at the same time.
- An use-once license is a consumable form of license. A license once used cannot be reused.

Network licenses are stored on a network license server and are shared among multiple network license clients. These licenses are not restricted to a single machine. Some of the types are described below:

- A concurrent license is a network license that can be temporarily granted to run the licensed application on a client.

- A reservable license is a network license that can be reserved for exclusive use of a user, a group, or a node.

Steps to Setup License for C for AIX

1. Configure your system as a LUM Nodelock License Server
2. Start the concurrent nodelock server daemon
3. Enroll a product license in your concurrent nodelock server
4. Confirm if the license for this product has been installed
5. Test the compiler to see if there are any licensing errors

Figure 2-9. Steps to Setup License for C for AIX

AU253.0

Notes:

The C for AIX compiler:

- Is a Licensed Program Product
- Uses LUM for licensing

Most other compilers on AIX also uses the IBM License Use Management (LUM) for licensing the product. The license files for the C for AIX compiler are in the /usr/vac directory (cforaix_c.lic, cforaix_cn.lic, and cforaix_n.lic).

LUM is installed by default on AIX.

To set up the C for AIX compiler license (in the case of a concurrent nodelock license), follow these steps:

1. Configure your system as a LUM Nodelock License Server

Use the command **i4cfg** or **i4config**. **i4cfg** is the GUI version of **i4config**. For example, this can be done as follow:

```
# /usr/opt/ifor/ls/bin/i4cfg -a n -S a
```

2. Start the Concurrent Nodelock Server daemon

Use the command **i4cfg** or **i4config**. For example, this can be done as follow:

```
# /usr/opt/ifor/ls/bin/i4cfg -start
```

3. Enroll a product license in your Concurrent Nodelock Server

Use the command **i4blt**. (Note: This command can take about 2 minutes. Depending on the system, this command may take up to 15 minutes or more to complete execution.)

For example, this can be done as follow:

```
# /usr/opt/ifor/ls/bin/i4blt -a -f /usr/vac/cforaix_cn.lic -R  
u -T 10
```

4. Confirm if the license for this product has been installed

Use the command **i4blt**. For example, this can be done as follow:

```
# /usr/opt/ifor/ls/bin/i4blt -s -l cn
```

5. Test the compiler to see if there are any licensing errors

Try to compile a small test program. The command **i4blt** can be used to delete the license, too. All the commands mentioned above are located at **/usr/opt/ifor/ls/bin**. If there is no license installed or if the license is set up incorrectly, the following can happen:

- The following warning message will be printed on the screen:

```
1506-507 (W) No licenses available. Contact your program  
supplier to add additional users. Compilation will proceed  
shortly.
```

- The compilation process may seem to hang.
- The compiler may dump core.

All the commands mentioned above are located at **/usr/opt/ifor/ls/bin** directory. The **i4blt** command can be used to delete the license too.

2.2 Compiler Commands and Options

The cc/xlc/c89 Commands

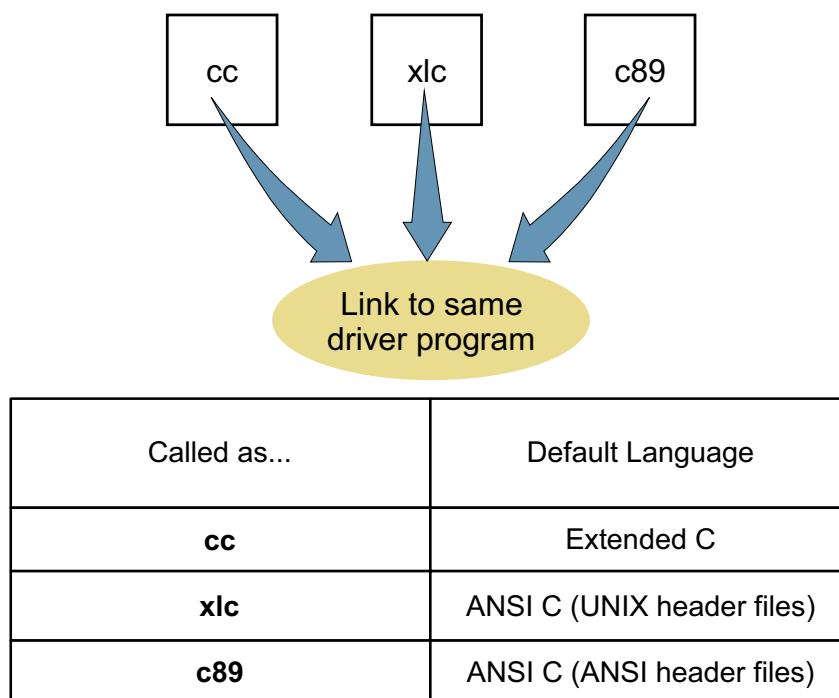


Figure 2-10. The cc/xlc/c89 Commands

AU253.0

Notes:

AIX has a single C compiler, but it can be called by at least three different names. AIX is shipped with three file names, **cc**, **xlc**, and **c89**, which are all linked to the same program. The compiler does, however, pay attention to how it is invoked and acts differently, depending on how it is invoked. The C ++ compiler may be invoked using **xlc**.

If called via the **cc** command, the compiler expects the source code to be in extended K and R (Kernighan and Ritchie) format. If called via **xlc** or **c89**, the compiler expects the source code to be ANSI-compliant. Appropriate errors and warnings are reported if violations in the expected level are found.

There are other important reasons for having different compiler invocation names. C for AIX supplies a configuration file named **/etc/vac.cfg** that defines how the compiler works with different invocations. The configuration file specifies default options, thus reducing the amount of information a user must supply when invoking the compiler. Various symbolic constants are defined for the preprocessor, based on which invocation is used to call the compiler. A programmer can add stanzas to the configuration file and define new ways to

invoke the compiler by linking the compiler to a new command name. This concept is described later in this lecture.

Compiler Configuration

The /etc/vac.cfg file:

```
xlc:      use          = DEFLT
          crt          = /lib/crt0.o
          mcrt         = /lib/mcrt0.o
          gcrt         = /lib/gcrt0.o
          libraries    = -lc
          proflibs     = -L/lib/profiled,-L/usr/lib/profiled
          options      = -qansialias

cc:       .
          .
          .
          options = -qlanglvl=extended, -qnor, -qnorconst

c89:     .
          .
          .
          options = -D_ANSI_C_SOURCE, -qansialias,
-qnolonglong, -qstrict_induction
          .
          .
          .
```

Figure 2-11. Compiler Configuration

AU253.0

Notes:

As mentioned earlier, configuration files exist for the C, C++, FORTRAN, and Pascal compilers. These files are found in the /etc directory.

Figure 2-11 illustrates an extraction from a typical /etc/vac.cfg file used for C.

xlc128, cc128, and xlc128 indicate support for the use of 128-bit floating point numbers defined as long double data types. Another way to control ANSI versus extended C is by using -qlanglvl during compile. All methods of invoking the C compiler are still linked to the same program. The differences become apparent when viewing the /etc/vac.cfg file.

Customization for thread safe C compilers are also present in the /etc/vac.cfg file. These are denoted by appending a _r to the names (that is, xlc_r, cc_r, and so on).

These files have a stanza format. Each stanza begins with a linked name for the C compiler (cc, xlc, and c89). Following the stanza name are lines of attributes and values for the attributes.

A system administrator can modify the attributes in these files to customize compiler behavior. New default compiler sets can be created by linking an existing compiler to a new

compiler name (see the man page for the **ln** command) and then creating a new stanza in the appropriate configuration file for the newly linked compiler name.

Note that the order of specifying the parameters in the configuration file is the order in which these options are passed on to the compiler.

cc/xlc/c89 Options (1 of 2)

Supported options from standard UNIX:

-O	Optimizes
-o filename	Specifies executable's name*
-c	Compiles without linking
-l key	Specifies library name for linking (in form libkey.a)*
-L libdir	Specifies directory for additional libraries*
-E	Sends preprocessor output to standard out
-P	Saves preprocessor output to .i file
-b option	Specifies option for the binder (ld)*
	* Option passed to the ld command
-S	Produce a .s (assembler) file

Figure 2-12. cc/xlc/c89 Options (1 of 2)

AU253.0

Notes:

These pages present various options available for the C for AIX compiler. Most of these option flags are identical to other UNIX systems' cc programs. Also, since the compiler calls the linkage editor (ld), many of the options used with the C compiler are passed directly to the linkage editor.

The -O option tells the compiler to optimize the resulting object code. Methods that the AIX compiler uses to optimize code are discussed later in this lecture. Optimized code is difficult, if not impossible to debug (since optimization techniques frequently rearrange the code!); therefore, the -O option should not be used until all debugging is completed.

The -O2 is an equivalent level of optimization as -O. The -O3 performs some memory and compile time intensive optimizations in addition to those executed with -O2. Note that the -O3 specific optimizations have the potential to alter the semantics of a user's program.

The -o option allows the user to specify a different file name for the resulting output of the linkage editor. By default, the resulting output file for executables is a.out.

The `-c` flag is used to tell the compiler not to call the linkage editor upon completing the compilation process. This results in unlinked object files that might be used to create libraries or used to link to other modules later. If the `-c` option is not used, the compiler calls the linkage editor and the object files (`.o` files) are automatically removed after linking occurs.

The `-l` and `-L` flags are used to specify libraries and directories where the libraries can be found by the linkage editor, respectively. The argument to the `-l` flag is the key portion of the library name (`libkey.a`). The argument to the `-L` flag is the path name of the directory to search for additional libraries. The linkage editor automatically searches the `/usr/lib` directory.

Example

```
cc -o myExecutable -O2 -L/home/self/lib -lmyLibrary myfile.c myobj.o
```

`-E` and `-P` are used to tell the preprocessor to pipe a copy of its results to either standard output (`-E`) or a specified file (`-P`). In either case, the preprocessor still creates the `.i` file used by the compiler. The `.i` file is always removed by the compiler after its use unless the `-P` option is given.

The `-b` option is used in AIX to pass special commands to the binder program.

cc/xlc/c89 Options (2 of 2)

Supported options from standard UNIX:

- v Verbose: shows compiler execution steps and arguments
 - p Enables run-time profiling
 - g Enables source level debugging
 - i Specifies the directory for additional include files *
 - D Defines a symbol to the preprocessor *
 - U Undefines a symbol to the preprocessor *
- * Option interpreted by the preprocessor

Figure 2-13. cc/xlc/c89 Options (2 of 2)

AU253.0

Notes:

The -v option tells the compiler to report the steps of compilation as they occur. As with other UNIX and AIX commands, the -v option means "verbose."

The -p option enables run-time profiling of CPU usage via the **prof** command. This allows a programmer to tune an application by collecting timing data on modules and called routines. This option should be used during the tuning phase of application development.

The -g option is also used during the testing/debugging phase to create a section of the XCOFF file used by the dbx symbolic debugger. This option must be specified when compiling programs in order to list the source code lines when using the dbx debugger on the programs later. Once a program has been fully debugged, recompiling without this option makes the executable smaller and faster.

The **gprof** command produces an execution profile of a program. To use **gprof** command on a program, the program must be compiled with the cc command using the -pg option.

The **tprof** command reports CPU usage for individual programs and the system as a whole. It does this by using the system clock and therefore does not require code to be

added to the program via recompilation. It is also capable of producing a source listing which includes the amount of time the CPU spent on each line. This functionality, however, does require the source to be recompiled with -g option. For more information, refer to AIX online documentation.

The -I option is used to tell the preprocessor to search a specified directory for include files (.h files) when the include file names are supplied within < > in the source code. By default, the preprocessor searches /usr/include/ for include files. Any include file can be specified in the source code with quotes ("") to indicate a relative path name from the current directory.

The -D and -U options are used to define and undefine (respectively) symbols to the preprocessor. The use of these options is described later in this lecture.

64-bit Compilation

- The default compilation mode is 32-bit.
- The `OBJECT_MODE` environment variable:
 - If equal to 32, generate/use 32-bit objects
 - If equal to 64, generate/use 64-bit objects
- Compiler option on command line:
 - `-q32`: Generate/use 32-bit objects
 - `-q64`: Generate/use 64-bit objects
 - Override the compiler mode set by the value of the `OBJECT_MODE`
- In 64-bit mode, the `__64BIT__` preprocessor macro is defined automatically.

Figure 2-14. 64-bit Compilation

AU253.0

Notes:

The `OBJECT_MODE` environment variable, if it exists, can set the default compilation mode.

The `-q32` and `-q64` options override the compiler mode set by the value of the `OBJECT_MODE` environment variable, if it exists. If the `-q32` and `-q64` options are not specified, and the `OBJECT_MODE` environment variable is not set, the compiler defaults to 32-bit output mode.

If the compiler is invoked in 64-bit mode, the `__64BIT__` (Note: two “`_`” present before and after “64BIT”) preprocessor macro is defined automatically.

To execute a 64-bit application, the target hardware should support 64-bit applications. Remember that it is possible to compile 64-bit applications on a hardware that does not support the execution of 64-bit executables.

Notes:

- If you mix 32-and 64-bit compilation modes, your XCOFF objects will not bind. You must recompile completely to ensure that all objects are in the same mode.

- Your link options must reflect the type of objects you are linking. If you compiled 64-bit objects, you must link these objects using the 64-bit mode.

Compiler Listings (1 of 3)

-qsource:	Generates source listing
-qattr:	Generates symbol attribute listing
-qxref:	Generates symbol cross reference listing
-qlist:	Generates object code listing
-qlistopt:	Generates list of compiler options in effect

- Any of these options create the file sourcename.lst.
- These options can be used in combination.
- Sections are added to sourcename.lst according to specified options.

Figure 2-15. Compiler Listings (1 of 3)

AU253.0

Notes:

An interesting feature of the AIX compilers is their ability to generate report listings of information pertaining to the compilation. The figure shows the options accepted by the **cc** command to request various forms of these listings.

Issuing any one of these options causes the compiler to create a file with the same base name as the original source file but ending in .lst (for list). Many, or all, of these options can be issued for a single **cc** command. Only one .lst file is created. Its contents depend on which listing options are specified.

You will have an opportunity to examine examples of these listings in the lab exercise accompanying this lecture.

The general structure of the listings is described on the following pages.

Some -q options do not create listings. One example is the -qcheck, which adds code into your program to test for a zero divide. Another is -qstrict, which is used with -O3 to prevent the aggressive optimization from altering a program's semantics (and results).

Compiler Listings (2 of 3)

<ul style="list-style-type: none"> • Header section <ul style="list-style-type: none"> • Identifies compiler • Identifies source files • Identifies time stamp
<ul style="list-style-type: none"> • Source section (only if -qsource used) <ul style="list-style-type: none"> • Includes source code with line numbers • Imbeds include files • Expands macros • Includes comments • Deletes preprocessor directives • Includes inline diagnostics
<ul style="list-style-type: none"> • Options section (only if -qlistopt used) <ul style="list-style-type: none"> • Identifies options specified to compiler
<ul style="list-style-type: none"> • Attribute and cross-reference section <ul style="list-style-type: none"> • Includes symbol names • Includes attributes (only if -qattr used) <ul style="list-style-type: none"> -Displays data type -Displays storage class • Provides cross-reference (only if -qxref used) <ul style="list-style-type: none"> -Includes file names and line numbers -Notes variable sets with #

MORE

Figure 2-16. Compiler Listings (2 of 3)

AU253.0

Notes:

Every compiler listing starts with a header section. This section identifies which compiler generated the listing and what source files were used. It also includes a date and time stamp.

If the -qlistopt option was issued on the command line, the listing will include an options section. It lists all options issued on the command line, as well as those invoked automatically, because they are in the configuration file (that is, /etc/vac.cfg, /etc/xlf.cfg, or /etc/xlp.cfg).

All symbol names are included in the listing. If the -qattr option is issued, the listing will include attributes of all symbols. The attributes would be the data type and storage class of each symbol. If the -qxref option is issued, the listing will include the source file name and line number for all external symbols.

Compiler Listings (3 of 3)

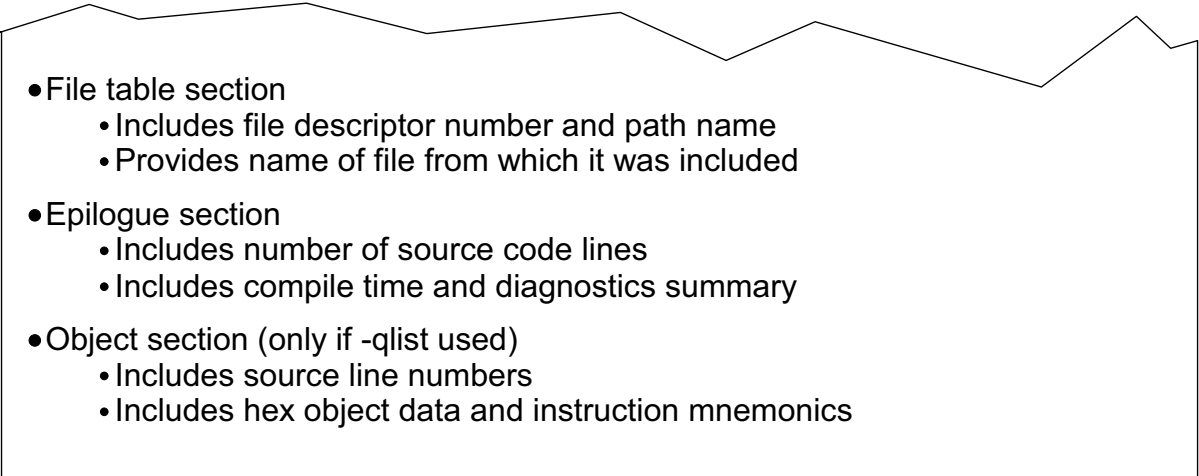
- 
- File table section
 - Includes file descriptor number and path name
 - Provides name of file from which it was included
 - Epilogue section
 - Includes number of source code lines
 - Includes compile time and diagnostics summary
 - Object section (only if -qlist used)
 - Includes source line numbers
 - Includes hex object data and instruction mnemonics

Figure 2-17. Compiler Listings (3 of 3)

AU253.0

Notes:

If the -qlist option is issued, the listing will include the pseudo assembler code.

By default, the listing will report which files the application opens and to which file descriptor the opened files are assigned.

All listings include a closing section, called the epilogue. This section indicates the number of source code lines compiled and includes compile time data and a summary of error and warning messages.

Compiler Diagnostics

- AIX standard compiler diagnostics style

```
$ cc buggy.c

"buggy.c",line  6.8: 1509-045 (S) Undeclared identifier to.

$
```

- AIX alternate compiler diagnostics style
-qsrcmsg flag

```
$ cc -qsrcmsg buggy.c

6 |   strcpy(to, from);
   |   .....a.....

a-1506-045 (S) Undeclared identifier to.

$
```

Figure 2-18. Compiler Diagnostics

AU253.0

Notes:

In the earlier C compiler, XL C in AIX Version 3.1, compiler error and warning messages displayed the source code line and used letters (here, a) to "point to" the trouble. The letters are then used to list the type of error or warning. While this style of error messaging is very useful, it does not comply with accepted UNIX standards.

For C in AIX Version 3.2 and later, compiler error and warning messages were modified to comply with these standards. This also allows the AIX compilers to properly interface with other UNIX program development tools.

The current C for AIX error and warning messages include the file name, line number, and character position (as in line:char), and the type of error warning.

To have compiler and error warning messages displayed in the alternate Version 3.1 fashion, use the -qsrcmsg option.

If you always want to use this flag, you may consider adding it to the compiler configuration file.

Compiler Modes

- `-qarch=pwr` POWER (RS1) mode
- `-qarch=pwr2` POWER2 (RS2) mode
- `-qarch=ppc` PowerPC mode
- `-qarch=com` Common mode

Figure 2-19. Compiler Modes

AU253.0

Notes:

The C for AIX compiler has several new flags to target code generation for the various instruction sets. A description of all the currently supported flags and their impact on binary compatibility and performance follows:

POWER (RS1) mode `-qarch=pwr`

This flag targets code generation to the POWER instructions set. Applications compiled with this flag will run on POWER, POWER2, and PowerPC-601 without the need for emulation. However, software emulation may be required when running on future PowerPC implementations, such as the 603 and 604.

This mode only supports 32-bit compilation.

POWER2 (RS2) mode `-qarch=pwr2` (same as `pwr`)

This flag targets code generation to the POWER2 instruction set. Applications compiled with this flag will *only* run on POWER2 based systems if any of the unique POWER2 instructions are generated, because there is no software emulation provided for these instructions on any other hardware.

This mode only supports 32-bit compilation.

PowerPC mode -qarch=ppc

This flag targets code generation to the PowerPC instruction set. Applications compiled with this flag will only run on PowerPC based systems (601, 603, and 604) because there is no software emulation provided for the new PowerPC instructions on POWER-based and POWER2-based systems.

This mode supports both 32-bit and 64-bit compilation.

Common mode -qarch=com (default)

Common mode targets code generation on the intersection subset of instructions, which are common between POWER, POWER2, and PowerPC.

This mode supports both 32-bit and 64-bit compilation.

It is possible to tune the performance for a particular processor. Performance improvement can be achieved by rearranging the sequential order of instructions to allow for more parallel execution among the various execution units. This type of optimization is referred to as *scheduling*. Because the number of execution units and their pipeline depth varies from one implementation to the next, it is important to be able to specify which target CPU is to be favored by the compiler for scheduling. This is accomplished by using the -qtune compiler flag.

A summary of the binary compatibility is shown in the following table:

Table 1: Binary compatibility

Code can run on	Common	POWER (RS1)	POWER2 (RS2)	PowerPC 32-bit
POWER (RS1)	OK	OK	NO	NO
POWER2 (RS2)	OK	OK	OK	NO
PowerPC 601	OK	OK	NO	OK
PowerPC 603, 604	OK	OK (Emulation)	NO	OK

Preprocessor Directives

<code>#define</code>	<code>#include</code>
<code>#ifdef</code>	<code>#undef</code>
<code>#else</code>	<code>#pragma</code>
<code>#endif</code>	<code>#elif</code>
<code>#ifndef</code>	<code>#error</code>
<code>#if</code>	

Figure 2-20. Preprocessor Directives

AU253.0

Notes:

The C for AIX compiler includes a built-in preprocessor. It does not call the standard UNIX X preprocessor (cpp). The built-in preprocessor directives are a superset of the standard UNIX C preprocessor directives.

Preprocessing is a step that takes place before compilation. The preprocessor is controlled by the following directives:

#define	Defines a pp directive
#undef	Removes a pp macro definition
#error	Defines text for compiler-time error messages
#include	Inserts text from another source file
#if	Conditionally suppresses portions of the source code
#ifdef	Conditionally includes source text if a macro name is defined
#ifndef	Conditionally includes source text if a macro name is not defined

#else	Conditionally includes source text if previous #if, #ifdef, #ifndef, or #else test fails
#elif	Similar to #else followed by a #if
#endif	Ends conditional text
#pragma	Specifies implementation-defined instructions to the compiler Example: #pragma langlvl (ANSI) - Sets C language level for compilation #pragma comment (-compiler or -date, and so on) - Places comment in object file

#pragma disjoint

The #pragma disjoint directive as a special command to the compiler. This directive allows a programmer to guarantee that a specified pointer will never reference a specified variable, thus reducing the number of reload instructions generated by the compiler.

```
int a,b, *ptr_a, *ptr_b;
#pragma disjoint (*ptr_a, b) /*ptr_a never points to b*/
#pragma disjoint (*prt_b, a) /*prt_b never points to a*/
one_function
{
    b=6;
    *ptr_a = 7;          /*assign. doesn't change value of b*/
    another_func(b);     /*arg b has value b*/
}
```

Preprocessor Options

- To define a symbol:

`cc -DSYMBOL NAME`

- To undefine a symbol:

`cc -USYMBOL NAME`

Figure 2-21. Preprocessor Options

AU253.0

Notes:

Figure 2-21 illustrates the use of preprocessor directives. They are frequently used to facilitate debugging routines.

For example: `cc -DEXTENDED`

```
#ifdef EXTENDED
#define MAX_LEN=75
#else
#define MAX_LEN=50
#endif
```


Optimization

- AIX compiler optimization techniques include:

- Interprocedural analysis (-qipa)
- Reassociation
- Commoning
- Code motion
- Strength reduction
- Dead code elimination
- Constant propagation
- Register allocation
- Branch optimizations
- Value numbering
- Common subexpression elimination
- Invariant IF code floating (unswitching)
- Store motion
- Dead store elimination
- Inlining (-Q option)
- Instruction scheduling
- Global register allocation

Figure 2-22. Optimization

AU253.0

Notes:

The C for AIX compiler does not optimize by default. Optimization is invoked by issuing the -O option. The -O3 option may be used to direct the compiler to be aggressive about optimization techniques and use as much memory as necessary for maximum optimization.

When the -O option is used, the compiler analyzes the control and data flow of each function, performing the following optimizations:

Reassociation: Rearranges the sequence of calculations in a subscript expression, producing more candidates for common expression elimination.

Commoning (Or Common Subexpression Elimination): Duplicate expressions are eliminated by using results from previous expressions. For example, if the following expressions are found in an application:

x = a + b;

. . .

y = a + b + c;

the result from the common expression $a + b$ is saved and used in the latter expression.

Strength Reduction: Replaces complex instructions, like multiply, with simple instructions, like add.

Dead Code Elimination: Eliminates code that cannot be reached or where results are not subsequently used.

Register Allocation: Global register allocation is implemented for entire functions by use of the "graph coloring" algorithm.

Branch optimization: Rearrange code to minimize branching logic.

Interprocedural analysis: Uncovers relationships across function calls.

Invariant IF Code Floating (Unswitching): Removes invariant branching code from loops to make more opportunity for other optimizations.

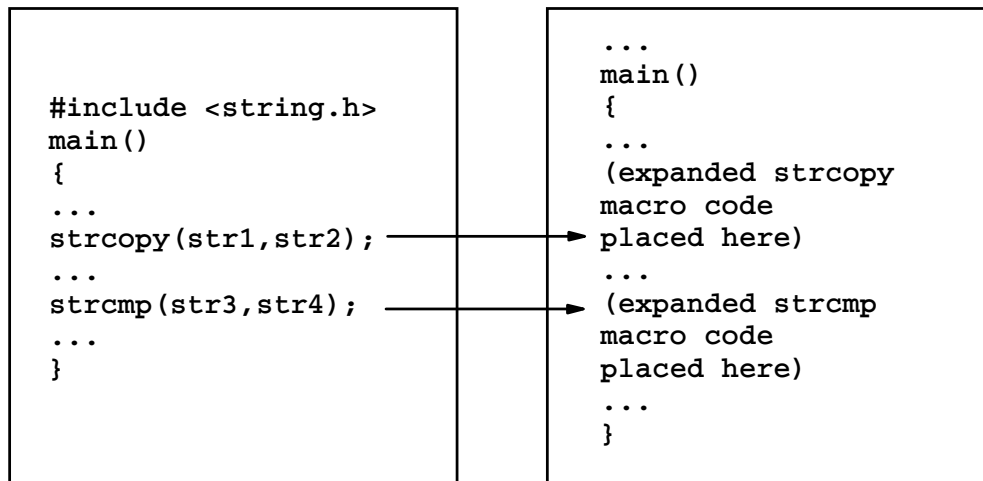
Example:

```
...  
return 0;  
x=1;
```

In the above code snippet, the line $x=1$ can never be reached. Hence, the assignment $x=1$ is removed by the optimizer.

The C for AIX compiler documentation explains each of the optimization techniques in detail.

Code Inlining



Preprocessor directives can control the degree of inlining.

Figure 2-23. Code Inlining

AU253.0

Notes:

Inlining is another form of optimization used by the AIX compilers. Inlining means that code from small functions or routines is placed directly into the code flow of the program by the compiler. This procedure reduces program run time by eliminating the need to branch to the subroutines.

The AIX C compiler provides a `-Q` for 20 or fewer lines default or a `-Q=#` option that can be used to control how inlining works for a particular compilation. While inlining speeds up execution time, if the inlined routines are called frequently, the result might be an undesirably larger program.

To maximize inlining, specify `-O` with `-Q`.

For example, `cc myprog.c -O -Q=12` (compiler attempts to inline functions of fewer than 12 lines).

The C for AIX compiler can improve optimization by generating substitute code for calls to some math and string functions available within the standard C run-time libraries. The

functions handled this way are defined as macros in /usr/include/math.h or /usr/include/string.h.

The special handling of these functions occurs by default. To prevent this special handling, the preprocessor macros `__MATH__` and/or `__STR__` should be undefined from the compiler command line. For example: **`xlc -c -U __STR__ file.c`**.

2.3 The Executing Program

Program and Process Images

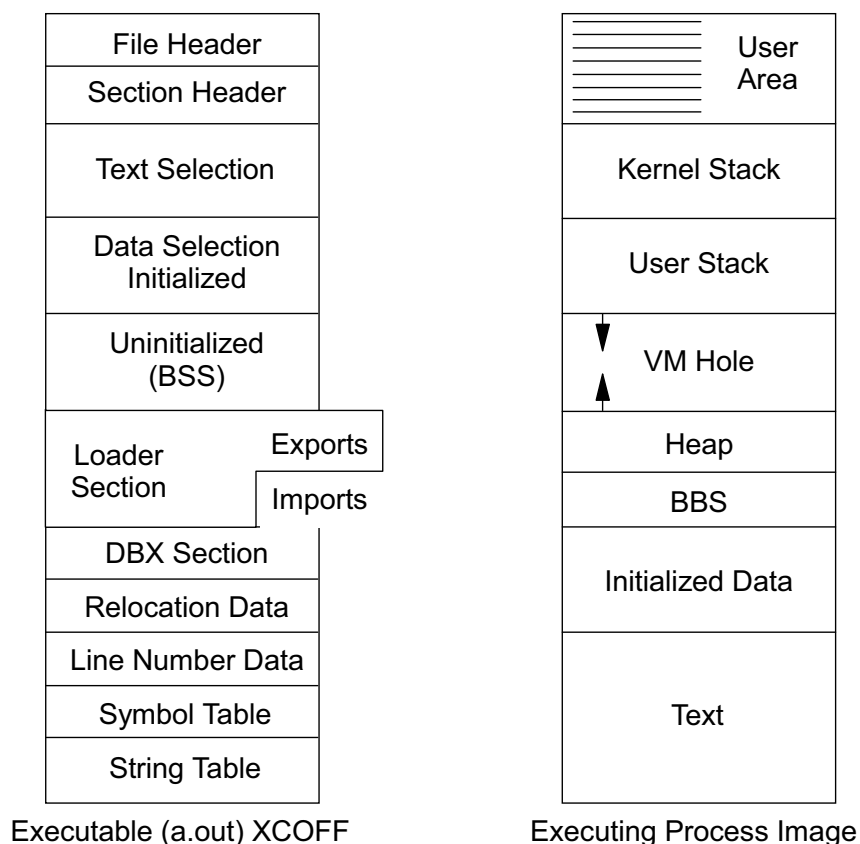


Figure 2-24. Program and Process Images

AU253.0

Notes:

Figure 2-24 illustrates two important images; the structure of the XCOFF file and the image of an executing process. The other important image, which is the virtual memory allocation of an executing process, will be described in Figure 2-25 on page 48.

The XCOFF image details the components of an executable program file. Sections include text, data (both initialized and uninitialized), loader (used for dynamic binding), and other sections for debugging and memory relocation. The only portion of this drawing not common to other UNIX systems' implementation of COFF is the loader section.

The executing process image illustrates how a running process accesses its allocated memory. Low memory is used to store the text of the program. Above that is the data region, with initialized data, then uninitialized data. Uninitialized data is referred to as BSS (block started by symbol). In the XCOFF file, the BSS only represents the definitions of the uninitialized data. During run time, this region is expanded to the required size for storing the uninitialized data elements.

To illustrate how data is stored, initialized data (such as `int x = 5;`) is stored by the compiler into the initialized data area, while uninitialized data (such as `struct emprec[100];`) is

recorded by the compiler into the BSS of the XCOFF file. At run time, enough memory is allocated to the process to handle the array of 100 employee records.

The user area of a process contains the environmental data (including a table of file descriptors) used by the process. This transient data is not part of the program's image (XCOFF), but is part of the process's image. (Remember that a process is an executing program!)

For 64-bit applications, a new executable file format, called XCOFF64, is available. It is an extension of the XCOFF format used in AIX for 32-bit executable files.

If you feel you may need some additional room for large data programs, look at the `maxdata` and `maxstack` options on the `ld` command. Also double check the data and stack options in the `/etc/security/limits` file.

Consider using the **strip** command to reduce the size of your program on disk. Remember that after the strip, no debugging can be done directly on that program.

More information about COFF is available in the book "*Understanding and Using COFF*", O'Reilly & Associates, Inc. The detailed structure of XCOFF can be found in the `/usr/include/xcoff.h` AIX header file.

Virtual Address Space

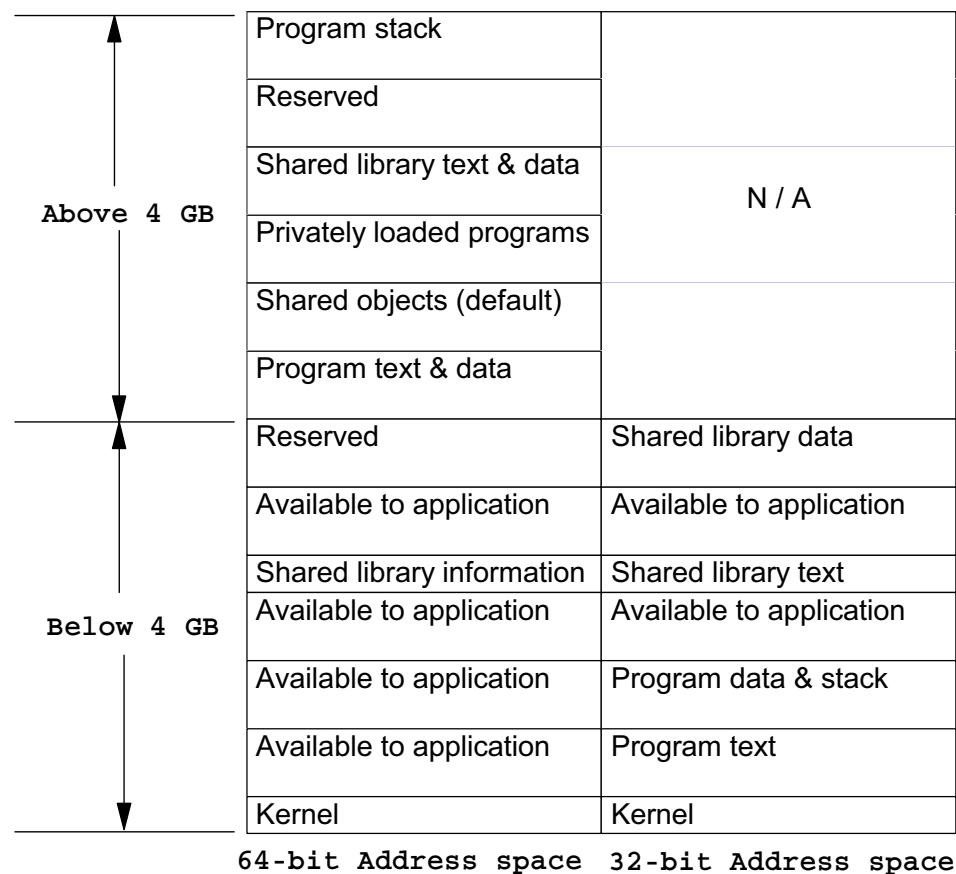


Figure 2-25. Virtual Address Space

AU253.0

Notes:

Figure 2-25 illustrates how virtual memory is used when the program is loaded and running.

In a 32-bit process, virtual memory is actually divided into 16 virtual memory segments. Each segment is 256 MB, for a total virtual address space of 4 GB. The important segments to remember for AIX application programming are:

Segment 1: Stores the text of the process.

Segment 2: Contains the data region (both initialized and uninitialized), users area, kernel and user stacks, and Heap (where new memory has been allocated via the malloc() system call).

The range of addressability in the 64-bit address space is one million terabytes (TB), or one billion gigabytes (GB), or 2^{60} bytes. More than a third of this range has been reserved for application program text and data. Shared memory regions, shared libraries, dynamically loaded programs, and the program stack are each allocated a subset of the address space up to 256 bytes, or approximately 64,000 TB.

64-bit Application Porting

- Porting challenges
 - Data type consistency and the different data model
 - Interoperation between applications using different data models
- 64-bit advantages
 - More virtual and physical address space
 - Larger data types and files support
- 64-bit disadvantages
 - More stack space to hold the larger registers
 - 64-bit applications do not run on 32-bit platforms

Figure 2-26. 64-bit Application Porting

AU253.0

Notes:

The first step in developing an application is to choose a programming model. The term programming model refers to the instruction set and data storage types (among other things) that are provided by the compilation tools and execution environment on a particular operating system. AIX 5L Version 5.1 provides the 32-bit and the 64-bit programming model. For example:

```
#include <stdio.h>
main()
{
    printf("Size of long datatype is %d\n", sizeof(long));
}
```

This program, when executed in a 32-bit mode, will produce the result:

Size of long datatype is 4

While in 64-bit mode, this program will produce the result:

Size of long datatype is 8

Note that the size of the **long long** data type is also 8 bytes in 64-bit execution mode.

If the application does not necessarily need any of the features present in 64-bit operating environments, there is little reason to force a transition, especially if you want the application to be ported as quickly as possible. The application can remain as a 32-bit application and still run on a 64-bit operating system without requiring any code changes or recompilation. In fact, 32-bit applications that do not require 64-bit capabilities should probably remain 32-bit to maximize portability.

When compiling in 64 bit mode, the compiler automatically defines the **__64BIT__** macro. This is useful for writing generic applications that can work both in the 32 bit as well as 64 bit mode.

Certain classes of applications exceed the 4 GB address space limitations of 32-bit systems. The primary objective for the development of 64-bit computing has been to make these and other large applications run efficiently. Therefore, applications that are clearly limited by the 32-bit address space should make the transition to 64-bit mode. These applications will perform better in a 64-bit environment.

64-bit performance considerations:

- 64-bit programs are larger.
- 64-bit long division is more time-consuming than 32-bit integer division.
- 64-bit programs that use 32-bit signed integers as array indexes require additional instructions to perform sign extension each time an array is referenced.

64-bit performance improvement:

- Avoid performing mixed 32-bit and 64-bit operations, such as adding a 32-bit data type to a 64-bit type. This operation requires the 32-bit type to be sign-extended to clear the upper 32 bits of the register.
- Avoid 64-bit long division whenever possible.

Unit Summary

- The AIX Compilation Process
- Other AIX Compilers
- The **cc/xlc/c89** Commands and Options
- Configuring the Compiler Commands (/etc/vac.cfg)
- Compiler Listings and Diagnostics
- Using the Preprocessor
- Optimization Techniques
- Program and Process Images
- 64-bit Programming Model

Figure 2-27. Unit Summary

AU253.0

Notes:

Unit 3. The dbx and idebug Debuggers

What This Unit is About

AIX supports several debuggers. The widely used ones are the popular UNIX debugger dbx and the GUI style debugger idebug. This unit introduces these two debuggers, as well as providing instructions on how to use them to debug simple C programs. A number of other potentially useful debugging tools are brought to the students' attention.

The lab exercise provides a buggy program to debug using the student's choice of debuggers.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Compile a program in such a way as to allow easy debugging
- Create a core dump with programming statements or hardware signals
- Use dbx and idebug to
 - Find the line that caused a program crash
 - Navigate through source code
 - Display and change variable values
 - Manage breakpoints
- Know about some of the other AIX facilities that can be used to debug programs

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 2

References

- *AIX 5L Differences Guide, Version 5.1 Edition (SG24-5765-01)*
- *AIX 5L Implementation Differences (AU37)*

Unit Objectives

After completing this unit, you should be able to:

- Compile a program in such a way as to allow easy debugging
- Create a core dump with programming statements or hardware signals
- Use dbx and idebug to:
 - Find the line that caused a program crash
 - Navigate through source code
 - Display and change variable values
 - Manage breakpoints
- Know about some of the other AIX facilities that can be used to debug programs

Figure 3-1. Unit Objectives

AU253.0

Notes:

3.1 The dbx Debugger

Introduction to dbx

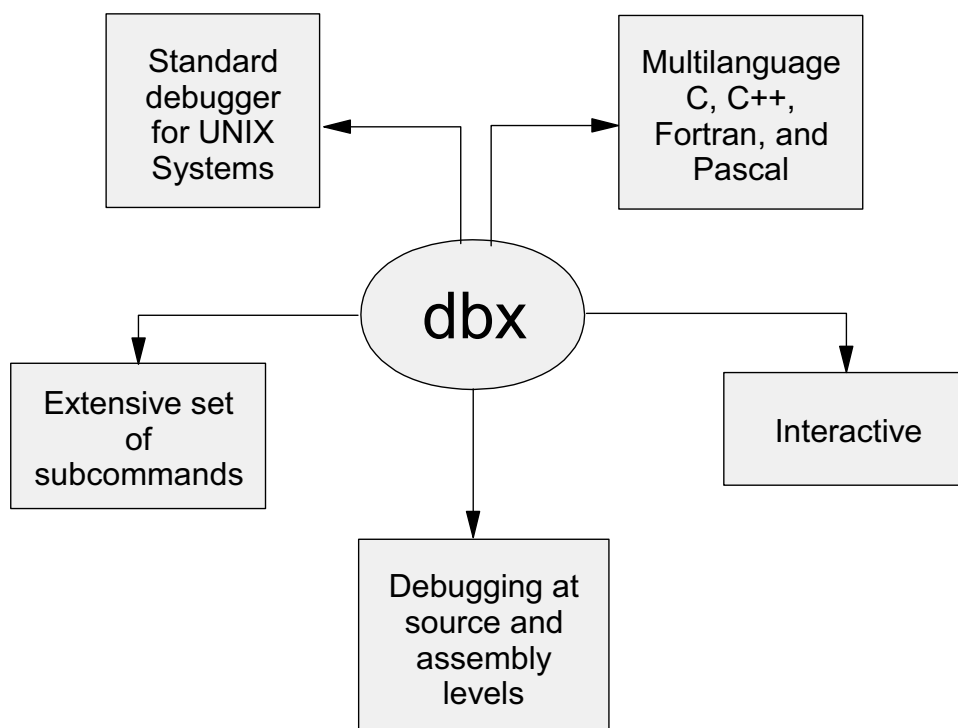


Figure 3-2. Introduction to dbx

AU253.0

Notes:

dbx is the current standard symbolic debugger for UNIX systems. As a symbolic debugger, it works with, and allows the user to work with, the source code of the program being debugged.

dbx is interactive, with an extensive, yet easy-to-use set of subcommands. We will discuss these subcommands during this lecture.

dbx supports debugging of C, C++, FORTRAN, and Pascal programs.

dbx allows you to debug a program at two levels: the source level and the assembler language level.

The fileset that installs **dbx** is bos.adt.debug, and comes along with the filesets for base operating system.

Using the dbx Debugger

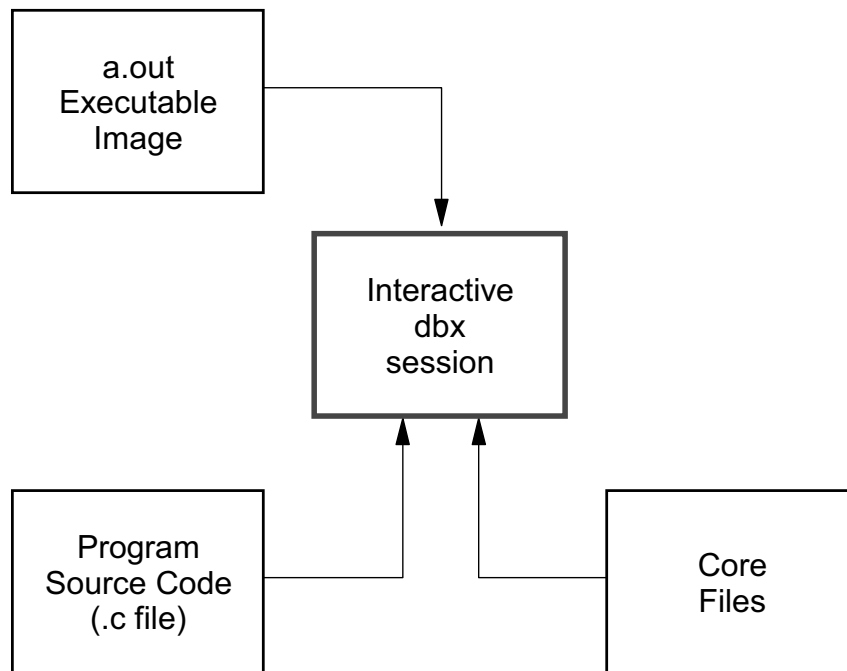


Figure 3-3. Using the dbx Debugger

AU253.0

Notes:

The dbx utility uses two, and sometimes three, files to help the user debug a program:

- The executable program (in a.out format), compiled with the -g option.
- The program source code (.c file). This allows the user to examine the source code lines for high-level debugging.
- A core image file (if one exists). A core file is often created when a program ends due to an error condition, such as memory address violations or illegal instructions. The core file contains the image of the program at the time of the crash. Core files are usually placed in the program's current directory. On many systems, if a core file exists, dbx can take you immediately to the line of source code that caused the crash.

Compiling for the Debugger

- During the debugging phase of program development, use the -g option when compiling with cc
 - Adds more information about variables and statements to the final executable (symbol table)
 - Especially useful for dbx
- Do not use the -O flag during compilation for debugging

```
$ cc -g myprog.c -o myprog
```

Figure 3-4. Compiling for the Debugger

AU253.0

Notes:

In order for dbx to work properly, the executable must be compiled with the -g option to the **cc** command. This option adds special information to the executable file. It obviously makes the final executable file larger, so -g is usually only used during the debugging phase of program development.

Debugging programs compiled with the -O option (optimize) is difficult with dbx because the optimization techniques rearrange portions of code for better performance. Use the -O option only when all debugging is completed.

If the -g option was not used in the compile, dbx will display the message "warning: no source compiled with -g".

Creating a Core Dump

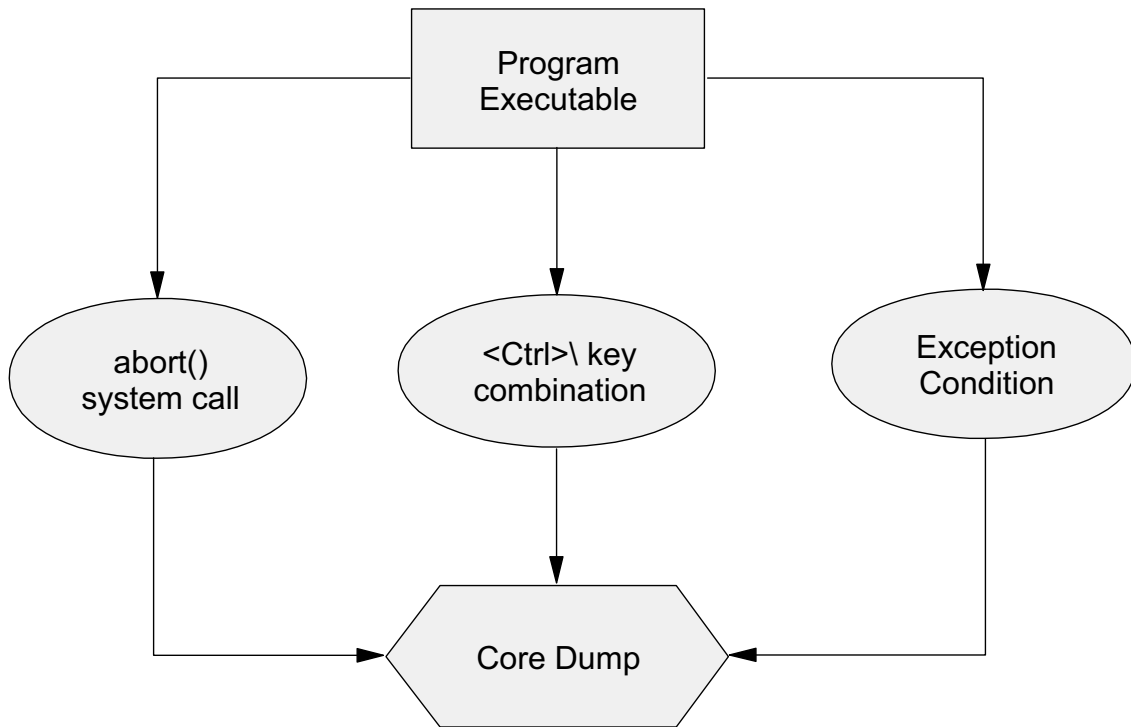


Figure 3-5. Creating a Core Dump

AU253.0

Notes:

A core file is handy for debugging programs with dbx. There are three ways a core file can be created:

1. If the program ends prematurely due to an error or exception condition.
2. If the user includes the `abort()` system call in his or her program. Include the `<stdlib.h>` header.
3. If the user presses the `<Ctrl>\` key combination while the program is running in the foreground.

The latter two options are available to intentionally cause a core dump for a program that is not working properly, but does not abend. To get more complete information about where problems occurred, refer to the appendix on core dumps. Also included in the appendix is a checklist on how to debug stripped/optimized code.

Core File Naming Conventions

- Core files can be created with an unique name.
- Set shell variable `CORE_NAMING=yes`.
- Export `CORE_NAMING` variable.
- Core file created in the name format `core.pid.ddhhmmss`.

Figure 3-6. Core File Naming Conventions

AU253.0

Notes:

The core file created has the name `core` by default and is located in the current directory. The next time a core file is created, the existing one is overwritten. If users prefer to save all core files created, then they have the overhead of either renaming the core file every time, or moving the core file to another path.

AIX 5L Version 5.1 provides an option to have unique names for all core files created.

This facility can be enabled by setting a shell environment variable `CORE_NAMING` to a value `yes` and then exporting it. Having done this, the core files are generated with the file names in the format `core.pid.ddhhmmss`.

- `pid` is the process ID of the process that caused the core dump.
- `dd` = Day of the month.
- `hh` = Hours.
- `mm` = Minutes.
- `ss` = Seconds.

Note that this facility of unique core file naming convention is a AIX 5L Version feature.

Contents of Core File

- Core header
- ldrinfo structures
- mstsave structures
- Default user stack
- Default data area
- Memory mapped regions
- vm_info structures

Figure 3-7. Contents of Core File

AU253.0

Notes:

- The core file contains all the information at the time the core dump occurred.
- The core header consists of the information about the core dump.
- The ldrinfo structures have the required loader information.
- The mstsave structures provide data about the kernel thread state.
- The default user stack is a copy of the user stack.
- The default data area consists data present in the data section at the time of core dump.
- The vm_info structures store the offset and size of information for memory mapped regions.

Invoking dbx

- General syntax:

dbx *[options] [object_file] [core_file]*

Example: dbx myprog

- Conditional debugging:

dbx -r *[options] [object_file] [command_arguments]*

Example: dbx -r myprog

- Attach to a running process:

dbx -a *pid*

Figure 3-8. Invoking dbx

AU253.0

Notes:

On most UNIX systems, there are two ways to invoke the dbx utility:

dbx *[object_file][core_file]* is used to invoke dbx on an existing executable (*object_file*). If the user fails to specify a *core_file*, dbx looks for a file named *core* in the current directory. dbx does not look for core files with the unique file name format. They have to be specified by the user. If an object file is not specified, the user is prompted to enter the name. The default object file is *a.out*.

dbx -r *[object_file] [command_arguments]* is used to run the executable program, automatically invoking dbx if the executable program abends. If the executable program completes without a terminating error, dbx is never invoked.

If a running program (process) is not acting properly (for example, stuck in an endless loop), one can start debugging the program by using the **-a** (attach) option of **dbx**. This is only practical if the user is running in a Windows environment or the debugger can be invoked from another terminal. The process ID number (PID) of the offending process is required as an argument. The process ID may easily be obtained by using the **ps -u userid** command.

Once started, the (dbx) prompt appears. You can exit the dbx program by issuing the **quit** subcommand at the (dbx) prompt.

3.2 dbx Subcommands

dbx Subcommands

- List of all subcommands and topics available:

Help

- Divided by functionality
 - Commonly-used subcommands
 - Topics
 - Startup
 - Execution
 - Breakpoints
 - Files
 - Data
 - Machine debugging
 - Environment
 - Threads
 - Expressions
 - Scope
 - Set-variables
 - Usage

Figure 3-9. dbx Subcommands

AU253.0

Notes:

One of the most useful subcommands in dbx is the **help** subcommand.

The **help** subcommand displays a list of all dbx-supported subcommands and their functions, as well as dbx topics.

Since many versions of dbx exist, it is recommended that you use the **help** subcommand to determine if all of the features described in this lecture are available on your version of dbx.

Debugging Options

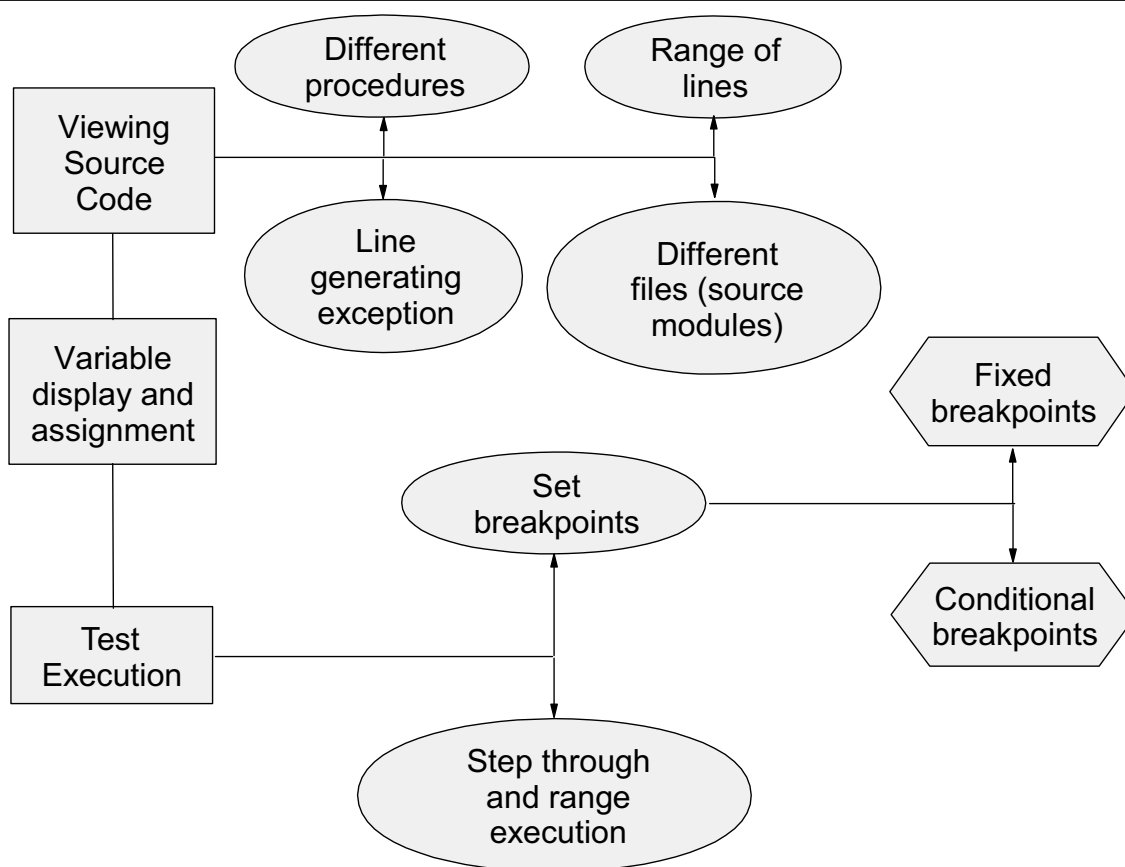


Figure 3-10. Debugging Options

AU253.0

Notes:

There are three basic options available to the programmer through dbx:

- Looking at the code.
- Test running, which includes the ability to instruct the debugger to stop at a certain point in the code, or when a certain condition exists. It also includes the ability to step through the program, executing a single line at a time.
- Examining and changing variables. Many times, the cause of programming bugs is incorrect variable values or assumed values. dbx allows you to not only look at the values of variables, but actually change them on the fly.

The best way to get accustomed to the dbx environment and its commands is by working through examples. The following example will be used as a reference to explain various subcommands in this section.

The example program performs basic arithmetic operations on integers. The functions defining the operations are put together in a file named functions.c. The main program, sample.c, calls these functions and prints the result of the operation to the standard output.

For example:

Source file sample.c:

```
# include <stdio.h>

main()
{
    int x = 20;
    int y = 5;
    int z;
    printf("\n\n This is a sample program that does basic arithmetic
operations on two integer variables\n\n");
    printf("Value of x = %d\n",x);
    printf("Value of y = %d\n",y);
    z = add_int(x,y);
    printf("\nAddition      : x + y = %d\n",z);
    z = sub_int(x,y);
    printf("\nSubtraction   : x - y = %d\n",z);
    z = mul_int(x,y);
    printf("\nMultiplication    : x * y = %d\n",z);
    z = div_int(x,y);
    printf("\nDivision        : x / y = %d\n",z);
}
```

Source file functions.c:

```
# include <stdio.h>

int add_int(int,int);
int sub_int(int,int);
int mul_int(int,int);
int div_int(int,int);

int add_int(int num1, int num2)
{
    int i;
    i = num1 + num2;
    return(i);
}

int sub_int(int num1, int num2)
{
    int i;
```

```
        i = num1 - num2;
        return(i);
}

int mul_int(int num1, int num2)
{
    int i;
    i = num1 * num2;
    return(i);
}

int div_int(int num1, int num2)
{
    int i;
    i = num1 / num2;
    return(i);
}
```

To compile the program, run:

```
$ cc -g -c functions.c
```

This creates the object file, functions.o. Then run:

```
$ cc -g sample.c functions.o -o sample
```

This creates the debug version of the program, in this example, sample.

Displaying the File

```
(dbx) list
1    # include <stdio.h>
2
3    main()
4    {
5        int x = 20;
6        int y = 5;
7        int z;
8        printf("\n\n This is a sample program that
does basic arithmetic operations on two integer variables
\n\n");
9        printf("Value of x = %d\n",x);
10       printf("Value of y = %d\n",y);

(dbx) list 15,17
15       z = mul_int(x,y);
16       printf("\nMultiplication      : x * y = %d\n
",z);
17       z = div_int(x,y);
```

Syntax:

```
list [starting source_line_number
      [, ending source_line_number]]
list procedure
```

Figure 3-11. Displaying the File

AU253.0

Notes:

The dbx utility maintains a line pointer. When invoked, dbx sets the line pointer to the first line of main(). The user can relocate the line pointer by entering another number.

The dbx program will display ten lines of source code when the **list** (or **l**, for short) subcommand is used. Also, a range of lines can be specified by supplying a starting line number and ending line number, separated by a comma, as arguments to the **list** subcommand.

Each time a **list** is performed, the dbx line pointer is relocated to the line following the last line listed. Therefore, to "page" through a program, the user need only continue to enter the **list** (or **l**) subcommand.

The code of procedures can also be displayed with the **list** subcommand. This subcommand displays a few lines before the beginning of the specified procedure and until the window is filled.

Moving Between Files and Procedures

```
(dbx) file
sample.c
(dbx) file functions.c
(dbx) list
  1  # include<stdio.h>
  2
  3  int add_int(int,int);
  4  int sub_int(int,int);
(dbx) func
main
(dbx) func add_int
(dbx) list
 11          i = num1 + num2;
 12          return(i);
 13      }
 14
```

Syntax:

file [*file_name*]

func [*function_name*]

Figure 3-12. Moving Between Files and Procedures

AU253.0

Notes:

Remember that an executable may be a compiled code linked from many different source code modules. As long as each module was compiled with the -g option, the dbx user can display and manipulate the code in these modules.

To change the current file (module), use the **file** subcommand. Line numbers always start with one for whatever the current file is. If the *file_name* argument is not specified, the name of the current file is displayed.

To change to a particular function, use the **func** subcommand. If the function is in a different source file, the file name also changes automatically. If the *function_name* argument is not specified, the name of the current function is displayed.

The dbx Stack Trace and String Search

```
(dbx) _  
(dbx) where  
add_int(num1 = 20, num2 = 5), line 11 in "functions.c"  
main(), line 11 in "sample.c"  
(dbx) _  
  
(dbx) file sample.c  
(dbx) /main  
3      main()
```

Syntax:

where [*> file_name*]

String search options:

/string

Search forward to next occurrence of *string*

?string

Search backward for next occurrence of *string*

Figure 3-13. The dbx Stack Trace and String Search

AU253.0

Notes:

Use the **where** subcommand to display the stack trace at the current line of execution. The stack trace gives a list of all functions that have been called, starting from main up to the current function. The function name at the top is the current function.

In addition, the trace also shows all the arguments and their corresponding values that are passed to the function. Also, the line number in the source file where the call to the function is made is displayed.

Output may be redirected to a file (*>filename*).

A forward search for a string within the source code is initiated with the **/string** subcommand (just like in vi).

A backward search for a string within the source code is initiated with the **?string** subcommand (also like in vi).

Displaying Variables (1 of 3)

```
(dbx) print 5+12
17
(dbx) print num1
20
(dbx) _
```

Syntax:

```
print expression
```

```
print variable
```

Figure 3-14. Displaying Variables (1 of 3)

AU253.0

Notes:

The current values of program variables can be displayed with the **print** subcommand. The **print** subcommand will also display the results of arithmetic expressions.

The **print** subcommand may also be used to print the return value from a procedure:

```
print procedure
print strlen("Hello")
```

The strlen() subroutine returns the length of the string argument passed to it, in this case 5.

Another technique that is useful is displaying memory. First, print out the address of a variable, then enter that address directly on the **dbx** subcommand area, followed by a slash and how many words you want to see at that address. An example follows:

```
(dbx) print &num1
0x2ff22b68
(dbx) _
```

```
(dbx) 0x2ff22b68/10
0x2ff22b68: 00000014 00000005 00000000 0000d0b2
0x2ff22b78: 00000000 21329000 00000000 145b1058
0x2ff22b88: 00000000 00000000
(dbx) _
```

In the above example, the address of `num1` is first found and then ten words starting from that address are displayed.

Note: The value at address `0x2ff22b68` is `00000014` (hexadecimal), the decimal equivalent of which is 20. It refers to the value of *num1*, which is at the same address.

Displaying Variables (2 of 3)

Normal display:

```
(dbx) _
(dbx) p d
(i = 75, c = 'V', iptr = 0x2ff22b80)
```

Display with `$pretty="on"`:

```
(dbx) set $pretty="on"
(dbx) p d
{
    i = 75
    c = 'V'
    iptr = 0x2ff22b80
}
```

Display with `$pretty="verbose"`:

```
(dbx) set $pretty="verbose"
(dbx) p d
i = 75
c = 'V'
iptr = 0x2ff22b80
```

Figure 3-15. Displaying Variables (2 of 3)

AU253.0

Notes:

There are three ways to display a complex variable, such as a structure:

- The usual default way, which displays all the variables of the structure in a single line. This could be very complex and confusing when the number of variables in the structure is large.
- By setting the `$pretty` variable to `on`. The display in this case is structured. All the variables are displayed within open and closed braces. If the structure has another structure defined within, the variables of that structure are also displayed within open and closed braces.
- By setting `$pretty` to `verbose`. This is very similar to the previous one, except that the display format varies, as seen in Figure 3-15. The open and closed braces are not displayed.

Note: The `$pretty` variable in dbx is an AIX 5L feature.

Displaying Variables (3 of 3)

```
(dbx) dump
add_int(num1 = 20, num2 = 5), line 11 in "functions.c"
i = 0
(dbx) _

(dbx) dump main
main(), line 11 in "sample.c"
x = 20
z = 0
y = 5
(dbx)
```

Syntax:

```
dump [procedure] [> filename]
```

Figure 3-16. Displaying Variables (3 of 3)

AU253.0

Notes:

A quick way to see the values of all variables in a procedure is to use the **dump** subcommand. If no procedure is specified, it will display all variables in the current procedure. Since the output may be extensive, and there is no way to pipe to the **more** command while in dbx, you may want to redirect the output of **dump** to a file via the > symbol.

Remember, this procedure looks at the local variables only by default. To view global variables, either use the **print** command or try:

- **dump . > myfile**
- **sh pg myfile** (this will show you all the variables, and the list is huge)

Change Variables

```
(dbx) _  
(dbx) assign y=10  
(dbx) assign x=y  
(dbx) assign y=1+1  
(dbx) print x  
10  
(dbx) print y  
2  
(dbx) _
```

Syntax:

assign *variable* = *expression*

Note:

Program must be stopped before *assign* can be used.

Figure 3-17. Change Variables

AU253.0

Notes:

The values of variables in your program can be changed during debugging by using the **assign** subcommand. This must be done when program execution has halted at a stop point. (We will discuss stop points, or breakpoints, shortly.)

The new value assigned can be a literal or the result of an expression.

For example:

```
assign x=5  
assign y="Hello"
```

Running a Program

```
(dbx) run
```

This is a sample program that does basic arithmetic operations on two integer variables.

```
Value of x = 20
```

```
Value of y = 5
```

```
Addition      : x + y = 25
```

```
Subtraction    : x - y = 15
```

```
Multiplication : x * y = 100
```

```
Division       : x / y = 4
```

```
execution completed
```

```
(dbx) _
```

Syntax:

```
run [args] [<filename>] [>filename]
```

rerun

Figure 3-18. Running a Program

AU253.0

Notes:

The **run** subcommand is used within dbx to run the program that you are debugging. Arguments that the program expects must be supplied as arguments to the **run** command in order for them to be passed to the program. Redirection of stdin and stdout can be specified.

The **rerun** subcommand is useful for restarting the program you are debugging and passing it the same list of arguments as was passed the last time you ran the program. This is particularly useful after reaching a breakpoint.

Using Breakpoints (1 of 2)

Breakpoints cause the executing program to stop at a desired line number, procedure, or under certain variable conditions.

Setting breakpoints:

- stop at** *source_line_number* [*if condition*]
 - stops when *source_line_number* is reached
- stop in** *procedure* [*if condition*]
 - stops when *procedure* is called
- stop if** *condition*
 - stops whenever condition is true
- stop** *variable* [*if condition*]
 - stops whenever *variable* changes value

Figure 3-19. Using Breakpoints (1 of 2)

AU253.0

Notes:

Breakpoints are one of the most useful tools in debugging a program. They allow you to specify a point in the code where execution will stop, giving you a chance to examine the contents of variables and machine registers.

The dbx utility gives you many ways to specify where to place a breakpoint and under what conditions it should take effect:

- Stopping at a specified line number
- Stopping at a specified procedure or function
- Stopping when a certain condition is true, such as when $x > 25$
- Stopping every time a specified variable's value changes.

A breakpoint must be set on a line of executable code, for example, not on a line that contains only a function name in a curly brace ({ or }).

A breakpoint must be used to initiate the step-through debugging technique.

Examples:

```
stop in main
stop at "functions.c":18 if num1=8
stop in sub_int if num2=22
stop if (x>y) and (x<200)
```


Using Breakpoints (2 of 2)

```
(dbx) status
[6] stop in main
[8] stop at "functions.c":18
[9] stop if x > 100
[10] stop y in main if x < y
(dbx) _
(dbx) delete 10
(dbx) status
[6] stop in main
[8] stop at "functions.c":18
[9] stop if x > 100
(dbx)
(dbx) file functions.c
(dbx) clear 18
(dbx) dump
[6] stop in main
[9] stop if x > 100
```

Syntax:

```
status [ > filename ]
```

```
delete all (or) delete event_number [event_number] [...]
```

```
clear line_number
```

Figure 3-20. Using Breakpoints (2 of 2)

AU253.0

Notes:

The setting of each breakpoint is called an event. (Remember that breakpoints can not always be associated with line numbers, as in the case of "stop if x < 25.")

To see all set breakpoint events, use the **status** subcommand. The output displays the event number within square braces and the event. Since the resulting output may be large, redirection to a file is possible via the > symbol.

Breakpoints can be selectively removed by using the **delete** subcommand. This subcommand expects an event number. Using all as an argument to the **delete** subcommand will remove all breakpoints. A related subcommand, **clear**, removes all breakpoints at a given source line. It must be given a line number as argument. If a breakpoint is set at the line number for the current file, it is cleared.

Continuing Execution

```
(dbx) stop in main
[2] stop in main
(dbx) run
[2] stopped in main at line 5 in file "sample.c"
    5    int x = 20;
(dbx) next
stopped in main at line 7 in file "sample.c"
    8    int y = 5;
(dbx) stop at 12
[5] stop at "sample.c":12
(dbx) cont
[5] stopped in main at line 12 in file "sample.c"
   14    z = add_int(x,y);
(dbx) step
stopped in add_int at line 11 in file "functions.c"
   11        i = num1 + num2;
```

Syntax:

cont

step

next

Figure 3-21. Continuing Execution

AU253.0

Notes:

One of the best debugging techniques is to step through the program line-by-line. The **step** and **next** subcommands provide this capability. The main difference between these two subcommands is that the **step** command will also step through line-by-line within a called procedure, whereas the **next** subcommand executes all lines of a called subroutine as a single instruction. A number may be specified with **step** and **next** to execute multiple lines.

The **stepi** and **nexti** subcommands are similar to the **step** and **next** subcommands. **stepi** runs one machine instruction, and **nexti** runs the application program up to the next machine instruction.

The **cont** (continue) subcommand is used after a breakpoint to move ahead, executing the code until the next breakpoint is reached or the program finishes.

Tracing

To have information displayed while program is executing, use these commands:

```
trace [in procedure] [if condition]
trace source_line_number [if condition]
trace variable [in procedure] [if condition]
```

Examples:

- To trace each call to the printf procedure, run:
`trace printf`
- To trace changes to the variable x within main, run:
`trace x in main`
- To display a message each time a procedure sub is called, run:
`trace mul_int`

Figure 3-22. Tracing

AU253.0

Notes:

Using **trace** is similar to stepping through a program except that detailed information is displayed for the line of code, which satisfies the trace condition. Unlike step through, the execution does not stop when any of the trace conditions are satisfied. Instead, messages are displayed when a trace condition is met and the execution continues.

Like breakpoints, **trace** can be specified to activate under certain conditions. These are also treated as events by the **status** and **delete** subcommands. After setting up the trace, issue the **run** subcommand. The **delete** subcommand removes traces.

Other dbx Commands

- To escape to a shell under dbx, run:
`sh [Command]`
- To edit the current source file, run:
`edit [Procedure | File]`
- To create dbx command aliases, run:
`alias name command_string`
- To release an alias, run:
`unalias name`
- To enable or disable multiprocess debugging, run:
`multiproc [on | off]`

Figure 3-23. Other dbx Commands

AU253.0

Notes:

Figure 3-23 describes some additional dbx subcommands. Consult your **dbx help** output for information on whether these subcommands are available on your system.

You may put aliases in the **.dbxinit** file. A list of subcommands can be included in the **.dbxinit** file. **dbx** runs all the subcommands specified in the **.dbxinit** file at the beginning of the debug session.

The **multiproc** command is used to debug multiple processes. This is required in situations when the process forks a child process that has to be debugged. You must be using X-Windows to follow a forked process with **multiproc** set to on.

Examples:

- To run the **ls** command while in dbx:
(dbx) sh ls
- To start an editor on the main .c file:
(dbx) edit main.c

Note: The EDITOR shell environment variable can be set to the path of the editor of your choice. The **edit dbx** subcommand then invokes that editor. If no editor is specified, dbx starts the vi editor by default.

- To substitute **rr** for rerun, run:

(dbx) alias rr rerun

- To alias two subcommands, print n and step:

(dbx) alias pas "print n; step"

- To remove the alias **rr**, run:

(dbx) unalias rr

- To enable multiprocess debugging for a program that forks a child process, run:

(dbx) multiproc on

3.3 The idebug Graphical Debugger

Introduction to idebug

- Graphical debugger
- Easy to use
- Has a lot of options
- Debugging at assembly level
- Multiple processes can be debugged

Figure 3-24. Introduction to idebug

AU253.0

Notes:

idebug is a graphical debugger. It has a number of features. The options available are similar to dbx. In addition, debugging is available at the assembly level, which makes it interesting to assembly programmers.

idebug also supports debugging of multithreaded applications.

The idebug Debugger

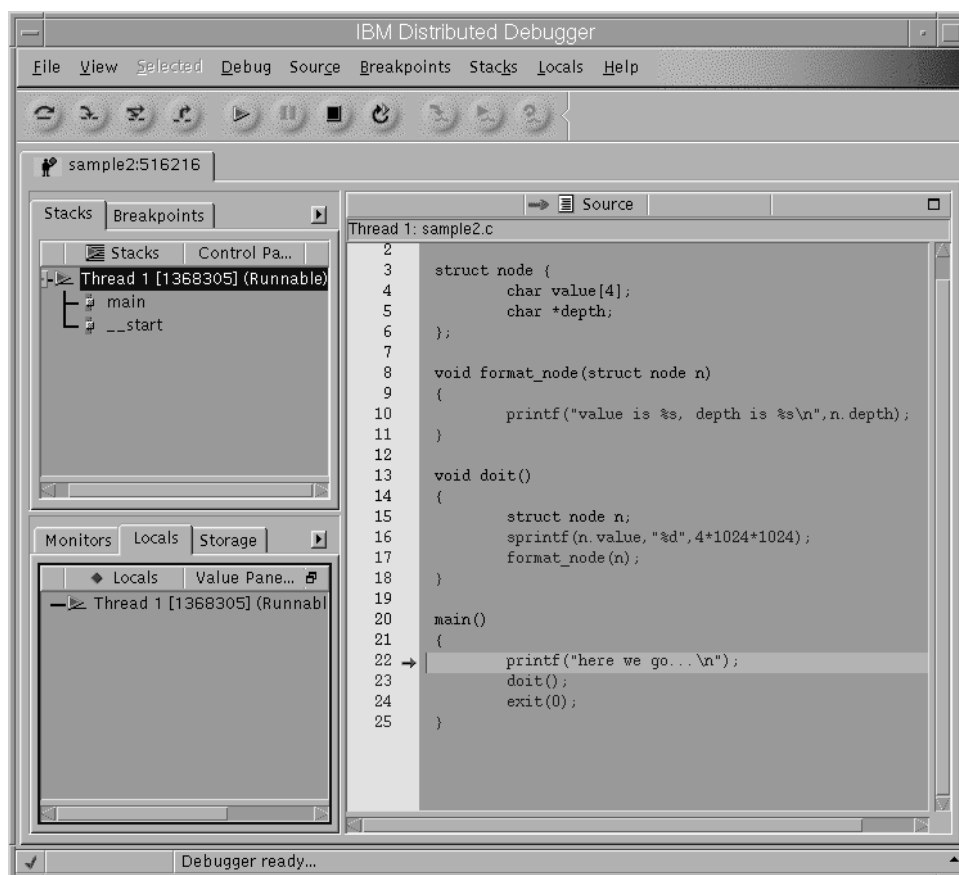


Figure 3-25. The idebug Debugger

AU253.0

Notes:

Figure 3-25 shows the main window of the idebug debugger. The window appears after compiling our program using the -g option and then invoking idebug on the load module:

```
$ cc -c -g sample2.c
$ cc -g sample2.o -o sample2
$ idebug ./sample2
```

Note that compiling a program for idebug uses the same -g option that is used to prepare a program for debugging with dbx.

The window in Figure 3-25 contains a number of sub-windows, many of which we will be looking at in more detail shortly. Let us have a look at each of the windows currently visible. There are three windows:

- The right half of the main window is the source code window. Right now, it is showing us the source for our main () function. The arrow pointing the highlighted line indicates that the next statement to be executed is the printf() at the start of main().

- The left lower window would normally show the local variables in the currently active function. It is called the Value panels. In this example, it is empty because main() has no local variables. It also shows the storage area.
- The left upper window displays the stack trace of the program at the time of current execution. It is called the Control panels. It has two other tabs for breakpoints and modules, which we will discuss as we go through the rest of the visuals.

Let us get started by clicking on pressing the F5 key. This is equivalent to a **cont** in dbx. Click on Debug on the menu bar. It has a lot of options for debugging.

For reference purposes, here is the program that we are debugging:

```
#include <stdio.h>

struct node {
    char value[4];
    char *depth;
};

void
format_node(struct node n)
{
    printf("value is %s, depth is %s\n",n.depth);
}

void
doit()
{
    struct node n;

    sprintf(n.value,"%d",4*1024*1024);
    format_node(n);
}

main()
{
    printf("here we go . . .\n");
    doit();
    exit(0);
}
```

Segmentation Fault in a Library Routine

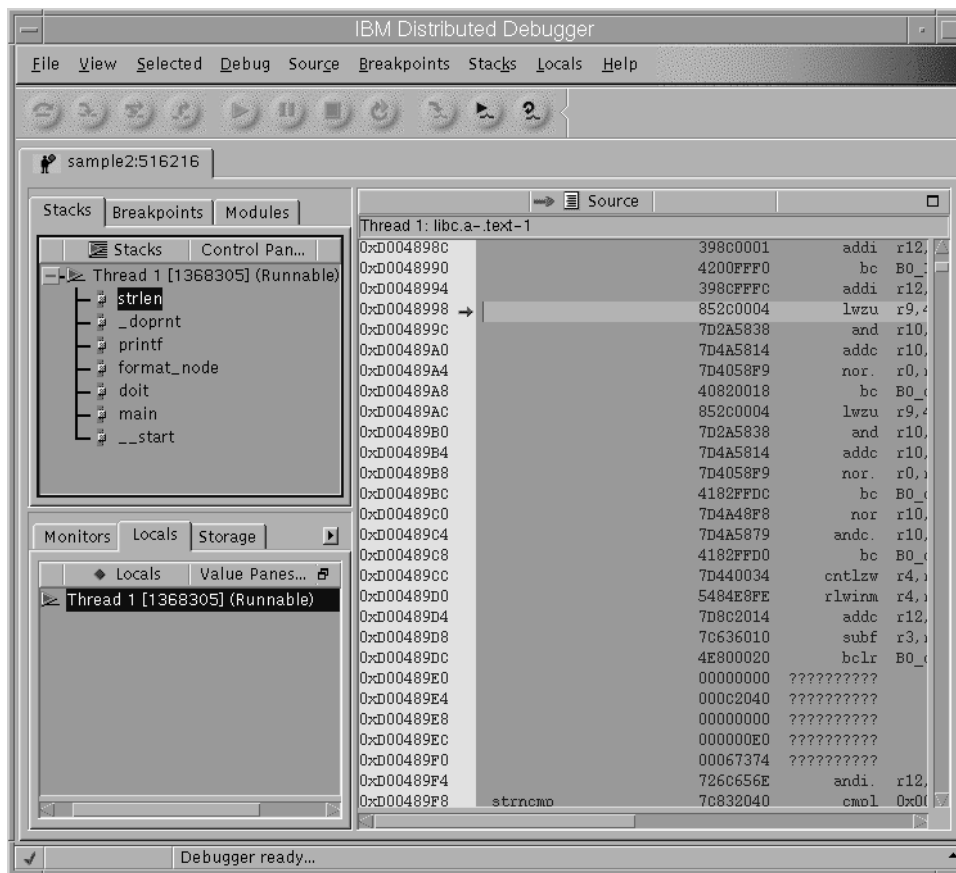


Figure 3-26. Segmentation Fault in a Library Routine

AU253.0

Notes:

When the F5 key is pressed, the debugger pops up a message saying an exception has occurred. This means that the program has suffered a segmentation fault. The stack trace shows that the program faulted while executing inside the `strlen()` routine. No source statement is visible in the source listing window because we do not have the source code for the AIX-supplied standard C library.

Looking at the stack trace, we see that `strlen()` was called from `printf()`, which was called from `format_node()`, and so on. There is not much point in looking at what is happening in `printf()`, because that is another standard C library routine that we do not have the source for. Let us click on the `format_node()` line in the stack trace.

Moving Through the Function Call Traceback

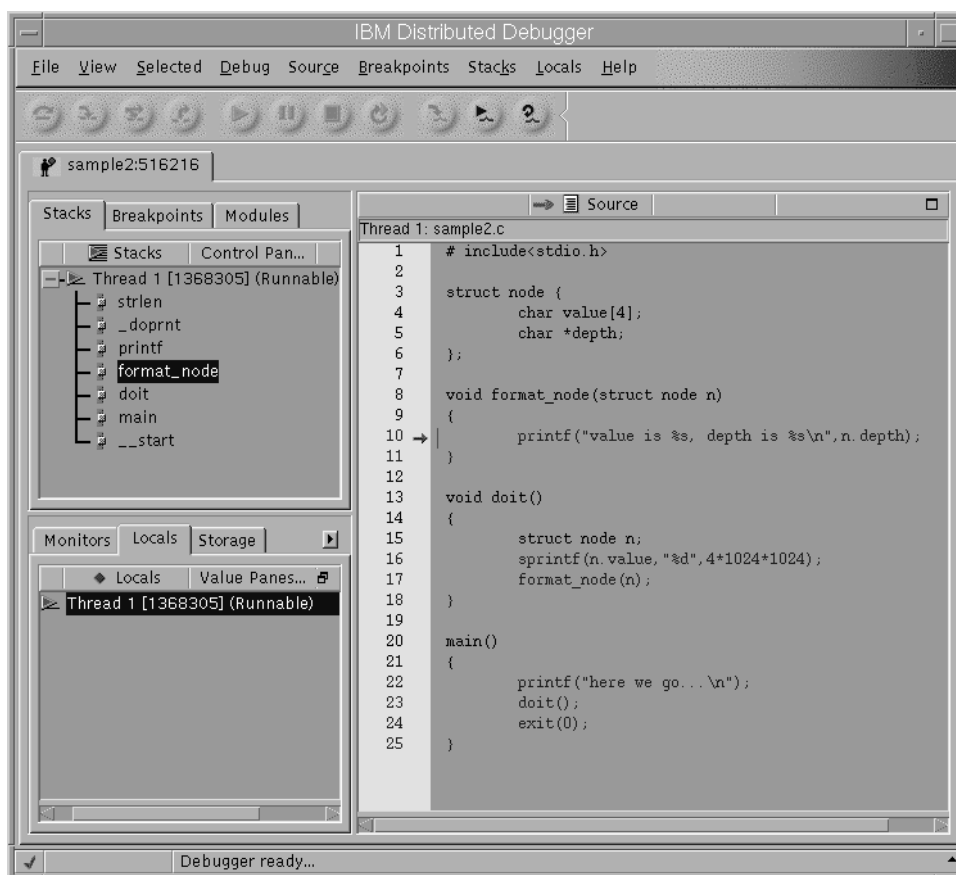


Figure 3-27. Moving Through the Function Call Traceback

AU253.0

Notes:

We have just clicked on the `format_node()` line in the stack trace. The visual shows us the C source code for `format_node()`. The arrow pointing the line of code in `format_node()` is the call to `printf()` that lead to the segmentation fault in `strlen()`.

Since we are not really sure what went wrong, let us take a different approach.

Setting the Breakpoint

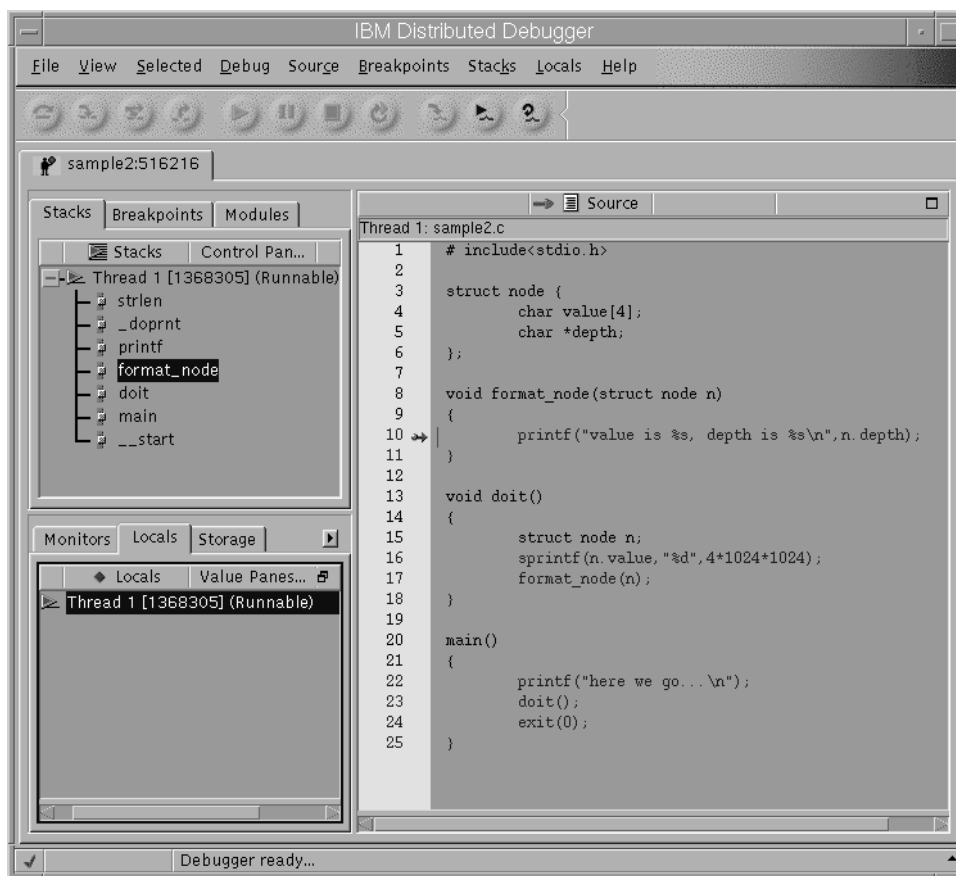


Figure 3-28. Setting the Breakpoint

AU253.0

Notes:

Let us set a breakpoint. Right click on the `printf` ("value is . . .") line in the source window. A menu pops that says "Set Breakpoint". A breakpoint is set at the line by clicking on the menu. A breakpoint can also be set by pressing F9 after clicking on the line. A red circle appears at the left corner of the line. This indicates that there is a breakpoint set on this line.

Listing the Currently Set Breakpoints

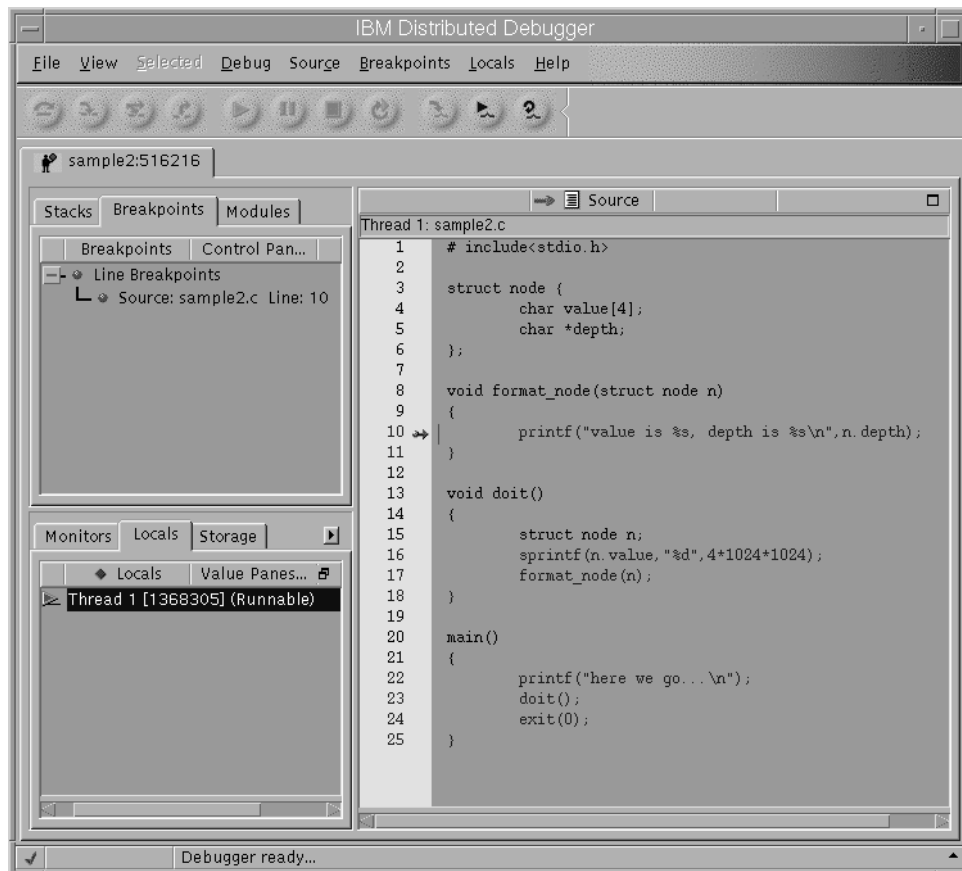


Figure 3-29. Listing the Currently Set Breakpoints

AU253.0

Notes:

A click on the Breakpoints tab on the control panel displays all the breakpoints that are currently set. Figure 3-29 shows us that we currently have a breakpoint set on line 10 of our sample2.c source file.

Viewing Modules in a Program

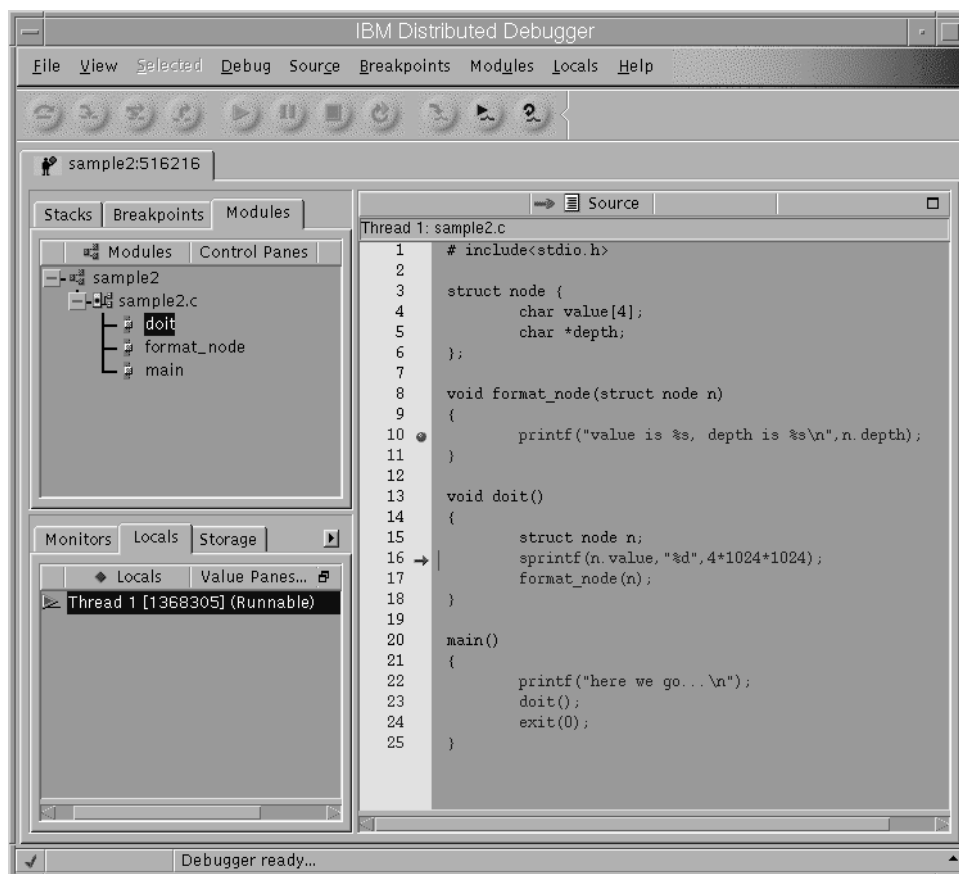


Figure 3-30. Viewing Modules in a Program

AU253.0

Notes:

We discussed the Control panel, that is on the left top corner of the window. The tabs corresponding to Stacks and Breakpoints are explained in the previous visuals. The third tab is the Module tab. This tab displays all the modules that are part of the program.

Because the program encountered a segmentation fault, it has terminated. The program must be reloaded to start execution again. You can reload the program by clicking on the File menu, or by quitting the current session of idebug, and invoking it again at the prompt.

Stopped at a Breakpoint

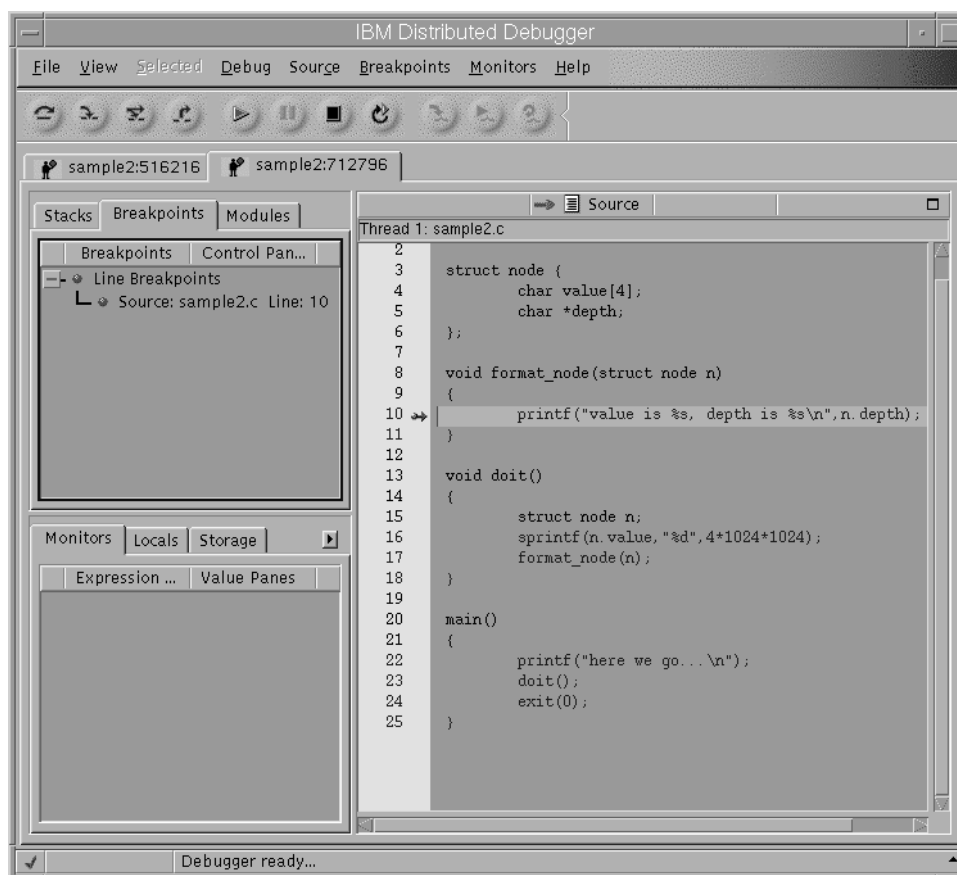


Figure 3-31. Stopped at a Breakpoint

AU253.0

Notes:

Having reloaded the program, continue the execution (press F5). The program stops at the breakpoint. The circle indicates that the current line has a breakpoint and the arrow on top of the circle indicates the line where we're currently stopped at, in this example, line 10 in sample2.c in the module format_node().

Let us now have a look at the n struct variable.

Looking at a Local Variable (1 of 3)

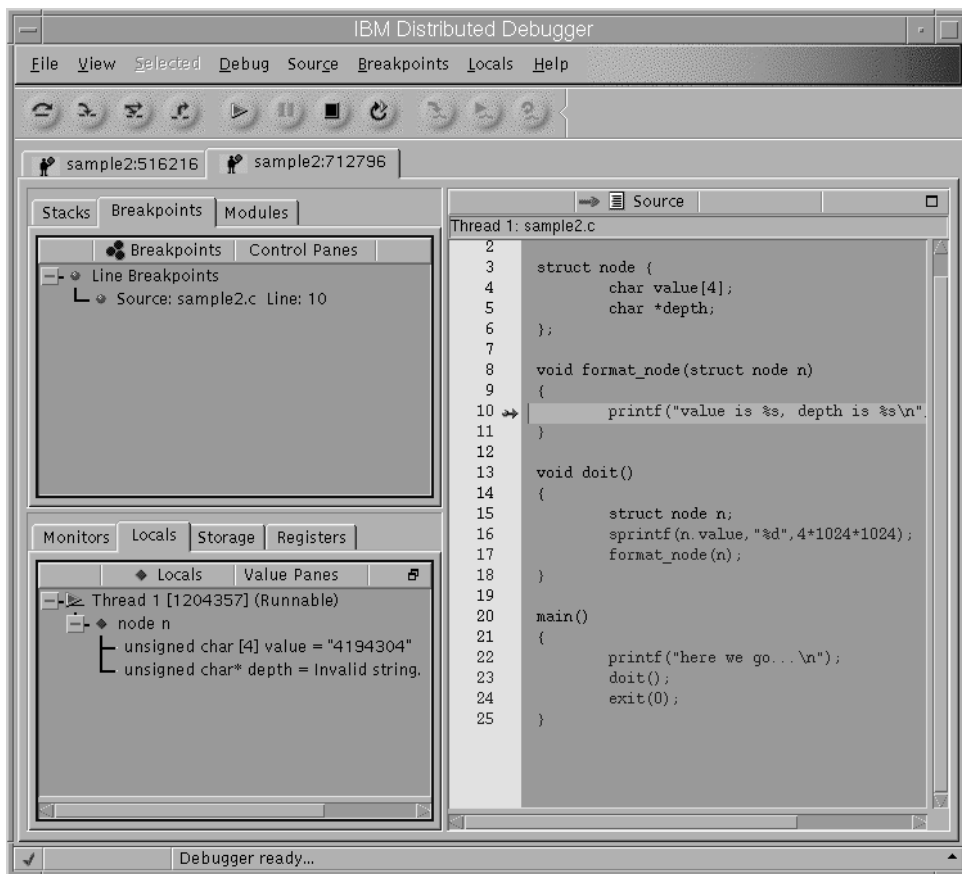


Figure 3-32. Looking at a Local Variable (1 of 3)

AU253.0

Notes:

Variables that are local to a module can be viewed by clicking on the Locals tab in the value panels. The module `format_node()` has only one local variable, `n`, which is displayed in the visual. The variable `n` is a structure. The display can be expanded further to view all elements within `n` - value field and depth field.

This is not a good situation. First, the four element char array value is supposed to contain a '\0' terminated character string. Moreover, the array contains seven characters, which goes beyond the array boundary. We decide to look at value as a character string.

The second point is that the debugger shows depth to be an invalid string.

Let us look at each one of them.

Looking at a Local Variable (2 of 3)

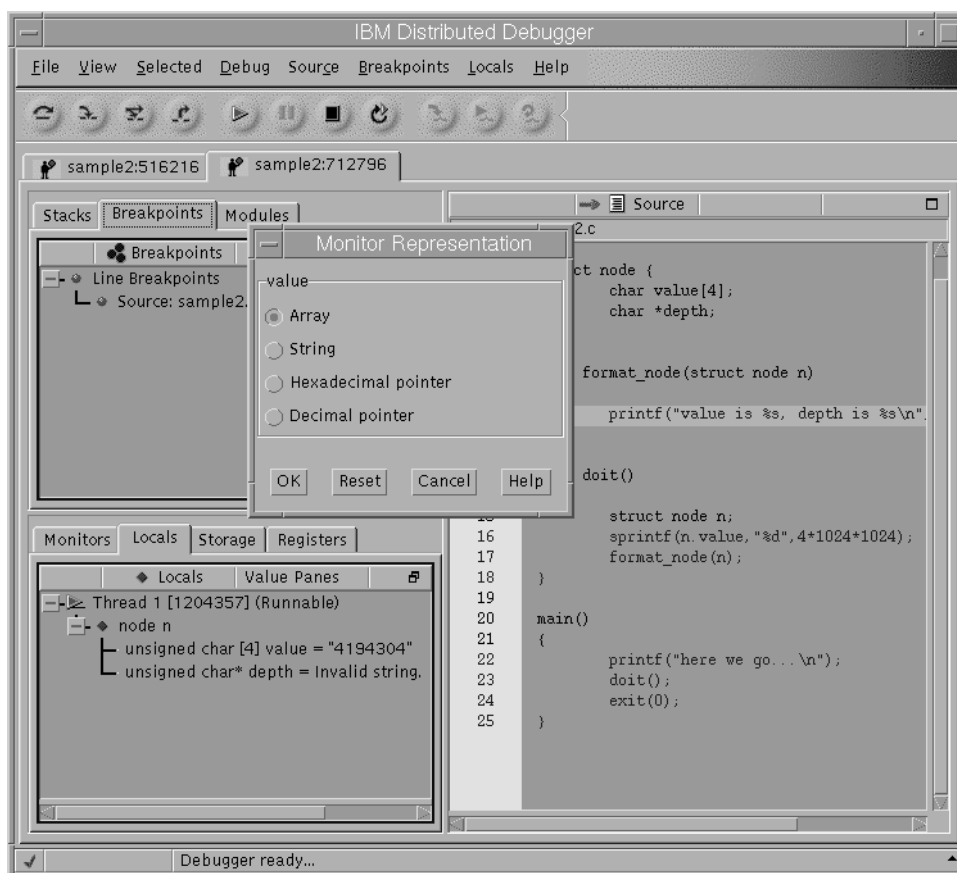


Figure 3-33. Looking at a Local Variable (2 of 3)

AU253.0

Notes:

The value field is displayed as a string. We could look at each element of the array. Right-click on value. A menu pops up with two items - Representation and Edit. Click on Representation. Another window pops up, from which we could select any of the representations.

Since value is an array, let us select the Array representation.

Looking at a Local Variable (3 of 3)

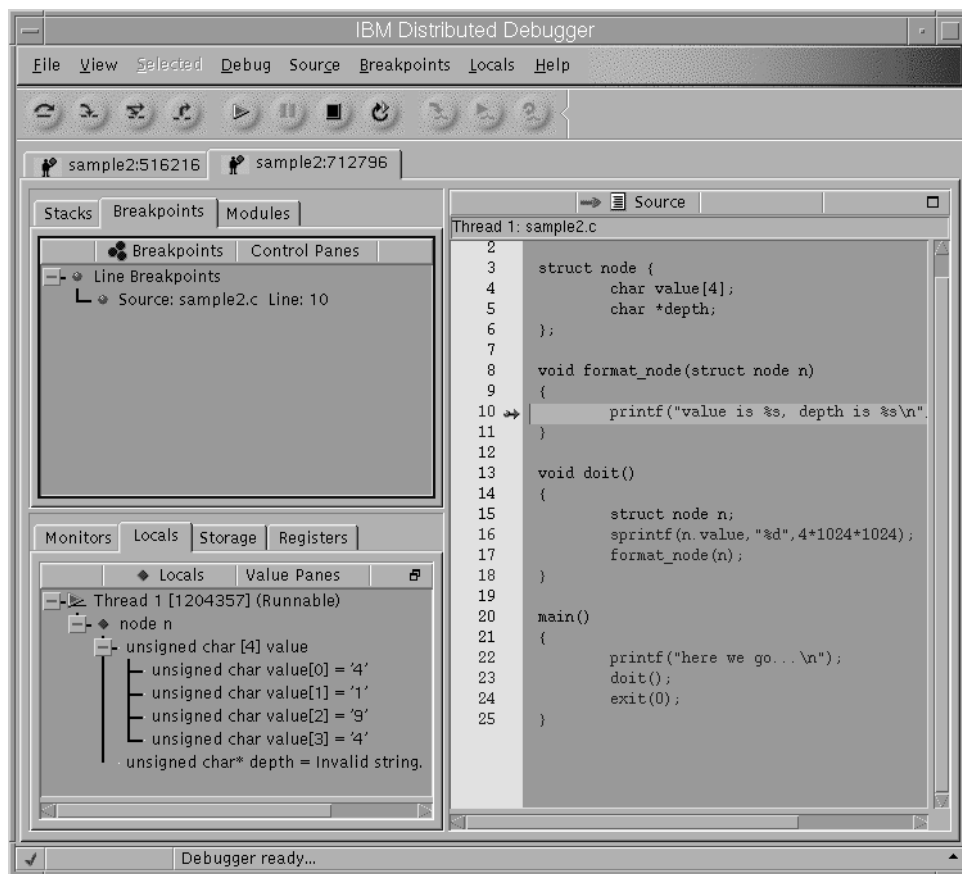


Figure 3-34. Looking at a Local Variable (3 of 3)

AU253.0

Notes:

The debugger displays only four elements of the array. This is because the array size is declared to be four, but value is consuming eight bytes (seven digits plus the terminating '\0').

Let us try to correct the situation.

Modifying a Local Variable (1 of 2)

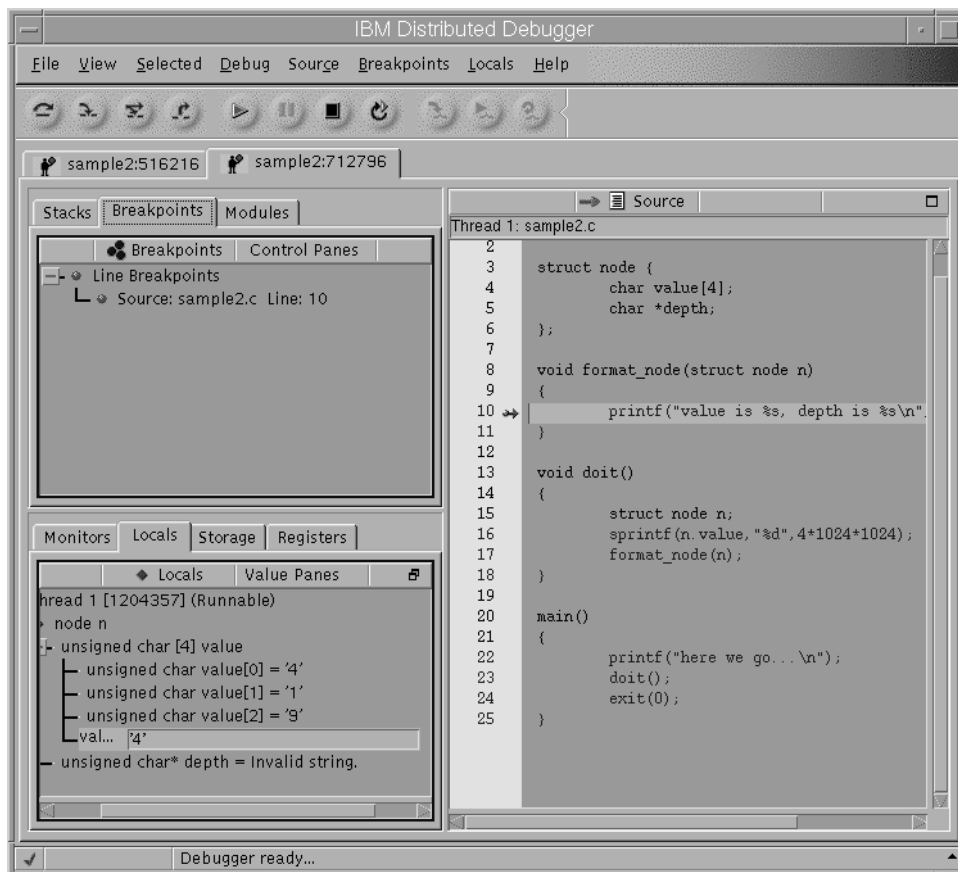


Figure 3-35. Modifying a Local Variable (1 of 2)

AU253.0

Notes:

Let us insert a '\0' at the end of the array. To do this task, we have to edit the existing field. Right-click on the last element of the array, '4', and then click Edit. A text box is displayed which allows editing of the value. Now, insert a '\0' in place of '4' and press <Enter>.

Modifying a Local Variable (2 of 2)

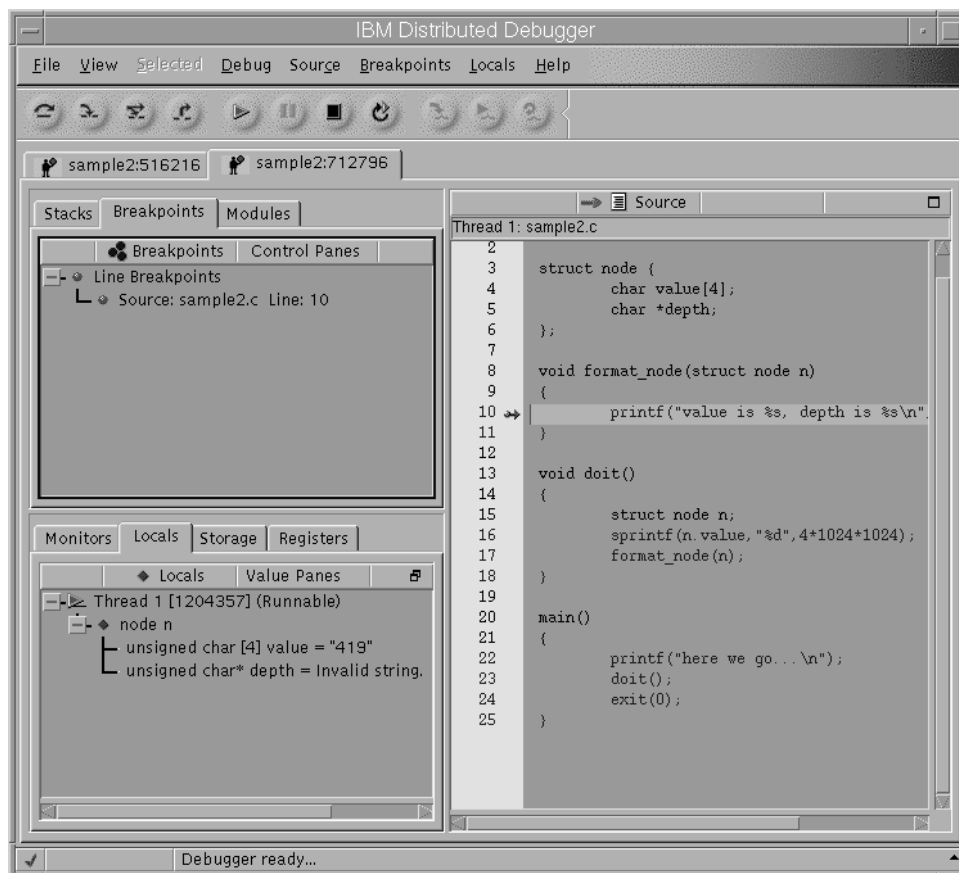


Figure 3-36. Modifying a Local Variable (2 of 2)

AU253.0

Notes:

The value field is now a '\0' terminated string (we could display it as a string if we wanted to be sure). Continue the execution of the program (press F5). The program still terminates with a segmentation fault.

Time to stop correcting symptoms and start looking for the cause (one could easily argue that we should have been doing that all along).

We click on `doit()` from the stack trace tab.

End of the Trail

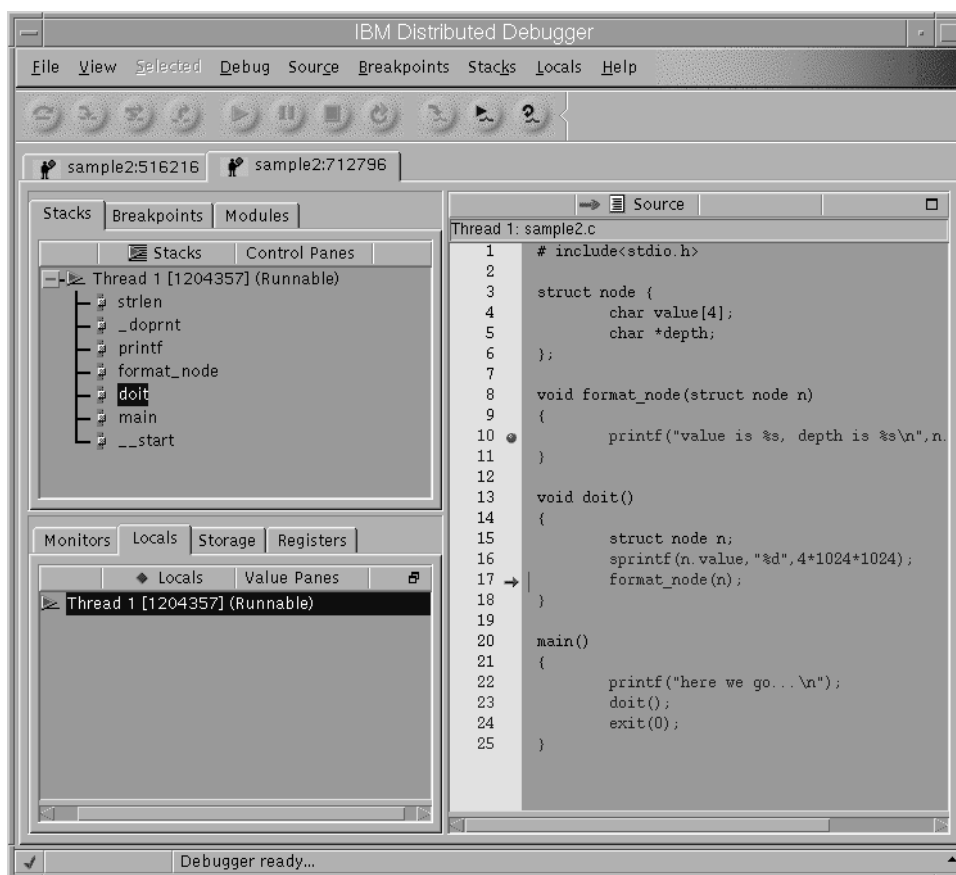


Figure 3-37. End of the Trail

AU253.0

Notes:

Looking at the code, we can see that not only is `sprintf()` being used to write too many bytes into the value field of `n`, but also the depth field is not even being assigned a value. At this point, we use our favorite text editor to fix the bugs so we can use idebug to find the next one.

That is it for our tour of idebug. There is actually quite a bit more to learn. Probably the best way to learn it is by working with idebug for a while.

3.4 Other Debugging Tools

Other Debugging Tools

Other useful debugging tools include:

- HeapView: Chases memory allocation errors
- gprof: Finds out where CPU time is being spent
- trace facility: Records and views kernel events
- audit facility: Records and views various events
- tprof: Looks for hot spots
- ZeroFault: Identifies memory leaks
- Great Circle: Performance testing tool
- Long walks: Useful for all sorts of problem solving

Figure 3-38. Other Debugging Tools

AU253.0

Notes:

Figure 3-38 shows a list of other tools that can be useful in chasing certain types of problems. Here is a bit more info on each of these tools (and where to find more):

HeapView

Probably the best source of information on HeapView is the IBM VisualAge C++ Professional for AIX library (on the CD-ROM).

gprof

gprof is a tool that can be used to identify where a program is spending CPU time. gprof is well covered by its man page. Just type **man gprof** for more information, or search for the same in the AIX online documentation. This can be used by installing bos.adt.prof fileset.

trace facility

The trace facility is intended to be used to record kernel events. As such, it is primarily focused on monitoring things, such as device driver events. It is sometimes useful as a

tool of last resort when chasing particularly strange problems that might be kernel related in some way.

The trace facility is described in the documentation library. Special system privileges are required to be able to use this facility. The `bos.sysmgt.trace` fileset contains this facility.

audit facility

The audit facility is part of the Trusted Computing Base (TCB). It is actually intended to watch out for potential security violations, although it can be quite useful in chasing certain classes of bugs (for example, identifying which programs are accessing a particular file). The audit facility is described in the documentation library and also requires special system privileges.

tprof

tprof is another tool that can be used to identify where CPU time is being spent in a program. tprof is described in the documentation library. It should be noted that **tprof** is packaged separately as part of the Performance Toolbox for AIX LPP.

ZeroFault

ZeroFault is a tool that can be used to identify and eliminate memory leaks in a program. It is very easy to use and has a graphical interface. The ZeroFault installable is part of the Bonus Pack for AIX Version 4.3. The AIX 5L Bonus Pack does not include ZeroFault. The installable can be downloaded from the Web site <http://www.zerofault.com>.

Great Circle

Great Circle is a testing tool that is used to identify performance and reliability problems in C and C++ application programs. It provides a Web-based interface and is easy to use. The evaluation version comes along with the AIX 5L Bonus Pack.

Long walks

Long walks is the most powerful debugging technique of all. If things are not working and you seem to be faced with a great deal of conflicting information, spend some time doing some strategic planning. Some things to consider (in no particular order):

- If you find yourself with multiple bugs, some of which seem to disappear as soon as you try to find them, chase all of the intermittent bugs at once. When one bug occurs, do what you can to gather more information on the bug (insert debug statements, look at potentially interesting variables, and so on). When that bug disappears and another bug appears, switch your focus to the new bug until another bug appears to replace it. You can make a surprising amount of progress by not fixating on any one particular bug. If some bugs are intermittent and others are pretty solid, you generally make the best overall progress if you focus on the intermittent ones as much as possible.
- Follow your intuition. If something feels wrong about a section of code, then it probably is wrong. Spend some time studying the code. Construct a few if statements that test the assumptions (for example, variable x must be between 1 and 75) and call `abort()` when the tests fail.

- Write transparent code; it will be easier to get it working correctly. It is actually quite rare for a section of code to be executed often enough that how efficiently it is written affects overall performance of the program. On the other hand, it can be very hard to write really efficient code that is also correct.
- Watch out for what the program is not doing. Programs do not always fail by doing something wrong; they often fail by not doing something right.
- If things start to get erratic, keep good notes on what you observe. These notes can prove invaluable once you start to see the pattern.
- Speaking of keeping notes, when you get involved in a major bug-hunt, you will find yourself making all sorts of assumptions. Write them all down as best you can and ask yourself from time to time which of the assumptions are starting to look weak.
- If your program has been compiled with an optimizer, compile it again without an optimizer. If it has not been compiled with optimization enabled, compile it again with the optimizer. The idea here is that an invalid program may change its behavior when optimized. Seeing what the program does with and without optimization might prove interesting (this particular technique does not yield results very often, but it can be useful when you are starting to get desperate).
- Write routines that validate key data structures and call the routines from all over your application. The routines should either abort or emit suitably obscure messages when they detect something wrong.
- Explain how your program works to somebody. It does not matter if the person has no idea what you are talking about (it does not hurt if they do, though!). The key is often to verbalize what you are doing and why.
- Design for failure. If the consequence of failure is severe, include code that either mitigates the failure if it happens (for example, do not retry a failed operation immediately, because looping in an infinite retry loop could consume a lot of resources) or include lots of self-checks so that you know when the failure has occurred and what was wrong.
- Try to write your software so that you try to preserve data values that were used in making key decisions. If the program then dies, you will be able to look at the variable to see why the program made the decisions it made. Obviously, this is a technique to be used sparingly. Focus on hanging onto values which were involved in key decisions.
- Implement a tracing facility of your own. One simple approach is to write a routine that takes a character string, makes a copy of it, and stores it into a circular buffer (old strings are freed as their slot is reused). When the program dies, you will be able to look at the table to see what happened lately.
- If you are worried about the cost of all the mallocs and frees, use an array of static buffers and just strcpy the strings in. It is important to copy the strings, because this tends to greatly simplify the calling of this routine (the caller can use sprintf to construct the string into a local buffer, which is then passed to your trace routine).

- Related to the last point, if feasible, crash the program when a self-check fails. A fault that results in the program failing is far more likely to get attention (that is, the problem will get reported and the bug will get fixed) than one which is a mere annoyance. Also if feasible, leave the self-checks with the `abort()` calls in place when you ship the code. Customers do not like programs that die, whereas it is amazing what sorts of soft-failures users will live with. As a result, programs which die get reported, the bug gets fixed, and the quality goes up.

Finally, do not become too dependent on fancy tools. A few carefully placed print statements and/or if statements that check your assumptions can often be the shortest path to success. This is particularly true of bugs that do not manifest themselves until a particular sequence of code has already executed correctly thousands or millions of times. Insert an if statement that detects the incorrect input and either call `abort()` or set a breakpoint in the body of the if. You will often quickly discover what is wrong or build confidence in what is right (be careful that your confidence is not misplaced!).

Unit Summary

- Compiling a program for debugging
- Creating a core dump
- Using dbx and idebug to:
 - Display source code
 - Identify point of program failure
 - Display and change variables
 - Manage breakpoints
- Other useful debugging tools and techniques

Figure 3-39. Unit Summary

AU253.0

Notes:

Unit 4. The make and SCCS Facility

What This Unit is About

This unit introduces the basic features of the UNIX **make** and SCCS (Source Code Control system) facility.

Since **make** is found on most UNIX systems, as well as DOS and OS/2 systems, there is a chance that you might already be familiar with the general concepts of **make**. This unit describes how **make** works in AIX. Targets, dependencies, and action lines are discussed. The construction of the makefile is explained, and **make** macros are introduced.

The second part of this unit discusses about SCCS.

At the conclusion of this unit and lab, you should be able to create makefiles of intermediate complexity to handle typical application scenarios. You should also be able to use SCCS to maintain your source files.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Create a makefile that properly lists the targets and dependencies for a typical C application.
- Use the **make** command to bring an executable file up to date.
- Create and execute pseudo-targets in a makefile.
- Code and use macros from within a makefile and from the command line.
- Use SCCS to maintain source code.

How You Will Check Your Progress

You can check your progress by completing:

- Lab exercise 3

References

- AIX 5L online documentation

Unit Objectives

After completing this unit, you should be able to:

- Create a makefile that properly lists the targets and dependencies for a typical C application.
- Use the **make** command to bring an executable file up to date.
- Create and execute pseudo targets in a makefile.
- Code and use macros from within a makefile and from the command line.
- Use SCCS to maintain source code.

Figure 4-1. Unit Objectives

AU253.0

Notes:

4.1 The make Facility

An Application Example

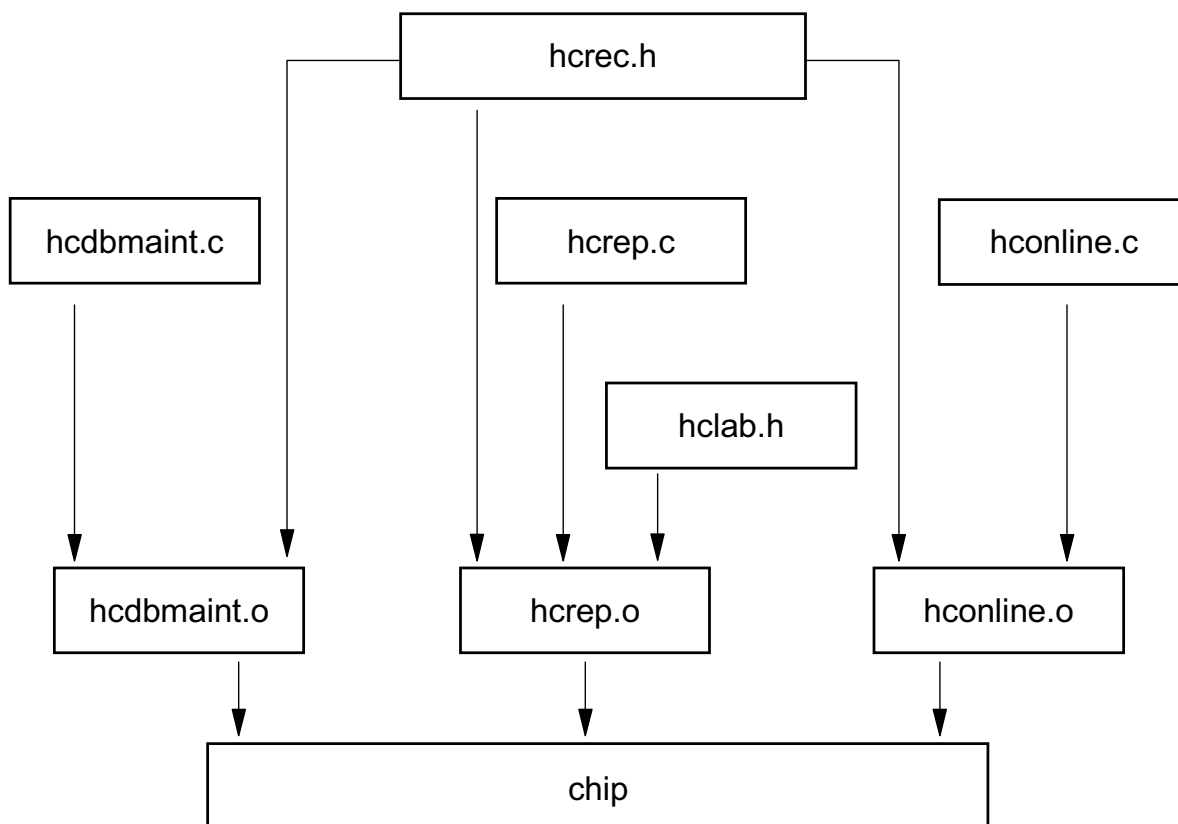


Figure 4-2. An Application Example

AU253.0

Notes:

The **make** utility and the makefile helps programmers manage software development projects. Generally, a software development project consists of many source files. Every time a source file changes, a set of commands need to be invoked to build the final software product. The **make** facility helps automate this process. A makefile is a representation of the steps to build the software product.

This facility is a part of the bos.adt.base filesset.

An application example:

A fictitious company uses a program called chip (chemical hazards information program). This program is actually a compilation of three source files: hcred.c, hcrep.c, and hconline.c. There are also two header files, hcred.h and hclab.h, used in the compilation process.

hcred.c is the code for maintenance routines for the hazardous chemicals database. hcrep.c is used for generating reports. hconline.c lets plant floor personnel access data on dealing with spills and accidental exposure.

The header files describe the structure of database records and the format for printing warning labels.

If any of these files becomes newer (modified more recently) than the chip program, then chip must be brought up to date. One way to do this would be to recompile everything. Suppose, though, that the only thing that was modified was hcrep.c. If .o files are maintained, there is no reason to recompile hcdbmaint.c and hconline.c. It would be more efficient to just compile hcrep.c and then relink everything. The **make** facility provides that capability.

Targets and Dependencies

- One can determine an order of dependency based on the diagram from the previous figure.
- Examples:
 - If a change is made to the hcrep.c source file, what other source code files need to be recompiled? What object files need to be relinked?
 - If the hclab.h file is changed, what source files need to be recompiled? What object files need to be relinked?
- The **make** utility uses a list of targets and dependencies to automate the process of bringing executables up to date.
- The **make** utility compares "last modification" dates of source, include, object, and executable files for an entire project.

Figure 4-3. Targets and Dependencies

AU253.0

Notes:

The **make** tool uses the concept of targets and dependencies. The dependencies for the chip example can be determined from Figure 4-2. If a dependency has a last modification time that is more recent than that of the target, action must be taken to make the target current.

In this example:

- chip depends on hcdbmaint.o, hcrep.o, and hconline.o.
- hcdbmaint.o (object) depends on hcdbmaint.c (source) and hcrec.h (header).
- hcrep.o depends on hcrep.c plus 2 headers, hcrec.h and hclab.h.
- hconline.o depends on hconline.c and hcrec.h.

The Makefile

- The **make** utility uses a file, usually named makefile or Makefile, to describe targets and dependencies.
- Makefile lines are either:
 - Dependency lines - Lists target and its dependencies
 - Action lines - Must begin with <Tab>; actual command used to compile or link dependencies.

```
chip: hcdbmaint.o hcrep.o hconline.o
    cc -o chip hcdbccmaint.o hcrep.o hconline.o

hcdbmaint.o: hcdbmaint.c hcrec.h
    cc -c -O hcdbmaint.c

hcrep.o: hcrep.c hcrec.h hclab.h
    cc -c -O hcrep.c

hconline.o: hconline.c hcrec.h
    cc -c -O hconline.c
```

- When **make** runs, the action line is executed for every dependency line where the last modification date of any dependency file is later than the target file.

Figure 4-4. The makefile

AU253.0

Notes:

A file called a makefile is used by **make** to establish the rules and relationships for a **make** procedure. Typically, this file is called either makefile or Makefile, but could be called anything. There are two primary types of lines in a makefile, namely dependency lines and action lines.

A dependency line establishes the relationship between files. The first line in the example is a dependency line and can be read "chip depends on hcdbmaint.o, hcrep.o, and hconline.o". The dependency line always starts with a label (usually a target file) followed by a colon, and is usually followed by a list of files upon which the target depends.

An action line defines the shell command to be executed should the target be out of date. It is often a compilation statement, but could be anything. This line **MUST BEGIN WITH A TAB**. Using spaces will not work. (Hint: Use ':set list' in vi, or the **cat** command with options -vte to verify invisible characters.) Normally, each action line will execute in its own shell.

Makefile Action Lines (1 of 2)

- Action lines:
 - Must immediately follow the dependency line they apply to
 - Must begin with a <Tab>
 - Can specify any command to execute

```
chip: hcdbmaint.o hcrep.o hconline.o
    @ echo "Creating the chip executable"
    @cc -o chip hcdbmaint.o hcrep.o hconline.o
    ...
```

- Normally, each executed action line is echoed to the screen as it runs.
- The @ symbol can be placed at the beginning of an action line to have the line run "silently."
- The **echo** command above still sends its args to stdout.

Figure 4-5. Makefile Action Lines (1 of 2)

AU253.0

Notes:

Action lines appear after the dependency line and **MUST** begin with a <Tab> (not spaces). The command can be any command that could be entered from the shell prompt.

Typically, each action line is echoed to the screen as it runs, but there are several ways to prevent this. One way is to precede each line with the @ character. Another is to use the -s (silent) option on the **make** command, which prevents the echo for each action line in the makefile. The third way is to use a fake dependency of .SILENT: in the makefile. Note, though, that standard output and standard error for each command will still be returned.

Makefile Action Lines (2 of 2)

```
chip: hdbmaint.o hcrep.o hconline.o
    @ echo "Creating the chip executable"
    # Saving the old chip program
    @-mv chip chip.old
    @cc -o chip hdbmaint.o hcrep.o hconline.o
    ...
```

- The # symbol indicates a comment:
 - When it is at the start of a line, it makes the entire line a comment.
 - When it is within a line, it makes all text to the right of it a comment.
- Normally, makefile action lines report an error if they fail and the **make** operation stops. The - symbol at the start of the action line ignores the error and does not print a message.
- Long action lines may use the \ symbol for line continuation.

Figure 4-6. Makefile Action Lines (2 of 2)

AU253.0

Notes:

Comments can be included in makefiles via the # symbol. All text to the right of the # is ignored by the **make** command. It is a good idea to comment complex relationships, the use of flags and macros, and anything else that might make the maintenance of a makefile difficult.

Normally, the **make** command will terminate if any executed command returns a non-zero return code. There are times when errors are acceptable.

For example:

If oldfile exists, move it to archive; otherwise, it does not matter. The command **mv oldfile archfile** would return a non-zero return code if oldfile does not exist and would abort **make**. To prevent **make** from terminating in this case, precede the **mv** command with a - (hyphen). Another alternative is to run the **make** command with the -i (ignore) option. Finally, the fake dependency line (.IGNORE:) within the makefile would also work.

As in the shell, the \ can be used as a line continuation character.

4.2 Using make

Using the make Command (1 of 2)

- The **make** command looks for the file makefile in the current directory.
- If makefile is not found, it looks for Makefile in the current directory.
- You may specify a different makefile by issuing the **make** command with the -f option:

```
$ make -f mymake
```

- You can tell **make** to run silently (same as preceding action lines with @) by using the -s option:

```
$ make -s -f mymake
```

- You can tell **make** to ignore errors in action lines (same as preceding action line with -) by using the -i option:

```
$ make -i -s -f mymake
```

Figure 4-7. Using the make Command (1 of 2)

AU253.0

Notes:

When the **make** command is issued with no options or arguments, it looks for a file called makefile in the current directory. If that file does not exist, it looks for Makefile in the current directory. Next, it checks for the SCCS versions of these two files (s.makefile and s.Makefile) in the current directory. The **get** command is used on an SCCS file to retrieve read-only copies, **make** uses the file, and then the copy is removed.

A different name can be used for the makefile. In order to have **make** recognize it, use the -f option. For example, **make -f /tmp/mycode/mymakefile**.


Other options to the **make** command include -s and -i. The -s is the silent option, causing **make** to not echo command lines to the screen. -i ignores non-zero return codes from individual command lines (does not terminate **make**).

The -k is similar to -i in that it continues processing after an error, but only on targets who do not depend on the target who caused the error.

Another useful option is the -n option. No actions actually execute. This just displays messages with respect to the steps necessary to bring the target up-to-date.

Using the make Command (2 of 2)

- If no dependency files have been modified since the last modification of the target file, the following message is reported:



'target file' is up to date

Figure 4-8. Using the make Command (2 of 2)

AU253.0

Notes:

Running **make** against a makefile in which the timestamp on each target is more recent than its corresponding dependencies results in a message on the screen indicating that everything is up to date and no commands had to be executed.

The **touch** command can be used to change the modification timestamp of files. This comes very handy while using the **make** facility. A common trick is to use the **touch** command on source files that need to be rebuilt. Refer to the AIX online documentation for more details about the **touch** command.

The **make** utility depends on the system clock heavily. It is important to ensure that the system clock is set correctly for the **make** utility to function correctly.

make with Command Line Targets

- A single makefile could be used to manage many projects.
- To keep control over which target files are checked, the desired target file name (first parameter of the dependency line) can be issued as an argument to the **make** command:

```
$ make -f mymake myprog
```

Figure 4-9. make with Command Line Targets

AU253.0

Notes:

It is possible to maintain multiple sets of targets and dependencies in a single makefile. In order to have **make** only work with certain parts of the makefile, provide target labels as arguments to the **make** command. That way, only the dependency line associated with the target, as well as any dependency lines associated with that target's dependencies, and so on, will be checked.

For example:

In the earlier chip example, if work was being done on hcrep.c and the programmer just wanted to compile to hcrep.o but was not ready to link to chip yet, the following would work:

```
$ make hcrep.o
```

Pseudo Targets

- Makefiles can contain pseudo targets:
 - Name on left side of colon on dependency line
 - Not associated with an actual file
 - Have no dependency list
 - Are always assumed to be "out-of-date" by **make**
 - Have associated actions

```
cleanup:
    @-rm hcdbmaint.o hcrep.o hconline.o
    @echo "Object files removed"

print_all: #print all source files
    @-pr *.c | lp
    @-pr *.h | lp
```

Figure 4-10. Pseudo Targets

AU253.0

Notes:

A pseudo target is a target with no dependencies and with a name that does not match an actual file. Because the file does not exist and the body of the target does not create the file, the dependency will be always out of date. Just like any other dependency line, it is followed by a series of actions to be taken. This can be useful for general cleanup, accounting, user notification (maybe mailing a status message), and printing.

Makefile Macros (1 of 2)

- Macros (variables) can be defined within a makefile:
 - Simplifies lists of file names or long expressions
 - Convenient for handling compilation options
 - Improves readability of makefile
- Definition syntax:

```
var_name=string_value
```

- Referencing the variable within the makefile:

```
$(var_name)
```

Figure 4-11. Makefile Macros (1 of 2)

AU253.0

Notes:

Using a macro in a makefile is similar to setting a variable in the shell. An example would be the inclusion of a CFLAGS macro in a makefile, as displayed. A user can override a macro definition from the command line. Therefore, if each compile command included \$(CFLAGS), then the user could choose options for a particular run of **make**, say:

```
$ make -f newmake "CFLAGS=-O"
```

A macro definition can include optional spaces on either side of the =. There must be no colon or tab before the =. When setting a macro on the command line, make sure to quote it if spaces or any shell variables that have to be passed to **make** are to be used.

Shell variables may be used within makefiles. This helps customize the execution of the makefile from outside the **make** utility.

Makefile Macros (2 of 2)

- Example:

```
OBJS=hcdbmaint.o hcrep.o hconline.o
CFLAGS=-O
chip: $(OBJS)
    @echo "Creating the chip executable"
    # Saving the old chip program
    @-mv chip chip.old
    @cc -o chip $(CFLAGS) $(OBJS)
    ...
```

Figure 4-12. Makefile Macros (2 of 2)

AU253.0

Notes:

This is an example of a makefile where two macros are being used. This makes it easy to change the makefile, and for the user to override macro settings from the command line.

Command Line Macros

- Values can be assigned to variables from the command line when make is executed.
 - This assignment supersedes any assignment made to that variable from within the makefile.
- Example:

```
$ make -f mymake chip CFLAGS=-g  
  
Creating the chip executable.  
  
$__
```

- This example creates the chip executable if needed, but with the -g option of the **cc** command instead of -O.

Figure 4-13. Command Line Macros

AU253.0

Notes:

Setting macro definitions on the command line overrides definitions in the makefile. Note that the compile option to be used for this run will be the -g flag instead of the -O, as defined in the makefile.

If any spaces are included in the definition, the string must be surrounded by quotes.

Internal Macros

- `$@` Name of the current target file
- `$$@` Label name on the dependency line
- `$?` Names of files more recent than target
- `$*` Name of the current parent file without suffix
- `$<` File name of the out-of-date file that caused a target to be created

Figure 4-14. Internal Macros

AU253.0

Notes:

If the `$@` macro is in the command sequence in the makefile, the **make** command replaces the symbol with the full name of the current target before passing the command to the shell to be run.

For example:

```
prog: prog.c
    xlc -o $@ prog.c
translates to:
    xlc -o prog prog.c
```

If the `$$@` macro is on the dependency line, the **make** command replaces this symbol with the label name.

For example:

```
prog: $$@.c
translates to:
    prog: prog.c
```

If the `$?` macro is in the command sequence, **make** replaces it with a list of parent files that have been changed since the target file was last changed.

For example:

```
prog: $$@.c
      xlc -o $@ $?
translates to:
      prog: prog.c
      xlc -o prog prog.c
```

If the `$*` macro is in the command sequence, the **make** command replaces it with the file-name part (minus the suffix) of the parent file.

For example:

```
prog: $$@.c
      xlc -o $@ $?
      @echo "$* has been created"
translates to:
      echo "prog has been created"
```

If the `$<` macro is in the command sequence, **make** replaces the symbol with the name of the file that started the file creation. It is used in suffixes, which we will discuss in Figure 4-15.

4.3 make Rules

Inference Rules

- Default rules for **make**
- Based on file suffixes
- Default rules file: /usr/ccs/lib/aix.mk
- Format:
 - rule:
 - <Tab> command
 - where rule is a single suffix (.c) or a double suffix (.c.o).
- File may be overridden on command line by MAKERULES variable.
- Rules may be added to makefile

Figure 4-15. Inference Rules

AU253.0

Notes:

The **make** command has a default set of inference rules, which you can supplement or overwrite with additional inference rule definitions in the makefile. The default rules are stored in the external file, /usr/ccs/lib/aix.mk. You can substitute your own rules file by setting the MAKERULES variable to your own file name from the command line. The following line shows how to change the rules file from the command line:

```
make MAKERULES=/pathname/filename
```

Inference rules consist of target suffixes and commands. From the suffixes, the **make** command determines the prerequisites, and from both the suffixes and their prerequisites, the **make** command determines how to make a target up-to-date. Inference rules have the following format:

```
rule:  
<Tab>command  
...
```

where *rule* has one of the following forms:

- .sl** A single-suffix inference rule that describes how to build a target that is appended with one of the single suffixes.
- .sl.s2** A double-suffix inference rule that describes how to build a target that is appended with .s2 with a prerequisite that is appended with the .sl.

The .sl and .s2 suffixes are defined as prerequisites of the special target, .SUFFIXES. The suffixes .sl and .s2 must be known suffixes at the time the inference rule appears in the makefile. The inference rules use the suffixes in the order in which they are specified in .SUFFIXES. A new inference rule is started when a new line does not begin with a <Tab> or # (number sign).

Makefiles can be made simpler by utilizing the information in the rules file.

For example:

```
creates the prog program:
# Make prog from 2 object files
prog: x.o y.o
# Use the cc program to make prog
    cc x.o y.o -o prog
# Make x.o from 2 other files
x.o:  x.c defs
#Use the cc program to make x.o
    cc -c x.c
# Make y.o from 2 other files
y.o: y.c defs
# Use the cc program to make y.o
    cc -c y.c
```

To make this file simpler, use the internal rules of the make program. Based on the internal rules, the **make** command recognizes three .c files corresponding to the needed .o files. This command can also generate an object from a source file by issuing a **cc -c** command.

Based on these internal rules, the description file becomes:

```
# Make prog from 2 object files
prog: x.o y.o
# Use the cc program to make prog
    cc x.o y.o -o prog
#Use the file, defs and the .c file
#when making x.o and y.o
x.o y.o: defs
```

Example Rules File

```
.SUFFIXES: .o .c .c~ ...

#PRESET VARIABLES
MAKE=make      LD=ld
LDFLAGS=       CC=cc
CFLAGS=-O      FC=xl f
FFLAGS=-O      AS=as
ASFLAGS=       GET=get
GFLAGS=
#SINGLE SUFFIX RULES

.c:
    $(CC)    $(CFLAGS)    $(LDFLAGS)    $< -o $@

.c~:
    $(GET)    $(GFLAGS)    -p $< > $*.c
    $(CC)    $(CFLAGS)    $(LDFLAGS)    $*.c -o $*
    -rm -f $*.c
...

#DOUBLE SUFFIX RULES

.c.o:
    $(CC)    $(CFLAGS)    -c $<

.c~.o:
    $(GET)    $(GFLAGS)    -p $< > $*.c
    $(CC)    $(CFLAGS)    -c $*.c
    -rm -f $*.c

.c~.c:
    $(GET)    $(GFLAGS)    -p $< > $*.c
...
```

Figure 4-16. Example Rules File

AU253.0

Notes:

The internal rules for the **make** program are in a file that resembles a description file. The internal rules file contains a list of file name suffixes (such as .o or .a) that the **make** command understands, plus rules that tell the **make** command how to create a file with one suffix from a file with another suffix. Also included is a set of preset variables. The suffixes and variables, plus those defined in a user's makefile, may be viewed by using the -p option with **make** (print). Output is sent to stderr.

The list of suffixes is similar to a dependency list in a description file. The **make** program creates the name of the rule from the two suffixes of the files that the rule defines. For example, the name of the rule to transform a .c file to a .o file is .c.o.

The **make** program has a set of single-suffix rules to build source files directly into a target file name that does not have a suffix (command files, for example). The **make** program also has rules to change the following source files having a suffix to object files without a suffix:

```
.C:    From a C++ language source file
.C~:   From an SCCS C++ language source file
.f:    From a FORTRAN source file
```

.f~: From an SCCS FORTRAN source file
.c: From a C language source file
.c~: From an SCCS C language source file
.sh: From a shell file
.sh~: From an SCCS shell file

The double suffix rules include:

.C.o: C source to object file
.C~.o: SCCS C source to object
.s.o: assembler source to object
.s~.o: SCCS assembler source to object

If an .a replaced the .o, the transition would be to archive instead of object.

To add more suffixes to the list, add an entry for the fake target name .SUFFIXES in the description file. For a .SUFFIXES line without any suffixes following the target name in the description file, the **make** command erases the current list. To change the order of the names in the list, erase the current list and then assign a new set of values to .SUFFIXES.

4.4 Source Code Control System (SCCS)

What is SCCS?

- A collection of commands used to control and track changes to text files.
- Facilities provided include:
 - Storing and retrieving various versions of text files
 - Recording the history of changes
 - Identifying the version of each source file from which a program was created
 - Controlling who has privileges to update files

Figure 4-17. What is SCCS?

AU253.0

Notes:

The Source Code Control System (SCCS) consists of 15 commands used to control and track changes to text files. It is often used for projects, such as application programming, where it is desirable to refer back to old versions. It also helps multiple users to edit the same file at the same time.

You need to install the `bos.adt.sccs` fileset to be able to use the SCCS facility.

SCCS stores a copy of the original file along with groups of changes called *deltas*. To retrieve a particular version of the source file, SCCS starts with the original file and applies appropriate deltas.

SCCS is useful for recording user comments associated with each delta of a source file. This aids in tracking development progress and history.

SCCS Terminology

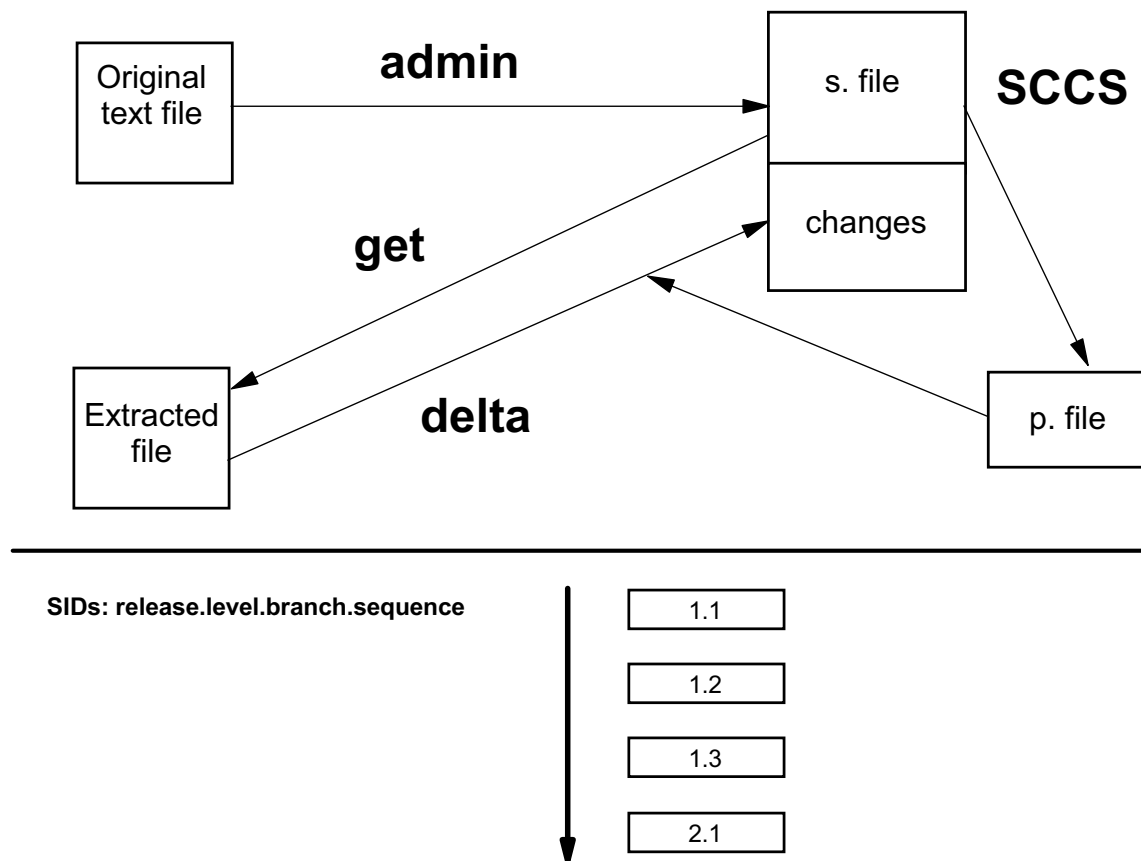


Figure 4-18. SCCS Terminology

AU253.0

Notes:

Some of the key SCCS commands are **admin**, **get**, and **delta**.

The **admin** command submits a file to the SCCS facility. From then on, SCCS controls the source file. This will create the s. file from the original source file. The s. file is the SCCS controlled file.

The **get** command is used to extract the original source file from the s. file. To view or change the source file, you need to extract the source file using the **get** command.

The **delta** command is used to make changes to the SCCS file. Using this command will change the latest version number of the SCCS file.

Each version of a file has an associated SID (SCCS Identification) number. The initial SID of a file is 1.1 (release 1, level 1). Each change (generally using the **delta** command), results in a new SID.

The **get** command can be used to retrieve any version from the SCCS file.

Creating a SCCS file

- Use the **admin** command to put a file under the control of SCCS.
- Syntax:
 - admin -i*orig_filename* *s.filename***
 - *orig_filename* follows -i (no spaces)
 - *s.filename* is the SCCS file to store the file *orig_filename*.
- The original file should be renamed or removed so that it does not interfere with the operation of SCCS.

Figure 4-19. Creating a SCCS file

AU253.0

Notes:

This command places a file in SCCS' custody.

For example:

To place a file `hello.c` under SCCS, use:

admin -ihello.c s.hello.c

where `s.hello.c` is the SCCS file created for the original source file `hello.c`

At this point, we have two copies of the file, the original file (`hello.c`) and the file under the SCCS custody (`s.hello.c`). It is a good idea to remove the original file or at least to rename the original file. Otherwise, there can be a conflict between SCCS and the original file name.

The **admin** command has a **-a** option that allows the assignment of users allowed to change the SCCS file. This option takes user IDs and group IDs. By not assigning these IDs, anyone can change the file.

get command

The **get** command:

- Retrieves a copy of a source file from SCCS.
- The retrieved copy is read-only by default.
- To retrieve a copy for editing and to lock others from retrieving a writeable copy, use the -e option.

Figure 4-20. get command

AU253.0

Notes:

The **get** command retrieves a copy of the source file. With no options, it retrieves a read-only copy of the file.

For example:

```
$ get s.hello
```

```
1.1
```

```
2 lines
```

```
$ ls -l
```

```
total 16
```

```
-r--r--r--  1 guest  usr          13 Mar 22 12:35 hello
```

```
-r--r--r--  1 guest  usr          156 Mar 22 11:00 s.hello
```

Once finished, the user can simply remove this file. Multiple users can get the same file in read-only mode.

To get a editable copy of the file, use the **-e** option.

For example:

```
$ get -e s.hello
1.1
2 lines
$ ls -l
total 16
-rw-r--r--  1 guest  usr      13 Mar 22 12:45 hello
-r--r--r--  1 guest  usr     156 Mar 22 11:00 s.hello
```

This prevents any other user from getting an editable copy of the same version until the current one is returned to SCCS or abandoned with the **unget** command. It does not keep people from working on different versions of the file at the same time.

Doing a **get -e** creates one additional file. A **p.** file is created in the same directory as the **s.** file. This contains a record of who checked out any versions of the file for update.

Note: We do not recommend that you edit an SCCS file directly. SCCS keeps track of SCCS file information within the same file and will mostly detect a conflict if the **s.** file is changed directly.

New releases can be started with the **get** command. This is done by specifying a new release number while extracting a file for editing.

For example:

```
$ get -e -r2 s.hello
1.5
new delta 2.1
3 lines
```

In this example, the current version of the file is 1.5. Note that the new version number would be 2.1 instead of 1.6.

A previous version of the source file can be retrieved using the **get** command by specifying an appropriate argument to the **-r** option.

For example:

```
$ get -r1.3 s.hello
1.3
3 lines
```

This example illustrates how an earlier version (1.3) is obtained from the SCCS file **s.hello**.

delta command

The **delta** command:

- Records changes made to the source file that was retrieved by **get -e**.
- Can add a comment about the change made.
- Removes the p. file and makes the source file available for someone else for editing.

Figure 4-21. delta command

AU253.0

Notes:

Changes are done to the source file retrieved by the **get -e** command. This changed file must be given back to SCCS using the **delta** command. This command provides an option to add a comment. It is suggested that a statement saying what was changed be included.

The **delta** command reads commands from standard input until it encounters a blank line or a end-of-file character. Comments must not be longer than 512 characters.

On completion, the record from the p. file makes that version available for others to edit. The entire file is removed if no other record is present in the file.

For example:

```
$ ls
hello p.hello s.hello
$ delta s.hello
Type comments, terminated with an End of File character
      or a blank line.
done
```

There are no SCCS identification keywords in the file. (cm7)

1.2

1 inserted

1 deleted

1 unchanged

\$ ls

s.hello

\$

unget command

The **unget** command:

- Prevents the creation of a delta after retrieving a writeable copy of the source file using the **get** command.
- Removes the writeable copy of source file and the p. file without affecting the version control.

Figure 4-22. unget command

AU253.0

Notes:

The **get -e** command issued by a user keeps the other users from editing the same file. The **delta** command is a way to release the lock. The **unget** command also releases this lock, but without checking the file back into the SCCS.

If you have checked out a file for editing, but do not want to check the file back in the SCCS, you can use the **unget** command. This can be handy when you corrupt the checked out version.

The **unget** command disregards the delta and removes the record of the **get -e** from the p. file.

For example:

```
$ unget s.hello
```

1.3

ID keywords

- Used to automatically embed information about the source file into the s. file.
- Place the ID keywords in a comment or declaration line, so that it does not interfere with the source code itself.
- The keywords are expanded when a **get** operation on the file is performed.
- Some common keywords:
 - %M%** The module name (name of file)
 - %I%** The SID
 - %Z%** The string **@(#)** as used by the **what** command
 - %W%** Embed **%Z%%M<Tab>%I%** together
- The **what** command:
 - Searches specified files for the **@(#)** string.
 - Prints text following the **@(#)** string until a double quote is reached.

Figure 4-23. ID keywords

AU253.0

Notes:

A user can find out the version of a source code by using the **what** command. However, for **what** command to work, the executable should contain ID keywords embedded in it.

ID keywords are macro definitions of various attributes of a source code. The ID keyword is translated by SCCS into meaningful information.

The **what** command searches for a line that begins with the characters **@(#)**. This command works with both ASCII files and binary files. It returns all text following these characters up to the first double quote. The **%Z%** ID keyword is used to embed the **@(#)** characters into the file.

ID keywords can work with shell scripts as well as compilable code (for example, C programs). With shell scripts, the keyword can be embedded into a command line. If the ID keywords were embedded in a comment in a C program, the compiler will strip out these keywords and they will not be a part of the resultant executable. Hence, these ID keywords are declared as a static character array within the program.

For example:

\$ cat my.sh

#!/usr/bin/ksh

#"@(#) my.sh 1.5"

command

\$ what my.sh

my.sh:

my.sh 1.5

Other Helpful SCCS Commands

- The **prs** command prints information about s. files managed by SCCS.
- The **rmDEL** command removes a delta.
- The **sact** command reports outstanding **get -e** operations.
- The **get** command uses the -x option to excluding deltas. Multiple deltas can be excluded.

Figure 4-24. Other Helpful SCCS Commands

AU253.0

Notes:

The **prs** command prints SCCS information to standard out. Many options are available for this command:

- a include details about removed deltas (**rmDEL** command).
- r gives SID details for a specific delta number.
- e gives all deltas created earlier than, and including that specified by the -r option.
- l gives all deltas created later than, and including that specified by the -r option.

Deltas can be removed only if no other deltas depend on them. This removal can be done with the **rmDEL** command. You will receive an error message if you try to remove a delta with dependencies.

For example:

```
$ rmDEL -r2.3 s.hello.c
```

Users may check files out of an SCCS file and forget to put them back. The **sact** command displays information about any outstanding **get -e** (edit) commands.

The **get** command may be used to retrieve a source file with particular deltas excluded. This is accomplished using the -x option of the **get** command. Multiple deltas may be excluded by listing them, comma separated, after the -x option. Ranges of deltas can be excluded using a hyphen to establish an inclusive range.

For example:

To exclude version 1.3, and 1.5 in the source version 2.3, use the following command:

```
$ get -x1.3,1.5 -r2.3 s.hello.c
```

To exclude version 1.3, 1.4, and 1.5 in the source version 2.3, use the following command:

```
$ get -x1.3-1.5 -r2.3 s.hello.c
```

Unit Summary

- An application example
- Targets and dependencies
- The makefile
- The **make** command
- Pseudo targets
- **make** Macros
- Using SCCS

Figure 4-25. Unit Summary

AU253.0

Notes:

Unit 5. AIX Dynamic Binding and Shared Libraries

What This Unit is About

This unit describes how to perform dynamic binding and create shared libraries in AIX. It provides an explanation of both static and dynamic binding, as well as the two types of dynamic binding: exec-time and load-time.

This unit also introduces the concept of shared libraries and contrasts them to non-shared libraries. The **ar** command is discussed.

Finally, many options of the **ld** command are explained.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Describe the differences between static and dynamic binding
- Create a dynamically-loadable C object file
- Create shared and nonshared C libraries
- Link application programs to shared and nonshared libraries
- List many AIX **ld** options and their functions
- Use the `dlopen()` and `dlclose()` system calls to dynamically bind object files during program execution

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 4

References

- *AIX 5L Version 5.1 documentation*
- *C and C++ Application Development on AIX* redbook, SG24-5674-00

Unit Objectives

After completing this unit, you should be able to:

- Describe the differences between static and dynamic binding
- Create a dynamically-loadable C object file
- Create shared and nonshared C libraries
- Link application programs to shared and nonshared libraries
- List many AIX **ld** options and their functions
- Use the `dlopen()` and `dlclose()` system calls to dynamically bind object files during program execution

Figure 5-1. Unit Objectives

AU253.0

Notes:

5.1 AIX Dynamic Binding and Shared Libraries

Overview

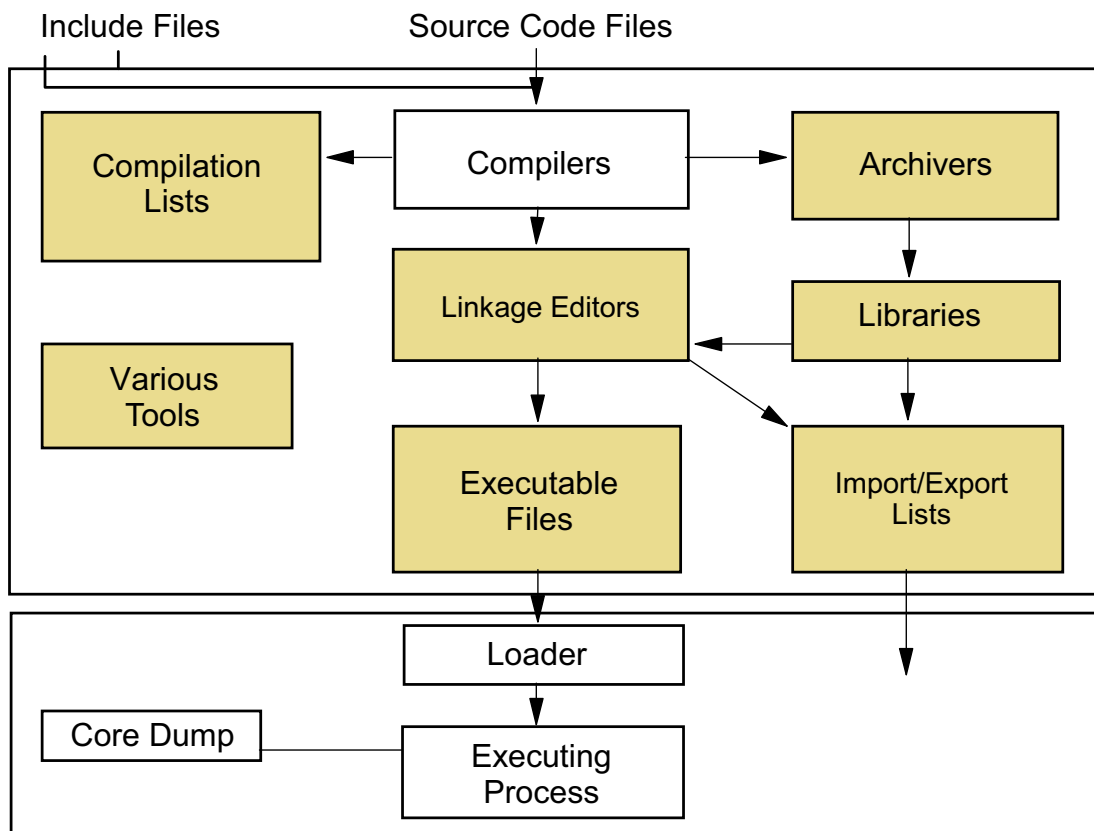


Figure 5-2. Overview

AU253.0

Notes:

AIX supports two popular UNIX programming concepts: dynamic binding and shared libraries.

Dynamic binding provides an efficient way of linking program modules. Instead of having the linkage editor statically bind all program modules and libraries into one potentially large executable file, the binder (which is called by the linkage editor) defers linking the modules until the program is executed. This results in smaller executables. It also allows external modules and libraries to be modified without a need to relink the modules to the applications that use them.

Shared libraries reduce the amount of memory required by applications by allowing many applications to share common code. When an application is linked to a shared library, the code for functions from the library is placed into a shared memory segment instead of the application's text segment.

While dynamic binding and shared libraries are two different concepts, they are often used together. Therefore, they are covered together in this unit.

This unit describes how AIX implements dynamic binding and shared libraries. Both implementations use the linkage editor. AIX dynamic binding uses import and export files to define symbols. The loader resolves these external symbol references at run time.

Libraries are created with the **ar** command. Libraries can also import and export symbols.

Static Binding Concepts

sample.c

```
main()
{
    print("This is main\n");
    funcA();
    funcB();
    funcC();
}
```

modA.c

```
funcA()
{
    printf("This is funcA\n");
}
```

modB.c

```
funcB()
{
    printf("This is funcB\n");
}
```

modC.c

```
funcC()
{
    printf("This is funcC\n");
}
```

sample

```
main()
{
    printf("This is main\n");
    funcA();
    funcB();
    funcC();
}
funcA()
{
    printf("This is funcA\n");
}
funcB()
{
    printf("This is funcB\n");
}
funcC()
{
    printf("This is funcC\n");
}
```

cc sample.c modA.c modB.c modC.c -o sample

Figure 5-3. Static Binding Concepts

AU253.0

Notes:

The traditional technique for linking program modules and library objects in UNIX is to statically bind all of the code into the text section of the executable file. The initialized data and BSS associated with all code sections are also included in the data section of the executable program. This means the executable file is self contained, but the file could be very large.

The example in Figure 5-3 builds an executable program called sample. (The resulting text of sample is shown as C source code purely for illustration.) The sample executable is created from the source file sample.c. The sample.c program makes calls to external functions from funcA(), funcB(), and funcC(). Each of these functions are external and are defined in the modules modA.c, modB.c, and modC.c, respectively. These individual functions are frequently referred to as "external symbols". In this example, the calls to these external symbols are resolved by the linkage editor by combining all symbols into the executable file. This is referred to as "static binding."

Static binding happens automatically when multiple source modules are provided to the **cc** or **ld** commands (remember that **cc** passes necessary parameters to the **ld** command).

Dynamic Binding Concepts (1 of 2)

- Objects are bound to the executable at run time instead of linkage time.
- Linkage editor (**ld**) calls the binder, which provides information needed to perform run-time binding:
 - The list of symbols to import from external modules (objects) is placed in the loader section of XCOFF file.
 - Glue code.
 - If binding an external module, the list of symbols to export is placed in the loader section of XCOFF file.
 - Import and export lists are used to declare imported or exported symbols.
- Types of dynamic binding:
 - exec() time
 - load() time

Figure 5-4. Dynamic Binding Concepts (1 of 2)

AU253.0

Notes:

When statically binding an application, all references to external symbols must be resolved at link time. If any external symbols or their modules are missing, the linkage editor (**ld**) reports an error.

AIX supports dynamic binding by allowing an application to be compiled and linked without needing all external symbols to be present. The programmer supplies an ASCII file that indicates the names of external symbols and the file in which to find them. The ASCII file is called the "import" file, and must be specified to the linkage editor when linking the application. The import file provides a way to "promise the resulting executable file that the external symbols will be available in the specific object files when the application is executed." Refer to Figure 2-24 for more information about XCOFF images.

The AIX binder is responsible for placing the list of external symbols to be imported into the loader section of the application's XCOFF file. The binder also places a small amount of code into the application's XCOFF file. This code is called "glue code" and is used to help the AIX loader resolve the external symbol references at run time. (The loader is part of the

AIX kernel and is responsible for loading and executing programs. The binder is a user program that is called by the linkage editor.)

AIX actually supports two kinds of dynamic bind: `exec()` time and `load()` time. With `exec()` time dynamic binding, the references to external symbols are resolved when the application is loaded and executed. This means that all text and data sections of the application are present throughout the application's execution. `load()` time binding means that the application explicitly loads a module to resolve references to external symbols on the fly. This is done via the `load()` system call. The application can `unload()` the module when it is no longer needed.

When creating a loadable object (a module that contains the code for external functions), the programmer specifies the function names (symbols) to be exported. This is also done via an ASCII file (referred to as an export file). The export file contains a list of the symbol names. The name of the export file must be provided to the binder (via the linkage editor) when linking the loadable objects.

The details of dynamic binding are to follow.

Dynamic Binding Concepts (2 of 2)

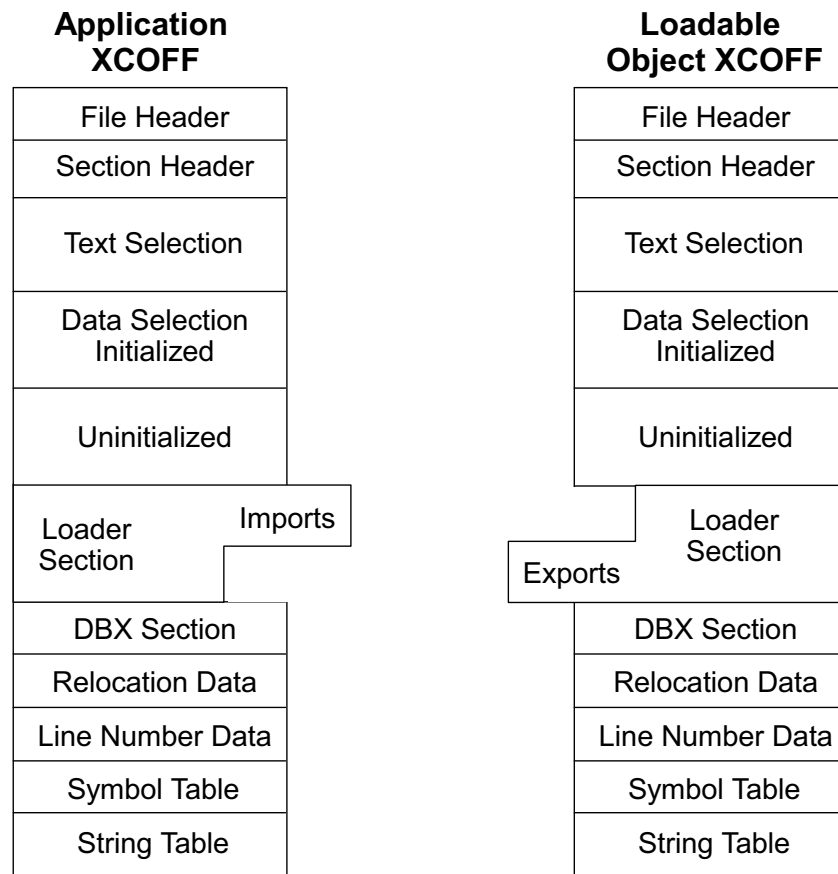


Figure 5-5. Dynamic Binding Concepts (2 of 2)

AU253.0

Notes:

Figure 5-5 illustrates the layout of AIX's XCOFF (Extended Object File Format) for an application file and loadable object file.

In this example, the application has information in the loader section to help the loader locate imported symbols.

The loader section of the loadable object contains a list of symbols to be exported and made available to applications.

Although, this example does not show it, a loadable object might also import symbols from another loadable object.

Import and Export Files

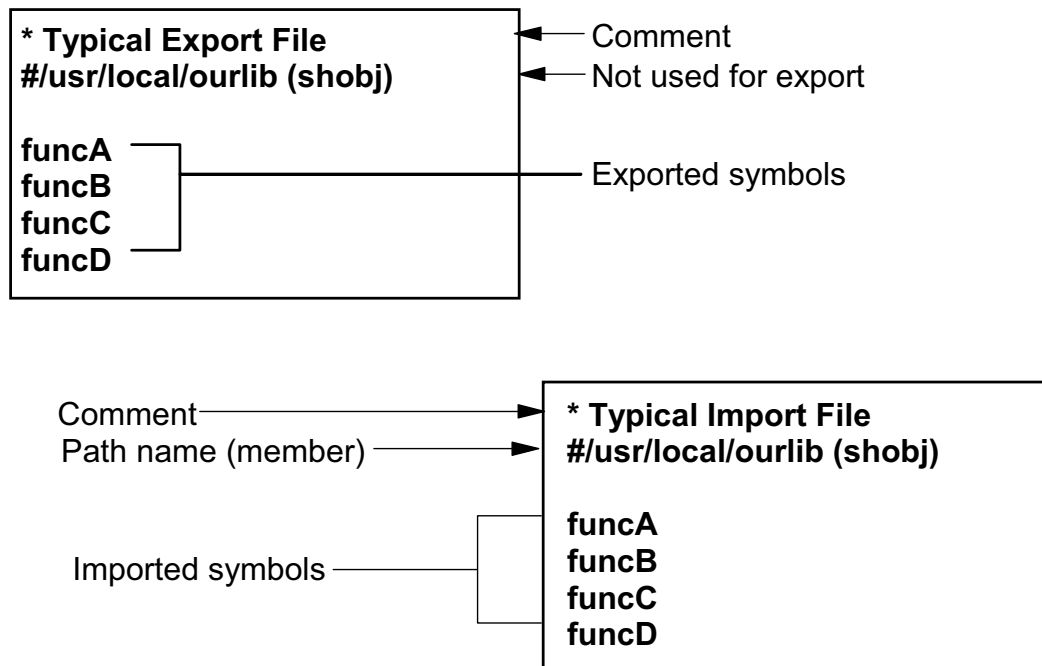


Figure 5-6. Import and Export Files

AU253.0

Notes:

Import and export files have a similar format. In fact, a single file can function as both the import and export file.

In an import or export file, any line starting with an asterisk (*) is treated as a comment.

The import file must specify the name of the file that contains the external functions. This is done with the syntax:

```
#!pathname
```

An archive contains one or more object files. A particular member object file of an archive can be specified as:

```
#!pathname (MemberObjectName)
```

The import file then lists the symbol names of the symbols to be imported.

The export file need only list the names of the symbols to export. If the `#!pathname` line is present, as in the case of using the same file for import and export, that line is ignored.

Creating a Loadable Object

myfuncs.c

```
funcA()
{
    printf("This is funcA\n");
}

funcB()
{
    printf("This is funcB\n");
}

funcC()
{
    printf("This is funcC\n");
}

funcXYZ()
{
    printf("This is funcXYZ\n");
}
```

myfuncs.exp

```
* Export file for myfuncs
#!/myfuncs
funcA
funcB
funcC
```

```
$ cc -c myfuncs.c -o
$ ld -bE:myfuncs.exp -o myfuncs.o -efuncA -lc
```

Figure 5-7. Creating a Loadable Object

AU253.0

Notes:

Here is an example of creating a loadable object. The source file myfuncs.c contains four functions: funcA(), funcB(), funcC(), and funcXYZ().

First, the source file is compiled into an object file, but not passed to the linkage editor (the -c flag means compile only).

Next, an export file is created. The export file must exist prior to passing the object file to the linkage editor, as its name must be included with the **ld** command. Notice that in this example, the symbol name for the function funcXYZ() is not exported. This means that any application wanting to reference the funcXYZ() function will need to be statically bound to the myfuncs object file.

The final step to creating the loadable object involves using the **ld** command. Here, the -b flag is an instruction to the binder, which is called by the linkage editor. The E used with the -b flag is used to specify the name of the export file (in this case, myfuncs.exp). The syntax of this flag is -bE:filename, where filename is the name of the export file.

Creating an Application

myapp.c

```
main()
{
    printf("Starting main...\n");
    printf("Calling funcC...\n");
    funcC();
    printf("Back in main...\n");
    printf("Calling funcB...\n");
    funcB();
    printf("Back in main...\n");
    printf("Ending program...\n");
}
```

myapp.imp

```
* Import file for myapp
#!/myfuncs
funcB
funcC
```

```
$ cc -bI:myapp.imp -o myapp myapp.c -O
```

Figure 5-8. Creating an Application

AU253.0

Notes:

Creating an application that dynamically binds to loadable objects is simply a matter of providing a list of external symbol names, and the names of the files from which they can be loaded at run time to the AIX binder. This list is given in the form of an ASCII import file.

The import file, here called myapp.imp, specifies the name of the file that contains the external symbols (in this case, myfuncs) and the symbol names of the functions to be imported (funcB and funcC). By using the -bl: flag to specify the import file, the references to funcB() and funcC() in myapp.c can be resolved at run time.

The -bl:impfilename option can be used with the **cc** or **ld** commands. In either case, it is passed on to the AIX binder.

The application can be compiled and linked to the loadable objects, as in this example, even if the loadable objects do not yet exist. The loadable objects must exist and have their symbols exported prior to executing the application; otherwise, the load/execution of the application will fail. The ./ in myapp.imp indicates a relative path. It is better to use a full path or use the binder option libpath or set the environment variable LIBPATH.

Execution - Time Binding

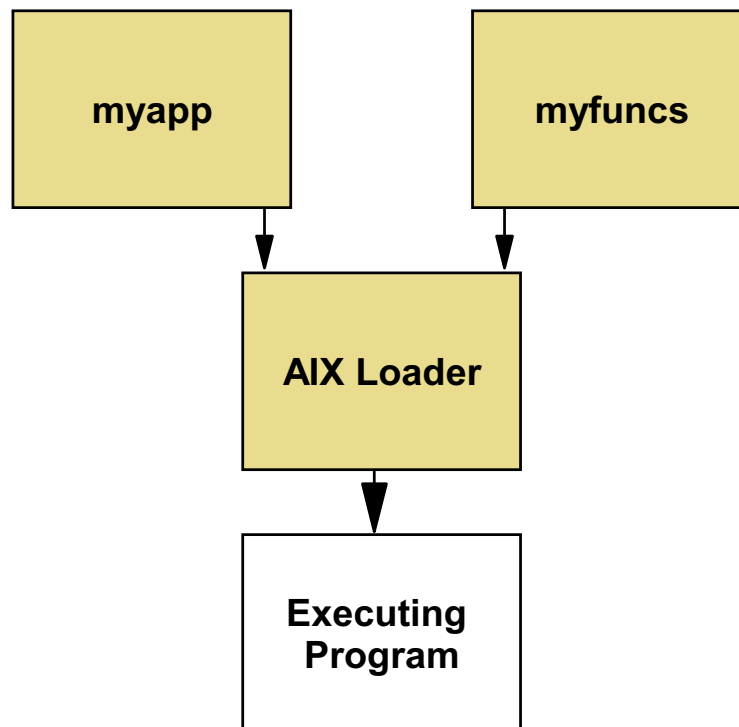


Figure 5-9. Execution - Time Binding

AU253.0

Notes:

The AIX loader is part of the kernel and is responsible for loading and executing programs. It is usually called by one of the exec system calls. The loader is also responsible for resolving dynamic binds to external functions.

Here we see how the loader resolves the external references to myfuncs made by myapp.

5.2 Shared Libraries

Library Terminologies

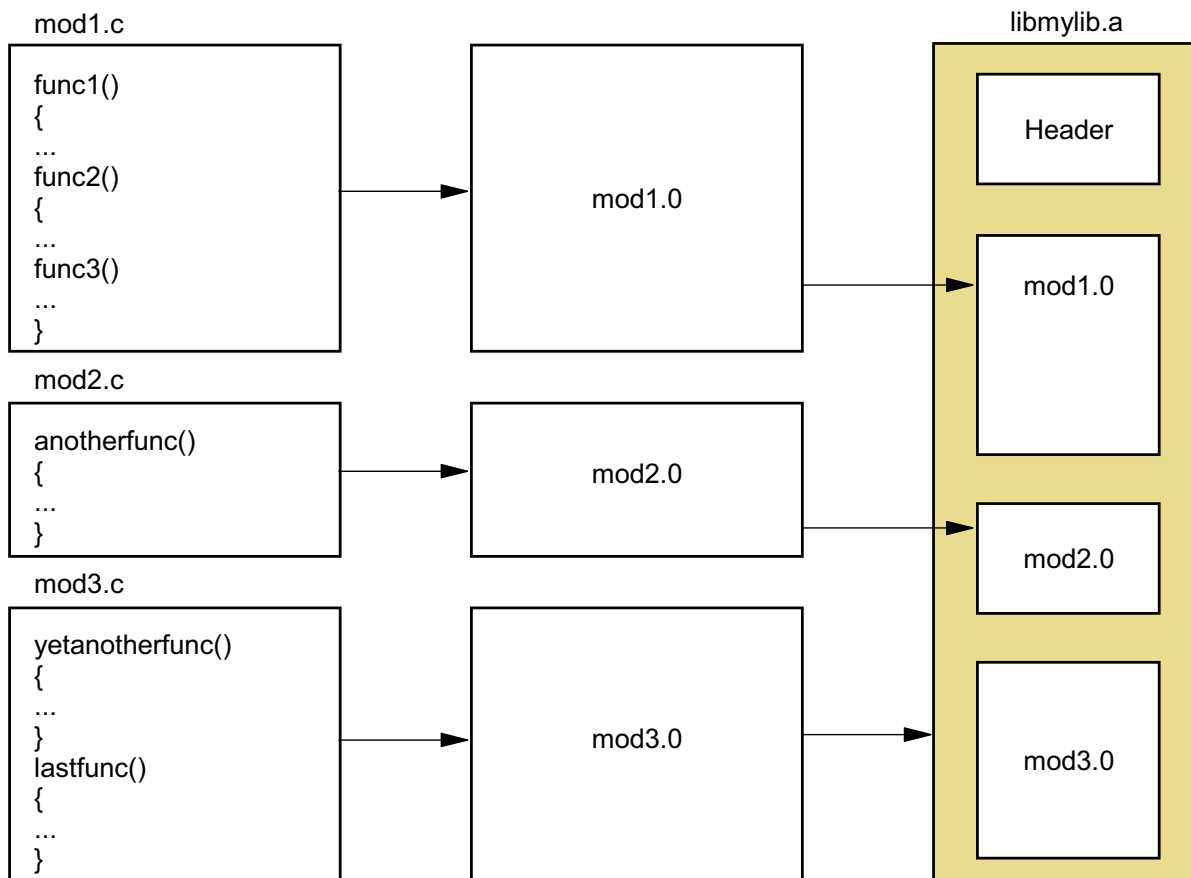


Figure 5-10. Library Terminologies

AU253.0

Notes:

Libraries contain one or more object files. Libraries are created and maintained with the **ar** command. Libraries include information in a header section, the object files, and a directory. The header contains offsets to certain archive file members.

The object files are compiled from source code using the `-c` flag. The resulting objects are then placed into libraries. The `/usr/include/ar.h` file describes the archive file format.

AIX Libraries

/usr/ccs/lib:

libc.a	libm.a
libcurses.a	

/usr/lib/:

libX11.a	libodm.a
libXt.a	libpthreads.a
liblvm.a	libblas.a

Figure 5-11. AIX Libraries

AU253.0

Notes:

Object files that contain functions can be archived into libraries. This way, frequently used functions can be accessed by many applications.

AIX includes the standard libraries shipped with many other UNIX systems. These libraries are found in the /usr/ccs/lib and /usr/lib directories. Examples of standard libraries in /usr/ccs/lib include:

- libc.a** The standard C library routines, such as printf()
- libcurses.a** The curses routine for screen and cursor manipulation tuning an application. The **libasl.a** is the super set of AIX screen library.
- libm.a** The standard UNIX C math routines, such as sqrt() and sin()

Application libraries in /usr/lib include:

- libX11.a** MIT's X-Windows routines
- libodm.a** Routines for manipulating the AIX Object Data Manager
- libXt.a** The X-Windows toolkit of routines

libpthreads.a POSIX compliant threads library

liblvm.a Routines for programming the Logical Volume Manager

libblas.a Basic Linear Algebra Subroutines, useful for performance

The AIX libraries contains both 32 bit and 64 bit versions of all functions.

Linking to Libraries

```
cc mysource.c -o myprog -lcurses  
  
or  
  
ld mysource.o -o myprog -lcurses  
-lc /usr/lib/crt0.o
```

- The -l option specifies the "name" of the library:
 - "name" is the key part of the libkey.a file name
 - Searches the /lib and /usr/lib directories
 - The -Ldir option adds the directories to the search.

Figure 5-12. Linking to Libraries

AU253.0

Notes:

The standard C library (libc.a) is automatically linked to any C program. Since this library is dynamically bound to most AIX programs, it must always be present in the /lib directory. As an example, removing the libc.a file would prevent one from booting the AIX kernel, because the kernel expects to dynamically link to the library at run time.

When compiling and linking an application that requires other libraries, these libraries can be specified with the -l flag (either to the **cc** or **ld** commands). The -l flag includes the key portion of the library name. The key portion is that part of the library name between the lib prefix and .a suffix. As an example, -lodm includes the library libodm.a. A space is not required between the -l and the key.

The linkage editor expects to find libraries in the /lib or /usr/lib directories. If a desired library exists in another directory, the linkage editor can be instructed to search that directory by specifying the directory name with the -L flag. An example might be:

```
cc -o myapp myapp.c -L/usr/local/lib -lourlib
```

where we are linking to a library named libourlib.a in the /usr/local/lib directory.

The -I and -L flags are associated with the **ld** command, and, if specified with the **cc** command, are passed to the **ld** command.

Static Linking to Libraries

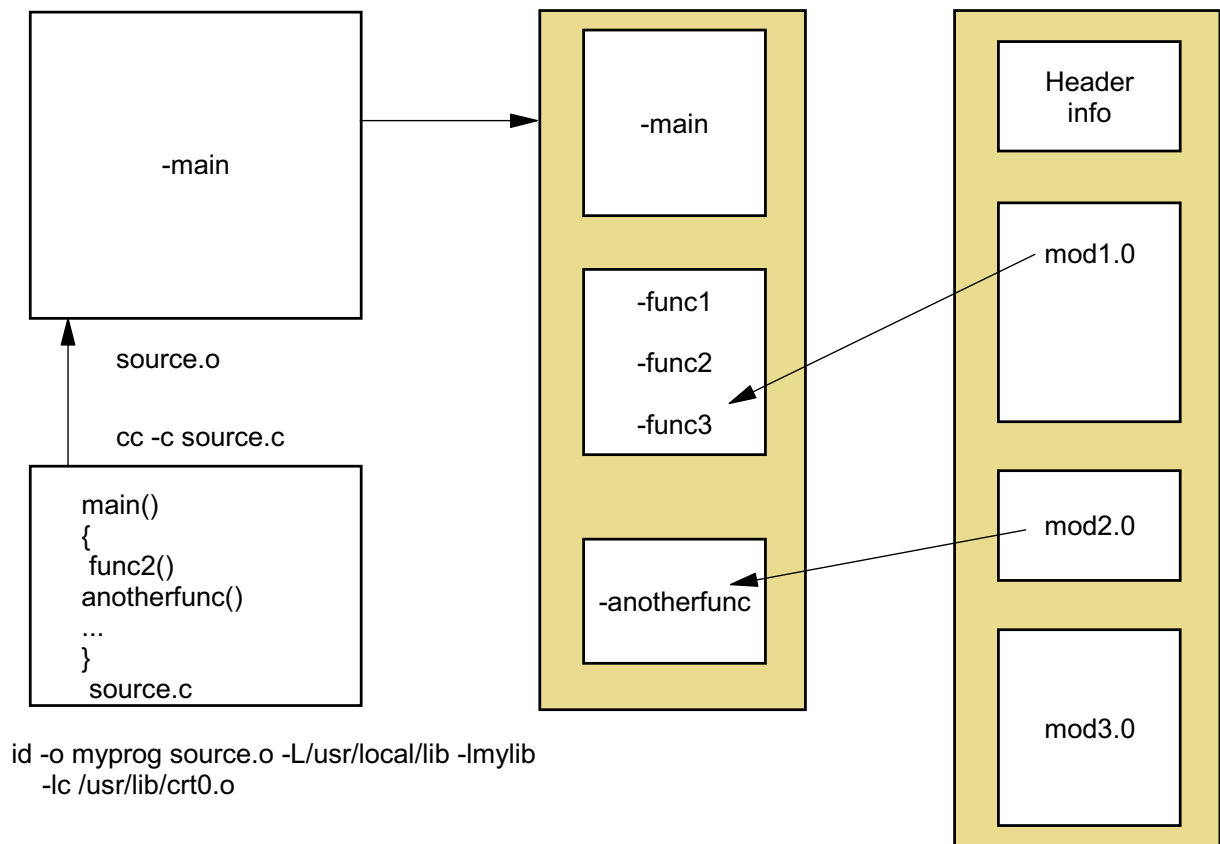


Figure 5-13. Static Linking to Libraries

AU253.0

Notes:

When an application statically links to a library, any object that contains symbols referenced by the application is linked into the executable. Remember, static linking means the text and code sections are combined into the executable.

Nonshared Libraries

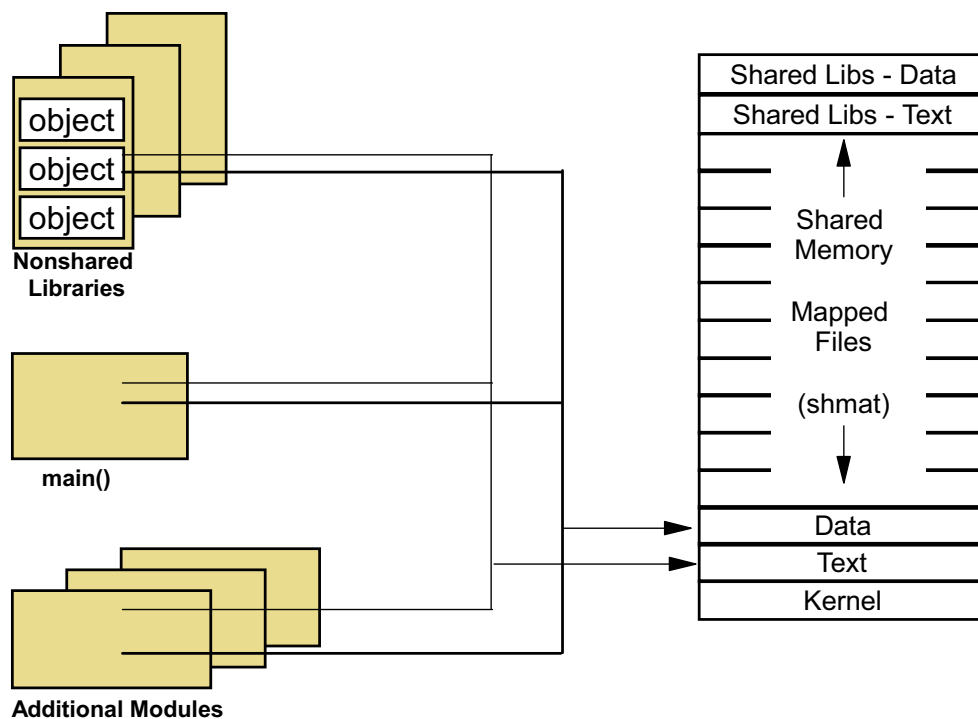


Figure 5-14. Nonshared Libraries

AU253.0

Notes:

Objects placed in a library can be specified as shared or nonshared. Here we see how nonshared objects are treated when bound (either statically or dynamically) into a process image. The text from a nonshared library object is loaded into the process's text segment. This is treated as an executable's own private copy of the text. The data section is loaded into the data segment of the process.

Simply put, linking to a nonshared library object means that the executable (application) gets a private copy of the object's CSECT. The CSECT is a control section containing an individual unit of coding or data.

Creating a Nonshared Library

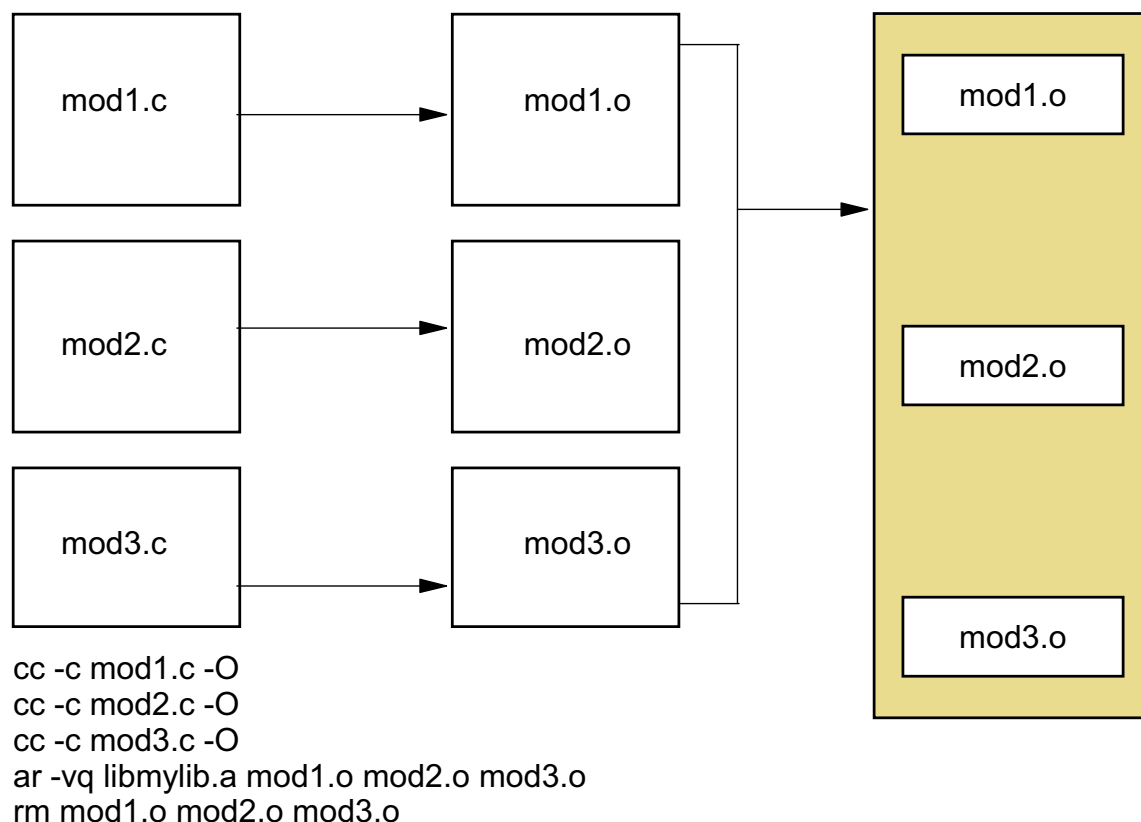


Figure 5-15. Creating a Nonshared Library

AU253.0

Notes:

Here we see an example of creating a new library. If the library does not currently exist, **ar** will automatically create it. Three objects are compiled from three source code files. The object files are then archived into the library using the **ar** command. The -q flag tells the **ar** command to append the files to the library. The -v flag provides verbose output (lists the objects as they are archived).

The syntax for a simple **ar** command is:

```
ar -flags libname objfile . . .
```

There are additional options to **ar** that include the ability to position new files before or after existing files in the archive. A -vt will list the contents of the library. For example:

```
ar -vt libmylib.a
```

The advantage to the -q option is that it is fast. One disadvantage is that it appends the file to the library, even if there is already a file in the library with the same name. An alternative is to use -ru, which replaces existing files if they exist and appends files to the end otherwise.

Shared Libraries

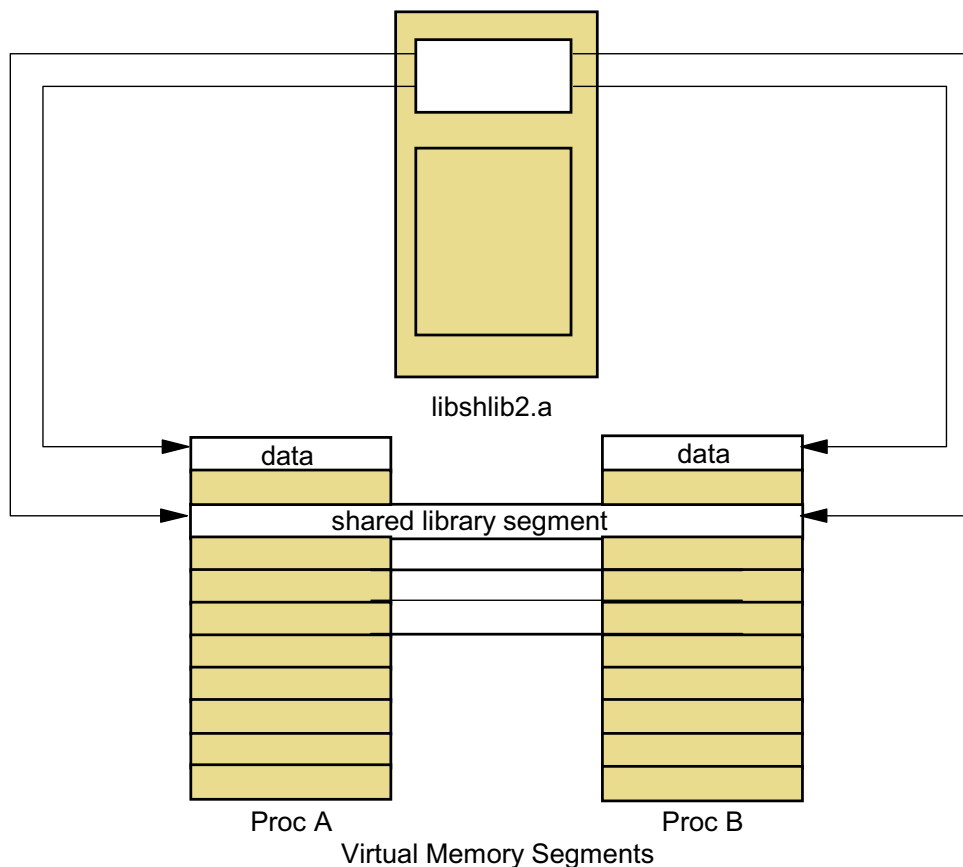


Figure 5-16. Shared Libraries

AU253.0

Notes:

The concept of shared libraries in UNIX provides more efficient use of system memory by allowing many processes to share the text from library routines. When a process links to an object in a shared library in AIX, the text from that object is mapped to shared library segment.

The data section of the shared library objects must be kept private for each process. Shared library data is put in a per-process shared library data segment.

The example in Figure 5-16 shows three different processes linking to shared objects in three different libraries. Relocation code enables each process to access the instructions from the routines in these libraries. The concept of reentrant code allows many processes to use the text at the same time.

Various commands, such as **genld**, **genkld**, **tprof**, and **svmon** provide information on shared libraries that are in use by processes.

Creating a Shared Library

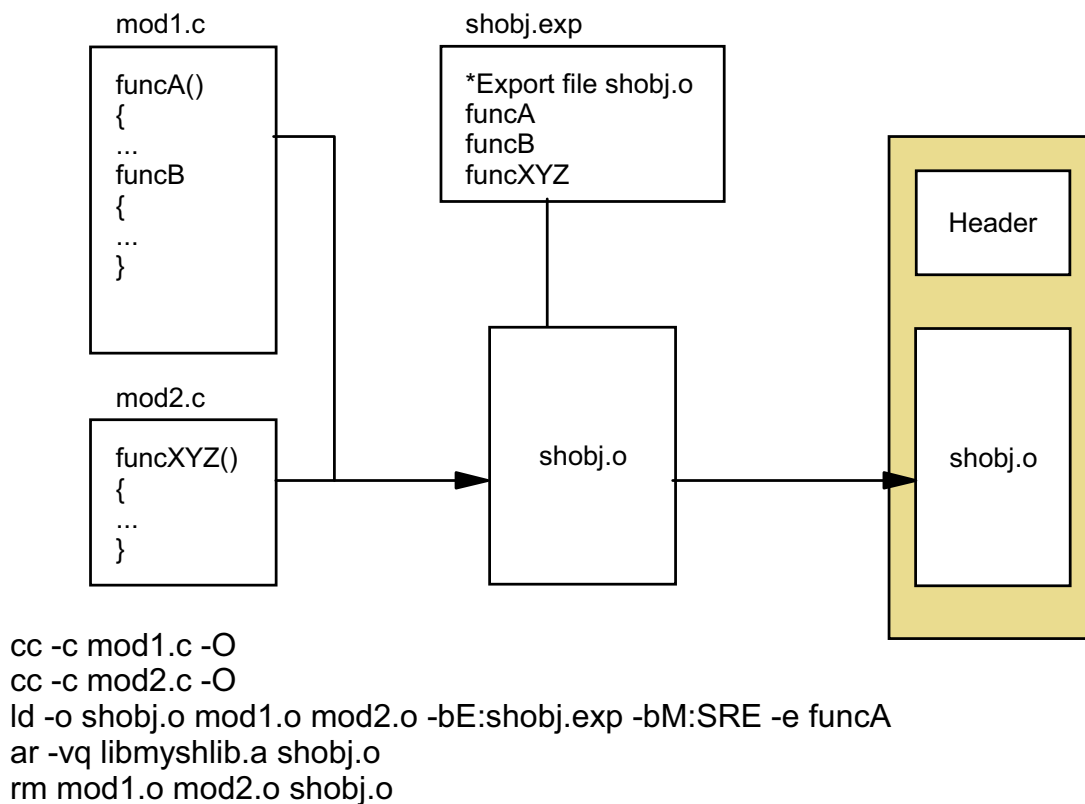


Figure 5-17. Creating a Shared Library

AU253.0

Notes:

A shared library contains one or more shared objects. Shared objects are created by compiling one or more source code files into object files (using the `-c` option of the `cc` command), then linking the object files into a single shared object. The shared object is then archived into a library.

A library may contain a combination of shared and nonshared objects. If an application links to a nonshared object, the code from that object is placed into the text segment of the application.

Applications may also be linked to shared objects in a nonshared manner by overriding the default shared linking procedure. This is accomplished by linking the application to the library using the `-bnso` option flag with the `ld` command. An example is given later in this lecture.

The key to making an object shared is to use the `-bM:SRE` option with the `ld` (or `cc`) command. Remember, the `-b` flag indicates an option to the AIX binder. The `M` specifies the "mode" of the link. `SRE` sets the mode to "shared reusable (reentrant)". This option can also be specified as `-bmode:SRE`.

Another important option flag illustrated Figure 5-17 is the definition of an entry point to the shared object (-efuncA). Because library objects do not contain a main() function, one must specify an entry point into the module. Actually, any function from the object can be designated. Note that only the symbol name is used with the -e flag. The parentheses are omitted.

Note that the example in Figure 5-17 includes the name of an export file (-bE:shobj.exp). Any objects used for dynamic binding must have their symbols explicitly exported.

Linking to a Shared Library (1 of 2)

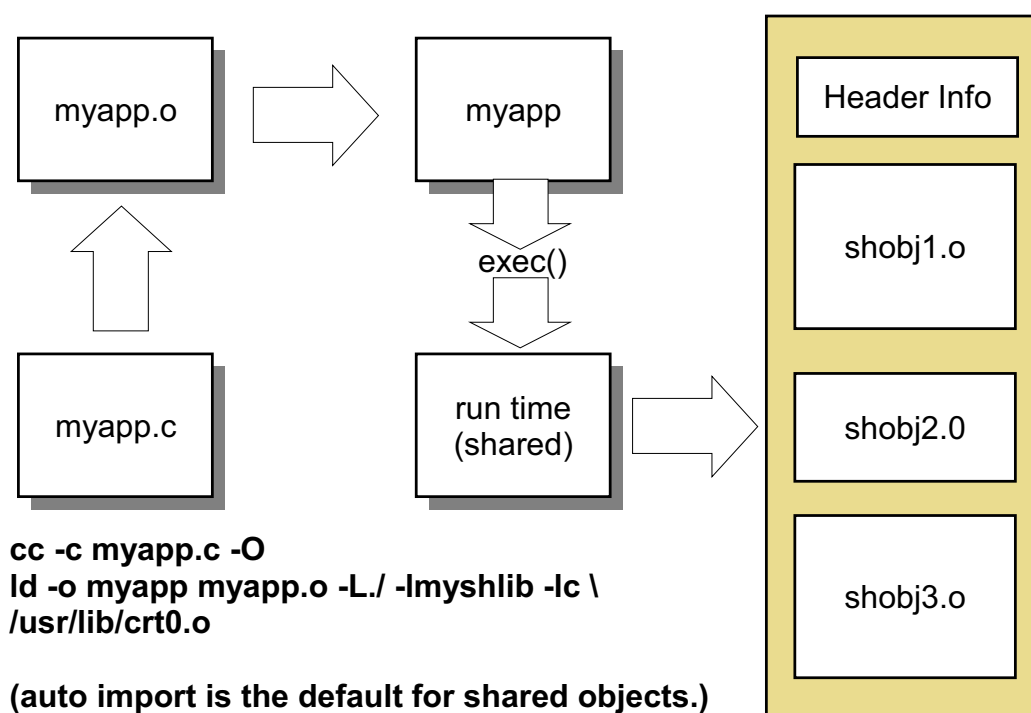


Figure 5-18. Linking to a Shared Library (1 of 2)

AU253.0

Notes:

When linking an application to a shared library, it is not necessary to use an import file. All required functions are automatically imported from the shared library. The **cc** command provides the flags `-bautoimp` or `-bso` to specify that all required symbols from shared libraries be automatically imported, but because this action is the default, you usually will not see these flags used. These `-b` flags are passed to the binder, called by the linkage editor.

The example in Figure 5-18 starts by compiling the source code file `myapp.c` into an object file name `myapp.o`. The `-O` flag instructs the compiler to perform optimization.

The file `myapp.o` is then linked (dynamically) to the necessary objects from the library `libmyshlib.a` (in this case, `shobj2.o`). No import file or list is required if `shobj2.o` was created as a shared object. Notice that the library `libmyshlib.a` is in the current directory, which is why the `-L./` option is used.

The executable file `myapp` contains information in its XCOFF loader section to resolve references to symbols in `shobj2.o`.

Upon execution, the AIX loader (called by the `exec()` system call) loads the `shobj2.o` object text and the data portion into the application's virtual address space. The text of `shobj2.o` is shared. The data of `shobj2.o` for this process is private to this process. The compile and link could have been done in a single step:

```
cc -o myapp -O -L./ -lmyshlib myapp.c
```

Linking to a Shared Library (2 of 2)

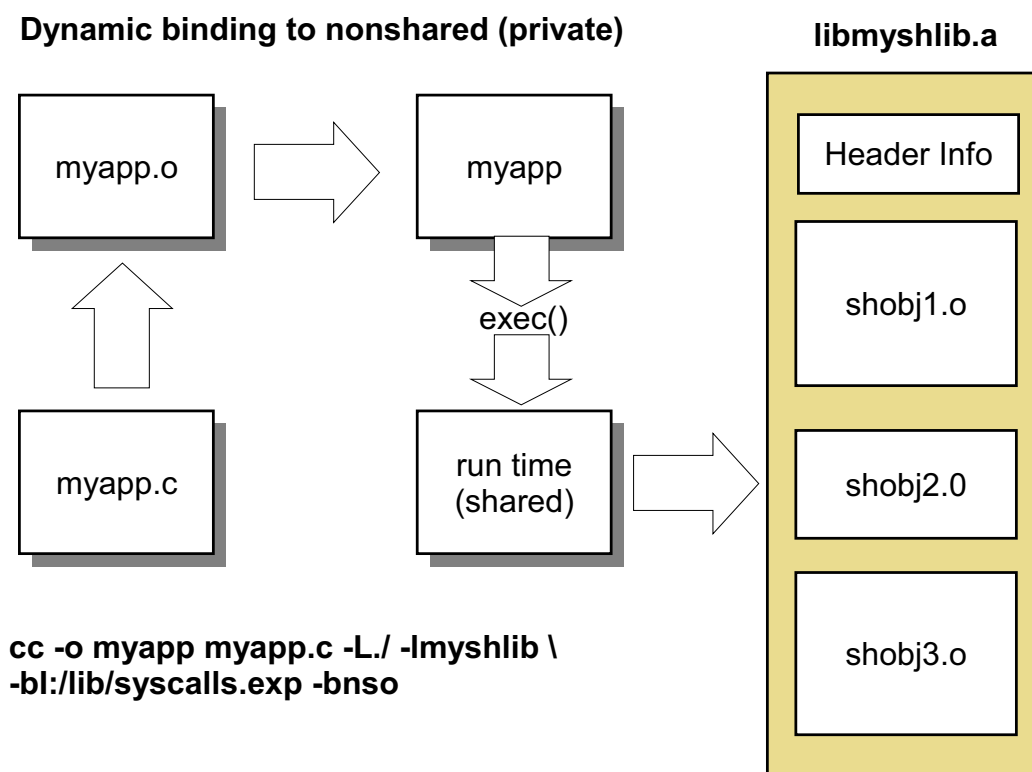


Figure 5-19. Linking to a Shared Library (2 of 2)

AU253.0

Notes:

The example in Figure 5-19 is similar to the previous one, except that a private copy of the text of shobj2.o is desired, so the flag `-bnso` is given to indicate non shared objects. This means the resulting executable will have a private copy of the text shobj2.o loaded into the application process's text segment at execution time. Note that this is still using dynamic binding principles.

There is something very important to note here. AIX requires that if any application binds with a private copy (nonshared) of a library object, all library objects must be bound in a nonshared manner. Since the application and functions in shobj2.o probably make calls to functions found in the libc.a library (which is automatically bound) and libc.a is a shared library, the exported symbols from libc.a must be manually imported by the application. IBM provides a default import file for the functions in libc.a. The file is `/lib/syscalls.exp`. It must be included with the `-bl:` flag in this example.

Using a Shared Library

- The method of linking decides how shared libraries are searched at run time
- The methods of using shared libraries are:
 - At compile time
 - Searching at run time
 - Shared or nonshared
 - Lazy loading

Figure 5-20. Using a Shared Library

AU253.0

Notes:

The most important point to remember about using shared libraries is that the way the application is linked will determine how the shared libraries will be searched for at run time.

At compile time, the method used will depend on the type of shared object being used. As far as the linker is concerned, there are three types of shared objects that it can handle:

- An archive library, which contains object files with the SRE bit set. To include an archive library, the `-l` and `-L` linker option can be used.
- An individual object file with the SRE bit set, for example, `shr1.o`. Specifying the absolute or relative path name is the only way to include it on the command line.
- A new style shared object of the form `lib name.so`. To include this type of shared object, the `-brtl` linker option should be used.

Searching at run time requires the `LIBPATH` environment variable, if the shared libraries exist in a different directory. If a relative or absolute path name is used to specify a shared object when the application is compiled, and the `-bnoipath` option is not specified, then the system loader will only look for the shared object using the exact pathname specified at link

time for that object. If a shared object cannot be found by the system loader when trying to start the executable, an error message is printed by the `exec()` subroutine.

Shared or non-shared

AIX supports the use of the `-bdynamic` and `-bstatic` linker options to determine how a shared object should be treated by the linker. If the `-bstatic` option is used, the shared object text is hardcoded as a part of the executable. These options are toggles and can be used repeatedly in the same link line.

Important Note: Remember to specify `-bdynamic` as the last option on the link line to ensure that the system libraries are treated as shared objects by the linker. If this is not done, and the system libraries are treated as normal archive libraries, the executable produced will be larger than normal. In addition, it will have the disadvantage that it may not work on future versions of AIX, because it is hardcoded with a specific version of system libraries.

Lazy loading is a mechanism for deferring the loading of modules until one of its functions is required to be executed. By default, the system loader automatically loads all of the module's dependants at the same time. Use the `-blazy` linker option to enable lazy loading.

Note: Lazy loading works only if the run-time linker is not enabled.

Binding at Load Time

Programatic control of loading shared objects

- `dlopen()`
 - Opens the shared object.
 - Dynamically maps it into the program's address space.
 - If successful, returns a handle.
- `dlclose()`
 - Removes access to a shared object that was loaded into the processes' address space with the `dlopen` subroutine.
- `dlsym()`
 - Searches for symbols in the loaded shared object.
 - If successful, returns a pointer to the address of the symbol.
- `dlerror()`
 - Obtains information about the last error that occurred in a dynamic loading routine.
 - The return value is a null terminated string.
 - Not thread safe.

Figure 5-21. Binding at Load Time

AU253.0

Notes:

The `dlopen()` family of subroutines is supported on the AIX operating system. When used appropriately, they allow a program to dynamically load shared objects into the address space, use functions in the shared object, and then unload the shared object when it is no longer required.

The `dlopen()` subroutine

The `dlopen` function is used to open a shared object, and dynamically map it into the running programs address space.

The `dlopen` subroutine has two parameters. The first parameter is the full path to a shared object. It can also be a path name to an archive library that includes the required shared object member name in parenthesis, for example, `/lib/libc.a(shr1.o)`.

The second parameter specifies how the named shared object should be loaded. The flags parameter must be set to `RTLD_NOW` or `RTLD_LAZY`. If the object is a member of an archive library, the Flags parameter must also set `RTLD_MEMBER`.

The subroutine returns a handle to the shared library that gets loaded. On failure, the subroutine returns NULL.

The dlclose() subroutine

The dlclose subroutine is used to remove access to a shared object that was loaded into the processes' address space with the dlopen subroutine. The subroutine takes the handle returned by dlopen as its argument.

The dlsym() subroutine

The dlsym routine searches for symbols in the loaded shared object. The dlsym subroutine accepts two parameters. The first is the handle to the shared object returned from the dlopen subroutine. The other is a string representing the symbol to be searched for.

If successful, the dlsym subroutine returns a pointer that holds the address of the symbol that is referenced. On failure, the dlsym subroutine returns NULL.

The dlerror() subroutine

The dlerror subroutine is used to obtain information about the last error that occurred in a dynamic loading routine (that is, dlopen, dlsym, or dlclose). The returned value is a pointer to a null-terminated string without a final newline.

This routine is not thread-safe. Details about thread-safe routines will be dealt later in the "Threads" unit.

For example:

A sample code snippet illustrating this concept is given below:

```
lib_handle = dlopen (FilePath, Flags);
lib_func = dlsym(lib_handle, "locatefn");
error = dlerror();
if (error)
{
    fprintf(stderr, "Error:%s \n",error);
    exit(1);
}
dlclose(lib_handle);
```

Run-Time Linking

- Ability to resolve undefined and non-deferred symbols in shared modules after the program execution has already begun
- Run-time linking should be used only when necessary
- Rebinding of symbols is made possible
- The -G linker option
 - × -berok
 - × -brtl
 - × -bsymbolic
 - × -bnortllib
 - × -bnoautoexp
 - × -bM:SRE

Figure 5-22. Run-Time Linking

AU253.0

Notes:

Generally, references to the symbols in the shared objects are bound at link time. That is, the output module associates an imported symbol with its definition in a specific object. The source of the definition can be seen by using the **dump -Tv** command on the executable or shared object.

To build shared objects enabled for run-time linking, use the -G flag and build the shared object with the **ld** command rather than the compiler **cc** or **xlc** commands. The -G linker option enables the combination of the following options:

- -berok: Enables creation of the object file, even if there are unresolved references.
- -brtl: Enables run-time linking. All shared objects listed on the command line (those that are not part of an archive member) are listed in the output file. The system loader loads all such shared modules when the program runs, and the symbols exported by these shared objects may be used by the run-time linker.
- -bsymbolic: Assigns this attribute to most symbols exported without an explicit attribute.

- `-bnortllib`: Removes a reference to the run-time linker libraries. This means that the module built with `-G` option will be enabled for run-time linking, but the reference to the run-time linker libraries will be removed. Note that the run-time libraries should be referenced to link the main executable only.
- `-bnoautoexp`: Prevent automatic exportation of any symbol.
- `-bM:SRE`: Build this module to be shared and reusable.

Another advantage of using run-time linking is that developers do not need to maintain a list of module interdependencies and import/export lists. By using the `-bexpall` option, all shared objects can export all symbols, and the run-time linker can be used to resolve the inter-module dependencies.

When using run-time linking, the order of specifying libraries and objects on the command line is important. In addition, all of the shared objects specified on the command line will be included in the header section of the resulting executable. This is the list of libraries and objects that will be searched in sequence to resolve symbols.

For example:

This example illustrates how the order of linking affects run-time linking. This example uses three source files: `f1.c`, `f2.c`, and `main.c`.

File `f1.c` and `f2.c` are identical. The content is as follow:

```
#include <stdio.h>
int func()
{
    printf( "\tinside %s...\n", __FILE__ );
}
```

File `main.c` is as follows:

```
main()
{
    func();
}
```

Compile, link, and execute as follows:

```
$ cc -c main.c
$ cc -c f1.c
$ cc -c f2.c
$ ld -o libf1.so f1.o -G -bnoentry -bexpall
$ ld -o libf2.so f2.o -G -bnoentry -bexpall
$ cc -o main main.o -lf1 -lf2 -brtl -L.

ld: 0711-224 WARNING: Duplicate symbol: .func
ld: 0711-224 WARNING: Duplicate symbol: func
```

Id: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.

\$./main

inside f1.c...

\$ cc -o main main.o -lf2 -lf1 -brtl -L.

Id: 0711-224 WARNING: Duplicate symbol: .func

Id: 0711-224 WARNING: Duplicate symbol: func

Id: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.

\$./main

inside f2.c...

\$

Rebinding System Defined Symbols

- The shared libraries shipped with the AIX operating system are not enabled for run-time linking.
- The shared libraries can be enabled by using the **rtl_enable** command.
- Not all symbols can be rebound:
 - If a symbol is referenced in the same file that also defines it, the symbol gets bound at link time and cannot be rebound.
 - References to symbols with the -symbolic attribute cannot be rebound.

Figure 5-23. Rebinding System Defined Symbols

AU253.0

Notes:

You may want to rebind function symbols so that the definitions may be picked up from a different location. In this case, the main program must be linked with the run-time linker and the module providing the alternate definition has to export the symbol.

Whenever a symbol is rebound, a dependency is added from the module using the symbol to the module defining the symbol. This dependency prevents modules from being removed from the address space prematurely. This is important when a module loaded by the `dlopen()` subroutine defines a symbol that is still being used when an attempt is made to unload the module with the `dlclose()` call.

It is possible to rebind the definitions of system library symbols.

For Example:

```
rtl_enable -o /usr/local/lib/libc.a /lib/libc.a
```

```
cc .... mymalloc.o -L /usr/local/lib -brtl -bE:myexports
```

In this example, mymalloc.o defines malloc and the export file myexports causes the symbol malloc to be exported from the main program. Calls to malloc from within libc.a will now go to the malloc routine defined in mymalloc.o.

Dynamic Binding and Shared Libraries (1 of 2)

		Dynamic Binding	
		NO	YES
S H A R E D L I B R A R I E S	NO (Flat File)	<pre>cc -c app.c; cc -c loadmod.c cc -o app app.o loadmod.o</pre> <p>PVT-TXT STATIC</p>	<pre>cc -c app.c; cc -c loadmod.c vi imp-exp -#loadmod -funcA cc -o loadmod loadmod.o -efuncA -bE:imp-exp cc -o app app.o -bl:imp-exp</pre> <p>PVT-TXT DYN. BIND</p>
	NO (LIB)	<pre>cc -c app.c; cc -c loadmod.c ar -vq libmylib.a loadmod.o cc -o app app.o -lmylib -L</pre> <p>PVT-TXT STATIC</p>	<pre>cc -c app.c; cc -c loadmod.c vi imp-exp -#loadmylib.a(loadmod) -funcA cc -o loadmod lomod.o -efuncA -bE:imp-exp ar -vq libmylib.a loadmod cc -o app app.o -bl:imp-exp -L/</pre> <p>PVT-TXT DYN. BIND</p>

Figure 5-24. Dynamic Binding and Shared Libraries (1 of 2)

AU253.0

Notes:

Figure 5-24 illustrates four scenarios that do not use shared libraries. The scenarios are:

- Without using libraries and dynamic binding
- Creating and using libraries, but not dynamic binding
- Using dynamic binding, but not creating or using libraries
- Creating and using libraries with dynamic binding

PVT-TXT means that the object text is not shared.

Dynamic Binding and Shared Libraries (2 of 2)

		Dynamic Binding	
		NO	YES
S H A R E D L I B R A R I E S	YES (LIB)	N/A	<pre>cc -c app.c; cc -c loadmod.c vi imp-exp -#libmylib.a(loadmod) -funcA cc -o loadmod lodmod.o -efuncA -bE:imp-exp -bM:SRE ar -vq libmylib.a loadmod cc -o app app.o -lmylib -L</pre> SHR-TXT DYN. BIND
	YES but BNSO (LIB)	<pre>cc -c app.c; cc -c loadmod.c vi imp-exp -#libmylib.a(loadmod) -funcA cc -o loadmod loadmod.o -efuncA -bE:imp-exp -bM:SRE ar -vq libmylib.a loadmod cc -o app app.o -bnso -lmylib -L/ -bl:/usr/lib/syscalls.exp</pre> PVT-TXT STATIC (c lib - dyn. bind)	<pre>cc -c app.c; cc -c loadmod.c vi imp-exp -#libmylib.a(loadmod) -funcA cc -o loadmod loadmod.o -efuncA -bE:imp-exp -bM:SRE ar -vq libmylib loadmod cc -o app app.o -bnso -bl:imp-exp -L/ -bl:/usr/lib/syscalls.exp</pre> PVT-TXT DYN. BIND

Figure 5-25. Dynamic Binding and Shared Libraries (2 of 2)

AU253.0

Notes:

Figure 5-25 illustrates three scenarios that use shared libraries. The scenarios are:

- Using a shared library for dynamic binding
- Using a non-shared object library, but not using dynamic binding
- Using a non-shared object library for dynamic binding

SHR-TXT means that the object text is shared.

5.3 Related Commands

AIX ld Options

Popular options of the AIX linkage editor:

- | | |
|---------------|---|
| • -o filename | Specifies output file |
| • -l name | Specifies library keys, in linkage order
(where library name is libname.a) |
| • -L libdir | Specifies directory for additional libraries |
| • -e label | Specifies the executable's entry point |
| • -b Option | Sets special processing options |
| • -s | Strips symbol table |
| • -v | Verbose; shows linkage execution steps and arguments |

Figure 5-26. AIX ld Options

AU253.0

Notes:

There are many options for the **ld** command. Some are listed here. Many can be given with the **cc** command, but are passed to the **ld** command. Many of these options have been discussed in other lectures.

The -o option is used to specify the name of the resulting executable file or object file. If omitted, the resulting file name is a.out.

The -l option specifies a library to include in the linking process. The -l option can be used many times to include many libraries. The -l option expects the key portion of the library file name, which excludes the lib prefix and .a suffix. In the case of -lmylib, the file name of the library would be libmylib.a.

The -L option is used to specify additional directories, beyond /lib and /usr/lib, to search for specified libraries.

The -e option is used to specify the entry point as a module. The default entry point is main, but modules that are not stand-alone executables, such as loadable shared objects, must have a different entry point designated. Otherwise, an error results.

The -b option sets special processing options, such as -bnso for the no share option.

The ar Command

```
ar -options archive_name mod1 [ mod2 ] . . .
```

- o Orders the members to compress space
- v Verbose
- u Updates only objects that have changed
- q Adds object to end of library
- d Deletes object from library
- r Replaces object in library
- t Table of contents
- x Extracts member

Figure 5-27. The ar Command

AU253.0

Notes:

Here is the syntax for the **ar** command. Included are some basic option flags.

For example:

```
ar -d lib obj.o (delete object)
ar -vx lib obj.o (extracts member)
ar -t lib (TOC)
```

The nm command displays information about symbols in object files, executable files, and object-file libraries. If the file contains no symbol information, the nm command reports the fact, but does not interpret it as an error condition.

```
nm lib (TOC w/functions)
```

Adding an Object to a Library

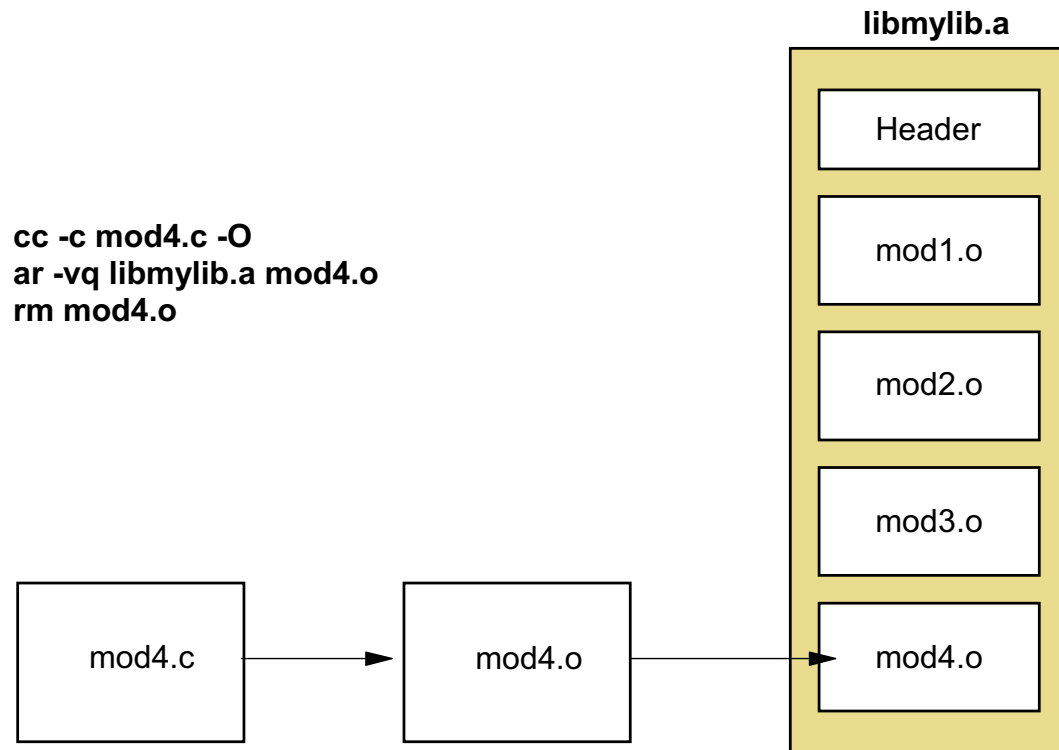


Figure 5-28. Adding an Object to a Library

AU253.0

Notes:

The **ar** command appends objects to a library via the -q flag. New objects are added to the end of the library. Other flags exist to control placement of the new object into the library, such as -bobjname, to indicate "before" the specified object.

For example:

```
ar -vb mod3.o libmylib.a mod4.o
```

Replacing an Object in a Library

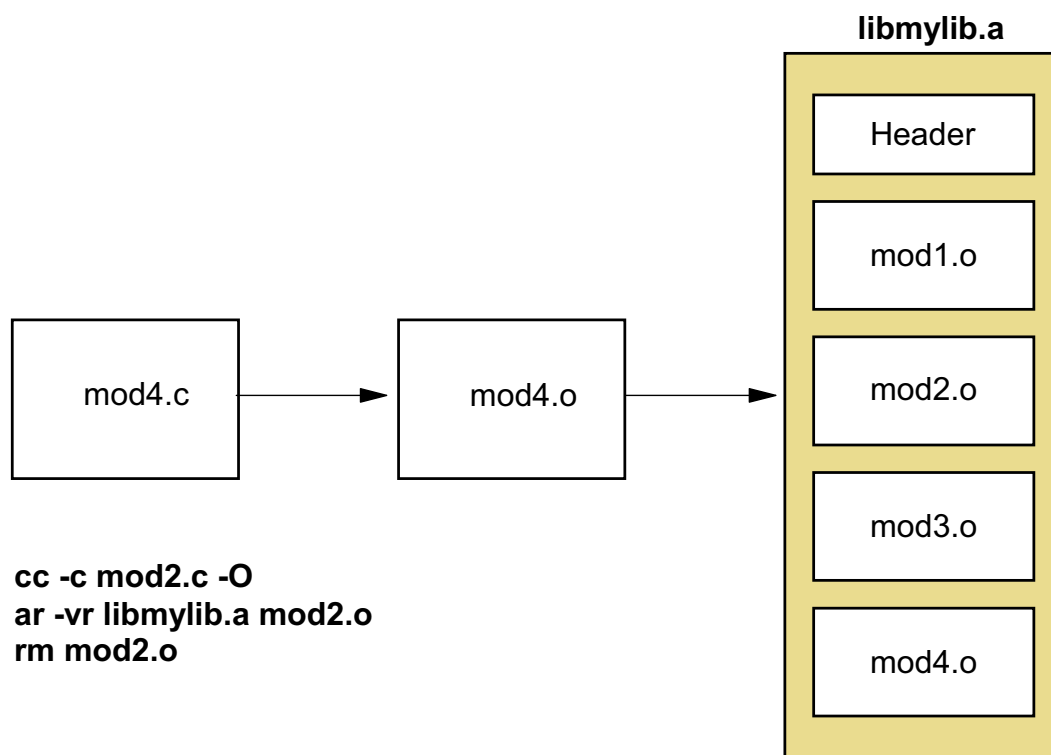


Figure 5-29. Replacing an Object in a Library

AU253.0

Notes:

The **ar** command uses the **-r** flag to replace an existing object in a library. Due to the effects of dynamic binding, one cannot replace an object in a library while any application is using the library.

The **dump -av lib_name** command may be used to dump the header file from the library to check the replacement date.

Some Useful Commands

- The **genkld** command
 - Lists the shared objects that are loaded in the system shared library segment.
 - Can only be executed by the root user or a user in the system group.
- The **slibclean** command
 - Unloads all shared objects with a use count value of zero from the system shared library segment.
 - Can be used by the root user.
- The **dump** command
 - Used to examine the header information of executable files and shared objects.
 - Important options
 - -H option
 - -Tv option

Figure 5-30. Some Useful Commands

AU253.0

Notes:

The **genkld** command is used to list the shared objects that are loaded into the system's shared library segment. The output of the command can contain multiple duplicate entries and be quite lengthy, so it is best to filter the output using the **sort** command or by performing a **grep** for the shared object you are investigating.

The **slibclean** command can be used by the root user to unload all shared objects with a use count value of zero from the system shared library segment. This command is useful in an environment when shared libraries are under development. You can run the **slibclean** command followed by the **genkld** command to ensure that the shared objects are not loaded in the system shared library segment.

The **dump** command is used to examine the header information of executable files and shared objects. The main options that are useful when working with shared libraries are the -H option and the -Tv options.

The **dump -H** command is used to determine which shared objects an executable or shared object depends on for symbol resolution at run time.

For example, the shared library libf1.so used in this example is based on the example in Figure 5-22 on page 34:

```
$ dump -H libf1.so
```

```
libf1.so:
```

```
***Loader Section***
```

```
Loader Header Information
```

VERSION#	#SYMtableENT	#RELOCent	LENidSTR
0x00000001	0x00000002	0x00000005	0x00000015

#IMPfilID	OFFfidSTR	LENstrTBL	OFFstrTBL
0x00000002	0x0000008c	0x00000000	0x00000000

```
***Import File Strings***
```

INDEX	PATH	BASE	MEMBER
0	/usr/lib:/lib		
1	..		

The **dump -Tv** command is used to examine the symbol information of a shared object or executable. It lists information on the symbols the object is exporting. It also lists the symbols the object or executable will try and import at load time and, if known, the name of the shared object that contains those symbols. The main columns to examine in the output are headed IMEX, IMPid, and Name.

The IMEX column indicates if the symbol is being imported (IMP) or exported (EXP). The IMPid field contains information on the shared object that the symbol will be imported from. The Name field lists the name of the symbol.

For example, the shared library libf1.so used in this example is based on the example in Figure 5-22 on page 34:

```
$ dump -Tv libf1.so
```

```
libf1.so:
```

```
***Loader Section***
```

```
***Loader Symbol Table Information***
```

[Index]	Value	Scn	IMEX	Sclass	Type	IMPid	Name
[0]	0x00000024	.data	EXP	DS	SECdef	[noIMPid]	func
[1]	0x00000000	undef	IMP	DS	EXTref	..	printf

Unit Summary

- Static and Dynamic Binding Concepts
- Shared and Nonshared Library Objects
- Import and Export Files
- Exec-time and Load-time Dynamic Binding
- The dlopen() and dlclose() System Calls
- Run-Time Linking
- The ar command
- AIX Linkage Editor (ld) and Binder Options

Figure 5-31. Unit Summary

AU253.0

Notes:

Unit 6. System Call Introduction

What This Unit is About

This unit is an introduction to system calls. The system call is defined, as well as the concept of a mode switch. Return values and the `errno` global variable are introduced. An overview of major system call groupings and header files is provided.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Explain the purpose of system calls
- Invoke system calls and evaluate their return values

How You Will Check Your Progress

There is no exercise for this unit.

References

- AIX 5L online documentation

Unit Objectives

After completing this unit, you should be able to:

- Explain the purpose of system calls
- Invoke system calls and evaluate their return values

Figure 6-1. Unit Objectives

AU253.0

Notes:

6.1 System Calls

Overview

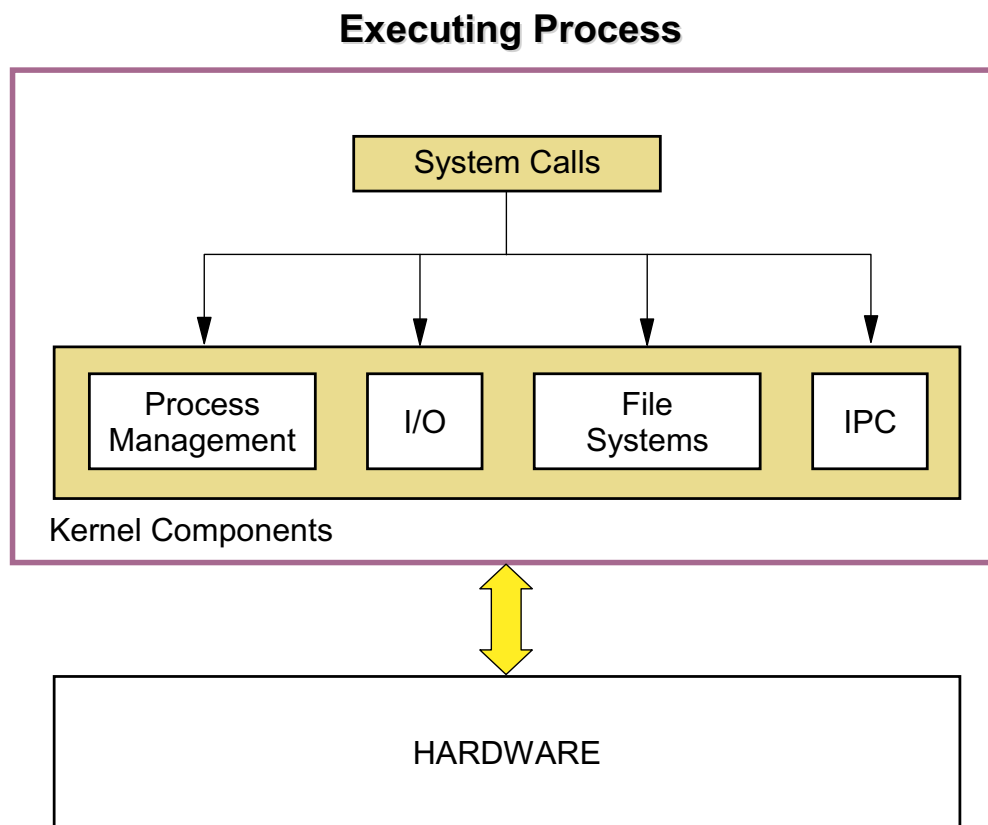


Figure 6-2. Overview

AU253.0

Notes:

This unit introduces the concept of system calls. UNIX system calls provide a user program with the ability to access structures within the UNIX kernel. (The kernel is generally protected from the user programs.)

System calls are actually portions of kernel code that run on behalf of the calling user program. System calls manipulate kernel structures and perform actions that a user program could not. For this reason, some system calls can only be successfully run by programs with a root effective user ID.

AIX supports most general UNIX system calls. In fact, there is some duplication of functionality in AIX system calls because System V, Berkeley (BSD), and POSIX system calls are included. Notes about differences in calls are given here.

Because system calls provide an application with an interface to the kernel, system calls can be grouped by the kernel services they provide. Specifically, system calls exist for process management tasks, I/O procedures, file system utilization, and interprocess communications. There are additional calls for timer facilities, resource reporting, and memory allocation.

System Call Overview

- System calls provide user processes access to privileged kernel routines.
 - User processes cannot normally access protected system resources.
 - Kernel routines can access such system resources.
- System calls switch from user mode of execution to kernel (system) mode.
 - The kernel executes system call code on behalf of the user process.
 - The return value is passed back to the user process.
 - Upon failure, the global integer variable `errno` is set to a value that can be referenced via the `/usr/include/sys/errno.h` header file.
- AIX allows the creation and loading of new system calls.

Figure 6-3. System Call Overview

AU253.0

Notes:

When a user application makes a system call, a hardware trap occurs. This trap causes a switch from user mode to kernel (system) mode. (You may have noticed that, when running UNIX performance monitoring tools such as **sar** or **iostat**, that information about CPU usage and run time is divided into user time and system time. These values relate to time spent running user code versus time spent running kernel code, or system calls.)

System calls can manipulate kernel data structures, such as the process table or file table. System calls provide a well known interface to the kernel.

The general format of a system call is:

```
return_value = syscall(paramlist)
```

Most system calls take a list of parameters. System calls return a value to the calling process. Usually, the return value indicates success or failure. Upon failure, the global variable `errno` is set to a value specific to the reason for the failure. The calling application can then examine the `errno` value to react to the failure.

AIX allows a user with root authority to add new system calls to the kernel. This can be done dynamically (without the need to rebuild and restart the kernel).

Executing System Calls

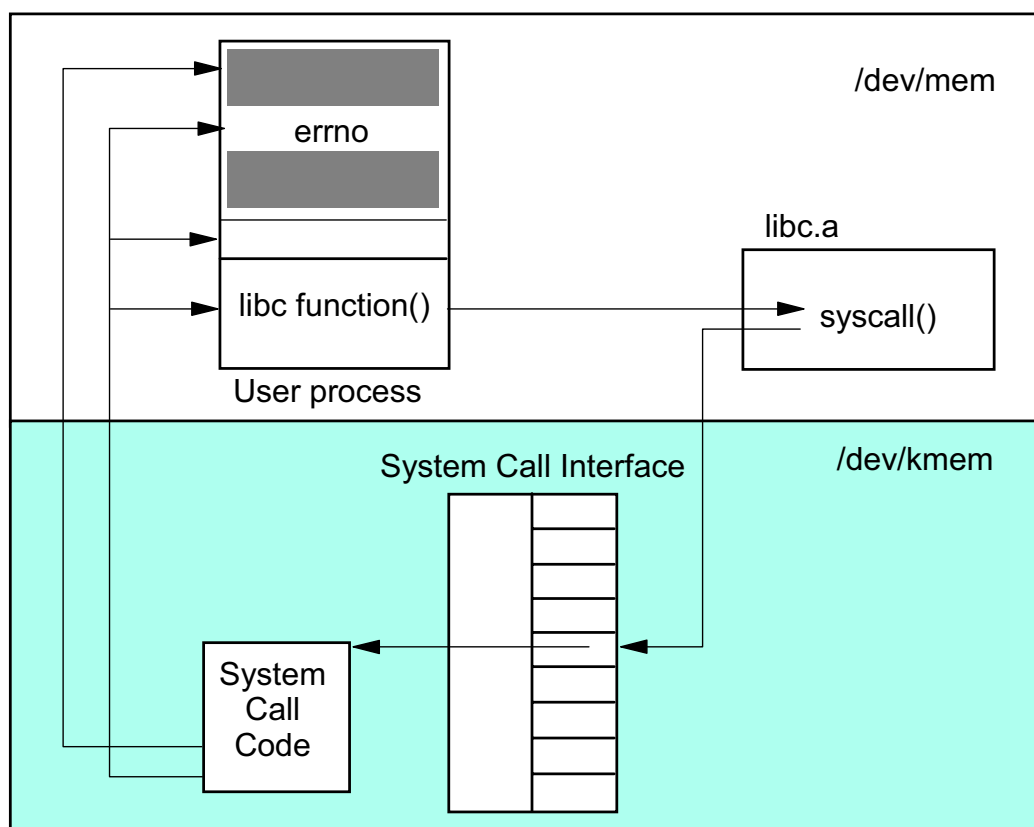


Figure 6-4. Executing System Calls

AU253.0

Notes:

Figure 6-4 illustrates what happens when a process makes a system call. All AIX system calls have a "front end" routine in the `/lib/libc.a` library. These routines are responsible for calling the actual system call and causing the mode switch.

System calls can call other system calls. They also make calls to kernel services (code in the kernel that is only available to system calls and other kernel extensions).

Unlike some UNIX systems, AIX system calls can be preempted. On traditional UNIX systems, a process running a system call is allowed to finish its system call before being preempted by a process with a more favored priority. To support real time programming, AIX system calls can be preempted. This is what is meant by a preemptable kernel. Also, portions of AIX system calls can be paged out. For these reasons, special care must be taken when creating your own system calls. Kernel structures should be locked to avoid having multiple processes updating these structures simultaneously. Creating system calls is beyond the scope of this lecture.

Because system calls may call other system calls or kernel services, and since a thread running a system call may be preempted, a kernel stack is provided on a per thread basis.

System calls return values to the caller or place error information in the calling process' user area. The variable holding this value is called the `errno`.

/usr/include/sys/errno.h

```

...
extern int errno;
...
#define EPERM    1      /* Operation not permitted      */
#define ENOENT   2      /* No such file or directory    */
#define ESRCH    3      /* No such process              */
#define EINTR    4      /* interrupted system call     */
#define EIO      5      /* I/O error                    */
#define ENXIO    6      /* No such device or address    */
#define E2BIG    7      /* Arg list too long            */
#define ENOEXEC  8      /* Exec format error            */
#define EBADF    9      /* Bad file descriptor          */
#define ECHILD   10     /* No child processes           */
#define EAGAIN   11     /* Resource temporarily unavailable */
#define ENOMEM   12     /* Not enough space             */
#define EACCES   13     /* Permission denied            */
#define EFAULT   14     /* Bad address                   */
#define ENOTBLK  15     /* Block device required        */
...

```

Figure 6-5. /usr/include/sys/errno.h

AU253.0

Notes:

Figure 6-5 shows a subset of the `errno.h` header file. You can see that it defines the `errno` variable. It also shows us many of the errors that our program can receive when we get a -1 return code.

We have a few choices on how we wish to handle these errors:

- Ignore errors (usually not a good idea)
- Treat any error as fatal and terminate the action or the program
- Inspect `errno` and take appropriate corrective action, depending on the error

Refer to each call in the AIX online documentation for the errors you might have. The complete set of C library errors is found in this header file.

Major System Call Groups

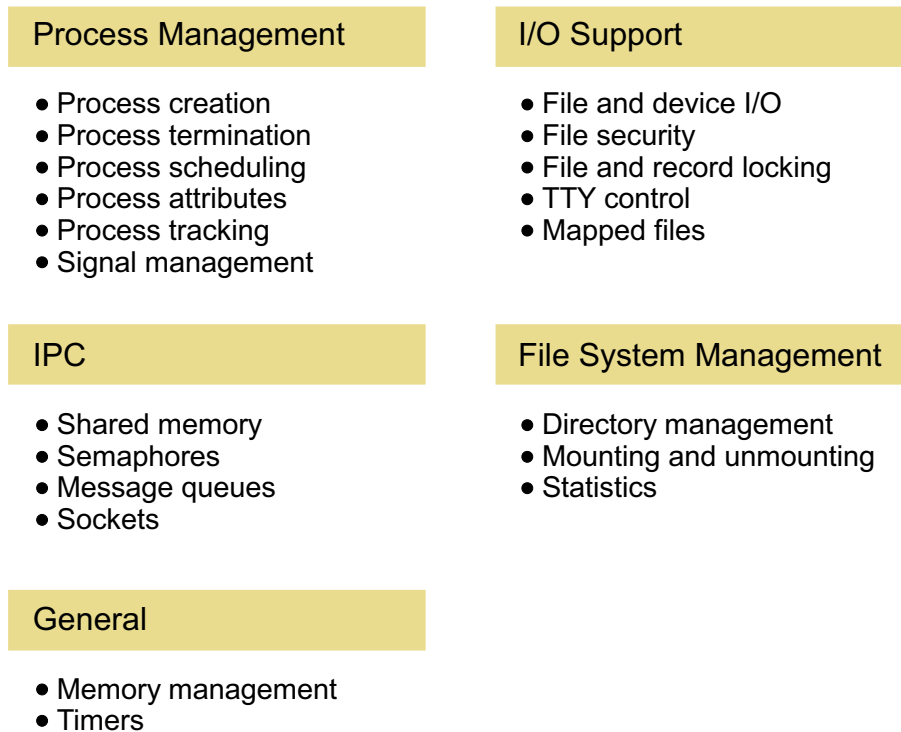


Figure 6-6. Major System Call Groups

AU253.0

Notes:

Figure 6-6 lists some of the tasks performed by AIX system calls.

Standard Headers Used with System Calls

- ANSI C

assert.h	locale.h	stddef.h
ctype.h	math.h	stdio.h
errno.h	setjmp.h	stdlib.h
float.h	signal.h	string.h
limits.h	stdarg.h	time.h

- POSIX

fcntl.h	sys/stat.h	termios.h
pwd.h	sys/types.h	unistd.h
sys/dir.h	sys/utsname.h	utime.h
	sys/wait.h	

Figure 6-7. Standard Headers Used with System Calls

AU253.0

Notes:

Standard headers

A header file contains the prototypes and types that are required to support one (or more) of the standard library subsystems supplied with a compiler. Rather than using one large header file, the standard defines 15 standard header files, each associated with a particular library subsystem. The reason for these separate files is easy to understand: header files must be compiled, and compilation takes time. Thus, by using separate files, your program need only include (and thus compile) the header information related to the functions that you actually use, instead of having to compile information about all functions in the standard library.

To include the POSIX header files, the preprocessor macro `POSIX_SOURCE` needs to be defined. Hence, when `POSIX_SOURCE` is defined, additional headers are used.

6.2 General System Calls

Memory Management Subroutines

`malloc (size)`

- Allocates memory

`calloc (numberofelements, elementsize)`

- Allocates space for an array of elements
- Initialize elements to zero

`realloc (pointer, size)`

- Changes the size of the block of memory pointed by the 'pointer' parameter

`free (pointer)`

- Frees a block of memory pointed by the 'pointer' parameter

Figure 6-8. Memory Management Subroutines

AU253.0

Notes:

The memory management routines are a part of the standard C library.

malloc() subroutine

`void *malloc (size_t Size)`

The malloc subroutine returns a pointer to a block of memory of at least the number of bytes specified by the Size parameter. On failure, this call returns a NULL.

free() subroutine

`void free (void * Pointer)`

The free subroutine frees a block of memory previously allocated by the malloc subroutine. Undefined results occur if the Pointer parameter is not a valid pointer. If the Pointer parameter is a null value, no action will occur.

calloc() subroutine

`void *calloc (size_t NumberOfElements, size_t ElementSize)`

The `calloc` subroutine allocates space for an array with the number of elements specified by the `NumberOfElements` parameter. Each allocated element of this array is initialized to zero.

realloc() subroutine

```
void *realloc (void * Pointer, size_t Size)
```

The `realloc` subroutine changes the size of the block of memory pointed to by the `Pointer` parameter to the number of bytes specified by the `Size` parameter and returns a new pointer to the block. The contents of the block returned by the `realloc` subroutine remain unchanged up to the lesser of the old and new sizes.

The pointer specified by the `Pointer` parameter must have been created with the `malloc`, `calloc`, or `realloc` subroutines and not been deallocated with the `free` or `realloc` subroutines. Undefined results occur if the `Pointer` parameter is not a valid pointer.

You need to include `stdlib.h` to use the memory management calls.

Example

A simple program that allocates a piece of memory to hold 100 characters:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    char *ptr;
    ptr = (char*) malloc(sizeof(char) * 100);
    if (ptr == NULL) {
        printf ("ERROR: malloc failed\n");
        return -1;
    }
    /* We have a char array to hold 100 elements */
    printf ("We have memory to hold 100 characters now\n");
    free(ptr);
}
```

Time Management Subroutines

`time(time)`

- Returns the time in seconds since 00:00:00 GMT, January 1, 1970.
- Includes sys/types.h.

`gettimeofday(time, timezone)`

`settimeofday(time, timezone)`

- Gets/sets current Greenwich time and the current time zone.
- The resolution of the system clock is hardware-dependent.
- Includes sys/time.h.

Figure 6-9. Time Management Subroutines

AU253.0

Notes:

time() subroutine

`time_t time(time_t *Tp)`

The time subroutine returns the time in seconds since the Epoch (that is, 00:00:00 GMT, January 1, 1970). The Tp parameter points to an area where the return value is also stored. If the Tp parameter is a null pointer, no value is stored.

gettimeofday() subroutine

`int gettimeofday (struct timeval *Tp, void *Tzp)`

`int settimeofday (struct timeval *Tp, struct timezone *Tzp)`

Current Greenwich time and the current time zone are displayed with the gettimeofday subroutine, and set with the settimeofday subroutine. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware-dependent.

The `timezone` structure indicates both the local time zone (measured in minutes of time westward from Greenwich) and a flag that, if nonzero, indicates that Daylight Savings Time applies locally during the appropriate part of the year. If this parameter has a value of 0, the time zone information is not returned or set.

Example:

A simple program to print the current date and time:

```
#include <stdio.h>
#include <sys/time.h>

main()
{
    time_t Tp;
    time (&Tp);
    printf ("Date and Time now is: %s\n", ctime(&Tp));
}
```

The **`ctime()`** routine is used to convert internal time representation (`time_t` data type) into a user readable string. More such routines are available to convert date and time to string representations. Refer to the AIX online documentation for more details.

The **`sleep()`** subroutine is an often used time management routine. This subroutine suspends a current process from execution. It takes the number of seconds to suspend execution as the argument.

```
unsigned int sleep ( Seconds );
```

A related call is the **`alarm()`** subroutine. The `alarm()` subroutine causes the system to send the calling thread's process a `SIGALRM` signal after the number of real-time seconds specified by the `Seconds` parameter have elapsed. Note that, since the signal is sent to the process, in a multi-threaded process another thread than the one that called the `alarm` subroutine may receive the `SIGALRM` signal.

```
unsigned int sleep ( Seconds );
```

Refer to the AIX online documentation for more details on these calls.

Unit Summary

- Purpose of system calls
- Invoking system calls
- Dealing with errors
- Header files
- Memory and time-related system calls

Figure 6-10. Unit Summary

AU253.0

Notes:

Unit 7. AIX File and I/O System Calls

What This Unit is About

This unit describes the functions and syntax of various system calls used to perform file and device I/O. The unit starts with a brief review of UNIX file systems and specific differences of the AIX file system.

Next, the basic system calls for the I/O and manipulation (`open()`, `close()`, `read()`, `write()`, and so on) are discussed. File locking techniques are introduced, as well as the memory mapping of files.

System calls used to manipulate file systems (`mount()`, `umount()`, `statfs()`, and so on) are also discussed.

The lab provides hands-on programming of practical file and file system programming.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Explain the functions of various file and I/O-related system calls
- Create simple C programs that manipulate data files
- Use file and record locking techniques to assure proper serialization of file access
- Use system calls to query and manipulate file and file system attributes
- Explain the benefits of explicit file mapping
- Use the `shmat()` system call to attach memory mapped files
- Know how to use files larger than 2 GB

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 5

References

- AIX 5L online documentation

Unit Objectives

After completing this unit, you should be able to:

- Explain the functions of various file and I/O-related system calls
- Create simple C programs that manipulate data files
- Use file and record locking techniques to assure proper serialization of file access
- Use system calls to query and manipulate file and file system attributes
- Explain the benefits of explicit file mapping
- Use the `shmat()` system call to attach memory mapped files
- Know how to use files larger than 2 GB

Figure 7-1. Unit Objectives

AU253.0

Notes:

7.1 The AIX File System

Overview

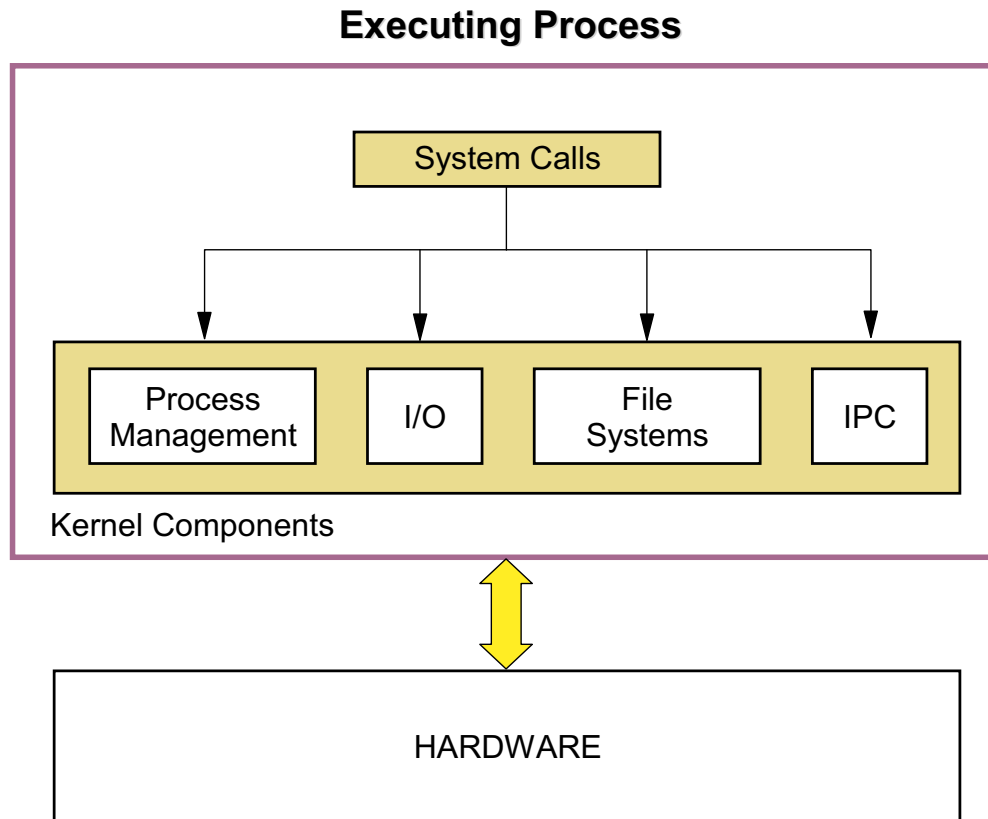


Figure 7-2. Overview

AU253.0

Notes:

This unit deals with system calls used to perform file and device I/O, and to manipulate file systems.

File System Overview (1 of 2)

- AIX files contain data and metadata:
 - The data is reason for the file (specific to file type).
 - Metadata is cntrl data for the file (stored in file's inode).
- AIX file types:
 - Ordinary files:
 - "Raw" string of bytes
 - No inherent format or EOF character
 - Directories:
 - Like ordinary file, but data formatted into file names and inode numbers
 - Special files:
 - File abstraction of physical and logical devices
 - Others
 - Symbolic links
 - FIFOs
 - Sockets

Figure 7-3. File System Overview (1 of 2)

AU253.0

Notes:

Let us start with some basic concepts about UNIX files.

Data or program text of UNIX ordinary files is simply stored as a string of bytes. Unlike other operating systems, UNIX does not use the concept of file records and record types. The controlling information about each file is stored in each file's inode.

Here are some UNIX file types:

Ordinary Files	Contain ASCII text, data, or executable program code. Ordinary files have no special structure, as far as UNIX is concerned.
Directories	Special files that store file names and inode numbers. UNIX directories do have a format.
Special Device Files	File abstractions of physical or logical devices. They are noted as block or character type and have major and minor numbers.
Symbolic Links	Files that "point" to other file names. A symbolic link file contains the full path name of the file that the link represents.

FIFO Files

First In, First Out Files. Also called Named Pipes, this type of file is used to share data between two or more processes. Its use is described in the IPC unit later in this course.

Unix Domain Sockets

A type of socket used to communicate with certain kinds of servers running on the local machine. These are briefly illustrated later in this course and much more information about them can be found in the AIX online documentation.

File System Overview (2 of 2)

- AIX files accessed through a "global" file system:
 - Hierarchical design.
 - Access controlled through permission settings.
- AIX file systems:
 - Logical partitionings of the "global" file system.
 - Each file system has its own root directory and set of inodes.
 - File system's mount-to-mount points within the "global" file system.
 - AIX file systems are journaled (JFS and enhanced JFS).
 - Remote mounting of file systems accomplished via Network File System (NFS).

Figure 7-4. File System Overview (2 of 2)

AU253.0

Notes:

There are two meanings to the term "file system" in AIX. The global file system is the entire hierarchical tree structure of directories and files. But physical file systems are made up of disk partitions, with their own root directory and set of inodes. Each physical file system is mounted on a directory within the global file system.

Basic File Manipulation Calls

- `creat()` Creates a new file
- `open()` Opens a file for reading or writing
- `read()` Retrieves data from a file
- `write()` Stores data to a file
- `lseek()` Positions read/write offset within a file
- `close()` Closes a file

Figure 7-5. Basic File Manipulation Calls

AU253.0

Notes:

Here are the basic system calls every program should use to manipulate and control file I/O.

Each of these system calls is described over the next few pages.

The open and creat Calls

```
int open(const char* path, int flag [,mode_t mode])
```

- Opens a file (can create a file if it does not already exist).
- Path is the absolute or relative path name of the desired file.
- Mode specifies the permission given to a new file created.
- Returns the assigned file descriptor number (integer).
- Returns -1 on failure.
- Include <fcntl.h>.
- Use open64() and creat64() calls for large files.

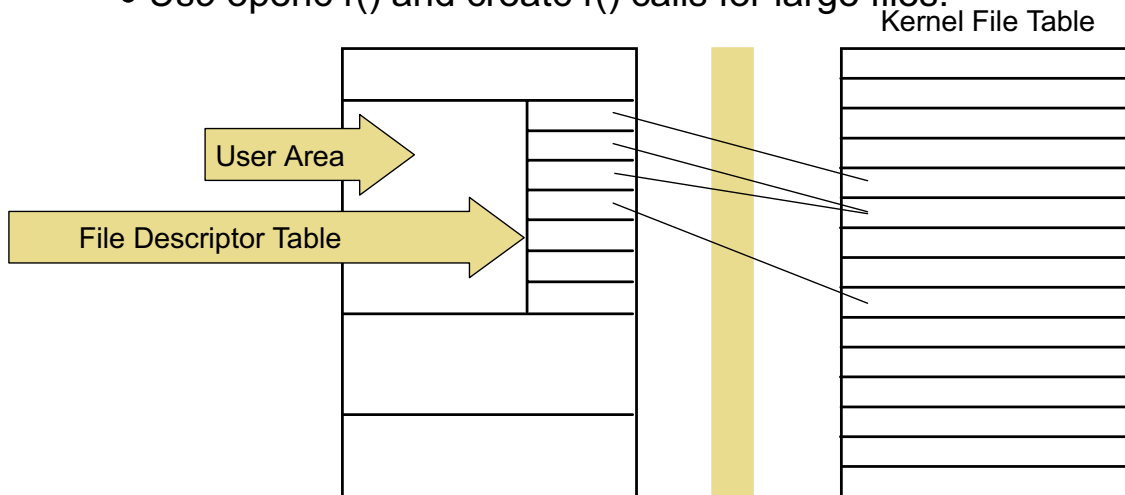


Figure 7-6. The open and creat Calls

AU253.0

Notes:

The open() system call opens a file for reading and/or writing.

Each process has a table that keeps track of the files that the process has open. This is called the file descriptor table and is part of the process' user area (see /usr/include/sys/user.h). An AIX process' file descriptor table has 32767 slots (0 - 32766).

The open() system call grabs the lowest available file descriptor number in the table and stores a pointer in that file descriptor entry. The pointer points to an entry in the kernel's file table. There is usually one entry in the file table for every instance of an open file. Note that if three different processes open the same file, there will be three different file table entries assigned. This is because the file table entry includes the read/write offset into the file.

The open() system call returns the file descriptor number to the calling process. This number is used as a tag to associate the file with future reads, writes and other operations.

The O_CREAT flag creates the file, if it does not already exist. Other flags include O_RDWR, O_RDONLY, and O_WRONLY. Multiple flags can be ORed by using the pipe (|) symbol. The header file fcntl.h is required to define these symbolic constants.

Mode can be set only when creating a new file.

For example:

```
int fd, fdnew;
```

```
fd=open("myfile", O_RDWR); /* Open for update */
```

```
fdnew=open("newfile", O_WRONLY|O_CREAT,0660);/* Create new */
```

The `creat()` library routine can be used to create files without having to deal with the various open flags. `creat()` takes two parameters. The first parameter is the path name of the file to create (if it does not already exist) and to empty (if it does exist). The second parameter is the permission settings that should be used if a new file is created. `creat()` returns a write-only file descriptor associated with the file or -1 on failure.

For example, the call:

```
fd = creat("myfile",0644);
```

is equivalent to the call

```
fd = open("myfile",O_CREAT|O_WRONLY|O_TRUNC,0644);
```

The `open64()` and `creat64()` subroutines are equivalent to the `open()` and `creat()` subroutines, except that the `O_LARGEFILE` flag is set in the open file description associated with the returned file descriptor. If the file needs to grow very large and eventually become a large file, it needs to be opened with the `open64()` call instead of the `open()` call. Details about large files are discussed later in this unit.

Alternatively, the Standard C Library routine **`fopen()`** may be used to create/open a file. Refer to AIX online documentation for more details.

The read and write System Calls

```
ssize_t read(int fdes, void* buffer, size_t numbytes)
```

- Reads numbytes of data from file associated with file descriptor *fdes* into memory pointed to by *buffer*
- Returns number of bytes actually read (integer) or zero if end-of-file encountered
- Returns -1 on failure
- Read/write offset moved to end of bytes read
- Include <unistd.h>

```
ssize_t write(int fdes, void* buffer, size_t numbytes )
```

- Writes numbytes of data from memory pointed to by *buffer* to the file associated with *fdes*
- Returns number of bytes (integer) actually written if successful
- Returns -1 on failure
- Read/write offset moved to end of bytes written
- Include <unistd.h>

Figure 7-7. The read and write System Calls

AU253.0

Notes:

The read() and write() calls move data in and out of the file.

Note the use of the file descriptor.

Reading and writing x number of bytes increments the read/write offset (stored in the file table entry for the specific file) by x bytes. If an error occurs with read() or write(), the errno variable is set to indicate the error.

Some simple examples follow. Note that there are numerous techniques used to both read and write.

```
int count,rc;
char buffer[80];
```

```
count=read(fd, buffer, sizeof(buffer));
rc =write(fdnew, buffer, count);
```

The Standard C Library routines **fread()** and **fwrite()** may be used to read from and write to a file that is opened using the `fopen()` call. Refer to AIX online documentation for more details.

The lseek System Call

```
off_t lseek(int fdes, off_t offset, int whence)
```

- Sets the r/w offset for the open file associated with *fdes*.
- Offset (integer) specifies the number of bytes to seek in conjunction with the *whence* parameter. Offset can be a negative integer.
- Whence sets the relativity of the seek:
 - SEEK_SET - Seek from start of file.
 - SEEK_CUR - Seek from current offset position.
 - SEEK_END - Seek from end of file.
- Returns the offset value from start of file upon success.
- Returns -1 upon failure.
- Include <unistd.h> for whence definitions.
- Include <sys/types.h> to typedef offset and return values as off_t (integer).
- Use lseek64 for large files.

Figure 7-8. The lseek System Call

AU253.0

Notes:

Here is how the lseek() system call works.

Each file descriptor has a current file position associated with it. When a read() or a write() is performed, the current position is incremented by the number of bytes read or written.

Normally, I/O is done sequentially on a file, but it is possible to change the current position to skip bytes forward or backward by use of the lseek() call. lseek() is equivalent to fseek() in the standard C library, except lseek() takes a file descriptor as its first argument instead of a file pointer.

If using large files, use the lseek64() call instead of the lseek() call. Large files are discussed at the end of this unit.

The close System Call

```
int close(int fdes)
```

- Closes the file associated with *fdes*
- Returns zero upon success
- Returns -1 upon failure
- Includes <unistd.h>

Figure 7-9. The close System Call

AU253.0

Notes:

The close() system call simply releases the reference that the process' file descriptor table has on the file and deallocates the slot in the kernel's file table.

Only one file descriptor can be closed with each invocation of the close() system call.

All remaining open files are implicitly closed when a process terminates.

The Standard C Library routine **fclose()** may be used to close a file that is opened by the **fopen()** call. Refer to AIX online documentation for more details.

File I/O Example

```

/* fileio.c */
#include <stdio.h> /*standard IO */
#include <fcntl.h> /*file attributes */
#include <sys/stat.h> /*file status values */
#include <sys/types.h> /*common typedefs */
#include <unistd.h> /*lseek constants */
main()
{
    int fdr, fdw;
    short i;
    char line[31];
    /* open file "mylist" in read mode */
    fdr=open("mylist",O_RDONLY);
    /* create new file "newlist" with read/write access */
    fdw=open("newlist",O_CREAT|O_WRONLY,0666);
    /* skip first five entries in "mylist" */
    lseek(fdr,(5*sizeof(line)),SEEK_SET);
    /* read the next five entries in "mylist" and
       write them to "newlist" */
    for(i=0;i<5;i++)
    {
        read(fdr,line,sizeof(line));
        write(fdw,line,sizeof(line));
    }
    /* close both files */
    close(fdr);
    close(fdw);
}

```

Figure 7-10. File I/O Example

AU253.0

Notes:

Here is a sample program that opens an existing file, creates a new file, reads records in from the first file, and writes the records out to the new file.

The data stored in the mylist file has a record structure only because the application defines it as such. To AIX, mylist is just a string of bytes.

The sizeof() routine in the read() and write() calls is an easy way of specifying the number of bytes to read and write.

The lseek() constants are found in unistd.h.

File I/O Example Output

mylist

Atlanta	GA	Peterson, Andrew	56
Baltimore	MD	Lewis, Marcia	21
Boston	MA	Williams, Glenn	89
Chicago	IL	Harris, Tom	32
Cleveland	OH	Bell, Lisa	74
Dallas	TX	Carlton, Len	90
Denver	CO	Dickson, Mike	88
Detroit	MI	Smith, Al	16
Fort Worth	TX	Snyder, Connie	76
Harrisburg	PA	Frost, Joanne	57
Los Angeles	CA	Geller, Nancy	60
Miami	FL	Podlin, James	06
...			

newlist

Dallas	TX	Carlton, Len	90
Denver	CO	Dickson, Mike	88
Detroit	MI	Smith, Al	16
Fort Worth	TX	Snyder, Connie	76
Harrisburg	PA	Frost, Joanne	57

Figure 7-11. File I/O Example Output

AU253.0

Notes:

Figure 7-11 illustrates the results of the program.

Writing to Disk Files (1 of 2)

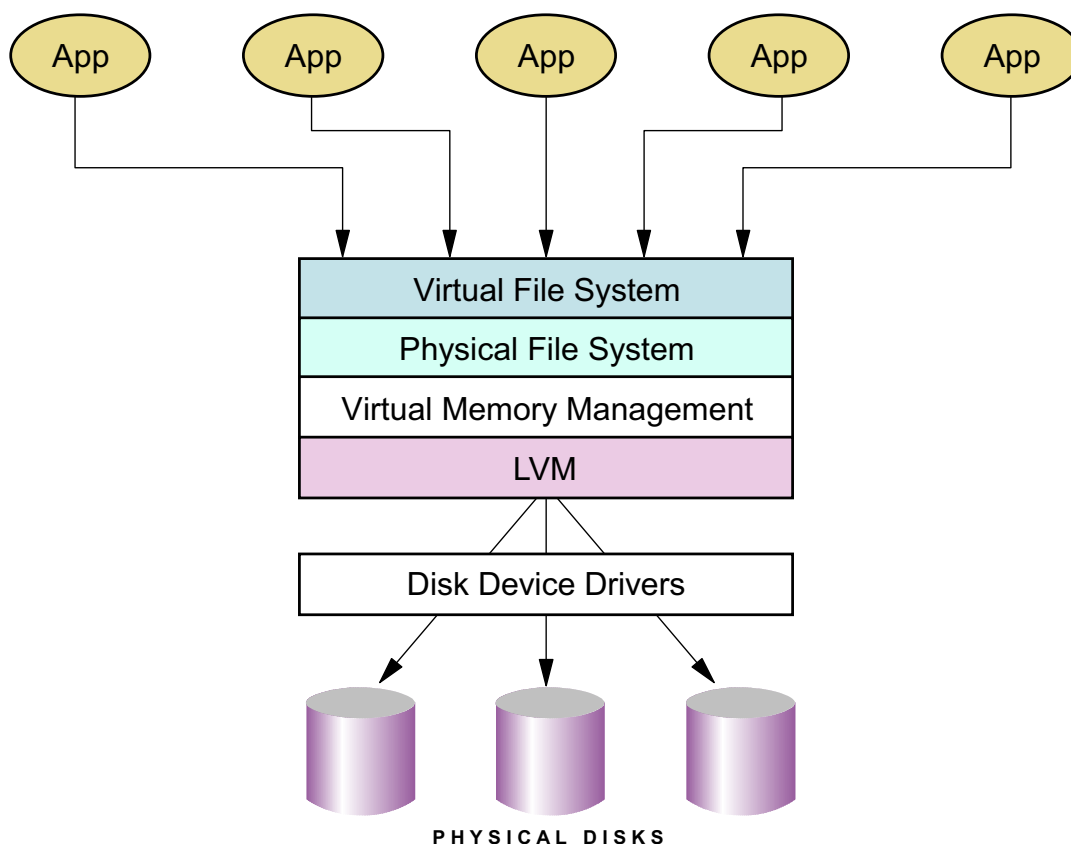


Figure 7-12. Writing to Disk Files (1 of 2)

AU253.0

Notes:

Figure 7-12 illustrates how AIX uses a layered approach to file I/O.

The virtual file system layer is concerned with the type of file system on which the file resides. AIX supports four default file system types: JFS (local disk file system), Enhanced JFS, NFS, and CD-ROM. This layer means that your application does not need to know what type of file systems it is accessing.

The enhanced JFS can hold a file size of up to 1Petabyte, or 1,048,576 GB, while the JFS can only hold a file size of up to 64 GB. Large file support should be explicitly enabled in JFS at the time of file system creation. There is no explicit enabling or disabling of large file support with respect to enhanced JFS. Details about large file support is discussed at the end of this unit. For the general file operations, these two file systems give a consistent view to the programmer.

The physical file systems layer includes directories, inodes, and logical data blocks.

The virtual memory manager is concerned with whether the desired page is already in memory.

The LVM layer is used to keep track of the actual physical locations of files on disk. It also performs mirroring and bad block relocation. Details on the VMM and LVM layers are available in other ILS courses.

Writing to Disk Files (2 of 2)

```
void sync()
```

non-POSIX

- Writes (or schedules to be written) all modified data and metadata to disks
- No arguments
- No return value (void)
- Includes <unistd.h>

```
int fsync(int fdes)
```

non-POSIX

- Forces all modified data for the file associated with fdes to be written to disk
- Returns zero upon success, -1 upon failure
- No include files needed

Figure 7-13. Writing to Disk Files (2 of 2)

AU253.0

Notes:

As mentioned earlier, just because your application writes data to a file does not mean that data has been written to disk. There are ways to make sure the data is written to disk; specifically, the sync() and fsync() system calls.

You may already be familiar with the **sync** command. The sync() system call does the same thing from within a program.

One problem with the sync() system call is that all dirty pages are written to disk. This is not always good for overall system performance. A better way for a programmer to assure that data written to a file gets written to disk is to use the fsync() system call, which only writes the dirty pages associated with a particular file descriptor to disk.

7.2 File Locking

File Locking

- System call that can lock and unlock portions of files:

- lockfx()
- lockf()
- flock()
- fcntl()

- These are integrated

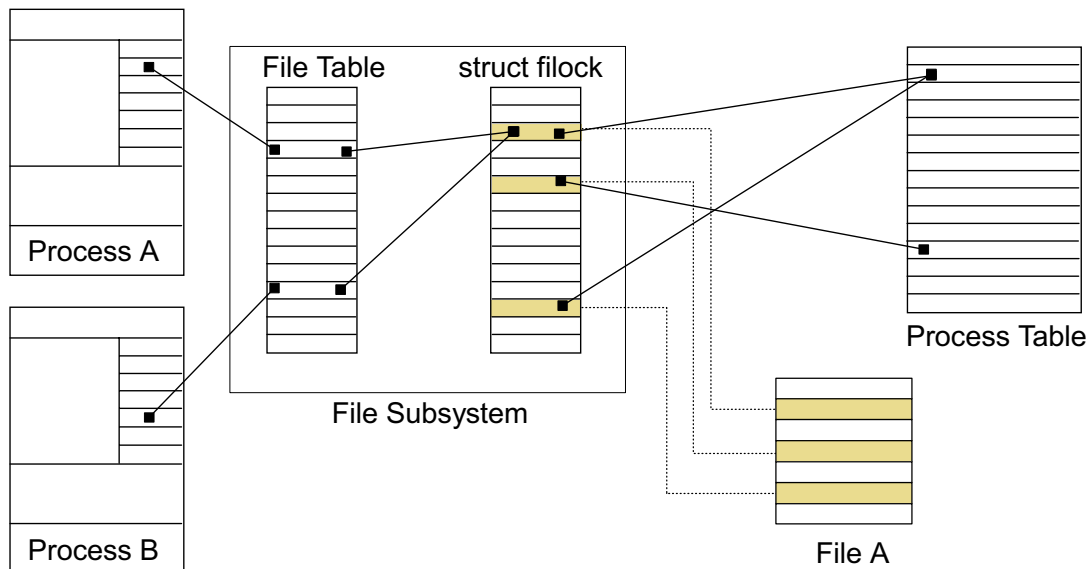


Figure 7-14. File Locking

AU253.0

Notes:

AIX allows more than one process to modify the same file simultaneously. It does, however, provide a facility for advisory locking of files.

Figure 7-14 illustrates how processes, using file descriptors to point to the kernel's file table, can issue locks on regions of a file (byte to byte). Many processes can have many locks on the same file, as long as none of the locked regions overlap. Because these are advisory locks, processes are supposed to cooperate by locking a region before they access or modify it.

A structure called the lock table (struct flock) manages the locks. The flock structure is defined in `/usr/include/sys/flock.h`.

Figure 7-14 also lists four system calls that can be used by processes to test, set, and unset locks. These calls come from different UNIX sources and there is some overlapping of functionality, but all four calls use the same underlying kernel structure and are, therefore, integrated.

The most important calls are detailed on the next few pages.

The lockf Call

```
int lockf(int fdes, int request, off_t size)
```

non-POSIX

- Locks a region of the file associated with fdes.
- Request is one of the following:
 - F_LOCK - Locks the region for exclusive use. Sleeps until lock is available.
 - F_ULOCK - Unlocks a previously locked region.
 - F_TEST - Tests to see if another process owns a lock on the specified region. Returns zero if not locked, -1 if locked.
- size indicates the number of bytes to lock. The region starts at the current read/write offset position. A negative number can be used to lock backwards. A zero locks the remaining bytes in the file.
- Returns zero upon success, -1 upon failure.
- Includes <unistd.h> and <sys/lockf.h>.
- Use lockf64 for large files.

Figure 7-15. The lockf Call

AU253.0

Notes:

An additional request, F_TLOCK, locks the region for exclusive use if another process has not already locked the region. If the region has already been locked by another process, a -1 is returned.

File Locking Example (non-POSIX)

```
/* lockf.c */
#include <sys/lockf.h>
#include <fcntl.h> /* file attributes */
#include <unistd.h> /* lseek constants */
#define RECSIZE 31 /* line length plus <CR> */
main()
{
    int fd;
    fd=open("mylist",O_RDWR);
    /* lock first two records (lines) */
    lockf(fd,F_LOCK,(2 * RECSIZE));
    printf("Parent has locked first two records.\n");
    /* create a child process */
    if(fork()==0) /* we are the child */
    {
        /* lock record 5 */
        lseek(fd,(4 * RECSIZE),SEEK_SET);
        lockf(fd,F_LOCK,RECSIZE);
        printf("Child has locked fifth record.\n");
        /* try locking record 2 */
        lseek(fd,RECSIZE,SEEK_SET);
        printf("Child trying to lock second record.\n");
        lockf(fd,F_LOCK,RECSIZE);
        printf("Child has locked second record.\n");
        exit(0); /* implicitly releases locks */
    }
    sleep(10); /* parent sleeps holding a lock */
    printf("Parent is unlocking first two records.\n");
    lseek(fd,0,SEEK_SET);
    lockf(fd,F_ULOCK,(2 * RECSIZE));
}
```

Figure 7-16. File Locking Example (non-POSIX)

AU253.0

Notes:

Here's an example of file locking.

Note that file locks are not inherited by child processes when the parent process does a `fork()` system call.

Note the use of the `sleep(10)` call to help synchronize the parent and child processes.

The output from this program is displayed in Figure 7-17 on page 25.

File Locking Example Output

Program output:

```
Parent has locked first two records.  
Child has locked fifth record.  
Child trying to lock second record.  
Parent is unlocking first two records.  
Child has locked second record.
```

Figure 7-17. File Locking Example Output

AU253.0

Notes:

Here is the output from the program in the Figure 7-16.

The fcntl Call

```
int fcntl(int fdes, int command, int arg)
```

- Performs controlling operations on the file associated with *fdes* (here we discuss file locking).
- *arg* is a pointer to a flock structure.
- *command* is one of the following:
 - F_SETLK - Sets or clears a lock. Returns -1 if lock cannot be set
 - F_SETLKW - Sets or clears lock, but blocks (sleeps) until lock is available
 - F_GETLK - Fill flock structure with data that describes current lock set.
- Returns zero upon success, -1 upon failure.
- Include <fcntl.h> and <unistd.h>.

Figure 7-18. The fcntl Call

AU253.0

Notes:

Here is a multipurpose system call known as `fcntl()`. One of the many things this call will do is manage file locks.

The major difference between this call and the `lockf` call is that this one uses a structure called "flock" that contains fields useful for managing the locks. The flock structure is defined in the `/usr/include/sys/flock.h` header file. The flock structure is illustrated in Figure 7-19. This header file is included by the `fcntl.h` header.

If the lock is not available, the process can sleep on the lock (`F_SETLKW`) or return an error (`F_SETLK`).

Other functions that `fcntl()` is used for include:

- Duplicating open file descriptors
- Setting/getting file descriptor flags
- Managing record locks
- Closing multiple files

flock Structure

- The flock structure (flock.h):

```
struct flock {
    short l_type;
    short l_whence;
    off_t l_start;
    off_t l_len; /*len = 0 means until end of file */
    ...
};
```

- Defined values for l_type:

```
/* file segment locking types */
#define F_RDLCK 01 /* Read lock */
#define F_WRLCK 02 /* Write lock */
#define F_UNLCK 03 /* Remove lock(s) */
```

Figure 7-19. flock Structure

AU253.0

Notes:

The flock structure described here is used on the fcntl file locking call:

l_type	Describes the type of lock. Possible values include F_RDLCK(read), F_WRLCK(write), and F_UNLK (unlock).
l_whence	Defines the starting offset. The value of this field indicates the point from which the relative offset, l_start, is measured. Possible values are SEEK_SET (start of the file), SEEK_CUR (current position), and SEEK_END (end of the file).
l_start	Defines the relative offset in bytes, measured from the starting point, l_whence.
l_len	Specifies the number of consecutive bytes to lock.

File Locking Example (POSIX)

```
/* filelock.c */
#include <fcntl.h>
#include <sys/flock.h>
#include <unistd.h>
#include <errno.h>
main()
{
    int lock;
    int rc;
    struct flock myflock;
    char buffer[256];

    lock = open("mylockfile", O_CREAT|O_RDWR,0666);
    printf("lock is %d\n", lock);
    myflock.l_type = F_WRLCK; /* write lock on whole file */
    myflock.l_whence = SEEK_SET;
    myflock.l_start = 0;
    myflock.l_len = 0;

    if (fcntl(lock,F_SETLKW,&myflock) < 0) { /* locking */
        perror("problems with lockfile"); exit(1);
    }
    read(lock,buffer,sizeof(buffer));
    printf("Lockf: The buffer says - %s\n", buffer);
    sleep(5);
    myflock.l_type = F_UNLCK; /* unlocking */
    if (fcntl(lock,F_SETLKW,&myflock) < 0) {
        perror("problems with unlockfile"); exit(1);
    }
}
```

Figure 7-20. File Locking Example (POSIX)

AU253.0

Notes:

This example utilizes the `fcntl()` system call to do locking and unlocking.

This example creates/opens a file with name *lockfile*. This file is kept locked for a period of five seconds. This file is read while it is in the locked state.

Output:

```
$ ./filelock
```

```
lock is 3
```

```
Lockf: The buffer says - Hello
```

7.3 File Characteristics

Controlling File Characteristics

- Various system calls for controlling a file's characteristics are:
 - `access()` Determines accessibility of a file
 - `chmod()` Changes permissions of a file
 - `chown()` Changes owner of a file
 - `chdir()` Changes the current directory of a process
 - `utime()` Changes file access and modification times
 - `umask()` Changes the default file creation mask

Figure 7-21. Controlling File Characteristics

AU253.0

Notes:

There are many system calls available for various file-related tasks.

The next few pages introduce these calls.

The access Call

```
int access(char* path, int accessmode)
```

- Determines the accessibility of the file pointed to by path
- accessmode specifies the criteria of the query:
 - R_OK Checks read permission (04)
 - W_OK Checks write permission (02)
 - X_OK Checks execute (search) permission (01)
 - F_OK Checks to see if file exists (00)
- Returns zero if access query is true, -1 if access is denied
- Includes <sys/access.h> and <unistd.h>

Figure 7-22. The access Call

AU253.0

Notes:

The access() system call is used to determine what access permissions the calling process has on a particular file.

An example program follows shortly and demonstrates the use of this call.

File Access Example

```
/* chkfile.c */
#include <stdio.h>
#include <sys/access.h> /*file access constants */
main(argc,argv)
int argc;
char *argv[];
{
    if(argc<2) {
        printf("A file name must be specified.\n"); exit(1);
    }
    if(access(argv[1],F_OK)!=0)
        printf("The named file does not exist.\n");
    if(access(argv[1],R_OK)!=0)
        printf("You have read access to the named file.\n");
    if(access(argv[1],W_OK)!=0)
        printf("You have write access to the named file.\n");
    if(access(argv[1],X_OK)!=0)
        printf("You can execute the named file.\n");
}
```

Possible output:

```
$ chkfile /home/dave/mygame
You have read access to the named file.
You can execute the named file.
```

Figure 7-23. File Access Example

AU253.0

Notes:

The program in Figure 7-23 illustrates the use of the `access()` system call. The program expects a file name as an argument, then checks the calling process' access permission to the specified file.

The output of the program is illustrated in Figure 7-23 as well.

The chmod Call

```
int chmod(char* path, mode_t mode)
```

- Sets permissions of the file pointed to by path
- mode specifies the permissions and can be bitwise ORed
- Call will succeed only when made by a process with the same UID as file or process with root authority
- Returns zero upon success, -1 upon failure
- Includes <sys/stat.h>

Note: Use of this call will disable ACL entries.

Figure 7-24. The chmod Call

AU253.0

Notes:

Here is a system call that doubles as a shell level command.

The chmod() system call can be used within a program like the **chmod** command is used from the command line.

The modes are defined in /usr/include/sys/mode.h, which is included by stat.h. They include:

- S_ISUID - Sets user ID on execution
- S_ISGID - Sets group ID on execution
- S_IRUSR - Allows owner to read the file
- S_IWUSR - Allows owner to write the file
- S_IXUSR - Allows owner to execute the file
- S_ISVTX - On a directory, restricts unlink and rename to the owner of the object being unlinked or renamed)

Similar permissions exist to allow anyone to read, write, or execute by substituting 'OTH' for 'USR' in the examples macros. Group permissions also exist by substituting 'GRP' for 'USR'.

For example:

```
chmod ("foo", 0600); /* R/W to owner */
```

There is also an `fchmod()` call that is identical to `chmod()`, with the exception that `fchmod()` takes a file descriptor instead of a path. `fchmod()` is slightly faster than `chmod()` if you already have a file descriptor for the file. It can also be very useful when you just have a file descriptor and do not know the name of the file.

The chown Call

```
int chown(const char* path, uid_t owner, gid_t group)
```

- Changes the UID or GID value of the file pointed to by path.
- owner specifies the desired UID. If -1, the UID is not changed.
- group specifies the desired GID. If -1, the GID is not changed.
- Returns zero upon success, -1 upon failure.
- Includes <sys/types.h> for typedefs uid_t and gid_t.

Figure 7-25. The chown Call

AU253.0

Notes:

The chown() system call is similar to the **chown** command and **chgrp** command, combining the functionality of both commands.

Only root can change the ownership of a file. Only root or the owner of a file can change the group of the file (non-root users can only change the group attribute to a group which they are a member of).

For example:

```
chown ("/tmp/file1",200,100);
```

OR

```
chown ("foo",0,0);
```

There is also an fchown() system call, which takes a file descriptor instead of a path.

The chdir Call

`int chdir(const char* path)`

- Changes the current directory of the calling process to the directory pointed to by *path*
- The calling process must have search access to the *path* directory
- Returns zero upon success, -1 upon failure
- No include files required

Figure 7-26. The chdir Call

AU253.0

Notes:

The `chdir()` system call works just like the **cd** shell built-in command.

For example:

```
chdir ("/home/workshop");
```

Similar to `chdir` is `chroot`. `chroot()` is a non-POSIX system call. For example:

```
chroot( path );
```

- Changes the effective root directory of the calling process to the directory pointed to by *path*.
- The calling process must have root authority.
- Returns zero upon success, -1 upon failure.
- No include files required.

For example:

```
chroot ("/usr/newrootdir");
```

The utime Call

```
int utime(const char* path, const struct utimbuf* times)
```

- Changes the access and modification times for the file pointed to by path.
- The times parameter is a pointer to a utimbuf structure, as defined in the utime.h file.
- Returns zero upon success, -1 upon failure.
- Includes <utime.h>.

Figure 7-27. The utime Call

AU253.0

Notes:

The utime() system call changes the access and/or modification date stamps in the specified file's inode.

The path parameter specifies the file.

For the utime subroutine, the times parameter is a pointer to a utimbuf structure, as defined in the utime.h file. The first structure member represents the data and time of last access, and the second member represents the data and time of last modification. The times in the utimbuf structure are measured in seconds since 00:00:00 Greenwich Mean Time (GMT), 1 January 1970. If a null (zero) pointer is passed as the times value, then the access and modification times are set to the current time.

For example:

```
#include <stdio.h>

#include <utime.h>
```

```
#include <time.h>
main(int argc, char *argv[])
{
    system("ls -l");
    if (utime(argv[1], NULL) == -1) {
        printf("error in changing time\n");
        exit (-1);
    }
    printf("time now being changed\n");
    system("ls -l");
    exit (0);
}
```


The umask Call

```
mode_t umask(mode_t mask)
```

- Changes or queries the calling process' file creation mask to mask.
- mask specifies the desired mask value by logically ORing the permissions defined in the <sys/mode.h> header file.
- Returns the previous mask.
- Includes <sys/stat.h> to use symbolic permission names (it includes mode.h).
- Includes <types.h> to use typedef mode_t.
- How file creation masks work:

System file value	=	666
Starting umask	=	022
Default file value	=	644 (rw-r--r--)

- After umask(S_IROTH):

System file value	=	666
New umask	=	026
New default file value	=	640 (rw-r-----)

Figure 7-28. The umask Call

AU253.0

Notes:

The umask() system call performs like the **umask** command.

File-permission bits defined in sys/mode.h are:

S_IRWXU	Permits the owner of a file to read, write, and execute the file.
S_IRUSR	Permits the owner of a file to read the file.
S_IREAD	Permits the owner of a file to read the file.
S_IWUSR	Permits the owner of a file to write to the file.
S_IWRITE	Permits the owner of a file to write to the file.
S_IXUSR	Permits the owner of a file to execute the file or to search the file's directory.
S_IEXEC	Permits the owner of a file to execute the file or to search the file's directory.
S_IRWXG	Permits a file's group to read, write, and execute the file.
S_IRGRP	Permits a file's group to read the file.
S_IWGRP	Permits a file's group to write to the file.

S_IXGRP	Permits a file's group to execute the file or to search the file's directory.
S_IRWXO	Permits others to read, write, and execute the file.
S_IROTH	Permits others to read the file.
S_IWOTH	Permits others to write to the file.
S_IXOTH	Permits others to execute the file or to search the file's directory.

These are the same permission bits used by the `chmod()` and `stat()` calls.

File Characteristics Example

```
/* filetest.c */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
main()
{
    int fdr, fdw;
    char line[31];
    chdir("/home/dave/workshop");
    fdr=open("mylist",O_RDWR);
    umask(S_IROTH); /* sets new umask value */
    fdw=creat("newlist",0644);
    /* copy the third record from mylist to newlist */
    lseek(fdr,(2 * sizeof(line)),SEEK_SET);
    read(fdr,line,sizeof(line));
    write(fdw,line,sizeof(line));
    /* close all files */
    close(fdr);
    close(fdw);
    /* change permissions on "mylist" file */
    chmod("mylist",S_IRUSR| S_IWUSR| S_IRGRP| S_IWGRP);
}
```

Figure 7-29. File Characteristics Example

AU253.0

Notes:

Here is a sample program that demonstrates some other file manipulation system calls.

Note the use of the or symbol (|) in the chmod() system call. You may find it easier to use the octal notation.

File Characteristics Example Result

```
$ ls -l /home/dave/workshop
...
-rw-r--r-- 1 dave staff    310 Feb 21 17:08 mylist
$ filetest
$ ls -l /home/dave/workshop
...
-rw-rw---- 1 dave staff    620 Feb 21 17:08 mylist
-rw-r----- 1 dave staff     31 Feb 22 13:21 newlist
$
```

Figure 7-30. File Characteristics Example Result

AU253.0

Notes:

Here is the effect of the program in Figure 7-29.

7.4 More Files I/O Calls

Other File and Device I/O Calls

- `fcntl()` Controls open file descriptors
- `ioctl()` Controls open file descriptors for block and character special files and sockets (non-POSIX)
- `mknod()` Creates an ordinary file, directory, fifo, or special file (non-POSIX)
- `mkfifo()` Creates a fifo file (named pipe)
- `pipe()` Creates an unnamed pipe
- `tcgetattr()` Gets terminal attributes
- `tcsetattr()` Sets terminal attributes

Figure 7-31. Other File and Device I/O Calls

AU253.0

Notes:

There are also system calls that control device I/O.

Only the `ioctl()`, `tcgetattr()`, and `tcsetattr()` calls are illustrated in this unit. Pipes are covered in the IPC lecture. FIFO files are also called pipes. Pipes are created by one process to temporarily allow communication with another process. These files cease to exist when the first process finishes.

ioctl() subroutine

```
int ioctl (int FileDescriptor, int Command, void* Argument)
```

non-POSIX

- Performs control functions associated with open file descriptors.
- Typically used with character or block special files, sockets, or generic device support.
- The control operation is specific to the object being addressed.
- The data type and contents of the *Argument* parameter are specific to the object being addressed.
- Performing an ioctl function on a file descriptor associated with an ordinary file results in an error.
- Includes sys/ioctl.h, sys/types.h, and unistd.h.

Figure 7-32. ioctl() subroutine

AU253.0

Notes:

The ioctl() call is an important device control mechanism. The ioctl() subroutine performs control functions associated with open file descriptors.

The ioctl() subroutine performs a variety of control operations on the object associated with the specified open file descriptor. This function is typically used with character or block special files, sockets, or generic device support, such as the termio general terminal interface.

The control operation provided by this function call is specific to the object being addressed, as are the data type and contents of the *Argument* parameter. The ioctlx() form of this function can be used to pass an additional extension parameter to objects supporting it.

Using these calls on a file descriptor associated with an ordinary file results in an error being returned.

Refer to the AIX online documentation for more details about this subroutine.

Device I/O Control Example (non-POSIX)

```
/* ioctl.c */
#include <termio.h> /* terminal character, i/o control */
#include <stdio.h>
struct termio origchar, newchar; /*struct to hold term char*/
main()
{
    ioctl(0,TCGETA,&origchar); /* get the current EOF char */
    newchar = origchar; /* Make a copy in newchar */
    newchar.c_cc[4]='\001'; /* change the EOF char*/
    ioctl(0,TCSETA, &newchar); /*it is now set to <Ctrl>A */
    printf("Executing cat > afile \n");
    printf("<Ctrl>A terminates...\n");
    system("cat > afile");
    ioctl(0,TCSETA,&origchar); /* resets original EOF char */
    printf("All done...\n");
}
```

Figure 7-33. Device I/O Control Example (non-POSIX)

AU253.0

Notes:

Here is a program that demonstrates a use for the `ioctl()` system call.

The `ioctl()` system call can perform miscellaneous functions on I/O devices or their drivers.

This program changes the normal end-of-input character (<Ctrl>D) to <Ctrl>A.

Two `termio` structures are defined. The `termio` structure holds tty control information. The `termio` structure is defined in `/usr/include/termio.h`.

The **TCGETA** command of `ioctl()` saves the current tty settings to each of the `termio` buffers, `origchar` and `newchar`.

The **TCSETA** command changes the current tty settings.

The caller must be root or must own the terminal device affected by the call.

Changing Terminal Characteristics (POSIX)

```
/* noioctl.c */
#include <termios.h>
#include <termio.h>
#include <stdio.h>
main()
{
    struct termios origtermios, newtermios;

    tcgetattr(0, &origtermios);
    newtermios = origtermios;

    newtermios.c_cc[4] = '\001';
    tcsetattr(0, 0, &newtermios);

    printf("Executing cat > afile \n");
    printf("<Ctrl>A terminates...\n");
    system("cat > afile");

    tcsetattr(0, 0, &origtermios);
    printf("All done...\n");
}
```

Figure 7-34. Changing Terminal Characteristics (POSIX)

AU253.0

Notes:

The example in Figure 7-34 uses POSIX calls to perform the same operations done by the previous (non-POSIX) example.

7.5 Even More File and File System Calls

File System Calls

- `stat()` Provides information about a file
- `statfs()` Gets file system statistics
- `mkdir()` Creates a directory
- `link()` Creates a directory entry for an existing file
- `unlink()` Removes a directory entry
- `tmpfile()` Uniquely creates a temporary file
- `vmount()` Mounts a file system
- `umount()` Unmounts a file system

Figure 7-35. File System Calls

AU253.0

Notes:

There are numerous system calls for managing file systems. Each of these calls is described over the next few pages.

The stat Call

```
int stat(const char* path, struct stat* buffer)
```

- Provides information about the file pointed to by path.
- *buffer* is a pointer to a structure of type `stat` (as found in `<sys/stat.h>`).
- Returns zero upon success, -1 upon failure.
- Includes `<sys/stat.h>`.
- The `stat` structure (`stat.h`):

```
struct stat {
    dev_t    st_dev;        /* device ID */
    ino_t    st_ino;        /* inode number */
    mode_t    st_mode;      /* file mode */
    short     st_nlinks;    /* number of hard links */
    uid_t    st_uid;        /* UID of owner */
    gid_t    st_gid;        /* GID of group */
    dev_t    st_rdev;       /* ID of device (for special files) */
    off_t    st_size;       /* file size in bytes */
    time_t    st_atime;     /* last access time stamp */
    time_t    st_mtime;     /* last modification time */
    time_t    st_ctime;     /* last time inode data changed */
    ulong_t   st_blksize;   /* block size (in bytes) for file */
    ulong_t   st_blocks;    /* number of blocks in this file */
    ...        /* other fields exist for */
    ...        /* extended versions of stat calls */
};
```

Figure 7-36. The stat Call

AU253.0

Notes:

The `stat()` system call allows you to examine the contents of a file's inode.

The concept is simple: it retrieves data from the inode associated with the file name and stores it in a buffer pointed to by the second parameter.

The buffer is a `struct stat`, as described in Figure 7-36. This structure is defined in the `/usr/include/sys/stat.h` header file.

There is a `fstat()` call that is equivalent to the `stat()` call, except that it takes a file descriptor instead of a path.

For example:

```
#include <sys/stat.h>
main()
{
    struct stat mystat;
```

```
if (stat("timefile", &mystat) < 0) {
    perror("Problems with stat"); exit(-1);
}
printf("Access time in seconds is %d\n", mystat.st_atime);
printf("Change time in seconds is %d\n", mystat.st_ctime);
printf("mod time in seconds is %d\n", mystat.st_mtime);
}
```

The stat64() subroutine is similar to the stat() subroutine, except that it returns file information in a stat64 structure instead of a stat structure. The information is identical, except that the st_size field is defined to be a 64-bit size. This allows the stat64() subroutine to return file sizes that are greater than OFF_MAX (2 GB minus 1).

The statfs Call (1 of 2)

```
int statfs(char* path, struct statfs* buffer)
```

non-POSIX

- Provides information about the file system that contains the file pointed to by path.
- Buffer is a pointer to a structure of type statfs (as found in <sys/statfs.h>).
- Returns zero upon success, -1 upon failure.
- Includes <sys/statfs.h>.

Figure 7-37. The statfs Call (1 of 2)

AU253.0

Notes:

The statfs() call is similar to the stat() system call, but the statfs() system call returns data about the file system that holds the specified file. The data is placed in the buffer pointed to by the second parameter.

The buffer set to receive the data must be defined as a struct statfs, as defined in /usr/include/sys/statfs.h. This structure is illustrated in Figure 7-38.

The statfs Call (2 of 2)

The statfs structure (statfs.h):

```
struct statfs {
    long f_version;      /* version type, 0 for now */
    long f_type;         /* type of info, 0 for now */
    long f_bsize;        /* file system block size */
    long f_blocks;       /* total data blocks in file system */
    long f_bfree;        /* free blocks in file system */
    long f_bavail;       /* free blocks available to non-su */
    long f_files;        /* total inodes in file system */
    long f_ffree;        /* free inodes in file system */
    fsid_t f_fsid;       /* file system ID */
    long f_vfstype;      /* type of vfs */
    long f_fsize;        /* file system block size */
    long f_vfsnumber;    /* vfs ID number */
    long f_vfsoff;       /* vfs data offset (reserved) */
    long f_vfsvers;      /* version of vfs (reserved) */
    char f_fname[32];    /* file system name (usually */
                        /* mount point) */
    char f_fpack[32]     /* file system pack name */
    long f_name_max;     /* max component name length
                        (POSIX) */
};
```

Figure 7-38. The statfs Call (2 of 2)

AU253.0

Notes:

Here is what the statfs structure looks like.

This information comes from the file system's superblock.

Example:

```
/* statfs.c */
#include <sys/types.h>
#include <sys/statfs.h>
main(int argc, char *argv[])
{
    struct statfs mystatfs;

    if (argc < 2) {
        printf("Usage: %s filename\n", argv[0]); exit(1);
    }
    if ((statfs(argv[1], &mystatfs)) < 0) {
        perror("Problems with statfs"); exit(1);
    }
}
```



```
    }  
    printf("Name of the file system for %s is %s\n", argv[1],  
    mystatfs.f_fname);  
    printf("Name of the vfstype for %s is %d\n", argv[1],  
    mystatfs.f_vfstype);  
    printf("Number of free blocks for %s is %d\n", argv[1],  
    mystatfs.f_bavail);  
    printf("Number of free blocks for superuser for %s is %d\n",  
    argv[1], mystatfs.f_bfree);  
    printf("File system pack name for %s is %s\n",  
    argv[1], mystatfs.f_fpack);  
    printf("File system name for %s is %s\n",  
    argv[1], mystatfs.f_fname);  
}
```

The mkdir Call

```
int mkdir(const char* path, mode_t mode)
```

- Creates a new directory specified by path.
- Mode specifies the desired permissions and can be given by bitwise ORing the permissions defined in the <sys/mode.h> header file.
- The calling process must have WRITE permission in the parent directory.
- Returns zero upon success, -1 upon failure.
- Includes <sys/stat.h> and <sys/mode.h> to use symbolic permission names.

Figure 7-39. The mkdir Call

AU253.0

Notes:

The mkdir() system call is used by the **mkdir** command to create a new directory.

This is a very simple system call. It creates a new directory, much like a new file is created. The permission bits are set from the mode. The owner ID is set to the process effective user ID. There is a corresponding POSIX call, rmdir(path), that removes a directory; just include the <unistd.h> header.

For example:

```
mkdir ("/tmp/test", 0700);  
rmdir ("/tmp/test");
```

The link Call

```
int link(const char* path1, const char* path2)
```

- Creates a new directory entry (*path2*) for existing file *path1*
- Increments the hard link count value for the file's inode
- Returns zero upon success, -1 upon failure

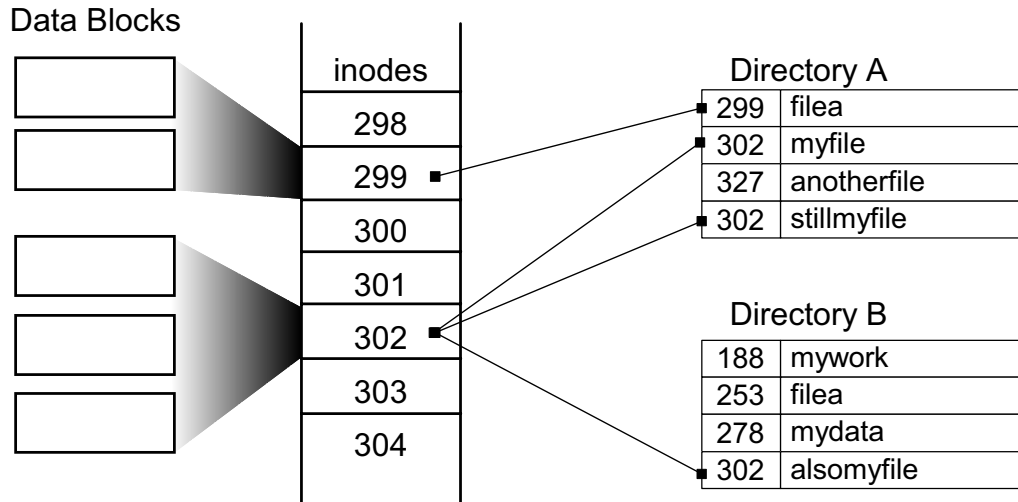


Figure 7-40. The link Call

AU253.0

Notes:

The `link()` system call is used in many instances. For example, when you create a new file, the `link()` system call is used to place that file into the desired directory. The `link()` system call is also called by the **ln** command when performing a hard link. Similarly, the `symlink()` system call is called by the **ln** command when creating a symbolic link (use **man symlink** or the AIX online documentation to get more information on the `symlink` system call).

A link is a connection between a file name and an inode (hard link) or between file names (symbolic link). Linking allows access to an inode from multiple file names. Directory entries pair file names with inodes. File names are easy for users to identify, and inodes contain the real disk addresses of the file's data. A reference count of all links into an inode is maintained in the `i_nlink` field of the inode. Subroutines that create and destroy links use file names, not file descriptors. Therefore, it is not necessary to open files when creating a link.

Hard links are created by the `link()` subroutine. Symbolic links are created by the `symlink()` subroutine.

- **Hard link**

Allows access to the data of a file from a new file name. Hard links ensure the existence of a file. When the last hard link is removed, the inode is released, and the data in the file becomes inaccessible. Hard links can be created only between files that are in the same file system.

- **Symbolic link**

Allows access to data in other file systems from a new file name. The symbolic link is a regular file that contains a path name. When a process encounters a symbolic link, the path name is appended to the path the process was searching, and the search continues. Symbolic links do not protect a file from deletion from the file system. This can be created with the **-s** option of the **ln** command. Refer to the AIX online documentation for more details.

The example illustrates the relationship of directory entries to inodes to data blocks, and shows how more than one directory entry can point to the same inode (via a link).

For example:

```
# ls -l
```

```
-rw-r--r-- 1 root system....filea
-rwxr--r-- 1 root system....myfile
-rw-r--r-- 1 root system....anotherfile
```

```
link("myfile", "stillmyfile");
```

```
# ls -l
```

```
-rw-r--r-- 1 root system....filea
-rwxr--r-- 2 root system....myfile
-rw-r--r-- 1 root system....anotherfile
-rwxr--r-- 2 root system....stillmyfile
```

The unlink Call

```
int unlink(const char* path)
```

- Removes the directory entry pointed to by path.
- Decrements the hard link count of the file.
- When the hard link count reaches zero and no processes have the file open, all resources associated with the file are reclaimed by the file system.
- Returns zero upon success, -1 upon failure.
- Includes <unistd.h>.

Figure 7-41. The unlink Call

AU253.0

Notes:

The reverse of the link() system call, the unlink() system call removes a file reference from a directory.

The unlink() system call is called by the **rm** command to remove a file reference from a directory. It also decrements the link count of the file.

For example:

```
#include <stdio.h>
#include <unistd.h>
main(int argc, char *argv[])
{
    if(unlink (argv[1]) == -1) {
        printf("cannot remove %s\n", argv[1]); exit(1);
    }
    exit(0);
}
```

The tmpfile Call

FILE* **tmpfile()**

- Creates a unique temporary file
- Opens the file for binary write update ("w+b")
- Returns a file pointer
- If file cannot be opened, returns NULL pointer
- Includes <stdio.h>

Figure 7-42. The tmpfile Call

AU253.0

Notes:

The tmpfile() system call generates a unique temporary file and opens a corresponding stream. The temporary file is automatically deleted when all references (links) to the file have been closed. The stream refers to a file that has been unlinked. If the process ends in the period between file creation and unlinking, a permanent file may remain. Refer to the AIX online documentation for more details.

For example:

```
/* mytemp.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
main()  
{  
    FILE *temp;  
    char buffer[50];  
    if ((temp=tmpfile()) == NULL) {  
        printf ("error creating file\n"); exit(1);  
    }  
}
```

```

    }
    fputs("I wrote some data to the file\n", temp);
    rewind(temp);
    fgets(buffer, sizeof(buffer), temp);
    fputs(buffer, stdout);
    exit(0);
}

```

Output:

```
$ mytemp
```

```
output: I wrote some data to the file.
```

Another similar but non-POSIX call is `mktemp(template)`:

```
mktemp( template )
```

- Replaces the string pointed to by *template* with a unique file name.
- The string in the *template* parameter is a file name with up to six trailing Xs. Since the system randomly generates a six-character string to replace the Xs, it is recommended that six trailing Xs be used.
- Returns a unique name upon success, NULL upon failure.
- No include files required.

For example: `mktemp`

```

main()
{
    char newfilename [16];
    strcpy(newfilename, "/tmp/ar.XXXXXX");
    mktemp(newfilename);
    fd = open(newfilename, O_WRONLY, 0666);
    .....
}

```

The vmount and unmount Call

```
int vmount(struct vmount* vmount, int size)
```

non-POSIX

- Mounts a file system, thereby making the file available for use.
- The *vmount* parameter is a pointer to a variable-length vmount structure. This structure is defined in the sys/vmount.h file.
- The *size* parameter specifies the size, in bytes, of the supplied data area.
- Returns zero upon success, -1 upon failure.
- Includes <sys/vmount.h> to define flags.

```
umount(char* device)
```

non-POSIX

- Unmounts (makes unavailable) the file system referenced by device.
- Returns zero upon success, -1 upon failure.

Figure 7-43. The vmount and unmount Call

AU253.0

Notes:

AIX supports the vmount() and umount() system calls. Both are similar to the **mount** and **umount** commands.

The vmount() subroutine only allows mounts of a block device over a local directory with the default file system type. This subroutine searches the /etc/filesystems file to find a corresponding stanza for the desired file system.

You can use the vmount() system call or the **mount** command from within your program. This can be used to mount:

- A local file over a local or remote file
- A local directory over a local or remote directory
- A remote file over a local or remote file
- A remote directory over a local or remote directory

File System Example

```
/* links.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
struct stat *statp;
main()
{
    statp=(struct stat*)malloc(sizeof(struct stat));
    chdir("/home/dave/workshop");
    sync();
    stat("newlist",statp);
    printf("The number of links for newlist is %d.\n",
           statp->st_nlink);
    printf("Linking newlist to altlist...\n");
    link("newlist","/home/dave/altlist");
    stat("newlist",statp);
    printf("Now the no. of links for newlist is %d.\n",
           statp->st_nlink);
    printf("Unlinking...\n");
    unlink("/home/dave/altlist");
    stat("newlist",statp);
    printf("The number of links is back to %d.\n",
           statp->st_nlink);
}
```

Figure 7-44. File System Example

AU253.0

Notes:

Figure 7-44 contains a program that illustrates many of the calls seen on the previous pages.

Possible output:

```
The number of links for newlist is 1.
Linking newlist to altlist...
Now the no. of links for newlist is 2.
Unlinking...
The number of links is back to 1.
```


7.6 Memory Mapped Files

Memory Mapped Files

- AIX implicitly mapped files:
 - Automatic when files opened.
 - Pages of file are mapped to virtual memory pages.
 - Reads and writes handled by kernel as paging operations.
- AIX explicitly mapped files:
 - Application issues `shmat()` on open file.
 - File is mapped into one of ten virtual memory segments.
 - `shmat()` returns the starting memory location of a mapped file.
 - The application can issue memory manipulation instructions. (through pointers) instead of reads and writes.
 - `mmap` (also used to map files to virtual memory).
 - `mmap (addr, len, R/W, flags, fd, offset)`.

Figure 7-45. Memory Mapped Files

AU253.0

Notes:

The final topic for file I/O system calls is memory mapping files.

When files are opened in AIX, the kernel maps the disk data blocks to a virtual memory segment. Then, all reads and writes are actually treated as page-ins and page-outs. This concept differs from the older UNIX kbuffers.

This automatic mapping is called implicit mapping, and while it improves the kernel performance on reads and writes and provides more memory support for files, it still requires system calls to read and write data. Remember that system calls are time consuming.

AIX also supports explicit file mapping via the `shmat()` and `mmap()` system calls.

Use the `shmat` services under the following circumstances:

- For 32-bit applications, where eleven or fewer files are mapped simultaneously and each is smaller than 256 MB.
- When mapping files larger than 256 MB.

- When mapping shared memory regions that need to be shared among unrelated processes (no parent-child relationship).
- When mapping entire files.

Use mmap under the following circumstances:

- Portability of the application is a concern.
- Many files are mapped simultaneously.
- Only a portion of a file needs to be mapped.
- Page-level protection needs to be set on the mapping.
- Private mapping is required.

Explicitly Mapped Files

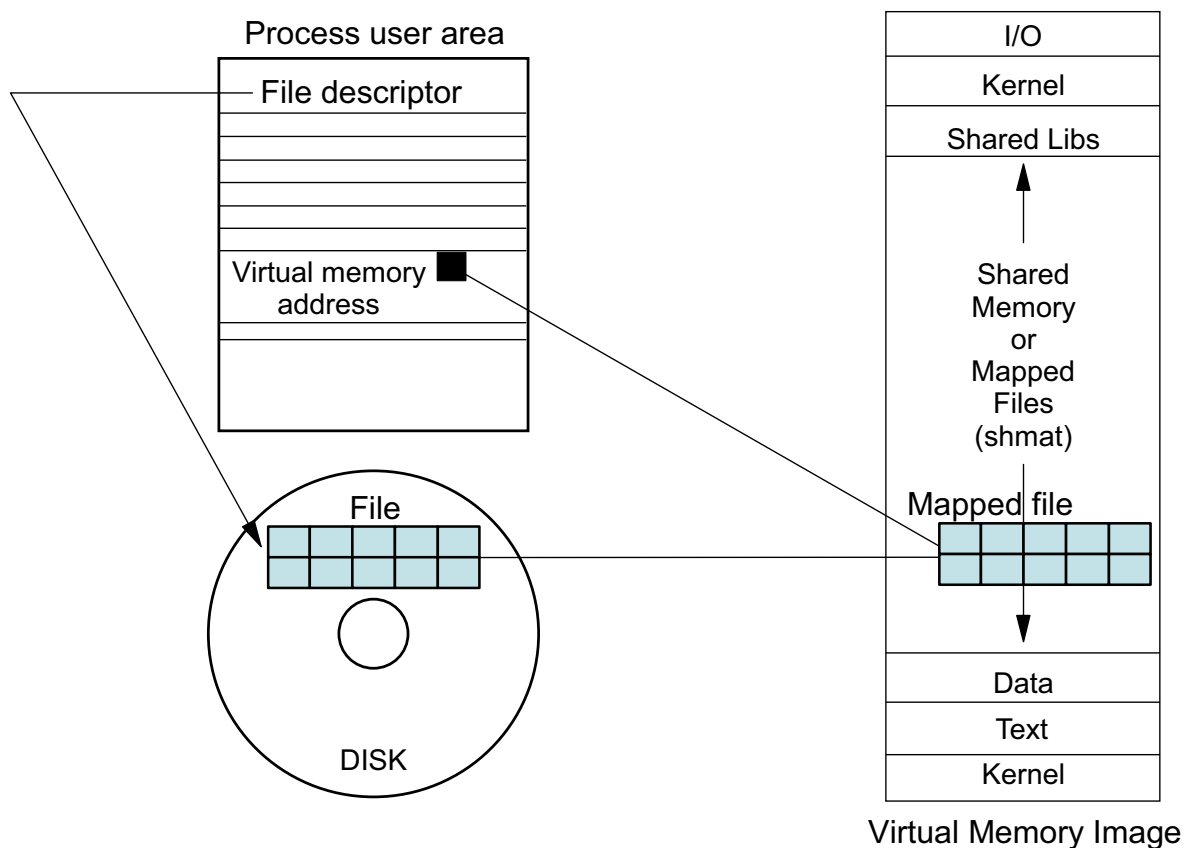


Figure 7-46. Explicitly Mapped Files

AU253.0

Notes:

Figure 7-46 illustrates how to explicitly map a file.

First, the file is opened in the normal way, by allocating a file descriptor (shown on the left side of the user area, pointing to the disk file).

Next, the program performs a `shmat()` system call (shared memory attach) to take the segment allocated for the file by the kernel and map it to one of the process' segments.

For 32-bit processes, the portion of address space available for mapping consists of addresses in the range of `0x30000000` to `0xCFFFFFFF`, for a total of 2.5 GB of address space.

For 64-bit processes, the address ranges with the process address space available for memory mapping files are in the range `0x07000000_00000000` to `0x07FFFFFF_FFFFFFFF`, `0x30000000` to `0xCFFFFFFF`, `0xE0000000` to `0xEFFFFFFF`, and `0x10_00000000` to `0x06FFFFFF_FFFFFFFF`. The last set specified is also made available to the system loader to hold program text, data, and heap, so only unused portions of the range are available for mappings.

Once the file is mapped within the virtual memory boundaries of a process, the process can access the data within the file directly in memory, instead of using read() and write() calls.

A program can still perform read() and write() calls on an explicitly mapped file.

It is possible to map files as a copy-on-write file. This creates a mapped file with changes that are saved in the system paging space instead of to the copy of the file on disk. You must choose to write those changes to the copy on disk to save the changes. Otherwise, you lose the changes when closing the file. Because the changes are not immediately reflected in the copy of the file that other users may access, use copy-on-write mapped files only among processes that cooperate with each other.

To create a Mapped Data File with the shmat subroutine, do the following:

Prerequisite Condition: The file to be mapped is a regular file.

Procedure: The creation of a mapped data file is a two-step process. First, you create the mapped file. Then, because the shmat subroutine does not provide for it, you must program a method for detecting the end of the mapped file.

1. To create the mapped data file:

a. Open (or create) the file and save the file descriptor:

```
if( ( fildes = open( filename , 2 ) ) < 0 )
{
    printf( "cannot open file\n" 0;
    exit(1);
}
```

b. Map the file to a segment with the shmat subroutine:

```
file_ptr=shmat (fildes, 0, SHM_MAP);
```

The SHM_MAP constant is defined in the /usr/include/sys/shm.h file. This constant indicates that the file is a mapped file. Include this file and the other shared memory header files in a program with the following directives:

```
#include <sys/shm.h>
```

2. To detect the end of the mapped file:

a. Use the lseek subroutine to go to the end of file:

```
eof = file_ptr + lseek(fildes, 0, 2);
```

This example sets the value of eof to an address that is 1 byte beyond the end of file. Use this value as the end-of-file marker in the program.

b. Use file_ptr as a pointer to the start of the data file, and access the data as if it were in memory:

```
while ( file_ptr < eof)
```

```
{  
    ....  
    (references to file using file_ptr)  
}
```

c. Close the file when finished:

```
close (fildes);
```

The shmat Call for Mapping Files

```
void* shmat(int fdes, const void* shmaddr, int flags)
```

non-POSIX

- Explicitly maps a file to the calling process' address space.
- fdes is the file descriptor for the previously opened file
- shmaddr specifies the segment to use. A value of zero assigns the first available segment (suggested use).
- flags must include SHM_MAP, to specify the mapped file request (instead of the shared memory attachment).
- flags may include SHM_RDONLY, to make the mapped file read-only.
- Returns the starting address of the mapped file upon success, -1 upon failure
- Includes <sys/shm.h>.

Figure 7-47. The shmat Call for Mapping Files

AU253.0

Notes:

Figure 7-47 shows the syntax of the shmat() system call.

Other versions of UNIX support shmat(), but each system's use of virtual memory differs.

The shmat() call returns a character pointer to the start of the mapped file.

The second parameter can be used to specify the starting address of the segment to map. More information on shared memory will be presented in the Interprocess Communications lecture.

Mapped File Example

```
/* writeit.c */
/* writes 100000 characters into a file */
#include <fcntl.h>
#include <sys/types.h>
main()
{
    int i, fdw;
    char c='A';
    fdw=open("wfile", O_CREAT|O_RDWR,0666);
    for(i=0;i<100000;i++)
        write(fdw,&c,1);
    close(fdw);
    exit(0);
}

/* mapit.c */
/* writes 100000 characters into a shmat-mapped file */
#include <fcntl.h>
#include <sys/shm.h> /* shmat structures */
main()
{
    int i, fdw;
    char *ptr, *beg_ptr;
    fdw=open("mfile",O_CREAT|O_RDWR,0666);
    ptr=shmat(fdw,0,SHM_MAP);
    beg_ptr=ptr;
    for(i=0;i<100000;i++)
        *ptr++ = 'A';
    shmdt(beg_ptr);
    ftruncate(fdw,100000);
    close(fdw);
    exit(0);
}
```

Figure 7-48. Mapped File Example

AU253.0

Notes:

Figure 7-48 shows two programs: writeit.c and mapit.c.

writeit.c writes 100,000 characters to a file, one character at a time, using the write() system call. Actually, the program does 100,000 write() system calls.

The mapit.c file opens the file, then maps it. Once mapped, 100,000 characters are stored in the file, one character at a time, by using memory pointers.

The shmdt() call detaches a shared memory segment. It takes a SharedMemoryAddress as its argument.

The ftruncate() subroutines change the length of regular files. It takes the file descriptor and the length as its parameters. The length parameter measures the specified file in bytes from the beginning of the file. If the new length is less than the previous length, all data between the new length and the previous end of file is removed. If the new length in the specified file is greater than the previous length, data between the old and new lengths is read as zeros.

You will create these programs in lab, then test them using the **time** command. You will be amazed at the speed of mapped files in contrast to using the write() system call.

The mmap Call

```
void* mmap(void* addr, size_t len, int
prof, int flags, int fildes, off_t off)
```

- Explicitly maps a file system object into virtual memory.
- *addr* specifies the starting address of the memory region to be mapped.
- *len* specifies the length, in bytes, of the region to be mapped.
- *prof* specifies the access permissions.
- *fildes* specifies the file descriptor.
- *off* specifies the file byte offset at which mapping starts.
- *flags* specifies attributes of the mapped region.
- Returns the starting address of the mapped file upon success, -1 upon failure.
- Includes <sys/types.h> and <sys/mman.h>.

```
void* mmap64(void* addr, size_t len, int
prof, int flags, int fildes, off64_t off)
```

- *off* is a 64-bit value specifying the file byte offset at which mapping starts.
- The rest of the fields and functionalities are the same as the `mmap` call.

Figure 7-49. The mmap Call

AU253.0

Notes:

The `mmap` subroutine provides a unique object address for each process that maps to an object. The software accomplishes this by providing each process with a unique virtual address, known as an alias. The `shmat` subroutine allows processes to share the addresses of the mapped objects.

The `mmap64` subroutine is identical to the `mmap` subroutine, except that the starting offset for the file mapping is specified as a 64-bit value. This permits file mappings that start beyond `OFF_MAX`. In the large file enabled programming environment, `mmap` is automatically redefined to be `mmap64`.

For applications in which many processes map the same file data into their address space, this toggling process may have an adverse affect on performance. In these cases, the `shmat` subroutine may provide more efficient file-mapping capabilities.

Note: A file-system object should not be simultaneously mapped using both the `mmap` and `shmat` subroutines. Unexpected results may occur when references are made beyond the end of the object.

The `munmap()` call unmaps a mapped region. This syntax is:

```
int munmap ( void* addr, size_t len)
```

The `munmap` subroutine unmaps regions created from calls to the `mmap` subroutine only. If an address lies in a region that is unmapped by the `munmap` subroutine and that region is not subsequently mapped again, any reference to that address will result in the delivery of a SIGSEGV signal to the process.

mmap Example

```
/* mmap.c */
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <errno.h>
main()
{
    int fd, i;
    caddr_t ptr, beg_ptr;

    if ((fd = open("mmapfile", O_RDWR)) < 0) {
        perror("problems with open"); exit(1);
    }
    ptr = mmap((caddr_t) NULL, 256,
        PROT_READ|PROT_WRITE,
        MAP_FILE|MAP_VARIABLE|MAP_SHARED,
        fd, (off_t) 0);
    if (ptr == (caddr_t) -1) {
        perror("problems with mmap"); exit(1);
    }
    beg_ptr = ptr;

    for (i=0; i<100; i++)
        *ptr++='A';
    if (munmap(beg_ptr, 256) < 0) {
        perror("problems with munmap"); exit(1);
    }
    ftruncate(fd, 100);
    if (close(fd) < 0) {
        perror("problems with close"); exit(1);
    }
    printf("done\n");
    exit(0);
}
```

Figure 7-50. mmap Example

AU253.0

Notes:

Figure 7-50 shows an example of using mmap to map a file.

This example program assumes that file to be memory mapped already exists and is not empty. The open call will fail if it does not exist. The first attempt to execute the `*ptr++='A';` statement will result in a core dump if the file is empty, because there is nothing to mmap.

7.7 Working With Large Files

How Large Is Large?

- Files larger than 2,147,483,647 (2 GB minus 1) bytes are considered to be large files.
- Largest possible file size in JFS (Journaled File System) is:
 - 2,147,483,647 bytes on file systems that are not large file enabled
 - 64 GB on large file enabled file systems
- Largest possible file size in JFS2 (Enhanced Journaled File System) is 1 Petabyte or 1,048,576 GB.

Figure 7-51. How Large Is Large?

AU253.0

Notes:

A large file is any file larger than 2,147,483,647 bytes. A large file enabled file system is a file system that is capable of managing a large file (we will discuss more details about large file enabled file systems shortly).

What Can Go Wrong?

- Some system calls take and/or return 32-bit file size or offset parameters.
- Most programs use 32-bit data types to hold file sizes and offsets
- Some programs store file sizes and offsets in files.
- A lot of programs do 32-bit arithmetic on file sizes and offsets.

Figure 7-52. What Can Go Wrong?

AU253.0

Notes:

The “files are never more than 2,147,483,647 bytes” assumption turns up in many places. Here are a few of them:

- The `lseek()` system call takes a 32-bit signed offset as its second parameter and returns a 32-bit signed result. The same problem occurs with the `fcntl()` `F_GETLK` operation and the standard I/O library's `fseek()` and `ftell()` routines (there are others).
- Programs that keep track of how many bytes they have written generally use a 32-bit datatype to hold the value.
- Programs that keep track of file offsets (for example, when building an index into a file) typically use either an `int` or a `long` to hold the offset. These frequently end up in data files as four byte binary values.
- Programs that take block offsets and use them to compute byte offsets (or any other arithmetic on file sizes) will suffer a loss of precision (that is, an overflow) if the values get too large.

Probably the most insidious result of this problem is that a program that is not designed to handle large files could accidentally corrupt a large file. For example, consider the following program:

```
int block_count;

.

.

lseek(fd, block_count * 512, SEEK_SET);
write(fd, data, data_len);

.

.
```

If `block_count` is too large, then the `lseek()` call could either fail or move the read-write pointer to an unexpected place in the file. Since `lseek()` practically never fails, many programmers do not check its return value.

Open Protection

To prevent data corruption (and other problems), AIX implements the following Open Protection scheme:

- Large files (that is, more than 2,147,483,647 bytes) must be opened using the new `open64()` system call (the old `open()` system call fails if the file being opened is a large file).
- The `read()` and `write()` calls fail if an attempt is made to read or write past the old size limit of 2,147,483,647 bytes (unless the file descriptor was opened using `open64()`). A few other system calls, such as `mmap()`, `ftruncate()`, and `fclean()` also fail in similar situations.
- Large files may only be created on file systems that have been defined to be large file enabled (something that must be done when the file system is created).

Figure 7-53. Open Protection

AU253.0

Notes:

This scheme mostly prevents a program that has not been designed to work with large files from getting into serious trouble if it encounters a large file.

The Open Protection scheme is not perfect. For example, file descriptors are inherited across `fork()` and `exec()` calls. Consequently, a program being executed could encounter a file descriptor that was opened using `open64()`. The bottom line is that a program that opens a large file should be careful about passing the resulting file descriptor to a program that is not designed to handle the file.

Failure as described above means that the system call returns -1 and sets `errno` to an appropriate value (usually either `EFBIG` or `EOVERFLOW`).

Large files are only allowed on large file enabled file systems. Note that it is not enough to have a multi-gigabyte file system; it must have been defined to be large file enabled when it was created by the system administrator.

Working with Large Files

There are two choices for working with large files:

- Use the new large-file-capable system calls, like `open64()` and `lseek64()`.
- Define `_LARGE_FILES` before including any system header files.

Note: In either case, the programmer must be very careful when doing arithmetic on file sizes or passing file sizes as parameters to other routines.

Figure 7-54. Working with Large Files

AU253.0

Notes:

The regular AIX file I/O system calls revolve around the `off_t` data type being a 32-bit signed integer. The `off64_t` data type is a 64-bit signed integer. A set of new 64-bit data types and system calls that use it are available. Some of the key ones are:

- `extern int open64(const char *, int, ...);`

Used to open a large file (or a file that might become large during the current use of the file).

- `extern int lseek64(int, off64_t, int);`

Used to move the read-write offset in a large file.

- `struct stat64;`

The `stat64` struct is identical to the `struct stat` struct, except that the `stat64` struct has an `off64_t` field for the file size.

- `extern int stat64(const char *, struct stat64 *);`

The large file counterpart of `stat()`.

By using these routines, an application programmer can deal with files that might be larger than 2,147,483,647 bytes.

The alternative is to compile the program with `_LARGE_FILES` defined (that is, by a `#define` statement or using the `-D` compiler option) before the inclusion of any system header files. The result is that `off_t` is defined as a 64-bit signed value and the old 32-bit system calls like `open()` and `lseek()` are re-mapped to their 64-bit counterparts.

If the application programmer has access to and control of enough of the source code for an application, then the `_LARGE_FILES` approach is often better in the long run. If the program is quite complex and not particularly careful about how file sizes are stored or manipulated, then the approach of only using the new 64-bit calls where absolutely necessary might be easier.

Probably the easiest and most common mistake to make when dealing with large files is to forget that a file's size might not fit in an `int`. ALWAYS use `off64_t` or the long 64-bit integer.

One classic mistake is:

```
int block_count;
off64_t offset;
...
offset = block_count * 512;
```

This does not work because `block_count * 512` is performed using 32-bit arithmetic. The result of the multiplication is then converted to a 64-bit integer and assigned to `offset`. One of a number of possible correct approaches is:

```
int block_count;
off64_t offset;
...
offset = block_count * (long long)512;
```

An alternative to `block_count * (long long)512` is `((off64_t)block_count) * 512`. The key is to ensure that at least one of the operands of the multiplication operator is a 64-bit value.

The File Size ulimit

- The default file size ulimit is 2097151 blocks (that is, 2097151*512 bytes).
- The limit can be increased using the **ulimit** command if the system administrator has increased the system-wide file size ulimit.
- The system-wide file size limit is defined in /etc/security/limits (only the system administrator can look at or change this file).

Figure 7-55. The File Size ulimit

AU253.0

Notes:

In the Korn shell, enter:

```
$ ulimit -f
```

to see what your file size ulimit is (in 512 byte blocks). You can increase your file size ulimit using:

```
$ ulimit -f new_limit
```

A new_limit of -1 removes the limit entirely.

Note that you cannot increase your ulimit beyond what the system administrator has defined as the system default. AIX 5L set the system wide file size ulimit to 2097151 blocks (or 1 GB minus 512 bytes) when the operating system is installed.

Unit Summary

- File System Concepts
- Basic File Manipulation Calls
- File and Record Locking
- Controlling File and File System Attributes
- Device I/O Control
- Explicitly Mapped Files
- Large Files

Figure 7-56. Unit Summary

AU253.0

Notes:

Unit 8. AIX Process Management Systems Calls

What This Unit is About

This unit has three parts.

The first part of this unit describes the properties of processes and how these properties can be managed with system calls. A system call syntax box is introduced in this unit. This syntax box includes an explanation of each parameter, the meaning of expected and unexpected return values, and the names of the header files that must accompany each system call. The syntax box cannot replace the AIX documentation and makes no attempt to do so. You are reminded to consult online documentation, the man pages, or the hardcopy documentation for further details.

The second part of this unit describes the life cycle of a process. The `fork()`, `exit()` and `wait()` system calls are introduced, as well as the complete `exec` family of calls.

The third part of this unit illustrates the creation of a process with an accompanying example.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Demonstrate the use of system calls associated with process IDs, user IDs, and group IDs
- List the system calls used with process scheduling
- Use the `fork()` and `exec()` system calls to create a new process
- Explain the process life cycle

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 6

Unit Objectives

After completing this unit, you should be able to:

- Demonstrate the use of system calls associated with process IDs, user IDs, and group IDs
- List the system calls used with process scheduling
- Use the fork() and exec() system calls to create a new process
- Explain the process life cycle

Figure 8-1. Unit Objectives

AU253.0

Notes:

8.1 AIX Process Management System Calls

Properties of a Process

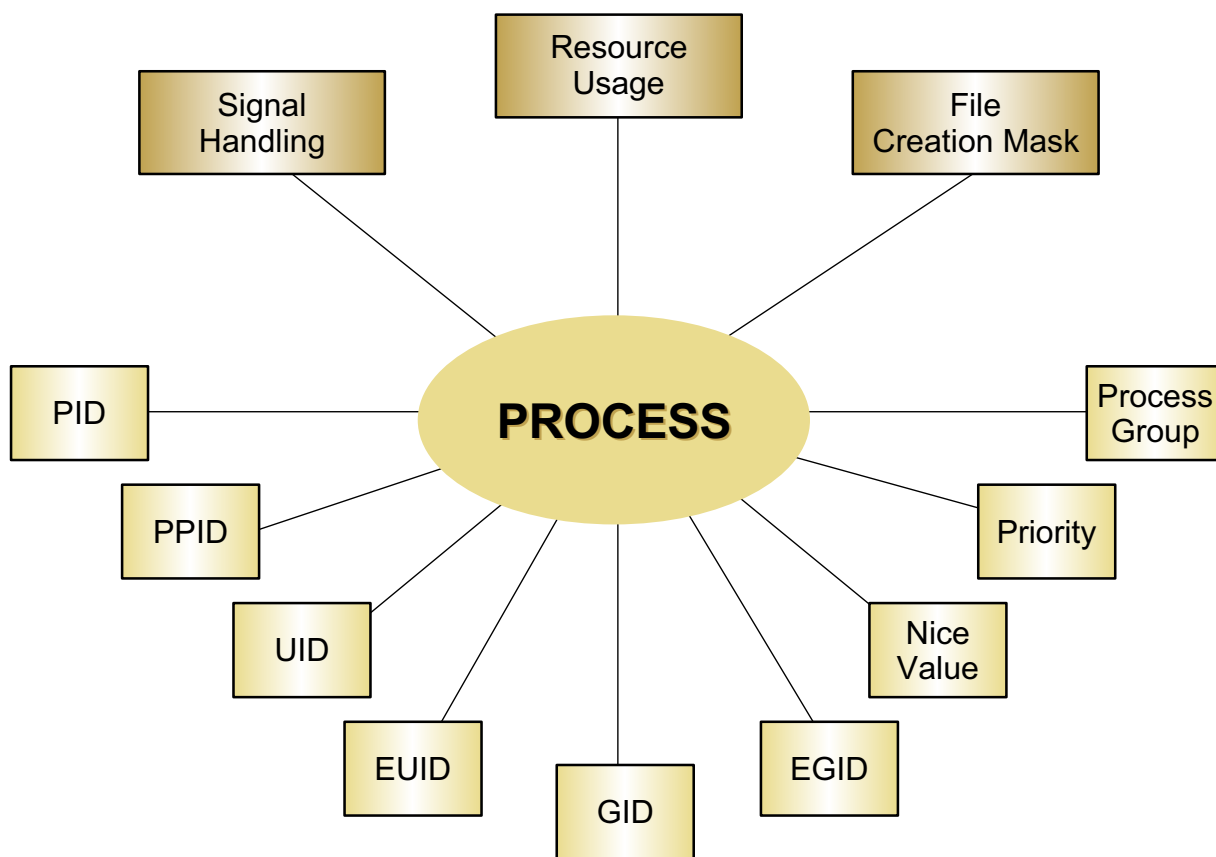


Figure 8-2. Properties of a Process

AU253.0

Notes:

A process is a program under execution. When a program is to be run, the shell creates a process (called a child process) and uses it for execution. The process that created the child process is called the parent process.

Every process has a set of attributes. Many of these attributes are inherited by a child process from its parent process when the child is created. Such attributes include the real and effective user IDs, real and effective group IDs, the process group, the priority and nice values, the file creation mask, and the signal handling masks.

Other process attributes are unique to the child, such as its process ID. Process resource values are also refreshed for the child process.

The following pages illustrate system calls used to query and change the process attributes described here.

Process IDs

```
pid_t getpid(void)
```

- Returns the process ID number of the calling process (integer)
- No arguments
- Includes <unistd.h> and <sys/types.h> to typedef returns value as pid_t

```
pid_t getppid(void)
```

- Returns the process ID number of the parent of the calling process (integer)
- No arguments
- Includes <unistd.h> and <sys/types.h> to typedef returns value as pid_t

Figure 8-3. Process IDs

AU253.0

Notes:

The `getpid()` system call allows a process to determine its own process ID number (PID). This value is returned from the process' user area. The call has no parameters.

Since PIDs are long integers in AIX, a variable used to store the return value can be declared as a long int. For compatibility, programmers may want to declare the variable used to store the return value as a type `pid_t`. This typedef value is defined in the `types.h` header file in `/usr/include/sys`. The variable that the `getpid` function returns is defined in `/usr/include/unistd.h`.

The `getppid()` system call returns the PID of the parent of the calling process.

For example: Create a temporary file name and store it in the buffer `fname` with the pid.

```
char fname[15];  
.  
.  
.  
sprintf (fname, "/tmp/x.%d", getpid( ));
```

User and Group IDs (1 of 4)

- Real User ID
- Effective User ID
- Saved User ID
- Real Group ID
- Effective Group ID
- Saved Group ID
- Supplementary Group ID

Figure 8-4. User and Group IDs (1 of 4)

AU253.0

Notes:

Real User ID - The real user ID identifies the user who was last authenticated. Setting the real user ID requires root privilege and sets the effective user ID to the same value. Your login initially sets your real user ID. An **su** can change it.

Effective User ID - The effective user ID indicates the current user has access rights to the process. It is used in access control decisions, and set as a side effect of explicitly setting the real user ID or of implicitly setting the saved user ID as a result of executing a **setuid** program.

Saved User ID - The saved user ID stores the user ID associated with a **setuid** program executed by the process. It is set at each **exec** call of a new program; if the new program has the **setuid** flag set, the saved user ID is set to the user ID of the owner of the file; otherwise, the saved user ID is set to the value of the effective user ID when the **exec** program is called.

Real Group ID - The real group ID indicates the primary group of the process. It is set when the user is authenticated upon login or by the **su** command. When the real group ID is set, the effective group ID is set to the same value.

Effective Group ID - The effective group ID indicates part of the current group access rights to the process. It, along with the supplementary group ID, is used in access decisions. It is set as a side effect of setting either the real or saved group IDs.

Saved Group ID - The saved group ID is used to store the group ID associated with a setgid program that the process has executed. It is set at each exec call of a new program. If the new program has the setgid flag set, the saved group ID is set to the group ID associated with the file; otherwise, the saved group ID is set to the value of the effective group ID when the exec program is called.

Supplementary Group ID - The supplementary group ID indicates the other part of the group access rights of the process (with the effective group ID). It is set to any subset of the (real) users group membership at login or when using the **su** command, and can be reset thereafter by the user as above. The group set is always inherited by new processes (fork) or by subsequent programs (exec).

User and Group IDs (2 of 4)

`uid_t getuid(void)`

- Returns the users ID associated with the calling process (integer)
- No arguments
- Includes <unistd.h> and <sys/types.h> to typedef returns value as uid_t

`uid_t geteuid(void)`

- Returns the effective user ID associated with the calling process (integer)
- No arguments
- Includes <unistd.h> and <sys/types.h> to typedef returns value as uid_t

Figure 8-5. User and Group IDs (2 of 4)

AU253.0

Notes:

The `getuid()` system call returns the real user ID of the calling process. The `geteuid()` system call returns the effective user ID of the calling process.

The real user ID reflects the ID of the user who executes the program, either as their login user ID or as the result of the **su** (switch user) command. The effective user ID represents the user ID associated with the execution environment of this process. The effective user ID is used for authority testing with AIX permissions. The effective user ID is different from the real user ID only when executing a program that has had the `setuid` feature activated.

The `I_YSUID` (set-user-id) flag is set on a program via the **chmod** command or system call. The program runs with an effective user ID of the owner of the program. The result is that the program runs with the privileges of the program file's owner. For example, if the program file's owner is root, then the program can perform privileged operations normally reserved to root.

For example: to add the set-user-id flag (`I_YSUID`) to the file `prog`:

```
chmod u+s prog
```


The following example demonstrates the change in user ID:

```
/* Program checkid1.c owned by user1 */
#include <stdio.h>
main()
{
    printf("Running program owned by user2\n");
    printf("UID = %d\n",getuid());
    printf("EUID = %d\n",geteuid());
}

/* Program checkid2.c owned by user2 */
#include <stdio.h>
main()
{
    printf("UID = %d\n",getuid());
    printf("EUID = %d\n",geteuid());
    execl("/home/user1/checkid1.c", "checkid1", 0);
}
```

Compile both the programs. Ensure that the executable of program checkid1.c has its set UID bit set. On executing the program checkid2.c, you will see that the effective user ID is different when it executes the program checkid1.c.

User and Group IDs (3 of 4)

`gid_t getgid(void)`

- Returns the group ID associated with the calling process (integer)
- No arguments
- Includes <unistd.h> and <sys/types.h> to typedef return value as gid_t

`gid_t getegid(void)`

- Returns the effective user ID associated with the calling process (integer)
- No arguments
- Includes <unistd.h> and <sys/types.h> to typedef return value as gid_t

Figure 8-6. User and Group IDs (3 of 4)

AU253.0

Notes:

The `getgid()` and `getegid()` system calls return the real and effective group IDs, respectively. The variable used for the return value can be declared as an integer or as a typedef `gid_t`.

The effective group ID is different from real group ID only when a executing a program whose `setgid` is active. The program then runs with the privileges of the program file's group. The `setgid` of a program is activated using the **chmod** command as follows:

```
chmod g+s <program name>
```

For example:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main ()
{
    uid_t uid;
```

```
gid_t gid;
uid = getuid();
printf("Real UID is %d\n", uid);
uid = geteuid();
printf ("Effective UID is %d\n", uid);
gid = getgid();
printf ("Group ID is %d\n", gid);
gid = getegid();
printf ("Effective GID is %d\n", gid);
}
```

User and Group IDs (4 of 4)

```
int setuid( uid_t UID )
```

- Sets the calling process real or effective user ID.
- UID is the desired user ID (integer).
- Returns zero upon success, -1 upon failure, and the errno global variable is set to indicate the error.
- Includes the file <unistd.h>.
- Includes the file <sys/types.h> to typedef argument as uid_t.

```
int seteuid( uid_t EUID )
```

- Sets the calling process effective user ID.
- EUID is the desired effective user ID (integer).
- Returns zero upon success, -1 upon failure, and the errno global variable is set to indicate the error.
- Include the file <sys/types.h> to typedef argument as uid_t.

Figure 8-7. User and Group IDs (4 of 4)

AU253.0

Notes:

With the setuid(uid) call, one of the following conditions occur:

- If the effective ID of the caller is the same as the root, the process real, effective, and saved user IDs are set to the value of UID.
- If the caller is not root, the effective user ID is reset if uid is equal to either the current real user ID or saved user ID.

The seteuid(euid) call resets the caller's effective user ID if one of the following conditions are met:

- The euid parameter is equal to either the current real or saved user IDs.
- If the effective user ID of the process is that of the root user.

Process Groups (1 of 2)

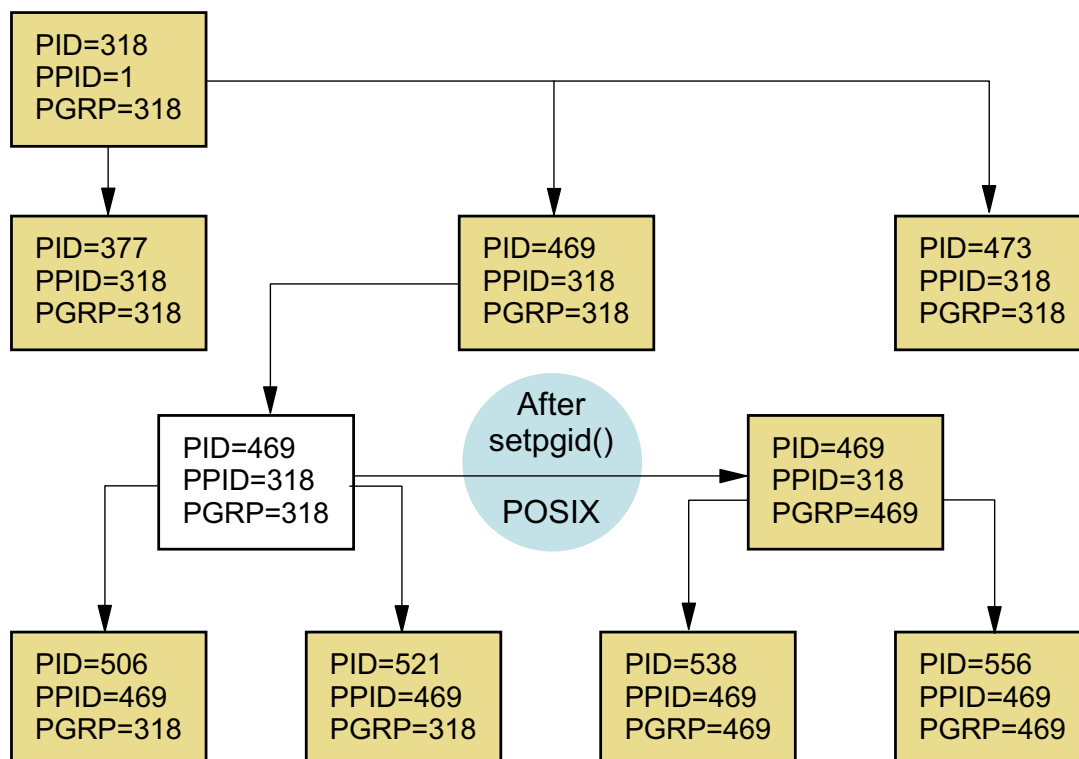


Figure 8-8. Process Groups (1 of 2)

AU253.0

Notes:

Every process belongs to a process group. Normally, a user's login shell is the process group leader for all child processes created during the login session. Each process has a reference to its group leader. The process group ID is the PID of the group leader.

Figure 8-8 illustrates a hierarchical relationship of processes in a process group.

The most important feature of a process group is the fact that any signal received by the process group leader is automatically distributed to all processes in the same process group.

A process can resign itself from a process group and start its own process group by using a **setpgid()** call. The resulting new process group ID is set to the process GID specified. If a zero is used as the new pgid, then the pgid is set to the PID of the calling process. To retrieve the current process group ID, the **getpgrp()** POSIX call may be used. It takes no arguments.

Processes that are children of the login shell terminate when the user logs off. The **nohup** command allows a process to continue execution even after the user logs off.

Process Groups (2 of 2)

```
int setpgid( pid_t ProcessID, pid_t ProcessGroupID )
```

- Sets process' group ID to a new or existing process group.
- *ProcessID* is the desired process (integer). If zero, the process ID of the calling process is used.
- *ProcessGroupID* is the desired process group ID (integer). If zero, the process group ID is set to PID.
- Returns zero upon success, -1 upon failure, and the `errno` global variable is set to indicate the error.
- Include files `<unistd.h>` and `<sys/types.h>` to typedef argument as `pid_t`.

Figure 8-9. Process Groups (2 of 2)

AU253.0

Notes:

The `setpgid(ProcessID, ProcessGroupID)` allows the calling process to create a new process group or attach an existing process to an existing process group. The *ProcessID* parameter specifies the desired process. If *ProcessID* is zero, the PID of the calling process is used. If *ProcessGroupID* is zero, the value of the *ProcessID* is used for the new process group ID. An example of using process calls follows.

Process ID, Group ID, and User ID Example (1 of 2)

```

sample.c:
#include <stdio.h>
main()
{
    printf("My pid is %d.\n",getpid() );
    printf("My parent's pid is %d.\n",getppid() );
    printf("My process group id is %d.\n",getpgrp() );
    printf("My real user id is %d.\n", getuid());
    printf("My effective user id is %d\n", geteuid());
    if (getpid() == getpgrp() )
    {
        printf("This group leader process is ending now...\n");
        exit(0);
    }
    printf("Starting my own process group now...\n");
    setpgid(0,0);
    printf("My process group id is now %d.\n", getpgrp() );
    printf("ending...\n");
}

sample.sh:
./sample

```

Figure 8-10. Process ID, Group ID, and User ID Example (1 of 2)

AU253.0

Notes:

There are two programs in Figure 8-10. The sample.c program displays the attributes of the process, such as process ID, parent process ID, user ID, effective user ID, and process group ID. The setpgid() system call tries to change the process group. This will fail if the process is also the process group leader, meaning if the PID and process group ID are the same.

The sample.sh is a shell script. It just invokes the sample program. This is to ensure that the PID and process group ID are different and the setpgid() call succeeds.

The getpgrp() call returns the process group ID of the calling process.

Process ID, Group ID, and User ID Example (2 of 2)

Possible program output:

```
My pid is 2341.  
My parent's pid is 2289.  
My process group id is 2289.  
My real user id is 204.  
My effective user id is 204.  
Starting my own process group now...  
My process group id is now 2341.  
ending...
```

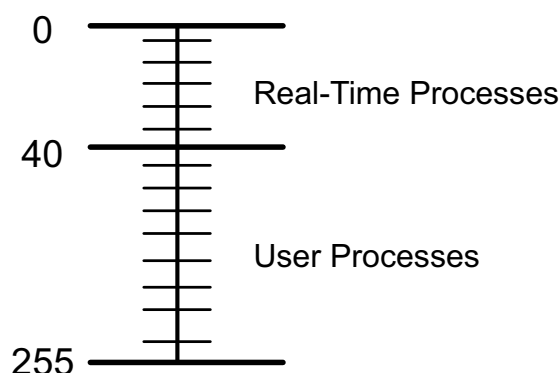
Figure 8-11. Process ID, Group ID, and User ID Example (2 of 2)

AU253.0

Notes:

Figure 8-11 illustrates possible output from the program example.

Process Scheduling



- Every AIX 5L process/thread has a priority value (0-255); a lower priority value means the thread is more favored.
- Process/thread priority values set with `setpri()` are fixed.
- Other process/thread priority values change as the process runs.
- Real-time processes can only be fixed at values less than 40 by using the `setpri()` system call

Figure 8-12. Process Scheduling

AU253.0

Notes:

Every AIX process is a schedulable entity. Therefore, every process has a priority value. In AIX 5L, priority values can range from 0 to 255 (0 - 127 in AIX Version 4.3), with priorities 0 through 39 being reserved for real-time processes. Regular user processes will have priorities between 40 and 254. Priority level 255 is reserved for the "idle" process, which runs whenever there are no other processes to schedule. The header file `/usr/include/sys/pri.h` defines some of the default priorities for system processes.

Each priority value equates to a run queue level. The kernel's process table maintains the fields necessary to manage process scheduling. It also maintains the run queues.

As processes run, their priorities are constantly recalculated, thus moving the processes from one queue level to another.

In an AIX process, there is only one stream of instructions that is in control of the process at one time. In a multithreaded process, the process starts out with one stream of instructions and may later create other instruction streams, called threads, to do tasks. This is very much like one process forking another process.

From AIX Version 4 onwards, the kernel supports multiple threads of control within a single process. An application can use this to bypass classical Inter-Process Communications (IPC). That is, data can be shared between the various threads of control using normal process storage, and access to this data is coordinated using library routines, thus removing the overhead of IPC associated with shared memory and semaphores.

Due to the support of threads, the scheduler from AIX Version 4 onwards is thread based rather than process based. This requires the runq to be thread based.

Scheduling Policies

- SCHED_FIFO
- SCHED_RR
- SCHED_OTHER
- SCHED_FIFO2
- SCHED_FIFO3

Figure 8-13. Scheduling Policies

AU253.0

Notes:

There are now five scheduling algorithms that can be selected when creating a thread:

SCHED_FIFO

This is a non-preemptive scheduling mechanism. A thread created with SCHED_FIFO will run at a fixed priority and will not be timesliced. It will be allowed to run on a processor until it voluntarily relinquishes by blocking or yielding.

A thread using SCHED_FIFO must also have root authority to use it. It is possible to create a thread with SCHED_FIFO that has a high enough priority that it could monopolize the processor.

SCHED_RR

This is a round-robin scheduling mechanism. The thread is timesliced at a fixed priority. At the end of the timeslice, it moves to the end of the priority queue.

SCHED_OTHER

This is normal AIX scheduling and is the default, where priority degrades with CPU usage.

SCHED_FIFO2

This policy is the same as SCHED_FIFO, except that it allows a thread that has slept for only a short amount of time to be put at the head of its run queue when it is awakened.

SCHED_FIFO3

A thread whose scheduling policy is set to SCHED_FIFO3 is always put at the head of a run queue.

Refer to the AIX 5L online documentation for more information on scheduling policies.

CPU Timeslicing

- Time given for threads to run.
- CPU usage is shared.
- Default timeslice is one clock ticks (10 ms).
- The **schedtune** command can be used to change the timeslice values.

Figure 8-14. CPU Timeslicing

AU253.0

Notes:

Timeslicing is the time for which a thread, scheduled with the SCHED_OTHER and SCHED_RR policies, runs before yielding the CPU to the next thread. This way, the CPU is available to all threads in the run queue. The timeslice is measured in units of clock ticks. Each clock tick is ten micro seconds.

The timeslice value can be changed using the **schedtune** command with the -t option. The syntax for the command is:

```
schedtune -t <number of clock ticks>
```

The CPU scheduler's priority calculations are based on two flags of **schedtune**, -r and -d. The -r and -d values are used to calculate the CPU penalty based on recent CPU usage, required for priority calculation.

Process Scheduling System Calls

AIX
(non-POSIX) `int getpri(pid_t ProcessID)`

- Returns the priority value of the process *ProcessID* (integer).
- *ProcessID* is the desired process (integer). If zero, the calling process' *ProcessID* is used.
- Returns -1 upon failure.
- Include file <sys/types.h> to typedef argument as pid_t.

AIX
(non-POSIX) `int setpri(pid_t ProcessID, int Priority)`

- "Fixes" priority of the process *ProcessID* (integer) to value *Priority* (integer).
- Process's nice and CPU values no longer examined.
- If *ProcessID* is zero, current process's priority is fixed.
- Can only be called by processes with root authority.
- Returns former process priority upon success, -1 upon failure.
- Include <sys/sched.h>.
- Include file <sys/types.h> to typedef argument *ProcessID* as pid_t.

Figure 8-15. Process Scheduling System Calls

AU253.0

Notes:

The `getpri(ProcessID)` system call returns the current priority value of the process or the first thread in the process specified by *ProcessID*. If *ProcessID* is zero, the calling process's priority is returned.

The `setpri(ProcessID, Priority)` system call allows a process to "fix" the specified process's priority and all its threads to a value in the range of 0 to 126. The *ProcessID* parameter specifies the desired process. If *ProcessID* is zero, the calling process's priority is set. Processes that have a fixed priority are immune to priority recalculations and requeueing.

Only root can use the `setpri()` call to change a process's priority. Special care must be taken to avoid setting a process's priority to a real-time value that results in using large amounts of CPU time. For example, fixing a process's priority to 5, then executing an endless loop that does nothing will appear to lock up your system, because no other processes will be able to run.

Two related non-POSIX calls are `getpriority()` and `setpriority()`, which gets or sets the nice value.

The system call `yield()` will place the caller on the end of the appropriate process list. Because the schedulable entity for AIX 5L Version 5.1 is the thread, this system call will only cause the current thread to relinquish the CPU.

Process Scheduling Example (1 of 2)

```
/* sample1.c */

#include <stdio.h>
main()
{
    int i, oldpri;
    printf("My pid is %d.\n",getpid() );
    printf("My current priority is %d.\n",getpri(0) );
    printf("Looping for a while...\n");
    oldpri = getpri(0);
    while ( oldpri == getpri(0) ); /* while loop with an empty body */
    printf("Now my priority is %d.\n",getpri(0) );
    printf("The priority of my parent is %d.\n",getpri(getppid()));
    printf("Attempting to set my priority...\n");
    if ( getuid()==0 || geteuid()==0 )
    {
        setpri(0,30);
        printf("My priority is now %d.\n",getpri(0));
        for(i=1;i<=1000;i++);
        printf("My priority is still %d.\n",getpri(0));
        exit();
    }
    printf("You do not have authority to set your priority!\n");
    exit(1);
}
```

Figure 8-16. Process Scheduling Example (1 of 2)

AU253.0

Notes:

The sample program in Figure 8-16 demonstrates the use of the `getpri()` and `setpri()` system calls.

Process Scheduling Example (2 of 2)

Possible program output:

```
My pid is 3227.  
My current priority is 66.  
Looping for a while...  
Now my priority is 67.  
The priority of my parent is 68.  
Attempting to set my priority...  
My priority is now 30.  
My priority is still 30.
```

Figure 8-17. Process Scheduling Example (2 of 2)

AU253.0

Notes:

Here is a possible output from the example program.

8.2 Creating New Processes

Creating a Process

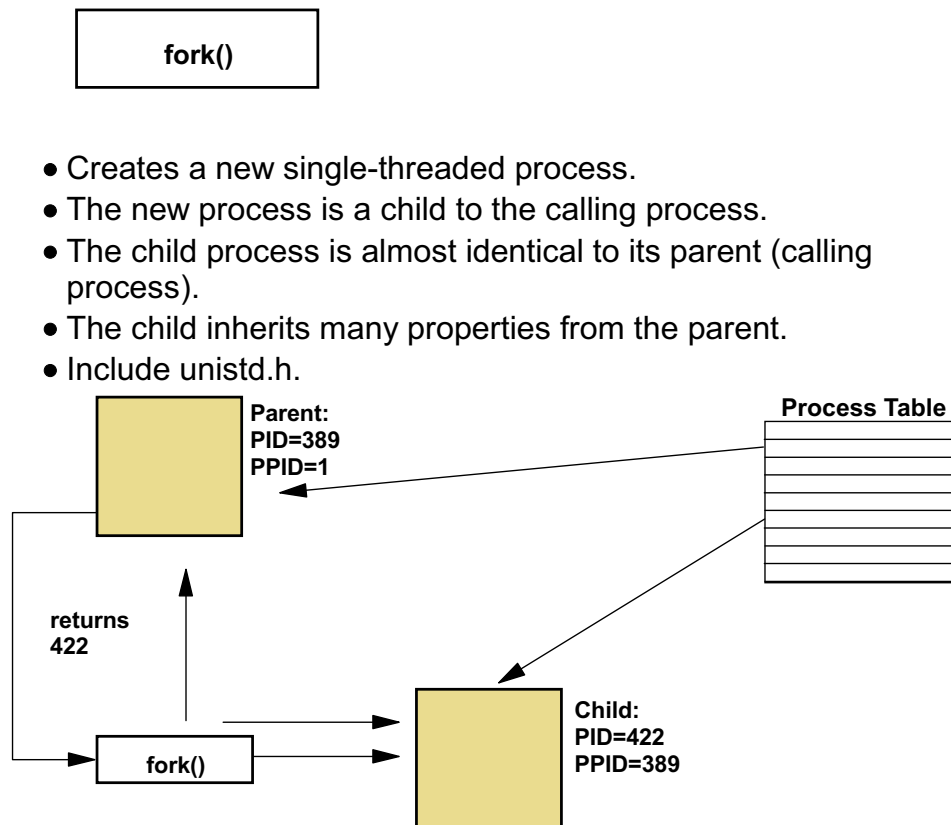


Figure 8-18. Creating a Process

AU253.0

Notes:

All processes, except PID 0 and PID 1, are created by other processes. The creating process is called the parent and the created process is called the child. The parent process creates the child process by issuing the **fork()** system call.

The **fork()** system call creates a child process that is almost an exact duplicate of the parent process. Some of the differences between the parent and child are the process ID number (PID), the parent process ID (PPID), and certain accounting values.

The **fork()** system call allocates a slot for the new process in the kernel's process table.

Because the child process is an almost exact duplicate of the parent, both processes are running the `fork()` system call when each gets scheduled to run. The `fork()` system call returns a zero to the child process and the PID of the child process to the parent. This way, each process can determine what to do next by evaluating the return value.

The **fork()** system call will fail, for example, in the following cases:

- If the caller has reached the maximum number of allowed processes for the user ID.
- When there is not enough paging space left for the process.

The default number of maximum processes per user is 128, but that is a tunable parameter (via SMIT; use the Change/Show Characteristics of the Operating System menu).

One important point to remember is that the child process inherits the open files of the parent.

The child process inherits the following from the parent process:

1. Environment
2. Close-on-exec flags
3. Signal handling (Block, Ignore, and so on)
4. SETUID and SETGID
5. Trusted State
6. Profiling on/off status
7. Nice value
8. All attached shared libraries
9. Process GID
10. tty GID
11. Current Directory
12. Root Directory
13. umask and filesize limit
14. Attached shmat segs
15. Audit Flags
16. Debugger process ID
17. Binding to processor

The child process differs from the parent process in the following:

1. One user thread (one that called the fork)
2. Unique PID and PPID
3. Different file descriptors (FDs), but common file pointer
4. Process locks
5. itime, stime, cutime, and cstime=0
6. Alarms cleared
7. Signals initialized (that were pending)
8. trace flags reset
9. Own copy of message catalog

Process Life Cycle

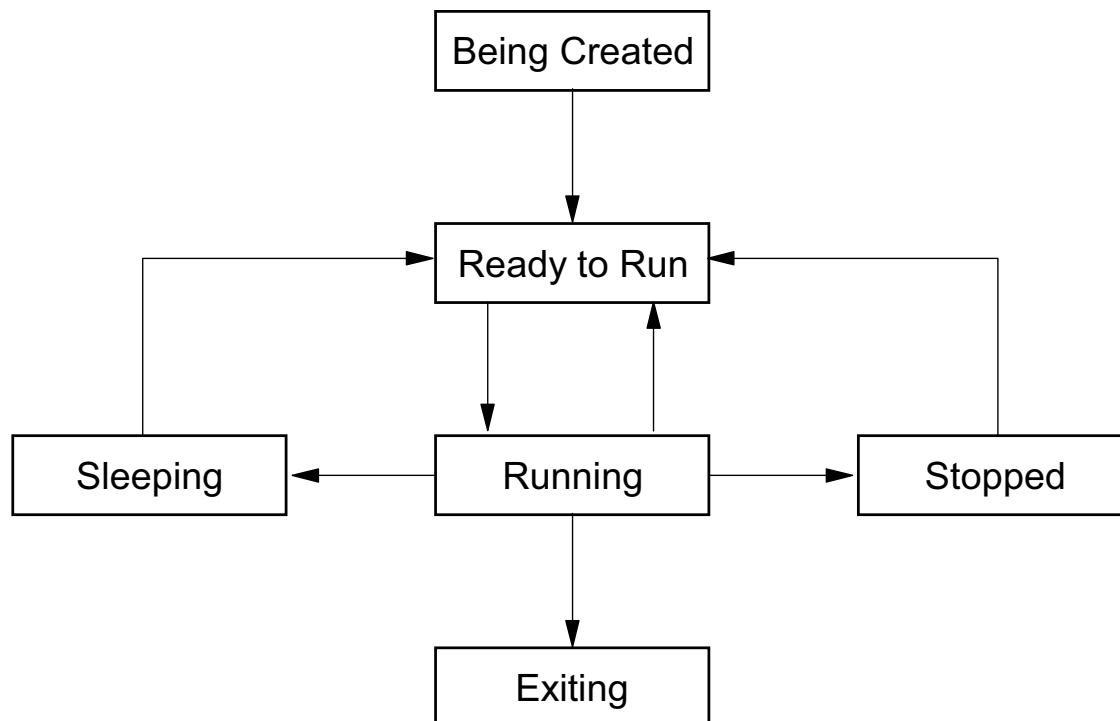


Figure 8-19. Process Life Cycle

AU253.0

Notes:

Figure 8-19 illustrates the life cycle of a process.

During the `fork()` system call, when a Process Table slot has been allocated for the new process, but the new process has not been created in memory, the state of the process is said to be in the state of being created.

All processes that are waiting to use the CPU and not waiting on any other system resource are said to be ready to run. In other words, they could run if given a chance.

Only one process can run at a time. The process that currently controls the CPU is called the running process.

When a process requests the services of a system resource, there is usually a delay. For example, when a process requests input from a disk or the keyboard, it will always take some amount of time for the input to become available. Instead of holding the CPU during this time, the process will move to a sleeping state, waiting for the desired event to occur. When the running process goes to sleep, another process can take control of the CPU.

Sleeping processes receive a signal when the event they are waiting on occurs. When the process wakes, it must contend for the CPU with other ready-to-run processes.

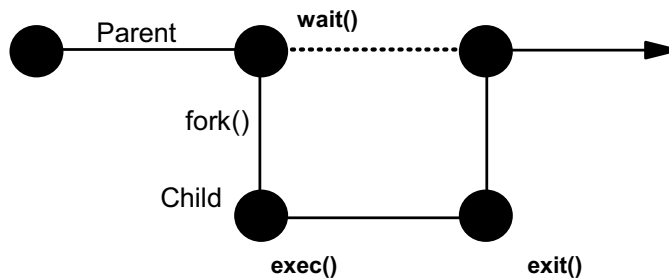
A running process may be stopped. For instance, the C and Korn shells support the ability to stop a foreground or background process as part of their job control features. A SIGSTOP or SIGTSTP signal is used to stop a running process. A SIGCONT signal will place the stopped process back on the ready-to-run queue.

When a process terminates, it moves to the exiting state (also called zombie state). The child issues the `exit()` POSIX system call, which sends a SIGCHLD signal to the parent process. The `exit()` system call also places a small amount of data in the Process Table so that the parent process can evaluate how the child process terminated.

The state of a process (or thread) may be seen by using the `l` flag with the **ps** command.

Process Life Examples

Example of foreground process:



Example of background process:

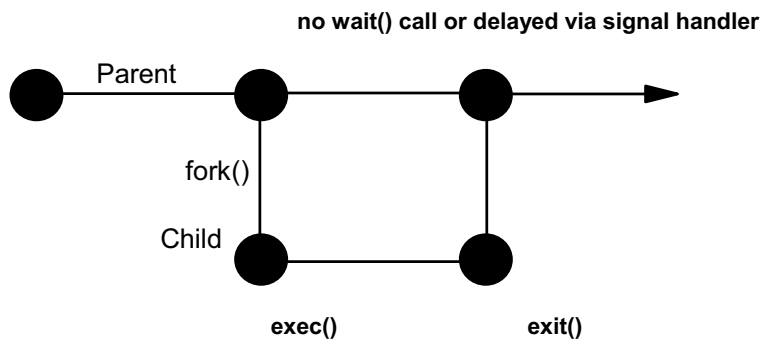


Figure 8-20. Process Life Examples

AU253.0

Notes:

Figure 8-20 illustrates examples of foreground and background processes. Processes that are started and may require user interaction are commonly known as foreground processes. Programs and commands are run as foreground processes by default. Processes that can run independently of a user are known as background processes. Here, the prompt comes back immediately, as there is no waiting on the conclusion of the process.

Also, note the differences between the system calls `fork()`, `exec()`, `wait()`, and `exit()`. The `fork()` system call creates a child process. The `exec()` system call overlays the existing process with another program. The `wait()` system call allows the parent process to wait for a child process to terminate. The `exit()` system call terminates the process.

These system calls are discussed later.

The exec System Calls (1 of 4)

- Various POSIX calls are used to replace the text of a calling process with text from another program - used with `fork()`.
- The call used depends on requirements and preferences:

<code>execl()</code>	<code>execv()</code>
<code>execle()</code>	<code>execve()</code>
<code>execlp()</code>	<code>execvp()</code>

- The "l" family passes arguments to the new executable via a list, while the "v" family passes arguments via pointers.
- The "e" family (ending with "e") includes the ability to pass environmental variables.
- The "p" family allows the use of PATH in searching for the file and the program may be a shell script instead of an executable.
- Includes `unistd.h`.

Figure 8-21. The exec System Calls (1 of 4)

AU253.0

Notes:

Upon completion of the **fork()** system call, both the parent and the child processes are running the same code. This is often not what is wanted. Another system call is used to load and execute a new program in the process. The **exec** family of system calls is used to replace the calling process' text and data regions with new program code and data.

There are actually six different calls. They are described on the following pages.

Even though all six **exec** routines are generally called system calls, **execve()** is the only actual system call. The others are library routines that transform their parameters into the form required by the **execve()** system call and then call **execve()**. The **exec** call fails for several reasons, some of them being due to wrong number of parameters, incorrect permissions for the program being executed and so on. Refer to AIX online documentation for more information on this one.

The POSIX call **getenv()** is also useful. For example:

```
main()
{
```

```
char * value;  
value=getenv ("HOME");  
printf("Home directory is %s\n", value);  
}
```

This call may be used to gather information about the environment that may be passed as arguments to an exec call.

The exec System Calls (2 of 4)

```
int execl(path, arg0 [, arg... ] , 0 )
```

- Path is the relative or absolute path name (or a pointer to such) of the file to execute.
- arg0 (required) is the first argument to the new executable and is usually the name of the program.
- Additional arguments can be passed in a comma separated list.
- A zero, or NULL, is required to terminate the list of arguments.
- If successful, there is no return from any exec call, because the calling program is replaced.
- Returns -1 upon failure.

```
int execlp(file, arg0 [arg1... ] , 0 )
```

- Same as execl, except that it uses a file name and the PATH environment variable.
- "file" can be a shell program instead of an executable.

Figure 8-22. The exec System Calls (2 of 4)

AU253.0

Notes:

execlp is the same as execl, except that it uses a file name without a full path and the file can be a shell script. NULL is defined in stdio.h and stdlib.h to be zero.

Syntax:

```
int execl (const char * path, const char * arg0 [, const char
* arg... ] , 0 )
```

```
int execlp(const char * file, const char * arg0 [ const char *
arg1... ] , 0 )
```

For example:

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

```
execl("/bin/cat", "cat", "file1", "file2", NULL);
```

For example:

```
#include <unistd.h>

.
execlp("ls", "ls", "-l", "/home/kelly/", NULL);
```

The exec System Calls (3 of 4)

```
int execl(const char *path, char * const argv)
```

- Path is the relative or absolute path name (or a pointer to such) of the file to execute.
- argv is an array of pointers to null-terminated character strings that make up the argument list.
- Returns -1 upon failure, nothing if successful.

```
int execve(const char *path, char *const argv, char *const envp)
```

- Path is the relative or absolute path name (or a pointer to such) of the file to execute.
- argv is an array of pointers to null-terminated character strings that make up the argument list.
- envp is an array of pointers to null-terminated character strings that make up the list of environmental variables.
- Returns -1 upon failure, nothing if successful.

Figure 8-23. The exec System Calls (3 of 4)

AU253.0

Notes:

execve is the same as execl, except that a third argument is used as a pointer to a list of environment variables.

For example:

```
#include <unistd.h>
.
.
.
static char *args[]={
    "cat",
    "file1",
    "file2",
    NULL
};
execl ("/bin/cat", args);
```

For example:

```
.  
.   
.   
static char *env[]={  
    "PATH=/usr/bin:/etc:/usr/sbin:/sbin:.",  
    NULL  
};  
static char *args[]={  
    "cat",  
    "file1",  
    "file2",  
    NULL  
};  
execve ("/bin/cat", args, env);
```

Here is a complete example:

```
main(int argc, char *argv[])  
{  
    int pid;  
    char arg0[] = "execpgm";  
    char arg1[] = "val_of_arg1";  
    char *myargv[3] ;  
    myargv[0] = arg0;  
    myargv[1] = arg1;  
    myargv[2] = 0;  
    if (printf("Before exec, my PID is who knows what\n") == -1)  
        perror("problems in first printf");  
    pid = fork();  
    if (pid == -1) {  
        perror("funtions failed, ret = %d\n", pid);  
        exit(1);  
    }  
    if (pid == 0) {  
        execv("execpgm", myargv);  
    }  
    printf("while execpgm runs, here is the parent \n");  
}
```

Note that the argv and envp arguments to the system calls always end with a NULL.

The exec System Calls (4 of 4)

```
int execvp(const char *file, char * const argv)
```

- file is the name (or a pointer to such) of the file to execute.
- If the file parameter is not a full path name, the directories in the PATH environmental variable is searched.
- argv is an array of pointers to null-terminated character strings that make up the argument list.
- Returns -1 upon failure, nothing if successful.

```
int execl(char *path, char *arg0, ..., char * const envp)
```

- path is the path of the file to execute, either absolute or relative.
- arg0 is the first argument, usually the name of the program.
- envp is the pointer to the environment variables.
- Returns -1 upon failure, nothing if successful.

Figure 8-24. The exec System Calls (4 of 4)

AU253.0

Notes:

execvp is the same as the execv call, except that you use a file name instead of the full path, and it can be a shell script.

For example:

```
#include <unistd.h>
.
.
.
static char *args[]={
    "cat",
    "file1",
    "file2",
    NULL
};
execvp ("cat", args);
```

`execle()` is the same as the `execl()` system call, except that it is passed an additional argument that consists of the environment variables.

The exit() System Call

exit(status)

Called by the terminating process:

Explicitly (that is, when error detected)

Implicitly, when closing brace of main() reached

Implicitly, when process terminated by signal

Status is the exit status returned to the parent process.

exit() sends a SIGCHLD SIGNAL (20) to the parent process.

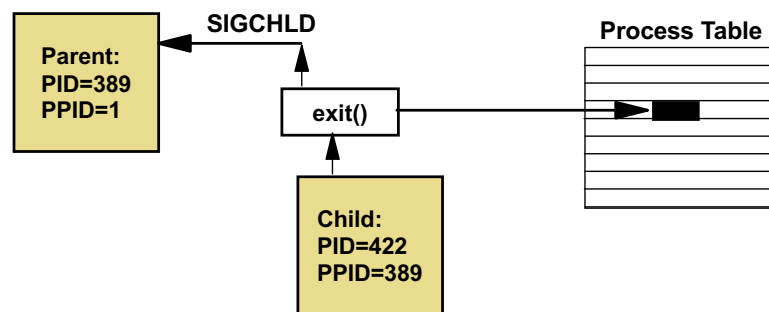


Figure 8-25. The exit() System Call

AU253.0

Notes:

A process terminates by issuing the exit() system call.

This call performs many actions. It sends a SIGCHLD signal to the parent process, closes all opened files, places the exit status information in the Process Table, and releases all resources held by the calling process.

The EXIT_SUCCESS and EXIT_FAILURE macros, defined in stdlib.h, may be used to indicate success or failure. EXIT_SUCCESS is set to 0 and FAILURE to 1. An errno may also be returned.

Strictly speaking, exit() is actually a library routine. It performs a variety of library related cleanup tasks and then calls the _exit() system call to actually terminate the process. One of the key cleanup tasks handled by the exit() library routine is the flushing and closing of all stdio buffers (that is, buffers associated with FILE * files). The programmer can use the atexit() library routine to specify additional cleanup routines that are to be called by the exit() library routine (refer to, "man atexit" or the AIX 5L online documentation for more information on atexit()).

The `exit()` library routine can be called explicitly within the user program and is called implicitly when the `main()` routine returns to its caller. On the other hand, if a process is terminated by a signal, then only the `_exit()` system call is called. Any buffered data in FILE * buffers is lost, and none of the cleanup routines requested using `atexit()` are called.

It should also be noted that `exit()` is often called a "system call", even though it is actually just a library routine. Fortunately, the distinction is usually moot. In keeping with this quite common convention, this course generally refers to the `exit()` system call.

For example:

```
#include <stdio.h>
#include <stdlib.h>
main ()
{
    if (fork() == 0)
        printf ("child\n");
    else
        printf ("parent\n");
    exit (EXIT_SUCCESS);
}
```

The output is:

```
parent
child
```

The wait() System Call

```
pid_t wait(int * statusloc)
```

- Called by the parent process to wait for the termination of the child process (responds to SIGCHLD).
- statusloc is a pointer to an int, where the exit status of child process can be stored and queried.
- Returns pid of a terminated child process.
- Clears the Process Table entry for the child process (removes defunct process).
- Included <sys/wait.h> for use of options described below.

```
pid_t waitpid(pid_t pid, int *statusloc, int options)
```

- Allows specification of the pid for which to wait.
- Options for waitpid() call include:
 - WNOHANG - Calling process does not wait if there are no terminated child processes.
 - WUNTRACE - Returns information about a child process stopped by SIGTTIN, SIGTTOU, SIGSSTP, and SIGTSTOP signals.

Figure 8-26. The wait() System Call

AU253.0

Notes:

The parent process can retrieve the exit status information about a terminated child process by issuing the wait() system call. The wait() system call also clears the zombie state Process Table slot. Normally, when a parent process issues the wait() system call, the parent process then sleeps until a SIGCHLD signal is sent from a terminating child process. By default, if wait() is not used, the parent ignores the child's signals and the child stays a zombie until init inherits it upon the death of the parent process.

wait3() is an AIX call, similar to waitpid, that returns resource usage.

For example:

```
#include <stdio.h>
#include <stdlib.h>
main ()
{
    if (fork() == 0)
        printf ("child\n");
    else {
```

```
        wait (NULL);  
        printf ("parent\n");  
    }  
    exit (EXIT_SUCCESS);  
}
```

The output is:

```
child  
parent
```

(Note the reversed order, due to wait)

Getting Status from Child

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
int status;
main()
{
    int pid;
    pid = fork();
    if (pid == -1) {
        perror("problems with fork"); exit(1);
    }
    if (pid == 0) { /* Child Code */
        execlp("ls", "ls", "-l", NULL);
        perror("problems with finding ls"); exit(1);
    }
    /* Parent continues */
    printf("Parent continues and does some code...\n");
    printf("Parent finishes and wants to find out ");
    printf("how child did....\n");
    wait(&status);
    if (WIFEXITED(status))
        printf("Child ended normally and status was %d\n",
            WEXITSTATUS(status));
    if (WIFSIGNALED(status))
        printf("child was terminated and signal was %d\n",
            WTERMSIG(status));
    if (WIFSTOPPED(status))
        printf("child stopped with stop signal of %d\n",
            WSTOPSIG(status));
    exit(EXIT_SUCCESS);
}
```

Figure 8-27. Getting Status from Child

AU253.0

Notes:

The WIF macros should be used to analyze the status from the child. There are three reasons a child can return information on a wait call:

- The child process exited normally.
- The child process was terminated by a signal.
- The child process was stopped.

Information is kept in different places in the status variable, depending on the type of termination of the child, and requires some bit manipulation to retrieve the status information you may be interested in. Hence, the WIF macros provide a portable way to retrieve your information.

The WIF macros are:

- WIFSTOPPED(ReturnedValue)
pid_t ReturnedValue;

Returns a nonzero value if the status returned for a child process is stopped.

- `int WSTOPSIG(ReturnedValue)`
`pid_t ReturnedValue;`
Returns the number of the signal that caused the child to stop.
- `WIFEXITED(ReturnedValue)`
`pid_t ReturnedValue;`
Returns a nonzero value if status returned for normal termination.
- `int WEXITSTATUS(ReturnedValue)`
`pid_t ReturnedValue;`
Returns the low-order 8 bits of the child exit status.
- `WIFSIGNALED(ReturnedValue)`
`pid_t ReturnedValue;`
Returns a nonzero value if status returned for abnormal termination.
- `int WTERMSIG(ReturnedValue)`
`pid_t ReturnedValue;`
Returns the number of the signal that caused the child to terminate.

Effects of waitpid() and wait()

WAITPID		
Condition of child	What happens on waitpid if 3rd parm of waitpid is 0	What happens on waitpid if third parm of waitpid is WNOHANG
No child has ended at the time of the waitpid call.	Parent will wait until one child finishes.	waitpid call returns a 0.
A child has ended and the waitpid call has not been issued for this child yet.	waitpid call returns the process ID of the child.	waitpid call returns the process ID of the child.
The waitpid call was already issued for all children.	waitpid call returns -1.	waitpid call returns -1.

WAIT	
Condition of child	What will happen with wait call
At least one child is still running and not finished yet.	Parent will wait until one child finishes.
A child has ended and the wait call has not been issued for this child yet.	wait call returns the process ID of one of the children that ended.
The wait call was already issued for all children.	wait call returns -1.

Check condition of child via statuslogic

Figure 8-28. Effects of waitpid() and wait()

AU253.0

Notes:

Figure 8-28 demonstrates the effects of waitpid() and wait(), depending on the value of the third parameter in waitpid() and the condition of the child process.

If you do plan to wait on multiple child processes, it can get quite complicated. Hence, you can let the process init handle your child processes if you do not care about their return codes. To do this, you must explicitly choose to ignore your child processes' SIGCHLD signals as follows:

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

main()
{
    signal(SIGCHLD, SIG_IGN);
    if (fork() == 0) {
```

```
    execl("/home/val/childprog", "childprog", NULL);  
    perror("exec failed");  
    exit(1);  
}  
/* parent continues and does no wait calls.... */  
}
```


8.3 Examples of Process Life Cycle

Process Creation Example (1 of 2)

```
/* sample2.c */

#include <stdio.h>
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>

void handle_it();
int status;
main()
{
    signal(SIGCHLD,handle_it);
    if(fork()==0) { /*We are the child... */
        execl("/usr/bin/ls", "ls",NULL);
        perror("exec failed");
        exit(1);
    }
    while(1);
}

void handle_it()
{
    wait(&status);
    printf("Exit status of child = %d\n",status);
    exit(0);
}
```

Figure 8-29. Process Creation Example (1 of 2)

AU253.0

Notes:

Here is an example of how to properly handle the process life cycle system calls discussed in this lecture.

The `signal()` is a system call and is used to define a signal handler (this is discussed later). The `handle_it()` subroutine is the signal handler. It is invoked when the `SIGCHLD` signal is received by the parent, indicating that the child process has terminated.

The child process is created with a call to the `fork()` system call. The parent enters into an infinite loop. Once the child terminates, the signal handler checks for the status of the child process and exits.

Process Creation Example (2 of 2)

```
void handle_it()
{
    int rv=1; /* used to track return from waitpid */
    signal(SIGCHLD,handle_it); /* reregister SH */
    while (rv > 0 ) /* waitpid returns zero if no zombies */
        rv = waitpid(0,0,WNOHANG);
}
```

Figure 8-30. Process Creation Example (2 of 2)

AU253.0

Notes:

The return code from `waitpid()` may be used to check for zombies. `waitpid()` will return a value of zero if there are no stopped or exited child processes, and `WNOHANG` is specified. The following table illustrates the possible values for `rv`, depending on whether the option is set to `WNOHANG` or zero.

Child Process State	Not ended	Just ended	Already ended
option=0	parent waits	rv=PID of child	rv=-1
option=WNOHANG	rv=0 rv=0	rv=PID of child	rv=-1

This example only shows how to get rid of zombie children if you do not want to take care of them. Although this is easy, it will not give you their return codes. To implement this scenario takes much more effort. Please refer to the `waitpid` table in Figure 8-28 on page 47 for assistance in creating your code. If this becomes too much of a hassle, then you may decide to let `init` take care of your children. Again, refer to the same page for comments.

Unit Summary

- Process Management System Calls
- Process Scheduling Control
- Process Creation and Termination
- Process Life Cycle

Figure 8-31. Unit Summary

AU253.0

Notes:

Unit 9. AIX Signal Management

What This Unit is About

This unit provides detailed information and examples about signal management techniques in AIX. Traditional UNIX has had many approaches to handling signals. AT&T System V-based systems use one set of system calls, while Berkeley-based systems use another set of calls. AIX combines both sets of calls. This leads to easier porting of code to AIX, but can lead to some confusion, as there are many calls that do the same thing.

This unit tries to keep things simple by introducing signals, how they are sent from one process to another, and how they can be handled by the receiving process.

What You Should Be Able to Do

After completing this unit, you should be able to:

- List and describe several AIX signals
- Use system calls to send signals
- Use signal handlers to ignore, block, and catch incoming signals

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 7

References

- AIX 5L online documentation

Unit Objectives

After completing this unit, you should be able to:

- List and describe several AIX signals
- Use system calls to send signals
- Use signal handlers to ignore, block, and catch incoming signals

Figure 9-1. Unit Objectives

AU253.0

Notes:

9.1 AIX Signal Management

Signals (1 of 2)

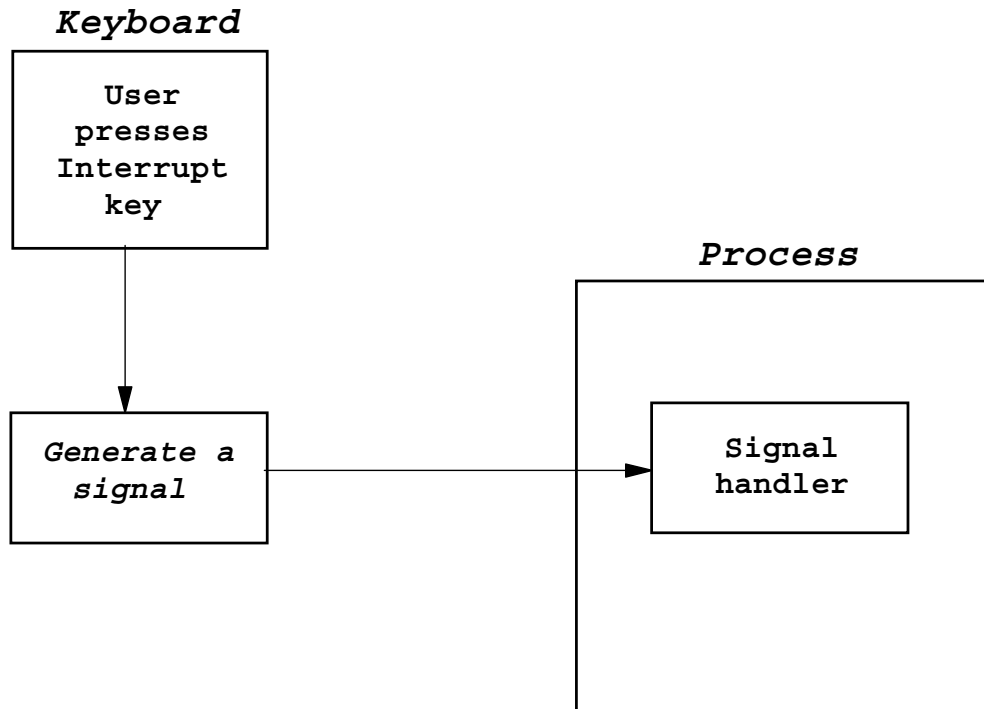


Figure 9-2. Signals (1 of 2)

AU253.0

Notes:

AIX processes can receive signals. Signals notify a process of a particular event or cause the process to take a specified action.

Figure 9-2 is a basic illustration of signal handling. There are more ways in which signals may be generated.

Signals are handled by a signal handler, which is a part of the process. Sometimes the process can be instructed by the executing program to ignore the signal or to take an action that is not normally taken when the signal is received.

Signals (2 of 2)

- Processes receive signals from:
 - Hardware (that is, the <Ctrl> key).
 - Operating system (exceptions caused a program request to do something illegal).
 - Other processes (primitive form of interprocess communication).
 - The usual default action is the termination of receiving process.
- Processes can choose to ignore or handle most signals, and there are many programming options available.
- A process can send a signal to any other process with the same user ID.
 - A process with root user ID authority can send signals to any process.
 - AIX supports up to 64 signals. although only 40 are currently defined.

Figure 9-3. Signals (2 of 2)

AU253.0

Notes:

Signals are sent to a process in one of four different ways:

- The signal is sent to a process from the hardware via the device driver interrupt handler. An example would be when a user presses the "Interrupt" key (<Ctrl>C in AIX).
- The signal is sent to a process by the kernel. When a user logs off, a particular signal is sent to all processes running in that terminal session. (Processes that were started with the AIX **nohup** command will ignore this type of signal.)
- The signal is sent to a process by the kernel on behalf of an exception handler. Exception handlers are portions of kernel code used to handle errors caused by a running program. Such errors would include segmentation violation.
- The signal is sent to a process by another process. The sending process must have the same effective user ID as the receiving process, unless the sending process' effective user ID is 0 (root).

The action usually taken by the receiving process is to terminate execution. This is typical of signals sent by logging off, pressing the interrupt key, or reacting to an exception.

AIX offers many ways for the programmer to ignore, block, or catch incoming signals.

Ignoring a signal means that the receiving process is not made aware of the fact that the signal was received.

Blocking a signal means that it is temporarily ignored by the receiving process, but later delivered when the process stops blocking it. This is useful during execution of portions of code where a process should not be interrupted.

Catching a signal means that the programmer has specified some non-default action to occur when the signal is received. Signal catching is accomplished by registering a signal handler. This is similar to using the **trap** command in Bourne and Korn shell programming.

Some signals cannot be ignored, blocked, or caught.

Examples of Signals

Here are some AIX signals (from `/usr/include/sys/signal.h`):

Sig. Num.	Sig. Name	Default Action
1	SIGHUP	Hangup, generated when terminal disconnects
2	SIGINT	Interrupt, <Ctrl>C from keyboard
3	SIGQUIT	Quit, <Ctrl>\ from keyboard (core dumps)
9	SIGKILL	Kill, can not be trapped or ignored (kill -9 command)
11	SIGSEGV	Segmentation violation detected (core dumps)
15	SIGTERM	Software terminate (default for shell kill command)
30	SIGUSR1	First user-definable signal
31	SIGUSR2	Second user-definable signal

Figure 9-4. Examples of Signals

AU253.0

Notes:

AIX supports 64 signals, POSIX defines less, and standard C only defines a subset of POSIX. To see the signals that are defined, look at the header file `/usr/include/sys/signal.h`. You will also want to include this header file in a program that uses signals.

The chart in Figure 9-4 lists some of the more commonly-used signals. Each signal has a number and a symbolic name. Programs using signal-oriented system calls can use either the numeric value or symbolic name for the signals if the `/usr/include/sys/signal.h` file is included.

The default action for all of these signals is to terminate the process. Those noted as "core dumps" also generate a core file.

As mentioned earlier, some signals cannot be ignored, blocked, or caught. A good example is the SIGKILL signal (number 9). You may be familiar with the AIX command **kill**, which, by default, sends a signal 15 (SIGTERM). Sometimes a process is running a program that ignores the SIGTERM signal. That is when you issue the **kill -9** command. The -9 option sends a signal number 9 (SIGKILL) to the desired process. This is what is sometimes called a "sure kill", because a SIGKILL signal cannot be ignored, blocked, or caught. The

header file `/usr/include/sys/signal.h` is referenced by an `#include` statement in `/usr/include/signal.h`.

There are two user-definable signals available. These are generally referred to as `SIGUSR1` and `SIGUSR2`.

Sending Signals

```
int kill(pid_t pid, int signal)
```

- Sends *signal* (integer) to *pid* (integer)
- *pid* is the desired process (integer)
- Returns 0 upon success, -1 upon failure
- Must include <signal.h> to use symbolic names of signals
- Include file <sys/types.h> to typedef arguments as pid_t.

```
killpg(int pgrp, int signal)
```

non-POSIX

- Sends *signal* (integer) to process group *pgrp* (integer)
- All other characteristics like kill()

Figure 9-5. Sending Signals

AU253.0

Notes:

The kill() system call sends the specified signal to the specified process. A process with root authority (effective user ID of 0) can send signals to any process. Otherwise, the sending process must have the same effective user ID as the receiving process in order for the signal to be delivered. The signal is delivered by the kernel to the Process Table slot for the receiving process.

Special process ID parameter values exist. A PID value of zero sends to all processes within the sender's process group (except PID 0 and PID 1).

A process ID parameter of -1 sends the signal to all processes, providing that the sender is root. (If the sender is not root, it is sent to all processes whose real UID is the same as the effective UID of the caller.

Any other negative value used for the process ID parameter causes the signal to be delivered to the process group whose ID is the absolute value of the parameter. In other words, the negative sign is ignored.

A special function of the `kill()` system call is the ability to send a signal 0 (zero). While a signal zero does not exist, the `kill()` system call will return a zero if the specified PID exists. Error checking is performed to see if the specified process exists. This is a tricky way of finding out if a particular process exists. This is like using signal zero as a dummy signal.

The `killpg()` system call sends the specified signal to all processes within the desired process group.

9.2 Handling Signals

Signal Actions

Processes can elect to take one of the following actions for each received signal:

- SIG_IGN (ignore the signal)
- SIG_DFL (take the default action for the signal)
- "Catch" the signal and call a routine (called a signal handler)

Note: Some signals, such as SIGKILL and SIGSTOP, cannot be ignored or caught.

Figure 9-6. Signal Actions

AU253.0

Notes:

Figure 9-6 reemphasizes the signal actions discussed earlier. Note the use of the symbolic constants for ignoring and taking the default action for signals.

Signal Handler

```
void signal(int signal, void(*) (int) action)
```

- Registers a signal handler.
- Performs action when a signal (integer) is received.
- The action can be SIG_IGN, SIG_DFL, or a label name for a routine to execute.
- Returns the value of previous signal action upon success, -1 upon failure.
- Must include <signal.h> to use symbolic names for signals.
- Both System V and BSD version of signal() are available.
- System V does not automatically reset the handler when the signal is caught.
- BSD does automatically reset the handler.

Figure 9-7. Signal Handler

AU253.0

Notes:

The signal() system call is used to instruct a program on how to handle specific signals.

The parameters of this call include the specific signal (either numeric value or symbolic constant) and the action to take. Actions include:

- SIG_IGN - Ignore the signal.
- SIG_DFL - To handle the signal in the normal fashion. (Default Action)
- routine - Names a subroutine to run when the signal is received. This is how a signal handler is registered.

System V signal handlers are cancelled (that is, the action reverts to the default for the signal) once the handler has been invoked once. If the programmer wants to be able to catch a series of signals, then the programmer must reestablish the handler each time (this is typically done inside the signal handler).

BSD signal handlers remain in effect until changed by another call to signal or sigaction. We will be discussing sigaction shortly.

A program linked with -lbsd will run with BSD-style signal handlers.

Be careful here: the term "reset" is sometimes used to mean that the signal handler only works once (that is, System V style) and is sometimes used to mean that the signal handler is reestablished each time it is used (that is, BSD style).

For example:

```
#include <signal.h>
main()
{
    signal (SIGINT, SIG_IGN);
    /* do anything */
    signal (SIGINT, SIG_DFL);
    /* do something else */
}
```

Signal Handler Example

```
#include <signal.h>

...

main()
{

    void handle_it();
    signal(SIGINT,handle_it);

    ...

}

void handle_it()
{

    signal(SIGINT,handle_it); /* System V version! */
    printf("You have asked to terminate this program.\n");

    printf("Please wait while I clean up a few things...\n");

    ... /* Code to remove temporary files, etc */

    exit(0);
}
```

Figure 9-8. Signal Handler Example

AU253.0

Notes:

Figure 9-8 illustrates how a signal handler routine is registered with the `signal()` system call. In this example, when a `SIGINT` signal (`<Ctrl>C`) is received, the program execution branches to the `handle_it` subroutine.

Notice that the first instruction performed within the `handle_it` subroutine is to reregister the handler. This is because the `signal()` system call resets the signal action to `SIG_DFL` when the handler is invoked.

Also note that another `SIGINT` might arrive before `handle_it` calls `signal()` to reregister the handler. If this happens, the program will terminate, because the default response to a `SIGINT` is termination.

The sigaction Call

```
int sigaction(int sigaction* signal, struct
sigaction* action, struct sigaction* oaction)
```

- Registers a signal handler or queries the current handler.
- The action is a pointer to a sigaction structure, and is used to define a new action to perform when a signal received.
- oaction is a pointer to a sigaction structure, where the current handler state (old action) is placed when sigaction() is called.
- If the action is NULL, no new handler action is defined for signal.
- Returns 0 upon success, -1 upon failure.
- Must include <signal.h> to use symbolic names for signals.
- sigaction structure:

```
void          (*sa_handler) ();
sigset_t      sa_mask;
int           sa_flags;
```

Figure 9-9. The sigaction Call

AU253.0

Notes:

A more reliable method of handling signals is to use the sigaction() system call. This call provides the ability to register a signal handler or query the state of one.

The main difference between the signal() system call and the sigaction() system call is that the action parameter points to a sigaction structure. The sigaction structure has fields for a pointer to the desired function, a signal mask, and special flags. Include the header file /usr/include/signal.h to prototype the sigaction structure.

The signal mask allows the handler to block specific signals while the handle code is running.

The sa_ flags include:

- SA_OLDSTYLE- Tells the handler to reset the signal to its default action before the handler is invoked. This means that the handler will act like the older signal() call. Otherwise, the signal handler is automatically reregistered by the sigaction() call.

- **SA_RESTART** - Allows the process to restart a specific call that may have been running when a signal was received. Refer to the AIX online documentation for a list of calls this option supports.
- **SA_NOCLDSTOP** - When a child process terminates or is stopped, a SIGCHLD signal is sent to the parent. If the parent sets the SA_NOCLDSTOP flag with the sigaction() call, any SIGCHLD signals sent from a stopped child are ignored. Only SIGCHLD signals sent from a terminated child process are acted upon.

SA_RESTART and SA_OLDSTYLE are not defined in the POSIX standard.

sigaction Example

```
/* sigaction.c */
#include <stdio.h>
#include <signal.h>
#include <memory.h>
void handle_it();

main()
{
    struct sigaction mysig;
    memset (&mysig,0,sizeof(mysig));
    mysig.sa_handler = handle_it;
    mysig.sa_flags = SA_RESTART;
    sigfillset(&mysig.sa_mask);
    /* register signals using mysig structure */
    sigaction(SIGINT,&mysig,NULL);
    sigaction(SIGTERM,&mysig,NULL);
    while(1) {
        printf("My pid is %d.\n",getpid());
        sleep(1);
    }
}

void handle_it(int signo)
{
    printf("Received signal %d...Quitting\n", signo);
    exit(0);
}
```

Figure 9-10. sigaction Example

AU253.0

Notes:

The main() subroutine registers the signal handler to catch the signals, SIGINT and SIGTERM. To register a signal handler, the steps are as follows:

- a. Allocate a variable of type struct sigaction.
- b. Reset the contents of this variable. This is usually done with the memset() subroutine.
- c. Set the signal handler's function address and the type of handler in this sigaction variable.
- d. Use this sigaction variable to register the signal handler. This is accomplished using the sigaction() call.

The function handle_it() acts as the signal handler in this example.

A signal can be ignored by setting the signal handler to SIG_IGN instead of specifying an actual handler. In this example, it can be accomplished as:

```
mysig.sa_handler = SIG_IGN;
```

Signal Masks

```
int sigprocmask(int how, const sigset_t* set,
               sigset_t* oset)
```

- Used to examine or change the signal mask of blocked signals for the calling process.
- *how* indicates how the mask should be changed:
 - SIG_BLOCK - Results in the union of the current set and the signals pointed to by set.
 - SIG_UNBLOCK - Results in the complement of signals that intersect the signals pointed to by set.
 - SIG_SETMASK - Signal mask is set to mask pointed to by set.
- *set* specifies the signal set.
- *oset* specifies where to save the previous signal mask when changing it.
- Returns zero upon success, -1 upon failure.
- Must include <signal.h> to use symbolic names for signals.
- See also sigsetmask() and sigblock() (more general, BSD non_POSIX calls).

Figure 9-11. Signal Masks

AU253.0

Notes:

The sigprocmask() system call is used to set the mask for a signal or set of signals. The how parameter specifies what action to take:

- SIG_BLOCK - Temporarily holds signals for later delivery.
- SIG_UNBLOCK - Deliver the held signals now.
- SIG_SETMASK - Sets the current mask to the values pointed to by the set parameter.

The sigprocmask() call allows the process to impose a signal mask to temporarily block specific signals. Unlike ignoring the signals, blocked signals are delivered when the mask is lifted.

The set parameter is a pointer to a signal mask set. See the AIX online documentation for information on the sigemptyset(), sigaddset(), sigdelset(), sigfillset, and sigismember() system calls.

Example

A code snippet to set a signal mask is as follows:

```
include <signal.h>

...

sigset_t mysigmask;
sigemptyset(mysigmask);
sigaddset(mysigmask, SIGINT);
sigprocmask(SIG_SETMASK, &mysigmask, NULL);

...
```

The sigset_t typedef is a structure of two integers, thus 64 bits.

Example

A code snippet to block all signals is as follows:

```
sigset_t mymask;
sigfillset(&mymask);
sigprocmask(SIG_BLOCK, &mymask, NULL);
```


Unit Summary

- AIX Signals
- Sending Signals
- Signal Handling Actions
- Registering Signal Handlers

Figure 9-12. Unit Summary

AU253.0

Notes:

Unit 10. Interprocess Communication

What This Unit is About

This unit lays the groundwork for working with common AIX IPCs. It will briefly mention many simple IPCs that you are already aware of, and will concentrate on pipes, System V IPCs (shared memory, semaphores, and message queues), and will briefly introduce you to sockets.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Implement pipes to pass data from one process to another
- Implement shared memory to share common data
- Implement semaphores to synchronize process activities
- Implement message queues to handle more complex data transfer
- Understand basic socket design for inter-CPU data transfer

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 8

References

- *UNIX Network Programming*, W. Richard Stevens, published by Prentice Hall, ISBN 0-13-949876-1.

Unit Objectives

After completing this unit, you should be able to:

- Implement pipes to pass data from one process to another
- Implement shared memory to share common data
- Implement semaphores to synchronize process activities
- Implement message queues to handle more complex data transfer
- Understand basic socket design for inter-CPU data transfer

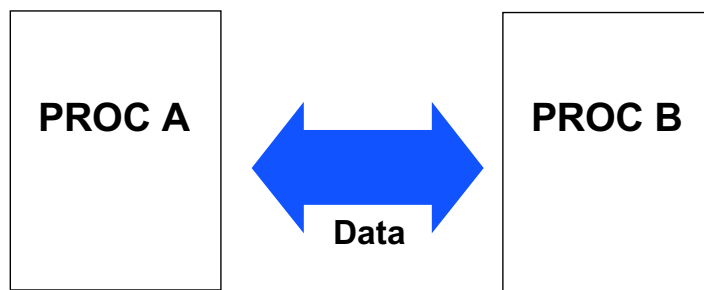
Figure 10-1. Unit Objectives

AU253.0

Notes:

10.1 Fundamentals of IPC

IPC Concepts



- IPCs (methods for Interprocess Communications) are available to allow processes to share information with other processes.
- Some IPCs allow two-way data flow and automatic synchronization.
- Generally, participating processes must be aware of the IPC mechanisms and programs be written to use them.

Figure 10-2. IPC Concepts

AU253.0

Notes:

Normally, UNIX processes exist without knowledge or interaction with other processes. Sometimes, however, it is desirable to have two or more processes share data. In UNIX, there are many options available for interprocess communication, or IPC.

Figure 10-2 introduces interprocess communication (IPC). Figure 10-2 points out two-way communications between two processes, but IPCs can involve communications between many processes and may be one-way or many-way (multiplexed). You may have heard of sockets, shared memory, semaphores, and message queues, but know very little about how to use them.

In order for processes to participate in IPC, the programs must be written to know about and use such IPC mechanisms. This unit describes these IPC mechanisms. POSIX has not set standards for IPC, so the calls discussed in this unit, though used widely, are all 'non-POSIX'.

Unofficial IPC Techniques

- **Disk Files** POSIX
 - Any processes can read from or write to files used to transfer data between processes.
 - These files persist.
 - Easy to implement.
 - Synchronization is difficult.
- **Exit Status** POSIX
 - Only from child to parent.
 - Child process must die to participate.
 - Simple to use.
- **Arguments on exec call** POSIX
 - One way/one time (parent to child).
 - Easy to implement.
- **Environment Variables** POSIX
 - One way/one time (parent to child).
 - Easy to implement.
- **Signals** POSIX
 - Only between processes with same UID.
 - Limited capabilities.
 - Relatively easy to use.

Figure 10-3. Unofficial IPC Techniques

AU253.0

Notes:

In the early days of UNIX, interprocess communication was limited to mechanisms such as the ones shown in Figure 10-3. Each of the forms of IPC listed in Figure 10-3 are available in AIX, but the use of each has its limitations.

Disk Files	Any process can be instructed to write to or read from a disk file. This provides a simple form of interprocess communication. Problems exist, such as how one process knows it is time to read after another process has written to the file. Also, because this form of IPC involves actual disk I/O, it is very slow.
Exit Status	When a child process terminates, it can send a single integer value back to its parent process. While this is a form of "unofficial" IPC, it is not practical, because the sending process must die to communicate with its parent process.
Arguments	This is a fairly easy way to pass information from parent process to child process as the child process is invoked. If the arguments are constant, the implementation is very easy. If they are variable

in number, then an `execv()` is required, which takes a little more effort. Remember that arguments are passed in character array format, regardless of the type of argument it is. Integer arguments must be translated back to integers in the child process before arithmetic can be performed. Remember that the child process retrieves this information by using the `argv[x]` array element (where `x` is the number of the argument you want).

Environment Variables These are easy to pass. Merely use `putenv()` in the parent process and `getenv()` in the child process. If you want to pass just a subset of your normal environment, consider using the `execve()` call.

Signals Signals are easy and well understood within the UNIX community. This form of IPC also has drawbacks. For instance, there is a limited set of signal meanings. They can only be sent to processes with the same UID, and the receiver does not know the identity of the sender.

10.2 Pipes

Pipes

- One-way flow
- 32 KB capacity
- Allows synchronization
- Uses file descriptors
- Peer-to-peer
- Client-to-server
- Unnamed - common ancestor
- Named - unrelated processes

Figure 10-4. Pipes

AU253.0

Notes:

Pipes, as the name suggests, are structures through which data flows. Data is written into the write end of a pipe and read from the read end. A pipe has a capacity of 32768 bytes (the capacity of a pipe varies between different variants of UNIX, although POSIX specifies that the capacity of a pipe must be at least 512 bytes).

A process that attempts to write more bytes into a pipe with a single request than can possibly fit in the pipe risks the possibility that the data in its request is interleaved with data from other write requests. On the other hand, a write request that could completely fit into the pipe if the pipe were empty is atomic (that is, the data will not be interleaved with data written by other processes).

Pipes can be used to communicate between peer processes or between processes in a client-server relationship. In a client-server relationship, the server typically reads from the pipe in a big while loop. If there's nothing in the pipe, it is suspended until data is available or until all the write ends of the pipe have been closed.

A very good example of pipes is the unix command:

```
# who | wc -l
```

Here, the output of **who** command goes to the write end of the pipe. The **wc** command reads from the read end of the pipe and processes the data. Pipes are a common usage for all UNIX users and shell programmers.

Unnamed Pipes

```
int pipe(int FileDescriptor[2])
```

- *FileDescriptor* is an integer array of two elements.
- Returns 0 on success and creates two file descriptors, -1 on failure.
- *FileDescriptor*[0] is for reading, *FileDescriptor*[1] is for writing.
- One read.
- Many writes.
- Child processes inherit both read and write.
- Close down the side you are not using.
- You must have read always open.

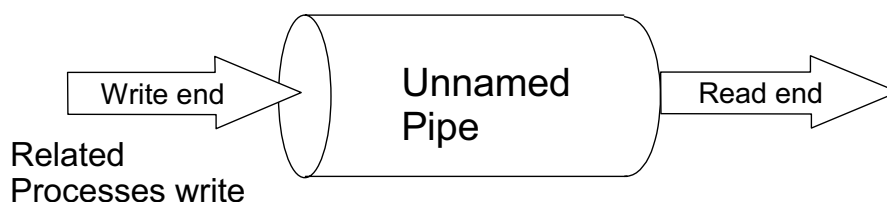


Figure 10-5. Unnamed Pipes

AU253.0

Notes:

When the pipe call finishes, the *FileDescriptor* array that passed on the pipe call will contain the read fd to the pipe (array element zero), and the write fd to the pipe (array element one). These will be used by anyone that wants to read or write to the pipe. Upon successful completion, the `pipe()` system call returns a 0 or -1 (with `errno` global variable set to the appropriate error number).

Typically, one process reads from the pipe and many can write to it.

Because all child processes inherit both FDs, convention says they should shut down the side they are not using.

If a write occurs and no process has the pipe open for read, then the write process will receive both the `SIGPIPE` signal and the `errno` will have the `EPIPE` error. Choose which strategy you plan to follow. At a bare minimum, you need to either ignore the `SIGPIPE` signal or write a signal handler routine for it.

Simple Unnamed Pipe Example

```

/* Program name - unnamed_pipe.c */

#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
    int p[2];
    int fd;
    char buf[80];
    pid_t pid;
    if (pipe(p) !=0){
        perror("pipe failed");
        exit(1);
    }

    if ((pid=fork()) == 0) {
        close(p[0]);
        sprintf(buf,"%d",getpid());
        write(p[1],buf,strlen(buf)+1);
        close(p[1]);
        exit (0);
    }
    close(p[1]);
    read(p[0],buf,sizeof(buf));
    printf("Child process said: %s\n",buf);
    close(p[0]);
    exit(0);
}

```

Figure 10-6. Simple Unnamed Pipe Example

AU253.0

Notes:

The pipe subroutine creates an interprocess channel called a pipe and returns two file descriptors: FileDescriptor(0) and FileDescriptor(1). FileDescriptor(0) is opened for reading and FileDescriptor(1) is opened for writing. A read operation on the FileDescriptor(0) parameter accesses the data written to the FileDescriptor(1) parameter on a first-in, first-out (FIFO) basis. **pipe** takes the address of an array of two integers, into which the new file descriptors are placed, as an argument.

In real life, the parent process would handle the read side of the pipe in a big while loop (that is, read, process, read, process). Also, the write side would insert some logic to handle SIGPIPE errors.

Here is an example of a more complete pipe example.

```

/* Program name - unnamed_pipe1.c */

#include <sys/signal.h>
#include <stdlib.h>
#include <fcntl.h>

```

```
int p[2];
char *whoami;

int main(void)
{
    int read_count;
    int pid;
    char parentbuf[25];
    char childbuf[25];

    struct sigaction mysig;
    sigset_t mymask;

    void end_pgm(int signo);

    memset(&mysig, 0, sizeof(mysig));
    mysig.sa_handler = end_pgm;
    sigfillset(&mymask);
    mysig.sa_mask = mymask;
    sigaction(SIGINT, &mysig, NULL);
    sigaction(SIGQUIT, &mysig, NULL);
    sigaction(SIGTERM, &mysig, NULL);

    pipe(p);
    pid = fork();
    if (pid == -1) {
        perror("parent: problems with fork"); exit(1);
    }
    if (pid == 0) {
        whoami = "child";
        close(p[0]);
        strcpy(childbuf, "Hi there from child!\n");
        if ((write(p[1], childbuf, sizeof(childbuf))) < 0) {
            perror("child: problems with writing to pipe"); exit(2);
        }
        close(p[1]);
        exit(EXIT_SUCCESS);
    }

    whoami = "parent";
    close (p[1]);
    for(;;) {
        read_count = read(p[0], parentbuf, sizeof(parentbuf));
        if (read_count > 0) {
```

```
        printf("processing request\n");
        printf("request is: %s\n", parentbuf);
    }
    else {
        sleep(1);
    }
}

void end_pgm(int signo)
{
    printf("%s dying off... Signal %d received.\n",whoami,signo);
    exit(EXIT_SUCCESS);
}
```

Named Pipes

```
int mkfifo(const char *PATH, int MODE)
```

- *PATH* parameter is the name of the file to be created.
- *MODE* determines the permissions of the file.
- Returns 0 on success, -1 on failure.
- One read.
- Many writes.
- Unrelated processes.
- Pipe can persist beyond current process.
- Open up just like a normal file.
- Must have read open always.
- Must have write open somewhere to avoid read hang.

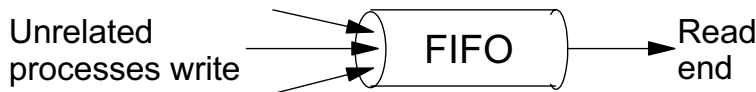


Figure 10-7. Named Pipes

AU253.0

Notes:

When the `mkfifo()` call finishes, it creates a file with the name in the `PATH` argument and permissions, as shown in the `MODE` argument. Then, just open it up like a normal file. Upon successful completion, the `pipe()` system call returns a 0 or -1 (with `errno` global variable set to the appropriate error number).

The same type of errors exist here as in unnamed pipes, and there now is also the problem of making sure at least one process has the pipe open for a write. It is not uncommon for the "server" to open the pipe for both read and write, even though it only plans to read from it.

Simple Named Pipe Example (1 of 2)

```
/* Program name - named_pipe.c */
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
    int fd,fd2,rc;
    char buf[80];
    pid_t pid;
    rc = mkfifo("myfifo",0620);
    if (rc < 0) {
        perror("problems with making the pipe");
        exit(1);
    }
    if ((pid=fork()) == 0) {
        sleep(1);
        fd = open("myfifo", O_WRONLY);
        if (fd < 0) {
            perror("problems with open on pipe - child");
            exit(1);
        }
        sprintf(buf,"%d",getpid());
    }
```

Figure 10-8. Simple Named Pipe Example (1 of 2)

AU253.0

Notes:

The sample program creates a FIFO named myfifo. The parent process then forks a child process. The child process writes its PID to the FIFO. The parent process then reads from the FIFO and prints the data onto the standard output.

The program continues in Figure 10-9.

Simple Named Pipe Example (2 of 2)

```
rc = write(fd,buf,strlen(buf)+1);
if (rc < 0) {
    perror("problems with write on pipe");
    exit(1);
}
close(fd);
exit (0);
}
/* Parent continues...*/
fd2 = open("myfifo", O_RDWR);
if (fd2 < 0) {
    perror("problems: pipe open - parent"); exit(1);
}
rc = read(fd2,buf,sizeof(buf));
if (rc < 0) {
    perror("problems with read on pipe"); exit(1);
}
printf("Child process said: %s\n", buf);
close(fd2);
remove("myfifo");
exit (0);
}
```

Figure 10-9. Simple Named Pipe Example (2 of 2)

AU253.0

Notes:

The biggest difference is that the `mkfifo()` call is needed for the final removal of the pipe with the `remove` call.

The real life implementation would have the parent process in a while loop (read, process, read, process, and so on), and the child process would have to handle SIGPIPE errors.

Here is another example of named pipes. It is similar to the one illustrated for unnamed pipes, except that FIFOs are used.

```
/* Program name - named_pipe1.c */

#include <sys/signal.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mode.h>

#define PERMS 0620
```

```
char *whoami;
int fdread;

int main(void)
{
    int read_count;
    int pid;
    int fdwrite;
    int rc;
    char parentbuf[25];
    char childbuf[25];

    struct sigaction mysig;
    sigset_t mymask;

    void end_pgm(int signo);

    memset(&mysig, 0, sizeof(mysig));
    mysig.sa_handler = end_pgm;
    sigfillset(&mymask);
    mysig.sa_mask = mymask;
    sigaction(SIGINT, &mysig, NULL);
    sigaction(SIGQUIT, &mysig, NULL);
    sigaction(SIGTERM, &mysig, NULL);

    umask(0);
    rc = mkfifo("myfifo", PERMS);
    if (rc < 0) {
        perror("parent: problems with create of myfifo"); exit(1);
    }
    pid = fork();
    if (pid == -1) {
        perror("parent: problems with fork"); exit(3);
    }

    if (pid == 0) {
        whoami = "child";
        fdwrite = open("myfifo", O_WRONLY);
        if (fdwrite < 0) {
            perror("child: problems with open of myfifo"); exit(4);
        }
        strcpy(childbuf, "Hi there from child!\n");
        if ((write(fdwrite, childbuf, sizeof(childbuf))) < 0) {
            perror("child: problems with writing to pipe"); exit(5);
        }
    }
}
```

```
    }
    close(fdwrite);
    exit(EXIT_SUCCESS);
}

whoami = "parent";
fdread = open("myfifo", O_RDONLY);
if (fdread < 0) {
    perror("parent: problems with open of myfifo"); exit(2);
}

for(;;) {
    read_count = read(fdread, parentbuf, sizeof(parentbuf));
    if (read_count > 0) {
        printf("processing request\n");
        printf("request is: %s\n", parentbuf);
    }
    else {
        sleep(1);
    }
}

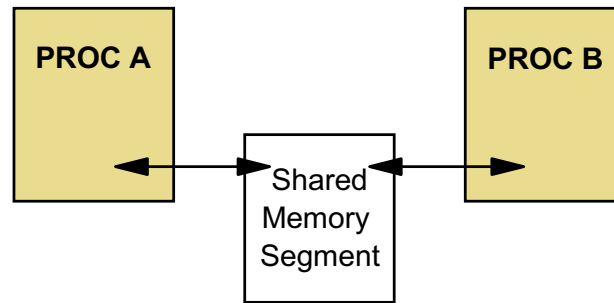
}

void end_pgm(int signo)
{
    remove("myfifo");
    printf("%s dying off... Signal %d received.\n",whoami,signo);
    exit(EXIT_SUCCESS);
}
```

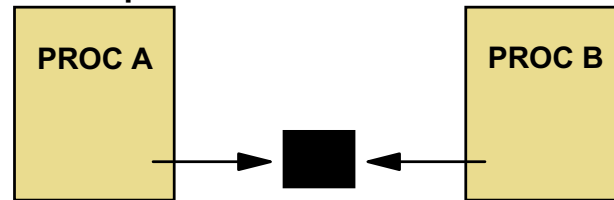
10.3 System V IPCs

System V IPCS

* Shared Memory:



* Semaphores:



* Message Queues:

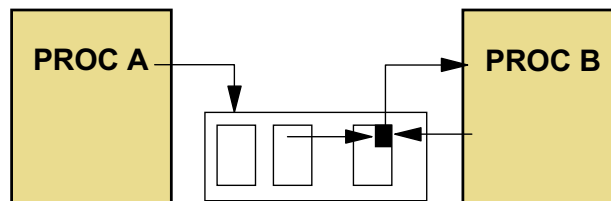


Figure 10-10. System V IPCS

AU253.0

Notes:

Some years ago, AT&T addressed the need for better forms of interprocess communication by developing three mechanisms. These became known as the System V IPCs.

Shared memory allows two or more processes to attach an AIX virtual memory segment to one of their own VM segments. (Remember that each process has sixteen segments, but ten of the segments are available for sharing or mapping files.) When two or more processes share the same segment, either process can read or write data into the segment and the other processes will immediately see changes made by the other processes.

Semaphores provide a mechanism for processes to express a condition or state to other processes. A semaphore is like a flag or set of flags within the kernel that can be queried or manipulated by processes.

Message queues are used to send packet-like messages to other processes. The messages are placed on a queue in the kernel. The queue is like a bulletin board. Other processes can read messages. Messages can also be addressed to specific processes, as demonstrated later in this unit.

The choice of an IPC is important when programming for IPC.

Shared memory is best used when the amount of data to be transferred between processes is large. The data is not synchronized, because more than one process can write into the shared memory segment simultaneously.

Semaphores are best used when the integrity of data being transferred is significant. Because the semaphores are mutually exclusive, data is always in sync.

Message queues are used for quick communication between processes. Message multiplexing makes it the best candidate for IPC.

Sockets are used for communication between CPUs and networks.

IPC Keys

- Processes sharing an IPC mechanism must use a common key to access the mechanism. Keys can be hard coded or generated with the `ftok()` routine.

```
#include <sys/ipc.h>
#define MYKEY 1234567L
...

#include <sys/types.h>
#include <sys/ipc.h>
...
main()
{
    int mykey;
    ...
    if ( (mykey=ftok("/home/ted/myapps/prog",'A') <= 0);
```

Figure 10-11. IPC Keys

AU253.0

Notes:

As mentioned earlier, processes participating in one of the System V IPC mechanisms must be written to know how to access the IPC mechanisms. But how do these different processes know how to access the same IPC mechanism as their corresponding partner processes? It is done by using special keys.

System calls used to establish the IPC mechanisms require a key that must be known by any process wishing to participate in the IPC. These system calls are discussed shortly.

Keys are long integers. The keys must be unique to each instance of an IPC. To avoid conflicts between applications on the same system, simple keys, such as 1234567 (as shown here) should be avoided.

The `ftok()` routine (file-to-key) can be used to generate a key based on a file name and a project identifier. This procedure can be practical for applications that can always count on the existence of a particular file. Refer to the AIX 5L online documentation for more on the `ftok()` routine.

The `ftok()` routine returns -1 upon failure.

10.4 Using Shared Memory Segments

Setting Up Shared Memory

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 1234567L
#define SHMSIZE 256

main()
{
    int shmid;
    char*shm_p;
    ...
    shmid=shmget(SHMKEY,SHMSIZE, IPC_CREAT|0660);
    shm_p=shmat(shmid,0,0);
    ...
}
```

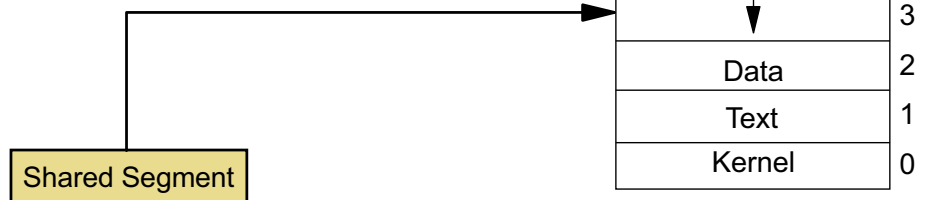


Figure 10-12. Setting Up Shared Memory

AU253.0

Notes:

Let us start by discussing the use of shared memory IPC.

Figure 10-12 introduces the `shmget()` and `shmat()` system calls and how they are used to create and attach a shared memory segment.

We have previously discussed memory mapped files and the use of a process' virtual memory segments three through twelve. The same ten segments that were shown as available for memory mapped files are also available for shared memory. In fact, here is the `shmat()` system call again. There are many similarities between memory mapped files and shared memory.

Note the use of the header files `ipc.h` and `shm.h`.

The `shmget()` call creates the shared memory segment and the `shmat()` system call attaches the shared memory segment to one of the calling process' segments. The precise syntax of these two calls is shown on the next page.

The following limits apply to shared memory:

- The maximum segment size is 2 GB, and 64 GB for 64-bit applications on AIX 5L.

- The minimum segment size is 1 byte.
- The maximum number of shared memory IDs is 131072.

Segments three through twelve are available for use as shared memory segments. Starting with AIX Version 4.2.1, segment fourteen is also available for use as a shared memory segment.

Also starting with AIX Version 4.2.1, an extended shmat capability is available. If an environment variable EXTSHM=ON is defined, then processes executing in that environment are able to create and attach more than eleven shared memory segments. The segments can be of size from 1 byte to 256 MB. The process can attach these segments into the address space for the size of the segment. Another segment can be attached at the end of the first one in the same 256 MB segment region. The address at which a process can attach will be at page boundaries (a multiple of 4096 bytes).

In order to be effective, this environment variable must have been set when the currently executing program was started (that is, when the `exec()` call was executed that loaded the current program into memory). Note that segment fourteen is available for use as a shared memory segment, regardless of whether or not the EXTSHM=ON environment variable is set.

If EXTSHM=ON is not defined (or the program is running on a version of AIX prior to AIX Version 4.2.1), then shared memory segments are always located at the start of a 256 MB virtual memory segment.

The smaller region sizes are not supported for mapping files. Regardless of whether EXTSHM=ON or not, mapping a file will consume at least 256 MB of address space.

Refer to the AIX shmat man page for information on this extended shmat capability.

Shared Memory System Calls

```
int shmget(key_t key, size_t size, int flags)
```

- Returns a shared memory identifier.
- key is used to identify a unique shared memory segment
- size indicates the number of bytes to allocate.
- flags can include ORed permission settings and the IPC_CREAT flag.
- Includes <sys/ipc.h>, <sys/shm.h>, and <sys/mode.h>.

```
void * shmat(int shmid, const void * addr, int flags)
```

- Attaches a shared memory segment to the caller's address space and returns a character pointer to the starting location of that segment.
- shmid is the shared memory identifier returned by the shmget() call.
- addr specifies the address to use for the shared memory segment (zero allows the kernel to assign a segment which is best for portability).
- The flag for specifying a mapped file is SHM_MAP.
- Includes <sys/ipc.h> and <sys/shm.h>.

Figure 10-13. Shared Memory System Calls

AU253.0

Notes:

Here is the syntax for the shmget() and shmat() system calls.

One of the participating processes has the responsibility of creating the shared memory segment by using the IPC_CREAT flag with the shmget() call. Other processes wishing to share the segment must also call the shmget() call, but do not use the IPC_CREAT flag. The size parameter defines the number of bytes to be shared. The shmget() call returns a shared memory identifier.

The addr parameter of the shmat() call allows the user to specify the address where the shared memory segment should map. However, the user should code a zero (0) for the addr parameter, allowing the kernel to decide where to attach the segment. This leads to better portability of the code, since every UNIX system implements virtual memory differently.

One point that catches programmers from time to time is that the shared memory segment could appear at a different address in each process attaching to the segment. For example, one process might see it at virtual address 0x30000000 and a second process might see it at 0x50000000. This usually only happens to programs that attach to multiple shared

memory segments or shared files, although the wise programmer will avoid assuming that the segment will have the same virtual address in all attaching processes.

The `sys/shm.h` header will automatically include `sys/ipc.h`, if it is not already present in the program.

Shared Memory Permissions

Permission Type	Symbolic Flag	Numeric Flag
Read by Owner	S_IRUSR	0400
Write by Owner	S_IWUSR	0200
Read by Group	S_IRGRP	0040
Write by Group	S_IWGRP	0020
Read by Others	S_IROTH	0004
Write by Others	S_IWOTH	0002

Figure 10-14. Shared Memory Permissions

AU253.0

Notes:

We mentioned how the creation of an IPC mechanism is like the creation of a file in that an IPC descriptor, like a file descriptor, is assigned and used to reference the IPC mechanism in other system calls. Another way that IPC mechanisms are like files is that they have permissions to control which processes can read (examine) and write (modify) the data in the IPC mechanism. Like files, permissions are based on user IDs associated with the processes.

Shared memory segments have permissions. The read permission allows a process to read the data in the shared memory segment. The write permission allows a process to modify the contents of the shared memory segment. Permissions apply to the user IDs associated with the owner of the IPC, the group associated with the IPC, and others.

These parameters are set by using flags when creating the IPC. You may find the numeric flags are easier to use. They work just like the numeric values used with the **chmod** command. Here is an example:

```
shmid=shmget(mykey,256,IPC_CREAT|0660);
```

This example creates a shared memory segment with read and write permissions for the owner and members of the owner's group. Note how the permissions are bitwise "ORed" with the IPC_CREAT flag by using the (|) symbol.

Using Shared Memory

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 123456L
...
int shmid;
char *shm_p;
...
if((shmid=shmget(SHMKEY,1000,IPC_CREAT|0660))<0) {
    perror("Could not get shmid..."); exit(1);
}
if((shm_p=shmat(shmid,0,0))<=0) {
    perror("Could not shmat..."); exit(2);
}
strcpy(shm_p,"This is fun.");
```

Figure 10-15. Using Shared Memory

AU253.0

Notes:

Here is an example of a program that creates a shared memory segment, attaches it, and writes a string of characters into it.

Controlling Shared Memory Segments

```
int shmdt(const void * addr)
```

- Detaches a shared memory segment.
- `addr` identifies the starting location of the shared memory segment.
- Includes `<sys/ipc.h>` and `<sys/shm.h>`.
- Returns 0 upon success, 1 on failure.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buffer)
```

- Provides control over shared memory segments.
- `shmid` is the shared memory identifier returned by the `shmget()` call.
- `cmd` specifies the command to perform; symbolic constants are defined in `<sys/shm.h>`.
- `buffer` is a pointer to a `shmid_ds` structure, as defined in `<sys/shm.h>`, where permissions and the size fields are defined.
- Includes `<sys/ipc.h>` and `<sys/shm.h>`.
- Returns 0 upon success, -1 on failure.

Figure 10-16. Controlling Shared Memory Segments

AU253.0

Notes:

Let us now look at two system calls used to control shared memory segments.

The `shmdt()` and `shmctl()` calls are illustrated in Figure 10-16. Both return a value of zero for success. `shmdt()` returns a 1 if the segment is not detached. `shmctl()` returns a -1 on error.

One of the participating processes should remove the shared memory segment upon completion of all desired actions. Before a segment is removed, however, all participating processes must either detach the segment or terminate. (Segments are automatically detached from a process when that process terminates.)

The `shmdt()` call detaches the specified segment. The segment is specified as an address parameter. This brings up an interesting point. It is best to save a copy of the starting address of an attached segment in a separate variable than the `shm` pointer, since the value of the `shm` pointer needs to change. (See the example on the next few pages.)

Once all processes have detached the shared memory segment, the segment can be removed by having one of the processes issue this system call:

```
shmctl(shmid, IPC_RMID, NULL) ;
```

Other commands include `IPC_STAT` (for status information) and `IPC_SET` (to set user and group IDs of the owner).

The **ipcs** and **ipcrm** are commands with which we could monitor the IPC activity on the system.

ipcs writes to stdout information about active IPC facilities, including what message queues, shared memory, and semaphores are in use. Following is a sample output of the **ipcs** command:

```
# ipcs
```

```
IPC status from /dev/mem as of Fri Mar 29 09:34:25 CST 2002
```

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

Message Queues:

q	0	0x4107001c	-Rrw-rw----	root	printq
q	3014657	0x0043951f	--rw-rw----	root	system
q	2	0x0074cbb1	--rw-----	root	system

Shared Memory:

m	0	0x58001307	--rw-rw-rw-	root	system
m	2490369	0x298ee665	--rw-rw-rw-	imnadm	imnadm
m	2490370	0x9308e451	--rw-rw-rw-	imnadm	imnadm
m	2490371	0x52e74b4f	--rw-rw-rw-	imnadm	imnadm
m	2490372	0xe4663d62	--rw-rw-rw-	imnadm	imnadm
m	2490373	0xc76283cc	--rw-rw-rw-	imnadm	imnadm
m	6	0xffffffff	--rw-rw----	root	system
m	7	0x0d001435	--rw-rw-rw-	root	system

Semaphores:

s	262144	0x58001307	--ra-ra-ra-	root	system
s	1	0x44001307	--ra-ra-ra-	root	system

ipcrm removes specified message queues, semaphore sets, or shared memory identifiers. Refer to the AIX documentation for more information on these two system administration commands.

Shared Memory Example (1 of 2)

```

/*  Prog shmem1.c -- Creates and writes to a shared memory segment  */

#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 5555555L

main()
{
    int shmid;
    char *shmw_p, *shm_start_p;
    int i;

    if((shmid=shmget(SHMKEY,10,IPC_CREAT|0660))<0) {
        perror("Could not get shmid...");
        exit(1);
    }
    if((shm_start_p=shmat(shmid,0,0))<=0) {
        perror("Could not shmat...");
        exit(2);
    }
    shmw_p=shm_start_p;
    for(i=1;i<=10;i++)
        *shmw_p++='A';

    if(shmdt(shm_start_p) != 0) {
        perror("Could not detach...");
        exit(1);
    }
}

```

Figure 10-17. Shared Memory Example (1 of 2)

AU253.0

Notes:

Figure 10-17 presents a final example of a program that creates and attaches a shared memory segment, then writes characters to the segment.

Note how the program detaches the segment.

Shared Memory Example (2 of 2)

```
/* Prog shmem2.c -- reads from the shared memory segment */

#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 5555555L
main()
{
    int shmid;
    char *shmr_p, *shm_start_p;
    int i;

    if((shmid=shmget(SHMKEY,10,0))<0) {
        perror("Could not get shmid...");
        exit(1);
    }
    if((shm_start_p=shmat(shmid,0,0))<=0) {
        perror("Could not shmat...");
        exit(2);
    }
    shmr_p=shm_start_p;
    for(i=1;i<=10;i++)
        printf("%c",*shmr_p++);
    if(shmdt(shm_start_p) != 0) {
        perror("Could not detach...");
        exit(1);
    }
    if(shmctl(shmid,IPC_RMID,NULL !=0)) {
        perror("Could not remove segment...");
        exit(1);
    }
}
```

Figure 10-18. Shared Memory Example (2 of 2)

AU253.0

Notes:

Here is another program that attaches and reads the data from the shared memory segment.

Note how this program uses `shmget()` without the `IPC_CREAT` flag, assuming that the segment already exists.

Note that this program removes the shared memory segment upon completion.

10.5 Semaphores

Setting Up Semaphores

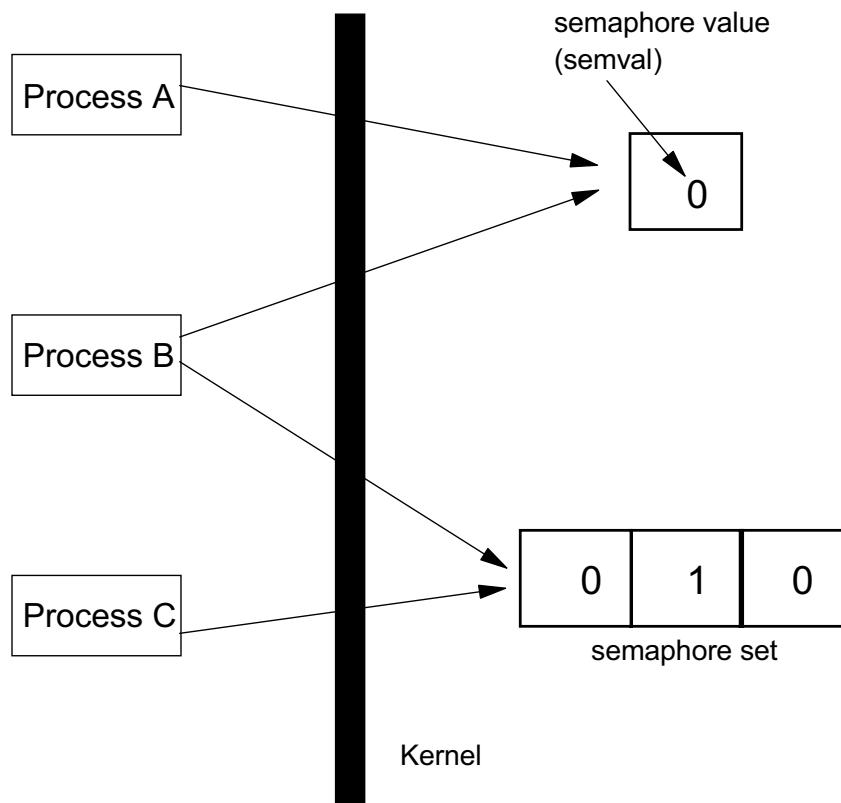


Figure 10-19. Setting Up Semaphores

AU253.0

Notes:

Figure 10-19 introduces semaphores.

A semaphore is a small structure within the kernel allocated by the `semget()` system call. The semaphore structure includes a semaphore value called "semval". Processes that access the semaphore can query the semval, or change its value.

Semaphores can be binary or positive. Binary semaphores will only have a semval of 0 or 1. Positive semaphores can have a semval value of any positive integer.

Semaphores are allocated in groups called semaphore sets. Each element of the set is a semaphore whose semval can be queried or changed.

Creating Semaphores

```
int semget(key_t key, int num, int flags)
```

- Returns a semaphore set identifier, -1 on failure.
- key is used to identify a unique semaphore set.
- num indicates the number of semaphores for the set.
- flags can include ORed permission settings and the IPC_CREAT or IPC_EXCL flag.
- Includes <sys/ipc.h> and <sys/sem.h>.

Figure 10-20. Creating Semaphores

AU253.0

Notes:

Semaphore sets are created with the `semget()` system call. `semget()` creates a data structure for the semaphore ID and an array containing num semaphores.

Figure 10-20 describes the syntax of the `semget()` system call.

The `semget()` call returns a semaphore identifier and requires a key. The key may be an IPC key constructed by the `ftok()` routine (or some other mechanism) or the value `IPC_PRIVATE`.

The flags that correspond to permission are reviewed in Figure 10-21.

Semaphore Permissions

Permission Type	Symbolic Flag	Numeric Flag
Read by Owner	S_IRUSR	0400
Alter by Owner	S_IWUSR	0200
Read by Group	S_IRGRP	0040
Alter by Group	S_IWGRP	0020
Read by Others	S_IROTH	0004
Alter by Others	S_IWOTH	0002

Figure 10-21. Semaphore Permissions

AU253.0

Notes:

Semaphores have permissions just like we saw with shared memory.

Note the use of "alter" instead of "write".

SEM_A is equivalent to IPC_W and is the same as S_IWUSR. SEM_R is equivalent to IPC_R and is the same as S_IRUSR. These flags are defined in sys/sem.h and sys/ipc.h and may be used instead of the corresponding S_FLAGS, if desired.

Controlling Semaphores

```
int semctl(int semid, int semnum, int cmd, [arg])
```

- Provides control over semaphores and semaphore sets.
- `semid` is the semaphore set identifier returned by the `semget()` call.
- `semnum` identifies the desired semaphore within the set (zero is the first semaphore as the set is treated as an array).
- `cmd` specifies the command to perform (symbolic constants defined in `<sys/sem.h>`).
- `arg` is either a pointer to a `semid_ds` structure as defined in `<sys/sem.h>`, where permissions and the number of semaphores are defined, or a value used by the `cmd` parameter.
- Includes `<sys/ipc.h>` and `<sys/sem.h>`.

Figure 10-22. Controlling Semaphores

AU253.0

Notes:

The `semctl()` system call is used to perform various actions on an existing semaphore set, such as changing the permissions or removing the semaphore set. Many actions access information stored in structures defined in `sys/sem.h`.

The `cmd` may be any of the following:

GETVAL	Returns the <code>semval</code> value
SETVAL	Sets the <code>semval</code> value to that specified by <code>arg</code>
GETALL	Stores all <code>semvals</code> in an array pointed to by <code>arg</code>
SETALL	Sets all <code>semvals</code> according to the array pointed to by <code>arg</code>

Additional GET commands access information in the `sem` structure. commands that get and set values in the `semid_ds` structure are also included

IPC_STAT	Obtains status information about a semaphore
IPC_SET	Sets the user and group IDs
IPC_RMID	Removes the semaphore ID and destroys the data structure

Using Semaphores

```
int semop(int semid, struct sembuf *opbuf, size_t numops)
```

- Performs operations on a semaphore.
- semid is the semaphore set identifier returned by the semget() call.
- opbuf is a pointer to a structure of type sembuf (as defined in the file <sys/sem.h>).
- numops specifies the number of operations to perform as an atomic action.
- Includes <sys/ipc.h> and <sys/sem.h>.

```
struct sembuf {  
    ushort sem_num; /* semaphore number */  
    short sem_op;   /* semaphore operation */  
    short sem_flg;  /* operation flags */  
};
```

Semaphore Operations:

- | | |
|----|---|
| 0 | Tests to see if the value of the semaphore is zero. |
| 1 | Increments the value of the semaphore. |
| -1 | Decrements the value of the semaphore. |

Figure 10-23. Using Semaphores

AU253.0

Notes:

The key to using semaphores is the semop() system call. This call performs the desired operations on the specified semaphore(s) within the semaphore set.

The key to using the semop() call is understanding the sembuf structure. The sembuf structure is defined in the sys/sem.h header file.

The number of operations to perform in an atomic semop() call is controlled by the numop parameter; each operation is defined in a sembuf structure placed in an array. The sembuf array is pointed to by opbuf.

The typical operations are:

- Test and set - A sem_op of 0 tests the semval for zero, a sem_op of 1 increments the semval.
- Unset - A sem-op of -1 decrements the semval.

The sem_num member of sembuf is used to indicate which semaphore (when there is more than one semaphore in a semaphore set). Return values are zero for success and -1 for failure.

Semaphores Example - Main Application

```
/* Program name - semph1.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEMKEY 888888L
int semid;
extern int locker(int num);
extern int unlocker(int num);
main()
{

    semid=semget(SEMKEY,1,IPC_CREAT|0660);
    if (locker(0) != 0)
        perror("Problems locking semaphore");
    else
        printf("Locked semaphore and pid is %d\n", getpid());

    sleep(10);
    if (unlocker(0) !=0)
        perror("Problems unlocking semaphore");
    else
        printf("Unlocked semaphore and pid is %d\n", getpid());

    semctl(semid,0,IPC_RMID,NULL);
}
```

Figure 10-24. Semaphores Example - Main Application

AU253.0

Notes:

Here is an example semaphore program that calls two functions, locker and unlocker, which are defined in Figure 10-25.

Semaphore Example - Called Functions

```
/* Program name - semph1_func.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
extern int semid;
locker(int num)
{
    static struct sembuf lock_it[2] = {
        0,0,0,
        0,1,0
    };
    lock_it[0].sem_num = num; /* sem number to test*/
    lock_it[1].sem_num = num; /* sem number to lock*/

    if (semop(semid,&lock_it[0],2) < 0)
        return 1;
    else
        return 0;
}
unlocker(int num)
{
    static struct sembuf unlock_it[1] = {
        0,-1,0
    };
    unlock_it[0].sem_num = num; /* sem number to unlock*/
    if(semop(semid,&unlock_it[0],1) < 0)
        return 1;
    else
        return 0;
}
```

Figure 10-25. Semaphore Example - Called Functions

AU253.0

Notes:

Here is a set of functions, `locker()` and `unlocker()`, which use semaphores to perform a type of advisory file locking.

This example treats the semaphore as a binary semaphore. It is also possible to treat the semaphore as a counting semaphore. Refer to the AIX 5L online documentation for more information.

10.6 Message Queues

Message Queues

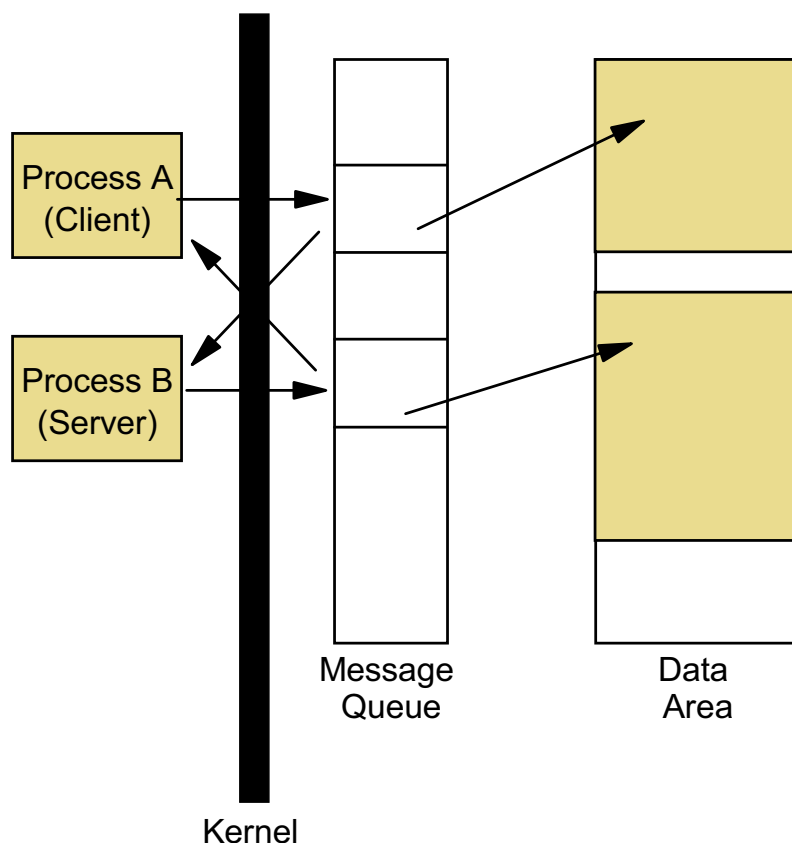


Figure 10-26. Message Queues

AU253.0

Notes:

The third and final System V IPC is the message queue. Figure 10-26 introduces the message queue.

Message queues allow processes to send messages (formatted packets of data) to other processes. One of the participating processes creates a message queue. A message queue is like a bulletin board, where messages can be posted. Actually, the message queue is allocated in the kernel to hold message headers. The message headers point to the message data placed in another area in the kernel.

A process uses the `msgsnd()` system call to post a message to the queue. Another process uses the `msgrcv()` system call to retrieve a message from the queue. When a message is retrieved, its header and data are removed from the queue.

Processes can be instructed to retrieve the oldest message on the queue or the oldest message of a particular type. Processes that send messages can define the message type. This method allows messages to be addressed to a desired receiver.

Setting Up Message Queues

```
int msgget(key_t key, int flags)
```

- Creates a message queue or accesses an existing queue.
- Returns a message queue identifier (integer).
- key is used to uniquely identify this queue.
- flags can include permission settings ORed with IPC_CREAT or IPC_EXCL.
- Includes <sys/ipc.h> and <sys/msg.h.>

Figure 10-27. Setting Up Message Queues

AU253.0

Notes:

Once again, the IPC is created by one of the participating processes by using a "get" function. In this case, the msgget() system call is used.

The msgget() call requires a key, the IPC_CREAT flag, and the permission set. Remember, the IPC_EXECL flag causes msgget to fail if the IPC_CREAT flag is also set and a data structure already exists.

A message queue identifier is returned.

Message Queue Permissions

Permission Type	Symbolic Flag	Numeric Flag
Read by Owner	S_IRUSR	0400
Alter by Owner	S_IWUSR	0200
Read by Group	S_IRGRP	0040
Alter by Group	S_IWGRP	0020
Read by Others	S_IROTH	0004
Alter by Others	S_IWOTH	0002

Defined in <sys/mode.h>

Figure 10-28. Message Queue Permissions

AU253.0

Notes:

Like the other System V IPCs, message queues have permissions. The permissions MSG_R and MSG_W defined in sys/msg.h are equivalent to S_IRUSR and S_IWUSR defined in sys/mode.h. These may be used instead, if desired.

Sending Messages

```
int msgsnd(int msgid, const void *buffer, size_t length, int flag)
```

- Places a message in the queue associated with msgid.
- Returns a zero upon success.
- msgid is the queue identifier returned by msgget().
- buffer is a pointer to an msgbuf structure, as defined in <sys/msg.h>.
- length indicates the number of bytes in the text of the message.
- flag is either IPC_NOWAIT or zero, where IPC_NOWAIT returns a -1 (failure), if there is not enough space left on the queue or not enough space left systemwide for the message.
- Includes <sys/ipc.h> and <sys/msg.h>.

Figure 10-29. Sending Messages

AU253.0

Notes:

The msgsnd() system call allows a process to post a message to the desired queue.

The IPC_NOWAIT flag means that the msgsnd() call should immediately return an error condition if there is no more room for messages on the queue.

The message must be sent through a buffer. There are some requirements of the format of the message buffer, which are described on the next page. The current process must have write permission to perform this operation. Upon successful return, various fields in the structure msgid_ds, defined in sys/msg.h, are updated, such as the number of messages in the queue (msg_qnum), the PID of the last message sender (msg_lspid), and the time the message was sent (msg_stime).

The msgbuf Structure

- The msgbuf structure, as defined in <sys/msg.h>:

```
struct msgbuf {
    mtyp_t mtype;          /* message type, must be > 0 */
    char mtext[1];         /* message data */
};
```

- mtyp_t is defined in sys/types.h to be long.
- The msgbuf structure is used as a template, because the system is not concerned with the layout of the body of the message.

```
typedef struct my_msgbuf {
    long mtype;            /* message type, must be > 0 */
    int mpid;              /* PID of sender */
    char mtext[256];       /* message data */
} msg_t;
```

Figure 10-30. The msgbuf Structure

AU253.0

Notes:

For the most part, messages can be defined with any format, as long as the first part of the message is a long integer. This long integer represents the message type. After that, the message can take any form.

The msgbuf structure found in the msg.h header file is only a template. Programmers can create their own message buffers to meet the needs of the message.

In this example, the my_msgbuf structure is created as a typedef. The typedef is called msg_t.

This message format includes the required long integer mtype. The message then contains the process ID number of the sender, followed by the text of the message, in an array of 256 characters.

Receiving Messages

```
int msgrcv(int msgid, void * buffer, size_t length, void * msgtype, int flag)
```

- Retrieves a message on the queue associated with msgid.
- Upon success, returns the number of bytes retrieved from the message, -1 upon failure.
- msgid is the queue identifier returned by msgget().
- buffer is a pointer to an msgbuf structure, as defined in <sys/msg.h>.
- length indicates the number of bytes in the buffer.
- msgtype indicates the type of message to retrieve (as defined by the m_type field in the msgbuf); zero indicates the oldest message on the queue, regardless of the m_type value.
- flag is either MSG_NOERROR (to truncate any message longer than length), IPC_NOWAIT (to return -1 (failure) if there is no message of the specified msgtype on the queue), or zero (to block (sleep) until a message of the specified msgtype appears on the queue).
- Includes <sys/ipc.h> and <sys/msg.h>.

Figure 10-31. Receiving Messages

AU253.0

Notes:

Processes retrieve messages from the queue by using the msgrcv() system call.

The process can be instructed to only receive messages of a certain type. This provides message multiplexing, where many processes can "talk" to each other through the queue. Telling the process to receive a message of type 0 retrieves the oldest message on the queue, regardless of the message's type value. This is why messages cannot be sent as type 0.

Usually, if no messages of the desired type are found on the queue, the msgrcv() call will sleep, waiting for a message of the desired type. The IPC_NOWAIT flag can be used to tell the msgrcv() call to return an error condition if no message of the desired type is present on the queue. Similar to msgsnd(), the msg_qnum, msg_lpid, and msg_rtime fields are updated upon successful execution.

Message Queue Multiplexing

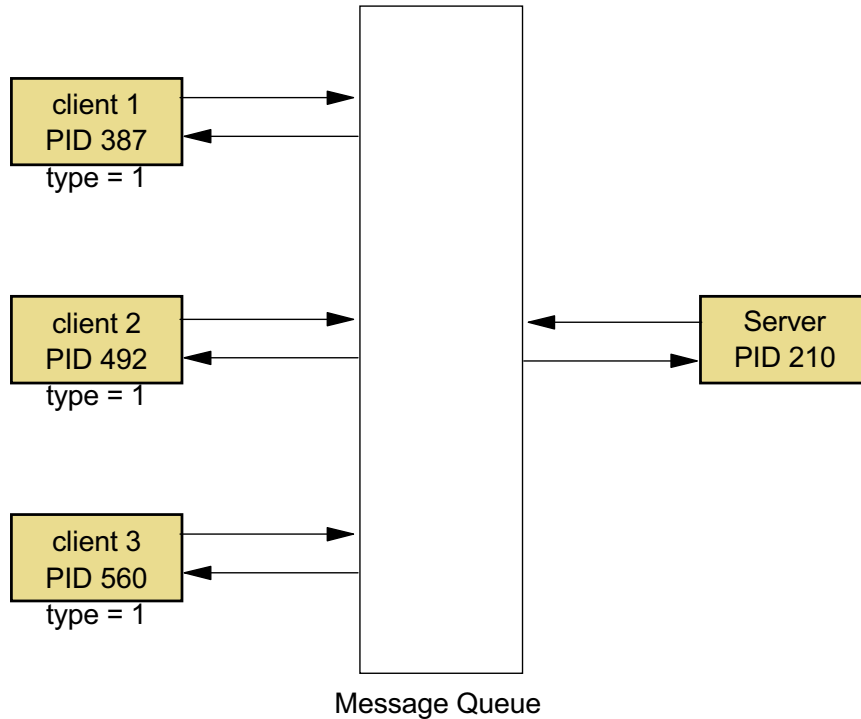


Figure 10-32. Message Queue Multiplexing

AU253.0

Notes:

Figure 10-32 illustrates message queue multiplexing. Messages addressed to the server process can be sent as type=1. Messages sent to the various clients can be addressed via the process' PID numbers.

Controlling Message Queues

```
int msgctl(msgid, cmd, buffer)
```

- Controls attributes of the queue associated with msgid.
- Returns zero upon success, -1 upon failure.
- msgid is the queue identifier returned by msgget().
- cmd is the operation to perform, and includes IPC_STAT (to retrieve queue status), IPC_SET (to alter permissions), and IPC_RMID (to remove the queue).
- buffer is a pointer to an msqid_ds structure, as defined in <sys/msg.h>.
- Includes <sys/ipc.h> and <sys/msg.h>.

Figure 10-33. Controlling Message Queues

AU253.0

Notes:

Message queues have a control system call like the other two System V IPCs.

This call can be used to check the status of the message queues, change the permissions of the queue, or remove the message queue.

Message Queue Example (1 of 4)

•A baseball team lookup example

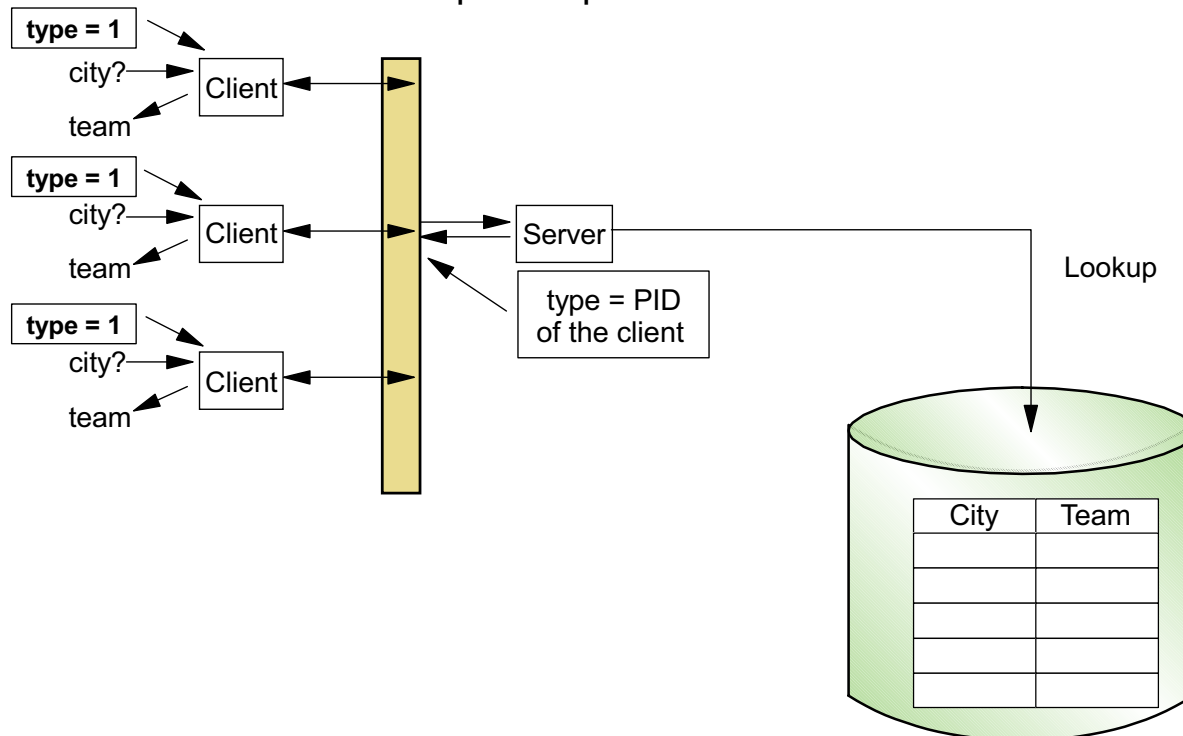


Figure 10-34. Message Queue Example (1 of 4)

AU253.0

Notes:

Let us build an application that demonstrates a multiplexing message queue.

Here is an illustration of a multiplexing message queue application. The point of the application may seem silly at first, but it is simple and fun.

Two programs are developed. One program is a client application named `bbcli.c`. This program runs in a loop, asking the user to enter the name of a city. The loop and program terminate when the user enters "END" (although the code that detects that is not shown here).

The client program then constructs a message and sends it to the message queue. The message contains the city name and some other data. Many client programs may be running at once.

The second program is called `bbsrv.c` and functions as a database server. This program picks up messages from the queue and looks up the city in a small database file. The database contains city names and the names of Major League baseball teams found in

each city. The server constructs and sends a message back to the client. The message contains the name of the baseball team for the client's specified city.

The server is also responsible for creating the message queue. The server and the client programs must both "know" the IPC key for the message queue.

The code in Figure 10-35 to Figure 10-37 only includes lines pertaining to the message queue. The code for looking up the city in the database, and some other code, has been omitted for simplicity.

Message Queue Example (2 of 4)

bbmsg.h header file:

```
/* bbmsg.h defines message format for example */

typedef struct my_msgbuf {

    long mtype;          /* message type, must be > 0 */

    int mpid;            /* PID of sender */

    char mcity[32];      /* client's city request */

    char mteam[32];      /* message data */

} bbmsg_t;

#define MSGKEY 4429087L
```

Figure 10-35. Message Queue Example (2 of 4)

AU253.0

Notes:

This application also requires a header file.

This header file, used by both server and client programs, uses a typedef to create a variable type called "bbmsg_t". This typedef defines the format of the messages sent and received by both programs.

The only requirement of the message buffer used in the msgsnd() and msgrcv() system calls is that the first piece of data in the message must be the message type. The remaining portion of the message can be defined as anything. In this case, the second field in the message buffer is an integer used by the client program to include a "return address" (client's PID) to the server. The last two fields have space for the city name and the returned baseball team name.

Message Queue Example (3 of 4)

```

/* msg queue example -- Client */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "bbmsg.h"
...
bbmsg_t msgout, msgin;
int msgid;
msgout.mtype=1;
msgid=mpid=getpid();
if((msgid=msgget(MSGKEY,0))<0) {
    perror("Could not get queue...");exit(1);
}
/* Ask user for city and assign it to msgout.mcity */
if(msgsnd(msgid,&msgout,sizeof(msgout),IPC_NOWAIT)<0) {
    perror("Could not send message...");exit(1);
}
msgrcv(msgid,&msgin,sizeof(msgin),getpid(),0);
/* Display the team name found in msgin.mteam */
...

```

Figure 10-36. Message Queue Example (3 of 4)

AU253.0

Notes:

Figure 10-36 shows the code for the bbcli.c client. Only the message queue code details are shown here.

Note the following:

- The **#include "bbmsg.h"**
- The creation of the msgin and msgout buffers, as well as the msgid
- How the message type and client's PID are assigned to the msgout buffer
- How the client accesses the queue via the msgget() call (note that the client does not create the queue)
- How the specified city is assigned to msgout.mcity (not shown)
- How the message is sent with the msgsnd() call
- The message length and the IPC_NOWAIT flag

- How the client waits for a reply from the server, looking for messages where type is the client's PID
- How msgrcv handles the "0" value in the flags field (waits for message)

Message Queue Example (4 of 4)

```

/* msg queue example -- Server */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "bbmsg.h"
...
bbmsg_t msgout, msgin;
int msgid;
if ( (msgid=msgget(MSGKEY,IPC_CREAT|0660)) < 0 ) {
    perror("Could not create queue..."); exit(1);
}
msgrcv(msgid,&msgin,sizeof(msgin),1,0);
msgout.mtype=msgin.mpid;
msgout.mpid=0;
strcpy(msgout.mcity,msgin.mcity);
/* Lookup the baseball team for msgin.mcity and assign the
results to msgout.mteam */
if (msgsnd(msgid,&msgout,sizeof(msgout),IPC_NOWAIT) < 0)
    perror("Could not send message...");
...

```

Figure 10-37. Message Queue Example (4 of 4)

AU253.0

Notes:

Figure 10-37 shows the code for the bbsvr.c server. Only the message queue code details are shown here.

Note the following:

- The **#include "bbmsg.h"**
- The creation of the msgin and msgout buffers, as well as the msgid
- How the server creates the message queue
- How the server waits for a client's request (using the "0" value for the flag portion of the msgrcv() call)
- How the server transfers the client's PID from the buffer of the received message to the type of the message to be sent
- How the results of the database lookup (the name of the city's baseball team) are stored in the msgout.mteam field (not shown)
- How the msgsnd() call replies to the client

10.7 Introduction to Sockets

What Are Sockets?

- A 2-way communication channel
- Referenced by socket descriptors
- Includes a socket type and one or more associated processes
- Exist within communication domains
- Are created, given names, and used to converse with other sockets

Figure 10-38. What Are Sockets?

AU253.0

Notes:

Sockets are a Berkeley (BSD) IPC facility. They are communication channels that allow unrelated processes to exchange data locally and over networks. Sockets have certain attributes that are unlike any other IPC mechanism.

Socket subroutines provide the following functions:

- Create and name sockets
- Accept and make socket connections
- Send and receive data
- Shut down socket operation
- Translate network addresses

IP Sockets - Addressed

- Network and Host Identified
- Four dotted decimal number
- Example: 10.19.96.14
- Client must identify server to talk to

Figure 10-39. IP Sockets - Addressed

AU253.0

Notes:

IP addresses tell TCP/IP what CPU to send the socket transfer to.

Actually, the IP address represents the adapter at the CPU that you need to get to. We will not discuss the logistics of the transfer at this point; we will leave that to the system administrator.

The first digit typically indicates how many digits represent the network versus host name on that network. If the number is between 1 and 126, then the first digit represents the network, and the last three represent the host on that network. If the number is between 128 and 191, then the first two numbers represent the network address and the last two represent the host on that network. If the number is 192 to 223, then the first three digits represent the network, and the last digit represents the host on that network.

Refer to the AIX TCP/IP course (ILS course AU07) for more information.

IP Socket Parts

- The unique application at server that you want to get to.
- Typically listed in /etc/services.
- You do not usually use anything below 5000.
- The server registers itself via the bind call.
- The client gets an automatic port number assigned.

Figure 10-40. IP Socket Parts

AU253.0

Notes:

The /etc/services file has a host of network services listed, and the port number on which to use those services.

An example /etc/services file is listed as follow:

```
# Network services, Internet style
#
ftp                21/tcp
telnet             23/tcp
smtp              25/tcp          mail
time              37/tcp          timserver
time              37/udp          timserver
rlp               39/udp          resource      # resource location
nameserver        42/tcp          name          # IEN 116
whois             43/tcp          nickname
domain            53/tcp          nameserver    # name-domain server
domain            53/udp          nameserver
```



```
mtp                57/tcp                # deprecated
tftp               69/udp
rje                77/tcp                netrjs
finger            79/tcp
#
# UNIX specific services
#
exec              512/tcp
biff              512/udp                comsat
login             513/tcp
who               513/udp                whod
shell             514/tcp                cmd                # no passwords used
syslog            514/udp
printer           515/tcp                spooler              # line printer spooler
talk              517/udp
ntalk             518/udp
efs               520/tcp                # for LucasFilm
route             520/udp                router routed

#
# Start of IBM added services ...
#
filesrv           2001/tcp
console           2018/udp
venus.itc         2106/tcp

# *** The CMVC entries are on the CMVC minidisk link via CMVCINIT ***
# CMVC entry for Lou in Boca (by Marcel)
# cmvctest        15555/tcp
# aei             5000/tcp
# iss             5001/tcp
# example         5002/tcp
```

Socket Calls - Server

- `socket(addrfam, type, protocol)` - Creates the socket
- `bind(socket, name, namelen)` - Binds a name to a socket
- `listen(socket, backlog)` - Listens for a socket connection
- `accept(socket, addr, addrlen)` - Accepts connection to create a new socket

Figure 10-41. Socket Calls - Server

AU253.0

Notes:

These are the main socket calls (others exist). Sockets include a variety of calls to send and receive data, including the usual read and write system calls. Other calls retrieve or set information about the socket.

Most calls require the `<sys/socket.h>` include file, `<sys/types.h>` and `<sys/socketvar.h>`.

The most common address family is `AF_INET`, which enables inter-CPU sockets.

The most common type is `SOCK_STREAM`, which will use the more reliable and connection-oriented TCP protocol for data transfer.

The name is typically a `sockaddr_in` structure. The most important fields are:

- `sin_family` - `AF_INET` for inter-CPU traffic
- `sin_port` - The port number where the server will register itself
- `sin_addr.s_addr` - `INADDR_ANY` (to allow the server application to be run from any CPU)

The `accept` call will grab the port number of the client application and store it in a structure of type `sockaddr_in`.

Socket Calls - Client

- `socket(addrfam, type, protocol)` - Creates a socket
- `connect(socket, name, namelen)` - Connects two sockets

Figure 10-42. Socket Calls - Client

AU253.0

Notes:

The only new call is the connect call. The name field is of data type `sockaddr_in`. In this case, the `sin_port` field should be the port number of the server. The `sin_addr.s_addr` field should be the IP address of the server.

Remember that other routines are available, so port numbers and IP addresses do not have to be hardcoded directly into your application.

Socket Example - Server (1 of 2)

```
/* server.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
main()
{
    int rc, pid, i, clientlen, sessionsd,sd;
    char buffer[80], str[20];
    struct sockaddr_in sockname, client;
    signal(SIGCHLD, SIG_IGN);
    strcpy(buffer,"This is the message from the server\n");

    memset(&sockname, 0, sizeof(sockname));
    sockname.sin_family = AF_INET;
    sockname.sin_port = htons((u_short )5001);
    sockname.sin_addr.s_addr = htonl(INADDR_ANY);

    sd=socket(AF_INET,SOCK_STREAM,0);
    if ( sd == -1) {
        perror("server:socket");exit(1);
    }
    printf("Server socket is %d\n", sd);

    rc = bind(sd,&sockname,sizeof(sockname));
    if (rc == -1) {
        perror("server:bind");exit(2);
    }
}
```

Figure 10-43. Socket Example - Server (1 of 2)

AU253.0

Notes:

Socket Example - Server (2 of 2)

```

if (listen(sd,3)== -1) {
    perror("server:listen");exit(3);
}
/* wait for a connection */
for(;;) {
    clientlen = sizeof(&client);
    sessionsd = accept(sd,&client,&clientlen);
    if (sessionsd == -1) {
        perror("server:accept");exit(4);
    }
    /* fork child to perform rest of actions */

    pid=fork();
    if(pid == -1) {
        perror("server:fork");exit(6);
    }
    if(pid == 0) {
        /* write a message to the session socket descriptor */
        printf("Server sending....\n");
        rc = write(sessionsd,buffer,sizeof(buffer));
        if (rc == -1) {
            perror("server:send");exit(5);
        }
        exit(0);
    }
    close(sessionsd);
}
}

```

Figure 10-44. Socket Example - Server (2 of 2)

AU253.0

Notes:

Here is an example of a server application using sockets. This would accept requests from multiple clients. The struct `sockaddr_in` is defined in `netinet/in.h`.

The `AF_INET` is used for inter-CPU transfer vs intra-CPU transfer. "htons" and "htonl" are calls to provide network architecture independent data. `INADDR_ANY` means anyone can come into the server. `SOCK_STREAM` represents that TCP is the method of transfer.

Refer to the AIX online documentation for more information on these calls.

Socket Example - Client (1 of 2)

```
/* client.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

main()
{
    int rc;
    int sd, sessionsd;
    char buffer[80];
    struct sockaddr_in toname;
    memset(&toname, 0, sizeof(toname));
    toname.sin_family = AF_INET;
    toname.sin_port = htons((u_short )5001);
    toname.sin_addr.s_addr = inet_addr("10.1.1.1");

    /*obtain a socket descriptor*/
    sd=socket(AF_INET,SOCK_STREAM,0);
    if ( sd == -1) {
        perror("client:socket");exit(1);
    }
}
```

Figure 10-45. Socket Example - Client (1 of 2)

AU253.0

Notes:

Socket Example - Client (2 of 2)

```
/*connect with a server process*/
rc = connect(sd, &toname, sizeof(toname));
if (rc == -1) {
    perror("client:connect");exit(2);
}

/*read a message from server*/
rc = read(sd, buffer, sizeof(buffer));
if (rc == -1) {
    perror("client:read");exit(3);
}
printf("%s",buffer);
close(sd);
}
```

Figure 10-46. Socket Example - Client (2 of 2)

AU253.0

Notes:

Here we have the client side of the socket application.

Note that the client makes a connection with the server by providing the server's IP address. If you try out this example, make sure that you provide the IP address of the machine on which the server is running.

After the connection with the server is established, the client waits for data from the server. The data is then printed.

Unit Summary

- Unofficial IPC Mechanisms
- Pipes
- Shared Memory
- Semaphores
- Message Queues
- Socket Basics

Figure 10-47. Unit Summary

AU253.0

Notes:

Unit 11. Writing AIX Daemons

What This Unit is About

UNIX daemon programs are extremely useful and relatively easy to write. With a few tricky points to be careful of, a program that creates a daemon process can be written in fewer than twenty lines of code.

This unit explains what daemons are and how they are used in AIX. The unit also describes the special characteristics of daemons and illustrates how to write a daemon.

The unit concludes with a complete example of a program that generates a daemon process, then executes the desired program in the daemon process.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Explain the function of daemon programs in the AIX environment.
- List special characteristics of daemon programs.
- Properly write and execute simple daemon programs, avoiding the common pitfalls and errors frequently made by improper daemon coding.
- Explain what the syslog facility is and how to use it.

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 9

References

- *Writing Reliable AIX Daemons*, SG24-4946

Unit Objectives

After completing this unit, you should be able to:

- Explain the function of daemon programs in the AIX environment.
- List special characteristics of daemon programs.
- Properly write and execute simple daemon programs, avoiding the common pitfalls and errors frequently made by improper daemon coding.
- Explain what the syslog facility is and how to use it.

Figure 11-1. Unit Objectives

AU253.0

Notes:

11.1 Just What Are Daemons?

What Is a Daemon?

- An UNIX program that performs repetitive tasks
 - Run continuously, but use minimal system resources
 - Not associated with a user's login session
 - Generally used to perform system-related tasks
- Typical AIX daemons:
 - cron
 - Kicks off user-created jobs at regular intervals
 - syncd
 - Regularly forces disk writes of all memory pages scheduled for page-outs
 - nfsd
 - Handles file I/O requests for a Network File System
 - qdaemon
 - Accepts print requests and kicks off queuing system back-end programs
 - errdemon
 - Responds to hardware and software errors reported by applications and device drivers

Figure 11-2. What Is a Daemon?

AU253.0

Notes:

The UNIX operating system has long featured a special kind of program called a daemon. Daemon programs are like any other type of program in that they perform some task that is usually of a repetitive nature. Daemon programs differ from ordinary UNIX programs in that they are not associated with a terminal login session. That is, the life span of a daemon process does not depend on the life span of the terminal session or any control key sequences generated from a terminal session. They frequently run continuously on the system, long after the user who started the program has logged off. Daemons can be identified using the **ps -ef** command. They have a “-” symbol in the TTY column of the **ps** output.

The AIX Operating System includes many daemon programs designed for system management. Examples include:

cron	A daemon program that wakes once each minute, looks for user jobs to start (users enter job requests into their crontab files), starts any jobs scheduled for that second, then puts itself back to sleep. The cron daemon schedules event types such as
-------------	--

crontab command events, **at** command events, **batch** command events, **ksh** command events, and so on. Refer to AIX online documentation for more information.

syncd

This daemon performs a sync() system call at regular intervals to force the operating system to write to disk all pages scheduled to be written. The syncd program takes an argument when invoked (the default is 60) that specifies how often, in seconds, to perform the sync().

nfsd

This is one of the many daemons used by the Network File System to handle requests for remote file access. There are many daemons used to handle various communications needs. The nfsd daemon runs on a server.

qdaemon

This program runs when the queuing system is up. It accepts jobs submitted to the queuing system via commands such as **enq** and **lp**. The qdaemon places the job request into the appropriate queue directory, then invokes a back-end program to service the queue.

errdemon

This traditional UNIX daemon catches information written by applications and device drivers to the /dev/error file. These messages contain information about hardware and software errors encountered by the applications and device drivers. The errdemon writes a record to the /etc/errlog file for each message. The system administrator can query the errlog using the **errpt** command. Note the spelling difference in errdemon.

Special Characteristics of a Daemon

Daemon programs (processes) :

- Run as background jobs.
- Have no controlling terminal.
- Must be immune to background job control I/O constraints.
- Ignore receipt of SIGTTIN and SIGTTOU signals.
- Files must not remain open any longer than necessary.
- Do not belong to a login session (not member of a login session process group).
- Do not interact with any user.

Figure 11-3. Special Characteristics of a Daemon

AU253.0

Notes:

Because daemons generally run for a long time, they are usually run as background processes. For this reason, they should not attempt to read from the stdin or write to stdout, as that action might interfere with the I/O being performed by the current foreground process on the terminal. In fact, the UNIX C and Korn shells will automatically stop any background process that attempts to read or write to the terminal. The shell does this by sending a SIGTTIN signal to a background process attempting to read from the terminal and a SIGTTOU signal to a background process attempting to write to the terminal. For this reason, these signals, and a few others, should be ignored by daemons.

When a process is created, it automatically becomes a member of a process group. Normally, the process group leader is the login process for the terminal. Any signal received by the process group leader is distributed to all processes in the process group. That is why all processes under your login process usually terminate when you log out. Logging out sends a SIGHUP signal to the login process, which is the process group leader. Action must be taken to prevent the daemon from receiving SIGHUP signals.

Daemon processes generally run for long periods of time. When possible, daemons should avoid tying up system resources. This includes file systems and files. If a daemon process runs with a certain file or set of files always open, the file system that those files are in is considered to be busy. This means that the file system cannot be unmounted. Daemons do not belong to a controlling terminal means that they do not belong to a login session either. The daemons do not interact with the users through a terminal. The `setpgrp()` and `setsid()` system calls are used to create a new process group and relinquish the controlling terminal.

On most of the UNIX versions, the `init` process is the parent of a daemon process. This is because either `init` started the daemon process, or adopted it when the actual parent process terminated. AIX, however, has a facility called System Resource Management (SRC). Among its other functionalities, the SRC also manages daemons. Hence, SRC could be the parent process for a number of daemons.

11.2 Writing A Daemon

Steps to Writing a Daemon

- Determine how the daemon was started
- Ignore specific signals
- Close all open files (especially stdin, stdout, and stderr)
- Lose controlling terminal
- Detached from process group
- Prevent reassociation with a controlling terminal
- Change the working directory
- Set desired environment

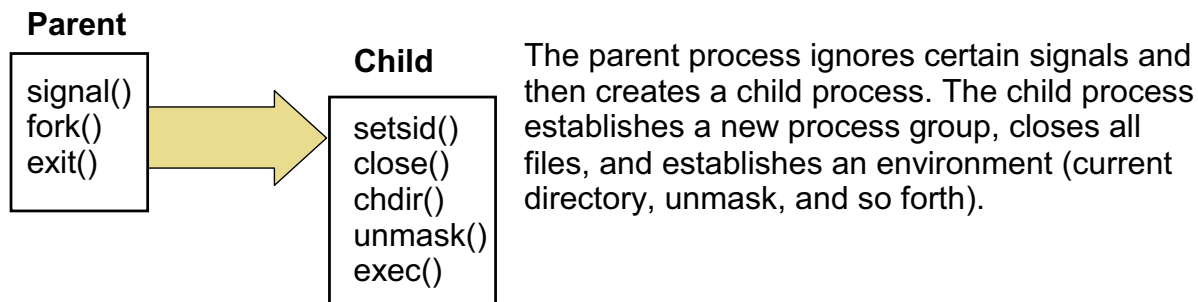


Figure 11-4. Steps to Writing a Daemon

AU253.0

Notes:

Figure 11-4 lists the steps of coding a UNIX daemon process. It also illustrates a practical approach to kicking off the daemon.

Each step of coding a daemon process is detailed on the following pages.

Here is the reason for the unusual way of kicking off the daemon process:

Daemons can be started from any other process, but they are usually run from a shell environment, which is part of a user's login session and terminal process group. As part of the daemon initialization process, it may be required to determine how the daemon was started. The goal of kicking off a daemon is to have the parent process "forget" about the child/daemon. This can be accomplished by having the shell fork and execute a child process that ignores the appropriate signals and creates its own process group, then forks and executes the daemon. The daemon then closes all open files, sets the environment, then executes the desired program.

It is sometimes required to change the current working directory of the daemon during the initialization process. We will discuss this later.

Determine How The Daemon Started

- Daemons are started in a number of ways.
- If started by the shell, a daemon has to be migrated to another process.
- There are no straightforward methods to determine how daemon was started. The methods used depend on the way certain programs work.

Figure 11-5. Determine How The Daemon Started

AU253.0

Notes:

Some of the daemon initialization steps are appropriate when the daemon is started in a certain manner. For example, when the daemon is started by `inetd`, `SRC` or `init`, it is not appropriate to detach the daemon from the process group, but when the daemon is started from a shell, it is appropriate to detach the daemon from the process group. Therefore, it is necessary to determine how the daemon was started.

There are no straightforward methods to determine how the daemon was started. AIX does not provide specific system calls for this purpose. The parent process ID (PPID) of a process is helpful in knowing how the daemon was started. For example, if the PPID of a process is 1, then there are two possibilities. One is that the process is the child of `init` process. The other possibility is that the parent is another process that terminated before the child process terminated. In that case, the child process is inherited by the `init` process.

Ignoring Signals

- Register signal handlers to ignore signals that affect background processes:

```
signal(SIGTTOU,SIG_IGN);
```

```
signal(SIGTTIN,SIG_IGN);
```

```
signal(SIGTSTP,SIG_IGN);
```

- Register signal handlers to ignore or react to other possible signals:

```
signal(SIGTERM,clean_up);
```

```
signal(SIGHUP, read_config_file);
```

Figure 11-6. Ignoring Signals

AU253.0

Notes:

Figure 11-6 shows how to ignore the signals that could cause problems for daemons. This is accomplished by registering signal handlers that ignore the signals.

The SIGTTOU signal is sent to background processes that attempt to send stdout to a terminal. The SIGTTIN signal is sent to background processes that attempt to read stdin from a terminal. In each case, the default action for the receiving process is to be stopped. This is a feature of csh and ksh job control that must be ignored by daemons.

The SIGTSTP signal is another signal that can stop a daemon and should be ignored.

When the system is shut down by the system administrator, every process is sent a SIGTERM and then given a few seconds to terminate gracefully. Any processes left running after a few seconds are sent a SIGKILL, which forcibly terminates them. Consequently, a daemon should generally handle SIGTERM signals by invoking a clean-up routine. The clean-up routine should do whatever tidying up is needed and then invoke the exit() system call to terminate the daemon.

Other signals can be ignored as needed, with the exception of SIGKILL and SIGSTOP, which cannot be ignored.

Closing stdin, stdout, and stderr

- Actually, all open files should be closed by the daemon process:

```
#include <limits.h>
...
    int  fd;
    ...
    for (fd=0; fd < OPEN_MAX; fd++)
        close(fd);
    ...
...
```

Figure 11-7. Closing stdin, stdout, and stderr

AU253.0

Notes:

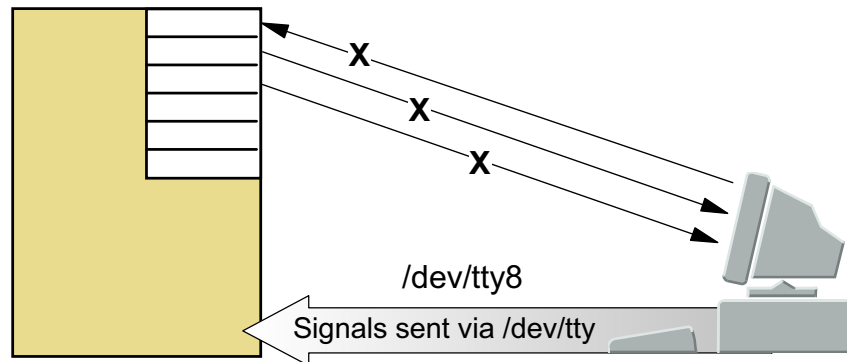
A daemon should close all open files. A simple looping routine will perform this action.

AIX supports a file descriptor table of 2000 slots. In this example, we use the `OPEN_MAX` symbolic constant from the `<limits.h>` include file for portability.

This operation also closes `stdin`, `stdout`, and `stderr`.

Closing open file descriptors is necessary and has a practical purpose. With a file descriptor opened by a process, the file system in which the file exists cannot be unmounted. Since daemon processes run for a long time, having file descriptors open could cause a significant problem to the system administrator.

Relinquishing the Controlling Terminal



- Every process starts with a controlling terminal (unless the parent process relinquished the tie to the controlling terminal prior to creating the child process).
- Even if stdin, stdout, and stderr are closed, the controlling terminal tie still exists.
- The POSIX `setsid()` system call relinquishes the controlling terminal.

Figure 11-8. Relinquishing the Controlling Terminal

AU253.0

Notes:

Whenever a terminal association exists or has existed, the running process is said to have a controlling terminal. Even if the program closes its own stdin, stdout, and stderr (file descriptors 0, 1, and 2), it still has a controlling terminal. The value of the controlling terminal is stored in the process's User Area and can be accessed by referencing `/dev/tty`:

```
fd=open("/dev/tty", ...);
```

This is sometimes desirable, but not with daemon processes, because having a controlling terminal means that signals generated by the controlling terminal may still be delivered to the daemon. A daemon should have no association with a terminal.

Traditionally, the `ioctl()` system call was used to force a daemon to lose its controlling terminal. That capability is still supported in AIX, but there is an easier way. AIX supports the `setpgrp()` system calls that relinquishes the tie to the controlling terminal.

AIX also supports a `setsid()` system call that not only causes the caller to become the leader of a new process group, but also loses the controlling terminal. This means that a simple call to `setsid()` (with no arguments given) performs two of the most important steps

of making the caller a daemon: starting a new process group and losing the controlling terminal. `setsid()` returns a new process group ID on success and -1 for failure.

Environmental Concerns

- For continuous daemons, change current directory to root, which allows all other file systems to be unmounted:

```
chdir("/");
```

- Set the file creation mask to zero :

```
umask(0);
```

Figure 11-9. Environmental Concerns

AU253.0

Notes:

As mentioned earlier, a daemon process must be careful of its own environment. Two typical environmental concerns involve the current directory and the file creation mask.

Figure 11-9 shows how the daemon should set its current directory to the root directory. This assures that the daemon will allow a System Administrator to unmount other file systems. For example, if the daemon is executed with a current directory of /home/dave, and it does not change its current directory, any attempt to unmount the /home file system will result in the error message "Device busy . . .". It is safe to change the current directory of a daemon to / (root directory), because the root file system is never unmounted during normal system operation.

Figure 11-9 also illustrates how a daemon should use the `umask()` system call to change its file creation mask, in case it had been changed previously by the caller. A `umask` of 0 allows the daemon code to have control of the permissions that apply to created files.

11.3 The Syslog Facility

The Syslog Facility

Q

How does a daemon, which has disassociated itself from all normal output channels, emit messages for an administrator?

A

By using the Syslog Facility.

Figure 11-10. The Syslog Facility

AU253.0

Notes:

The syslog facility is intended to be used to record and manage messages emitted by daemons. With a bit of help from your system administrator, you can easily emit messages from your daemon that are recorded in log files. This solves the otherwise troublesome problem of arranging for a daemon to be able to communicate with the outside world after it has closed all of its file descriptors.

Sending Messages to Syslog

```
void openlog(const char *id,int options,int facility)
```

- Initializes a connection with the syslog daemon.
- *id* is a character string attached to the beginning of every message.
- *options* is a set of flags.
- *facility* specifies what kind of facility is sending the message.
- Includes <syslog.h>.

```
void syslog(int priority, const char * format,...)
```

- Sends message to the daemon.
- *priority* indicates a severity level.
- *format* specifies a set of values as in printf subroutine.
- Includes <syslog.h>.

Figure 11-11. Sending Messages to Syslog

AU253.0

Notes:

Using syslog is a two-step process:

1. Initialize a connection to the syslog daemon by calling `openlog()`. *id* is a character string that will appear at the start of each logged message. *options* is a set of flags. Common flags include:

- LOG_CONS

Sends messages to the system console if they cannot be sent to the syslog daemon (quite useful in a daemon that has otherwise disassociated itself from all other output channels).

- LOG_PID

Includes the caller's PID in emitted messages (quite useful in distinguishing between multiple daemons that all happen to use the same id string).

facility specifies what kind of facility is sending the messages. Although there are a number of choices, the most common choice for daemons is LOG_DAEMON (see the AIX online documentation or the syslog man page for more choices).

2. Send messages to the daemon using `syslog()`. The `syslog()` routine is a `printf`-like routine that takes a message priority (that is, how important is this message), a format string, and however many additional parameters are required to satisfy the format string. The possible message priority and roughly what they mean are:

- `LOG_EMERG`

Something catastrophic has just happened and the entire system is almost certainly going to crash.

- `LOG_ALERT`

Something pretty dramatic has just happened. The sender of the message does not expect to be able to continue.

- `LOG_CRIT`

Something quite serious has just happened (for example, an important operation failed), although the sender expects to be able to continue functioning.

- `LOG_ERR`

An error has occurred, although the sender is able to continue normal processing.

- `LOG_WARNING`

Something unusual has happened, although little or no harm was likely done.

- `LOG_NOTICE`

Something interesting has occurred, although there is probably no reason for concern.

- `LOG_INFO`

This message is purely informational. It really does not matter if anybody ever sees this message.

- `LOG_DEBUG`

Used for purely debug-level messages.

Each process should establish its own connection to the syslog daemon. If a process calls `fork()` and the child expects to send messages to the syslog daemon, then it should first close the inherited connection by calling `closelog()` with no parameters and then calling `openlog()` again with appropriate parameters.

Configuring the Syslog Daemon

- What the syslog daemon does with messages that it receives is determined by the contents of the `/etc/syslog.conf` configuration file. This line causes all daemon messages of any priority to be sent to `/var/adm/debug`:

```
daemon.debug      /var/adm/debug
```

Figure 11-12. Configuring the Syslog Daemon

AU253.0

Notes:

A lot of AIX systems are configured to not record syslog messages anywhere. The `/etc/syslog.conf` file will arrange for at least the daemon messages to be sent to `/var/adm/debug`. (Check with your system administrator.) If you do not have syslog already configured to record the kinds of messages that you want to send, we suggest that you should add the above line to the `/etc/syslog.conf` file. Once the `/etc/syslog.conf` file has been changed, the syslog daemon must be refreshed by the system administrator using the following command:

```
# refresh -s syslogd
```

Note: Once you have added this line, all sorts of daemon messages are going to start to accumulate in `/var/adm/debug`. Your system administrator is going to probably also have to setup a cron job that periodically cycles the log files (otherwise, the `/var/adm/debug` file will keep growing and, finally, the `/var` file system will be full). Something like this might get run every night:

```
mv /var/adm/debug.6 /var/adm/debug.7 2>/dev/null
mv /var/adm/debug.5 /var/adm/debug.6 2>/dev/null
```

```
mv /var/adm/debug.4 /var/adm/debug.5 2>/dev/null
mv /var/adm/debug.3 /var/adm/debug.4 2>/dev/null
mv /var/adm/debug.2 /var/adm/debug.3 2>/dev/null
mv /var/adm/debug.1 /var/adm/debug.2 2>/dev/null
mv /var/adm/debug /var/adm/debug.1 2>/dev/null
```

This will preserve a week's worth of daemon debug messages on an ongoing basis. Any error messages are discarded, because nobody is likely to really care if one of the mv's fail. Otherwise, you could redirect the error messages to another file instead of /dev/null.

11.4 Using The Daemon

Starting the Daemon

- Can be started as normal processes.
- Users can automate daemon startup with at or cron.
- System daemons can be kicked off via /etc/rc* files.
- System daemons could be started via /etc/inittab, but be careful:
 - Daemons started with inittab option to respawn will cause rapid respawning due to init thinking daemon has completed, since parent usually exits after starting daemon as child.

Figure 11-13. Starting the Daemon

AU253.0

Notes:

Daemons might do a simple task, then terminate, or daemons may run for a long time. The way a daemon runs must be taken into account when deciding how to invoke the daemon.

Daemons can also be started using inetd, the System Resource Controller (SRC), or the init process.

The /etc/inittab file has four fields: identifier, runlevel, action, and command. The action field in the /etc/inittab file can be set to respawn (if the process dies), once (start the process, do not wait), or wait (start the process and wait for its termination). Other actions are available for boot processes, system initialization, run-level default, and power failures. The following is a sample entry of the /etc/inittab file:

```
cron:23456789:respawn:/usr/sbin/cron
```

Daemon Example

```

/* daemon.c */
#include <stdio.h>
#include <signal.h>
#include <limits.h>
main()
{
    int fd;

    signal(SIGTTOU,SIG_IGN);
    signal(SIGTTIN,SIG_IGN);
    signal(SIGTSTP,SIG_IGN);
    signal (SIGHUP,SIG_IGN);
    if (fork()!=0) /* parent */
        exit(0);

    setsid(); /* detach from process group and
               relinquish controlling terminal */

    for (fd=0;fd < OPEN_MAX;fd++)
        close(fd);

    chdir("/");
    umask(0);

    /* do the daemon's work */
}

```

Figure 11-14. Daemon Example

AU253.0

Notes:

Figure 11-14 ties it all together and presents the code for starting a daemon.

Be careful! Only execute the fork and setsid if the daemon is being kicked off from somewhere other than the /etc/inittab file. The easiest way to check for this is to check if getppid() returns a PID of 1. If so, then your daemon is getting kicked off from /etc/inittab (because init's PID of 1 is the parent).

Unit Summary

- Daemon programs and their characteristics
- Common AIX daemons
- Steps to writing daemons
- Syslog Facility
- Invoking daemons
- Example program

Figure 11-15. Unit Summary

AU253.0

Notes:

Unit 12. The Remote Procedure Call (RPC) Facility

What This Unit Is About

This unit teaches the students how to use the ONC (Open Network Computing) remote procedure call facility to write client server applications. The concept of remote procedure calls is introduced and justified. The unit then demonstrates the use of the **rpcgen** tool to build ONC RPC-style applications.

The lab consists of having the students use **rpcgen** to implement a client for a simple ONC RPC application.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Explain what a remote procedure call is
- Explain some of the kinds of problems remote procedure calls can be used to solve
- Write simple ONC RPC applications using **rpcgen**

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 10

References

- AIX Version 5 online documentation
- *Power Programming with RPC* by John Bloomer, published by O'Reilly & Associates, ISBN 0-937175-77-3

Unit Objectives

After completing this unit, you should be able to:

- Explain what a remote procedure call is.
- Explain some of the kinds of problems remote procedure calls can be used to solve.
- Write simple ONC RPC applications using rpcgen.

Figure 12-1. Unit Objectives

AU253.0

Notes:

There are a number of different Remote Procedure Call (RPC) facilities available. The one that we will be looking at in this unit is the Open Network Computing (ONC) RPC facility. ONC RPC was originally developed by Sun Microsystems Inc. and is available on a wide variety of platforms, including all modern UNIX systems, MS Windows, OS/2, MacOS, and mainframe operating systems. The widespread adoption of ONC RPC makes it an excellent choice for developing portable RPC-style applications.

There are no RPC facilities specified by the Single UNIX Specification.

12.1 RPC and Distributed Computing

Remote Procedure Call Flow

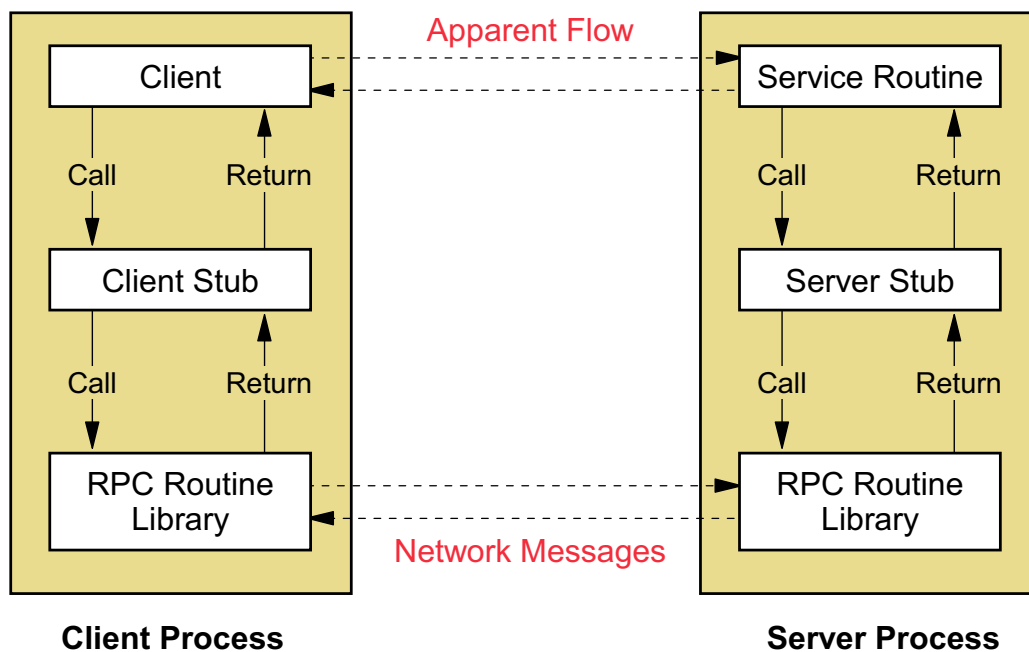


Figure 12-2. Remote Procedure Call Flow

AU253.0

Notes:

Calling a procedure and getting a result back is something that happens all the time in any C program. The Remote Procedure Call model allows a programmer to continue to use this process, even though the procedure being called is actually invoked by a different process that is often running on another machine on the network.

From the programmer's perspective, the apparent flow is directly between the calling procedure and the called procedure.

The reality is quite different. The calling routine, generally called the client, calls a client stub routine. This client stub routine packages a request that indicates which remote service routine is to be called and what parameters are to be provided to the routine. This request is passed to the RPC library routines, which pack parameter values into a standard form and pass the request to the server process using network programming calls.

The server process receives the request in RPC library routines that unpack the parameters and call the appropriate server stub routine. This stub routine finishes unbundling the request and calls the service routine.

The service routine performs the requested operation and returns a value to the server stub routine. The return value is packaged up and sent back to the client stub routine via the RPC libraries and the network. The client routine then returns the value to the original caller.

The RPC protocol is built on top of the eXternal Data Representation (XDR) protocol, which standardizes the representation of data in remote communications. XDR converts the parameters and results of each RPC service provided. The conversion of data to and from a standard form mentioned above deals with issues related to the fact that not all machine architectures store values in the same way. For example, one architecture might store int values into memory with the most significant bytes appearing before less significant bytes. Another architecture might store int values into memory with the least significant byte first. By passing data values in a standard form, the receiver of the data always knows how to interpret the data.

Advantages of Remote Procedure Calls

- The server can:
 - Run on a very powerful machine.
 - Use information not available to the client.
 - Take advantage of privileges not available to the client.
 - Coordinate requests from multiple clients.
- The client can:
 - Run on a less capable machine than the server.
 - Gain access to information and capabilities not normally available to it.
 - (Often) ignore what other clients may be doing.

Figure 12-3. Advantages of Remote Procedure Calls

AU253.0

Notes:

Probably the key advantage to RPC-style programming is that it allows an application to be distributed over a network without forcing the programmer to spend a lot of time dealing with the problem of actually communicating over the network.

Some of the other reasons why RPC-style programming is attractive are:

- The client/server model allows an application to take advantage of the capabilities of a large central server machine without losing the advantages of running on the user's local machine. This is particularly important on PC operating systems, which generally require that a GUI based program run on the user's local machine.
- The programmer can usually ignore the fact that multiple programs or machines are being used to solve the problem at hand.
- It is easier to create an application that can run cleanly in a heterogeneous environment, because the RPC facility provides an architecture-neutral way for clients and servers to communicate.

- The RPC protocol allows the users to work with remote procedures as if the procedures were local.
- The RPC provides an authentication mechanism by which the client and the server identify each other.

Where is RPC Used Today?

- By the operating system:
 - The Network File System (NFS) facility
 - The Network Information Service (NIS) facility
- By distributed applications:
 - Ray tracing
 - Distributed databases
 - Imaging systems

Figure 12-4. Where is RPC Used Today?

AU253.0

Notes:

ONC RPC is used by the Network File System (NFS) and the Network Information System (NIS). The widespread popularity of NFS is probably one of the primary reasons why the ONC variant of RPC has been ported to essentially all modern operating systems.

ONC RPC and other RPC variants have been used to implement a wide variety of client-server applications:

- Ray tracing is a very CPU-intensive technique used to generate realistic images of 3-D scenes. Fortunately, it is also easy to parallelize. An RPC-based ray tracer typically has a single client and multiple servers. The client asks each server to compute a different part of the scene and assembles the results as they come back.
- Another approach is to have a single server and multiple clients. Each client contacts the server to request a part of the scene to compute. When the client completes its assigned portion, it sends the result to the server, which then assigns the client another part of the scene.

- Modern database applications frequently need to be accessible from a wide variety of platforms. RPC-style programming allows relatively simple client applications running on a wide range of system types to access central or distributed database servers.
- Document imaging systems often manage hundreds of gigabytes of images. The images are typically stored on a central machine and accessed by clients scattered around a network. RPC-style programming is a natural way to simplify the task of writing the clients and communicating with the image server and related database servers.

One theme which is common to all of these examples is that the clients and servers often run on a range of computers running a variety of different operating systems. For example, the server might run on a large UNIX box and the clients might run on Macs, PCs and desktop UNIX systems. Distributing an application across a wide variety of computer architectures often leads to problems in sharing binary data due to differences in how data is represented on the various machines. Let us have a closer look at this issue and how it is addressed within the ONC RPC model.

12.2 Writing An ONC RPC Application

Writing an ONC RPC Application

The steps in writing an ONC RPC application are:

- Define the protocol using the RPC Language .
- Compile the protocol description using **rpcgen**.
- Implement the application specific parts of the client and server programs.

Figure 12-5. Writing an ONC RPC Application

AU253.0

Notes:

The RPC Language (RPCL) is a vaguely C-like language which is used to describe the protocol client programs will use to communicate with the server. We will be looking at it in more detail shortly.

rpcgen takes a protocol description in a file with a .x suffix and produces a .h file and associated .c files that implement most of the protocol.

The application programmer must write the client programs (except for the stub routines described earlier) and finish the server by implementing the remote procedures defined by the protocol.

For example, in the hypothetical example given earlier, the application programmer would implement the client procedure that calls the remote service routine and the service routine itself. All other parts of the ONC RPC application are either in the .c files produced by **rpcgen** or in precompiled ONC RPC library routines.

rpcgen

rpcgen compiles a protocol description into:

- A .h file containing various useful C declarations.
- A .c file containing functions that implement most of the client side of the protocol.
- Another .c file containing functions that implement all of the server side of the protocol.
- Yet another .c file containing functions that pack and unpack data structures defined by the protocol.

Figure 12-6. rpcgen

AU253.0

Notes:

If the protocol file is called abc.x, then the files produced by **rpcgen** will be called:

abc.h	The include file
abc_clnt.c	The client-side protocol routines
abc_svc.c	The server side protocol routines
abc_xdr.c	Routines that pack and unpack data structures defined in the .x file

The *.h file contains C-style declarations of the data structures and routines defined by the protocol. This file can be #included by the programmer when writing code that manipulates the data structures defined in the .x file.

The *_clnt.c file contains C code for the client-side ONC RPC stub routines.

The *_svc.c file contains the main() routine for the server side. This main() routine registers the ONC RPC server with the portmap daemon running on the local host and then waits for and deals with requests that arrive from client programs.

The *_xdr.c files contain XDR translation routines that are used by both the client and server code. An XDR translation routine knows how to pack, unpack, or free instances of a particular data type. The *_xdr.c file will contain an XDR translation routine for each data type defined in the *.x file.

These XDR translation routines are used as follows:

- The client-side stub routines use the appropriate XDR translation routines to pack its parameters into a message, which is then sent to the server.
- The server uses the same XDR translation routine to unpack the parameter and pass it to the server-side procedure that implements the requested remote procedure call.
- When the server-side remote procedure returns, an appropriate XDR translation routine is used to pack the return value so that it can be sent back to the client. Also, the XDR translation routine corresponding to the parameter received from the client is called to free any memory that might have been dynamically allocated when the parameter was unpacked earlier.
- When the reply arrives back on the client, the stub routine uses the same XDR translation routine to unpack the return value and return it to its caller.
- If any memory was dynamically allocated when the return value was unpacked, the client frees the memory using the same XDR translation routine.

These files contain source code and declarations that have been generated by **rpcgen**. Consequently, they can be rather hard to understand.

RPC Protocol Identifiers and Versions

- Each ONC RPC protocol has an ID number which is:
 - (Theoretically) unique across all ONC RPC protocols.
 - Used by the client to indicate which server on a particular machine it wants to communicate with
- Each ONC RPC protocol can also have one or more versions:
 - Versions are distinguished by the version ID in the program clause of the RPCL protocol description.
 - A server can support multiple simultaneous versions of the protocol.
 - Version IDs can be used to ensure that the client and the server are using the same version of the protocol.

Figure 12-7. RPC Protocol Identifiers and Versions

AU253.0

Notes:

The RPC identifier is supposed to be unique worldwide. At a bare minimum, it must not be the same as any other RPC protocol running on any of the machines that a client or server for this protocol might run on. There is an official database of registered protocol numbers maintained by Sun Microsystems, the original developers of the RPC facility described by this unit.

Otherwise, look in `/etc/rpc` on your system and pick a number that is not already there. If you get your system administrator to add your number to the `/etc/rpc` file, then the next person to write an RPC application will not think that your number is available. Note that this technique is not foolproof; you might pick a number that is used by a software product that gets installed next week. The result is likely to be pretty confusing.

Protocol version numbers are unique within the protocol. This is a lot easier to manage, because the usual convention is to define the first version of the protocol as version 1 and then simply increment the version number as the protocol evolves in response to changing needs. A single server might support multiple versions of the same protocol or two or more servers might each support different versions of the protocol.

Some technical details on what happens when a server starts up might be appropriate here:

- When a server starts up, it registers itself with a daemon on the local machine called the portmap daemon. This registration consists of the server telling the portmap daemon which protocol ID and protocol version ID it is implementing and how clients should contact it (that is, what network protocol they should use). The portmap daemon assumes that the last server to register with a particular protocol ID and version ID combination is the valid server.
- When a client starts up, it contacts the portmap daemon on the machine that is supposed to be running the server and asks for the registration for a server with the desired protocol ID, version ID, and network protocol mechanism. The portmap daemon returns the information for the last server that registered itself for the desired protocol ID, version ID, and network protocol mechanism.

The bottom line is that if two servers start up on the same machine for the same protocol ID, version ID, and network protocol, then the last one to register will get all the clients and the first one will get no clients.

The Protocol Description File

- Here's an example RPC protocol description file - baseball.x

```
const MAX_STR = 32;          /* Maximum city or team name size */

struct reply {
    string city<MAX_STR>;    /* city name */
    string team<MAX_STR>;    /* team name */
    int founded;             /* Year team was founded */
};

program TEAMDB {
    version V1 {
        reply GETCITY(string) = 1; /* Get record for a city */
        reply CHAMPION(int) = 2;  /* Who won the world series? */
    } = 1;                      /* version number */
} = 0x1234;                    /* protocol RPC ID */
```

Figure 12-8. The Protocol Description File

AU253.0

Notes:

This is the RPC protocol definition file for a simple application written in the ONC RPC language (RPCL). The name of the protocol definition file must end with a .x and, by convention, is based on or derived from the name of the application.

This protocol defines a constant, a struct, and a program. The RPCL language also has ways of defining enumerations, unions, and typedefs.

RPCL enumerations, structs, and typedefs are essentially identical to their C counterparts. const clauses are used to define symbolic constants (much like the const constructs supported by Pascal and C++). RPCL unions are discriminated unions. Unlike a C union, they have a field whose value indicates which of the union variants is active right now (refer to the AIX online documentation or the *Power Programming with RPC* book by John Bloomer mentioned earlier for more information).

In this example, the reply struct is returned by the GETCITY service routine. This struct contains two string fields and an int. The string base type is used to describe fields that point to '\0' terminated character strings. The value within the angle brackets indicates the maximum size of the string (excluding the terminating '\0').

The program clause names the protocol (TEAMDB in the example), specifies its protocol ID, and describes the set of remote procedures (often called service routines) supported by the protocol. In addition, each program clause ends with an equal sign followed by the protocol's RPC identifier, (more on this shortly). Although fairly unusual, it is possible for an .x file to contain more than one protocol.

A program clause contains one or more version clauses, each of which describes a particular version of the protocol currently supported by the server. A version clause names the version (V1 in this example), specifies the version number (1 in this example), and contains a description of the remote procedures provided by the version of the protocol. In this example, both of the remote procedures return a reply struct and take a parameter. The number after the equal sign in each of the remote procedure declarations is the code that is passed from the client to the server to indicate which remote procedure is to be invoked. These code numbers must be positive integers and each remote procedure within a version must have a different code number associated with it.

One important limitation to keep in mind is that remote procedures must take exactly one parameter. If more than one parameter is required, define the routine to take a struct containing the real parameters.

An Example Problem

The baseball team lookup example (again)

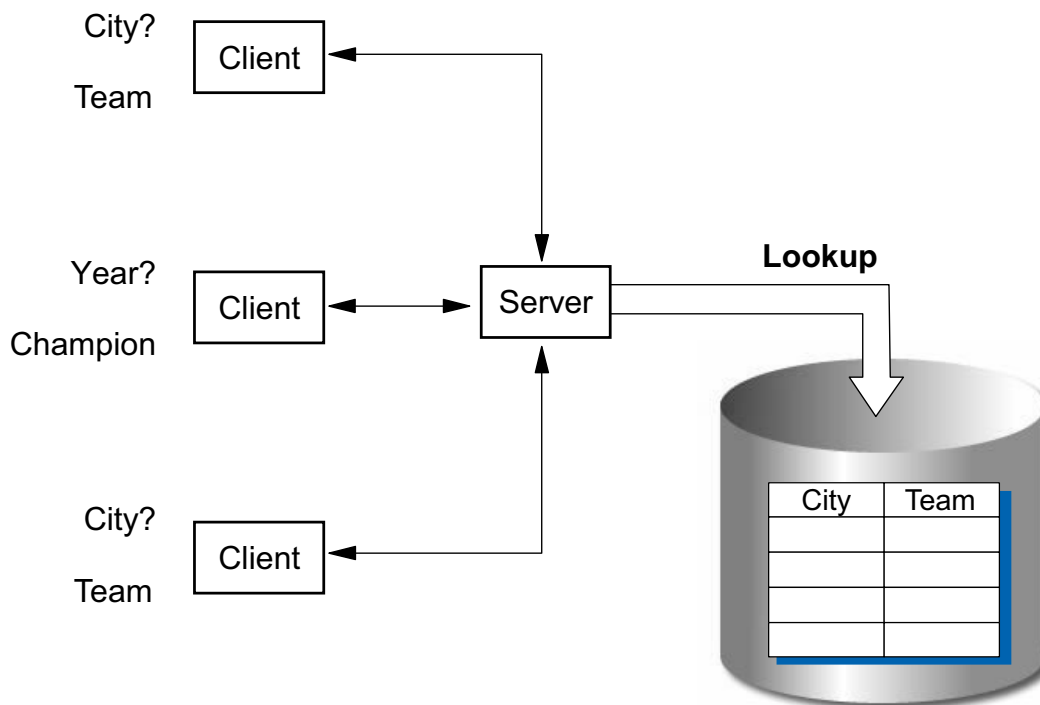


Figure 12-9. An Example Problem

AU253.0

Notes:

Let us revisit the baseball team lookup example which we implemented earlier with message queues. As this is a client server application, it is natural to implement it using RPC.

One change that we will make is to specify that the reply is to include the year that the team was founded. We'll also provide a remote procedure that returns the record for the team that won the league championship in a specified year. These changes allow us to look at more of the ONC RPC facility.

We will first define the protocol to be used to communicate between the client and the server. Next, we will use **rpcgen** to generate the utility routines that implement the protocol. Finally, we will fill in a few gaps and end up with a working application.

For reference purposes (and so that you can follow along as the instructor explains how to implement this application), the files that implement this application can be found in /home/workshop/AU25/example/unit12 directory. Otherwise, you can ask your instructor for a proper directory.

The baseball.x File

```
const MAX_STR = 32;          /* Maximum city or team name size */

struct reply {
    string city<MAX_STR>;    /* city name */
    string team<MAX_STR>;    /* team name */
    int founded;             /* Year team was founded */
};

program TEAMDB {
    version V1 {
        reply GETCITY(string) = 1; /* Get record for a city */
        reply CHAMPION(int) = 2;   /* Who won the world series? */
    } = 1;                       /* version number */
} = 0x1234;                     /* protocol RPC ID */
```

Figure 12-10. The baseball.x File

AU253.0

Notes:

This is the RPC protocol definition file for our application. Following the convention described earlier, this file is named `baseball.x`.

We first declare a constant that specifies the maximum size of any string that we will be using. We next declare a C-style struct that we will be using as the return value of one of our service routines.

Finally, we use a program statement to define the protocol itself. There are two remote service routines. The `GETCITY` routine takes the name of a city and returns a reply struct containing the city name, team name, and the year the team was founded. The `CHAMPION` routine takes a year and returns a reply struct describing the team that won the championship in that year.

Having defined the protocol, the next step is to run **rpcgen** on the `.x` file. This will result in the creation of several files.

rpcgen `baseball.x`

The files created are `baseball.h`, `baseball_clnt.c`, `baseball_svc.c`, and `baseball_xdr.c`.

baseball.h - The protocol Header File

```
#define MAX_STR 32

struct reply {
    char *city;
    char *team;
    int founded;
};
typedef struct reply reply;
bool_t xdr_reply();

#define TEAMDB ((u_long)0x1234)
#define V1 ((u_long)1)
#define GETCITY ((u_long)1)
extern reply *getcity_1();
#define CHAMPION ((u_long)2)
extern reply *champion_1();
```

Figure 12-11. baseball.h - The protocol Header File

AU253.0

Notes:

The baseball.h file is by far the simplest of the four files produced by **rpcgen**. Fortunately, it is also the only one that we really need to have a close look at.

Comparing this to the baseball.x file, we see that the definitions in the .x file have been transformed into their C language equivalents:

- MAX_STR is now a #define symbol with a value of 32.
- The reply struct is unchanged, except that the city and team strings are now char * 's (character pointers).
- The program clause in the .x file has been turned into #define statements and C-style function declarations.

Note that each of the function names has been suffixed with the version number (that is, 1) from the program statement. This convention makes it possible for a single server to support multiple versions of the same protocol by ensuring that function names in different versions have different names.

bbfuncs.c - Server Side Implementation Routines

```
#include <stdio.h>
#include <string.h>
#include <rpc/rpc.h>
#include "baseball.h"

reply *
getcity_1(city_pp)
char **city_pp;
{
    char *cityname = *city_pp;
    static reply r;

    /* pretend that all cities have a team called the Mud Hens */

    r.city = cityname;
    r.team = "Mud Hens";
    r.founded = 1997;
    return(&r);
}

reply *
champion_1(year_p)
int *year_p;
{
    int year = *year_p;
    static reply r;

    /*
     * The Sherwood Park Mud Hens won in 1997.
     * Nobody won in any other year.
     */

    if ( year == 1997 ) {
        r.city = "Sherwood Park";
        r.team = "Mud Hens";
        r.founded = 1997;
        return(&r);
    } else {
        r.city = NULL;
        r.team = NULL;
        r.founded = 0;
        return(&r);
    }
}
```

Figure 12-12. bbfuncs.c - Server Side Implementation Routines

AU253.0

Notes:

These are the somewhat simplified routines that we need to write to complete the server. Note that each routine takes a pointer to its parameter and returns a pointer to its return value. As mentioned earlier, all ONC RPC remote procedures take a single parameter and return a single value.

Unlike the .c and .h files which we just discussed, we are responsible for writing the routines in this file, because they are application specific.

Once we have written these routines, we can compile the server:

```
$ cc baseball_svc.c baseball_xdr.c bbfuncs.c -o bbserver
```

bbclient.c - A Simple Baseball Client Program

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "baseball.h"

main(int argc, char **argv)
{
    CLIENT *cl;
    reply *rp;

    if ( argc != 3 ) {
        fprintf(stderr, "Usage:  bbclient server_host parm\n");
        fprintf(stderr, "\tserver_host : which host the server is running on\n");
        fprintf(stderr, "\tparm : either a year or a city name\n");
        exit(1);
    }

    /* Create the client record */
    cl = clnt_create(argv[1], TEAMDB, V1, "udp");
    if ( cl == NULL ) {
        /* Can't find the server */
        clnt_pcreateerror(argv[1]);
        exit(1);
    }

    if ( isdigit(argv[2][0]) ) {
        int year;
        year = atoi(argv[2]);
        rp = champion_1(&year, cl);
        if ( rp->city == NULL || strlen(rp->city) == 0 ) {
            printf("Nobody won in %d.\n", year);
        } else {
            printf("The %s %s won in %d.\n", rp->city, rp->team, year);
        }
        xdr_free(xdr_reply, rp);
    } else {
        rp = getcity_1(&argv[2], cl);
        if ( rp->city == NULL || strlen(rp->city) == 0 ) {
            printf("There is no team in %s.\n", argv[2]);
        } else {
            printf("The %s team is called the %s and was founded in %d.\n",
                rp->city, rp->team, rp->founded);
        }
        xdr_free(xdr_reply, rp);
    }
    exit(0);
}
```

Figure 12-13. bbclient.c - A Simple Baseball Client Program

AU253.0

Notes:

This is a simple client program for our baseball protocol. There are a few statements here which required explanation:

```
cl = clnt_create(argv[1], TEAMDB, V1, "udp");
```

This statement contacts the portmap daemon on the host specified by the first parameter and asks it to find a particular ONC RPC server. The server in question is identified by a protocol ID (TEAMDB), a protocol version number (V1), and the networking protocol that the client wants to use to communicate with the server (udp). Additional information on the portmap daemon is available in the AIX online documentation or the *Power Programming with RPC* book.

If the attempt to find the server fails, then `clnt_create()` returns NULL. In this event, this client program emits an appropriate error message using `clnt_pcreateerror` (an ONC RPC routine roughly analogous to `perror`) and terminates.

```
rp = champion_1(&year, cl);
rp = getcity_1(&argv[2], cl);
```

These statements are the actual remote procedure call statements. They each call a client-side subroutine, passing it the address of a single parameter and a pointer to the CLIENT object returned by `clnt_create()`. The routines return a pointer to the value returned by the remote procedure, which the client software interprets as appropriate.

The `_1` suffix on each name corresponds to the version number of the protocol that this routine comes from. In order to make it easier to change the version number of a protocol, ONC RPC programmers often write their own client-side subroutines with unsuffixed names. These routines are responsible for calling the real ONC RPC subroutine.

```
xdr_free(xdr_reply, rp);
```

`xdr_free` is a precompiled ONC RPC library routine. It takes an XDR translation routine and a pointer to an instance of the data type understood by the translation routine. `xdr_free` uses the XDR translation routine to free any dynamically allocated memory associated with the pointer at object (it does not free the object itself).

This convention of not freeing the object itself allows the client-side subroutine to use a static variable for its return value. This in turn means that the code that calls the client-side subroutine does not need to call `xdr_free` unless the return value is a struct that has fields that point to other objects. Unfortunately, it also means that calls to an ONC RPC client subroutine must be protected by a mutex if called in a POSIX threads environment (this last sentence will make more sense once we have completed the threads programming units that come later in this course).

The rest of this client program is relatively conventional C code. It first ensures that the user specified two parameters. Once it has found the server, it checks if the second parameter starts with a digit. If it does, then it passes what it assumes is a year to the `champion_1` routine to find out who won the league championship in the specified year. Otherwise, it calls the `getc_1` routine to find the name of the team that plays in the specified city.

Because the rest of the client program is in `baseball_clnt.c`, `baseball_xdr.c`, or in the precompiled ONC RPC libraries, the next step is to compile the client program:

```
$ cc baseball_clnt.c baseball_xdr.c bbclient.c -o bblclient
```

Running the Programs

Start the server (executed as a background process on a host called yoho)

```
$ ./bbserver &
```

What team plays in Townsville? (executed anywhere in the network)

```
$ ./bbclient yoho Townsville
The Townsville team is called the Mud Hens and was founded in 1997.
```

Who won in 1997? (also executed anywhere in the network)

```
$ ./bbclient yoho 1997
The Sherwood Park Mud Hens won in 1997.
```

Figure 12-14. Running the Programs

AU253.0

Notes:

The server does not take any parameters and can be started by just specifying its path name, as shown in Figure 12-14.

The client requires the name of the host that the server is running on and either a year or a city name. Note that the client commands can be run on any host on the network, including the host that the server is running on.

For completeness, here is a Makefile that could be used to compile the application that we just developed:

```
DBX = -g
CFLAGS = $(DBX)

all : bbserver bbclient

bbserver: baseball.h baseball_svc.o baseball_xdr.o bbfuncs.o
        $(CC) $(DBX) baseball_svc.o baseball_xdr.o bbfuncs.o -o bbserver
```

```
bbclient: baseball.h baseball_clnt.o baseball_xdr.o bbclient.o
        $(CC) $(DBX) baseball_clnt.o baseball_xdr.o bbclient.o -o
bbclient

baseball.h baseball_svc.c baseball_xdr.c baseball_clnt.c : baseball.x
        rpcgen baseball.x

clean :
        rm -f *.o

clobber : clean
        rm -f baseball.h baseball_clnt.c baseball_svc.c baseball_xdr.c \
        bbserver bbclient
```

This Makefile and all the other files required to implement this application can be found in /home/workshop/AU25/example/unit12 directory. Otherwise, you can ask your instructor for a proper directory.

Unit Summary

- Distributed computing
- ONC RPC
- **rpcgen**
- Example application

Figure 12-15. Unit Summary

AU253.0

Notes:

The Open Network Computing (ONC) Remote Procedure Call (RPC) facility can be used to develop client server applications.

The **rpcgen** program is used to compile a protocol description in a *.x file. The result is a *_clnt.c file containing client-side subroutines, a *_svc.c file containing the server's main() routine, a *_xdr.c file containing routines to pack and unpack data structures, and a *.h file. The *.h file contains various declarations needed by the above *.c files and the source code written by the application developer.

The ONC RPC programmer must implement the server-side remote procedures as well as any client programs required by the application.

ONC RPC allows the programmer to focus primarily on the application by taking care of most of the communication details between client and server.

Unit 13. Threads: Getting Started

What This Unit is About

This lecture introduces the major thread concepts and discusses enough basic thread calls to get a simple program operational.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Explain thread concepts.
- Produce a simple POSIX threads program.

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 11

Unit Objectives

After completing this unit, you should be able to:

- Explain thread concepts.
- Produce a simple POSIX threads program.

Figure 13-1. Unit Objectives

AU253.0

Notes:

13.1 Thread Concepts

Threads versus Processes

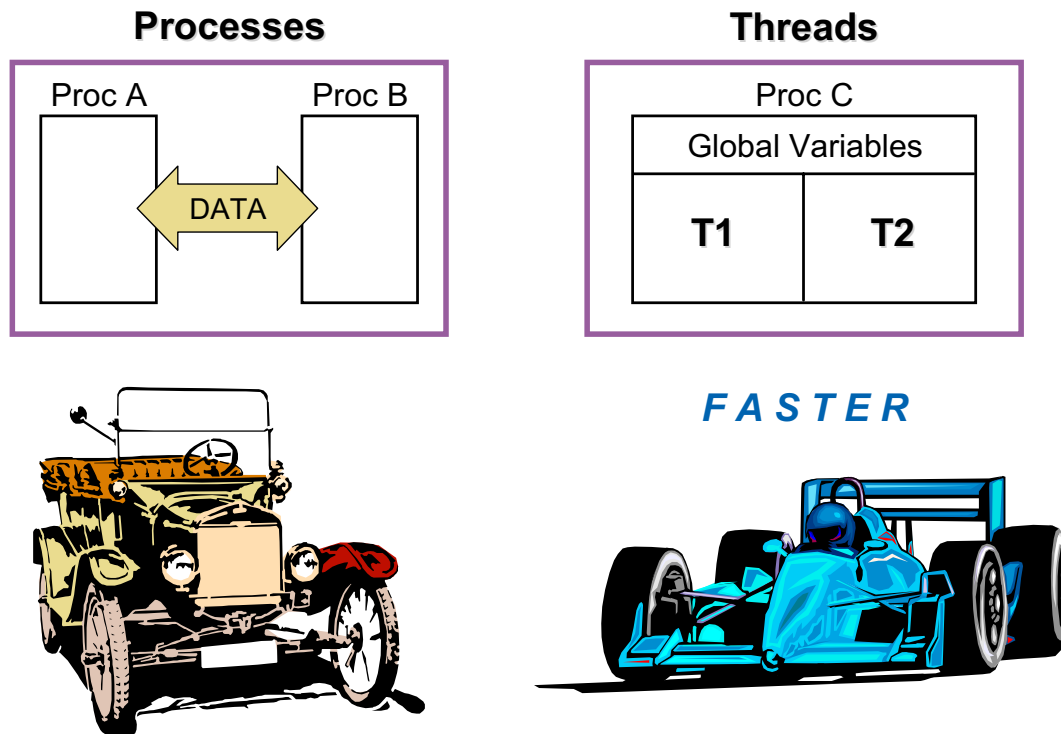


Figure 13-2. Threads versus Processes

AU253.0

Notes:

A process in a multithreaded system is a changeable entity. It has all traditional process attributes, such as:

- Process ID, process group ID, and so on.
- Environment.
- Working directory.
- A common address space and common system resources, such as file descriptors, shared libraries, and IPC tools (semaphore, message queues, pipes, and shared memory).

A thread is a schedulable entity. It has the following properties required to ensure its independent flow of control:

- Stack.
- Scheduling properties, such as priority and policy.
- Set of pending and blocked signals.
- Some thread specific data.
- They have global resources and characteristics.

Threads are must faster than processes because of local data resources.

Meeting POSIX Standards



IBM

POSIX

POSIX 1003.4a Draft 7 (POSIX.1c)

Figure 13-3. Meeting POSIX Standards

AU253.0

Notes:

AIX Version 4 supports multithreaded programming based on the POSIX 1003.4a Draft 7 specification.

POSIX.4a has been renamed to POSIX.1c. The reason for this renaming is to associate it to the POSIX programming base (POSIX.1).

POSIX.4a Draft 10 is already out and is now a standard.

The bottom line is that programs should be widely portable.

Traditional Approach: Direct Manipulation

```
main()  
{  
    struct data mydata;  
    mydata.field1=0;  
    mydata.field2=5;  
  
    funca();  
    .  
    .  
    .  
}
```

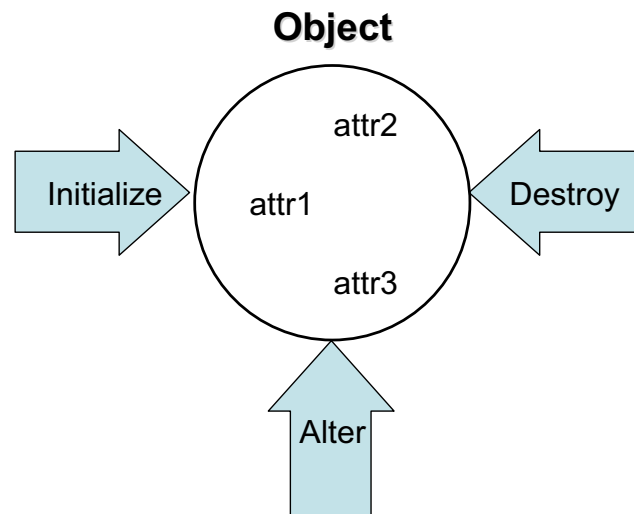
Figure 13-4. Traditional Approach: Direct Manipulation

AU253.0

Notes:

Before object-oriented principles, it was hard to conceive of portability of threads for many architectures. Low-level structures had to be implemented directly on all platforms.

Global Approach: Objects



Examples:

Thread	(pthread_t)
Thread Attributes	(pthread_attr_t)
Mutex	(pthread_mutex_t)
Mutex Attributes	(pthread_mutexattr_t)

Figure 13-5. Global Approach: Objects

AU253.0

Notes:

Structures and other data describing characteristics are kept inside an object.

Objects are typically unknown data types that we affect through calls versus altering the data directly.

Object characteristics are often changed by changing the attribute object associated with the object (that is, changing the thread's attribute versus changing the thread directly). Again, this is still done by using calls versus direct manipulation.

Most objects are initialized, used, and then destroyed. These are all done typically by using a call. Words like create, init, delete, destroy, cancel, get, and set on the call should give you an indication of what the call does.

A normal method of creating an object is:

- Initialize the attribute.
- Set the attribute's characteristics.
- Use the attribute on the object creation call.

More Thread Terminology

- Reentrant
- Thread Safe
- Thread Efficient

Figure 13-6. More Thread Terminology

AU253.0

Notes:

Reentrant functions are functions that can be shared by many processes at the same time. This would mean that when one process is executing the reentrant function, another process can begin execution in parallel. This is because reentrant functions do not contain static variables, or return pointers to static variables. They also must not call non-reentrant functions.

Thread safe functions are functions that protect shared resources from concurrent access. These functions lock access to global resources (that is, global variables, file descriptors, and other shared resources). They obviously also must call nothing but thread safe functions.

Thread efficient functions are functions designed from the ground up with parallel efficiency in mind.

Thread safe libraries include the standard C library, the libbsd.a library, and the new libc_r.a library.

Some functions in the standard C library are non-reentrant (such as strtok() and ctime()). Hence, the newer C library (libc_r.a) should be linked to instead.

Protecting Data Resources

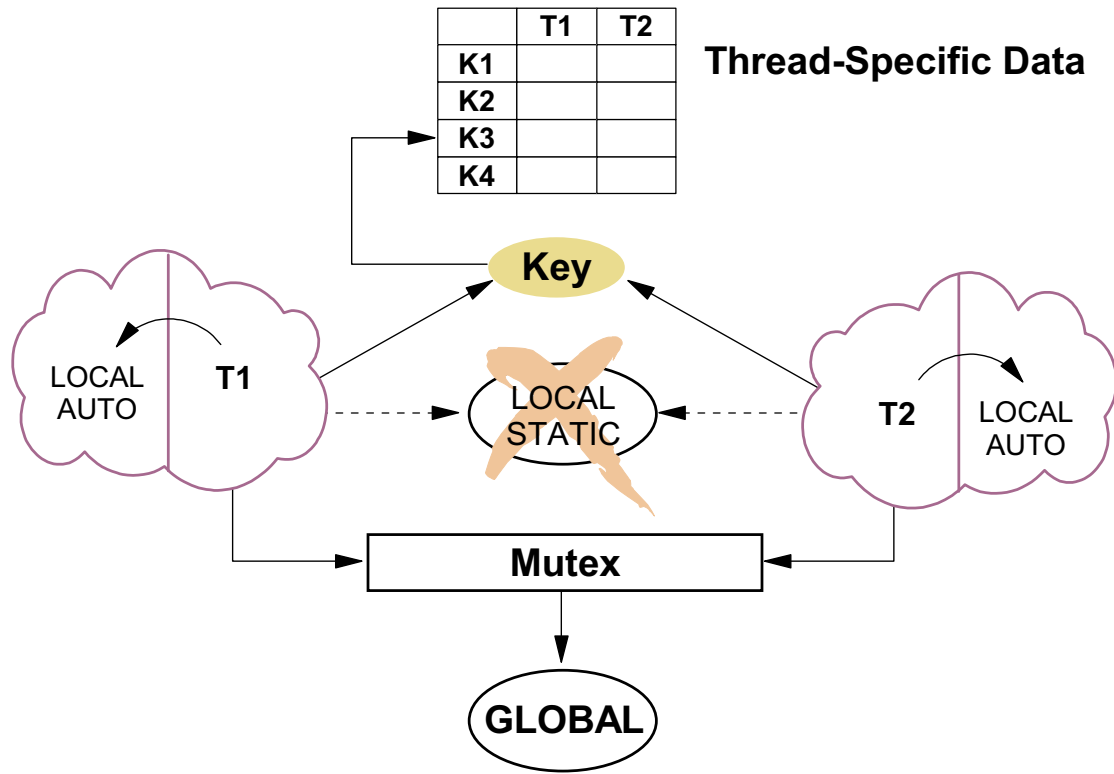


Figure 13-7. Protecting Data Resources

AU253.0

Notes:

Local Auto Variables are handled on the stack and are separate per thread. These variables are inherently safe from harm. However, they only last while executing the function they were defined in.

Global Static Variables are kept in the process's global data area, but can be easily protected by using a locking mechanism such as mutexes. This provides a thread safe approach for variables that are shared across the process.

Local Static Variables are kept in the process's global data area because they are stagnant across function calls. Because of this, they should be protected by a locking mechanism such as mutexes or eliminated.

Thread Specific Data (TSD) is kept in the process's common global data area. A `malloc()` call is typically used to set up the space for the data. A common *key* is the mechanism used to retrieve the data. The *thread ID* determines which thread's data to get. Hence, it provides a thread safe mechanism to access global variable data that is unique to each thread.

Controlling Thread Execution

- Create
- Exit
- Wait/Interrupt
- Cancel
- Schedule
- Other

Figure 13-8. Controlling Thread Execution

AU253.0

Notes:

Wait is a broad topic. It can mean waiting on a data resource (that is, probably by mutexes), time (maybe by sleep), another thread's completion of a given task (condition variables), on a signal (via a new sigwait call), or even normal I/O. Interrupts typically happen when one of these conditions is met from the outside.

There are many ways to cancel a process. Typically, you would either send a thread kill signal, or issue a cancel call. Cancellation handling can become a big task if you use it to its utmost capability.

Other items include special capabilities to make sure a piece of code is executed only once, and special forking considerations.

Thread Inheritance

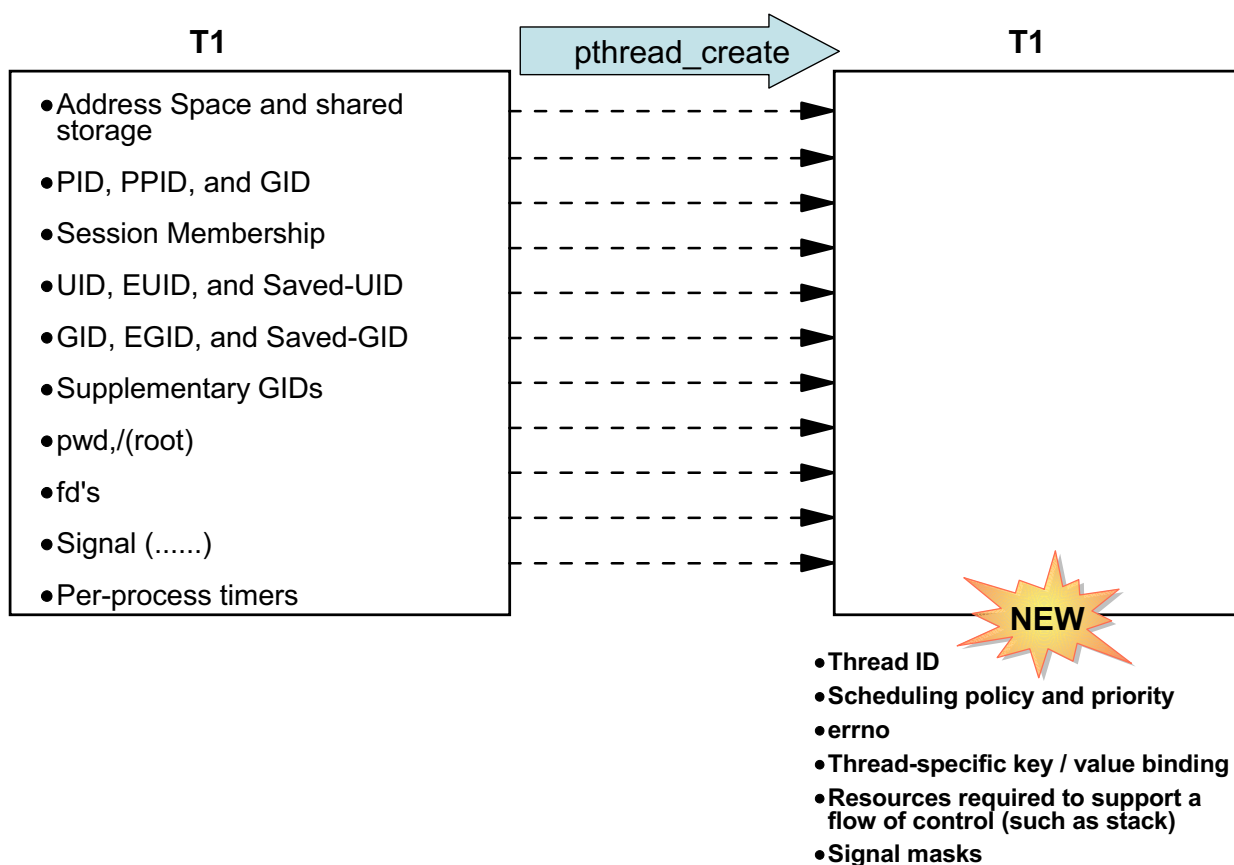


Figure 13-9. Thread Inheritance

AU253.0

Notes:

The major emphasis of this foil is to understand where is the need to handle particular resources, whether on a thread basis or on a process basis. For example, signal handlers are on a process wide basis. Hence, the main thread may want to set these up, or you may have a dedicated thread to handle signals. Signal blocking, on the other hand, is controlled on a individual thread basis.

Scheduling policy and priority are inherited from the parent, but can change on a per-thread basis.

TSD key/value bindings means that there is another column added to the TSD table when your thread starts. This allows you to set up your own data using the common process key.

Signal handlers are process wide (that is, SIG_IGNs and signal handler routines). Only blocking a signal is unique to the thread. Use the `sigthreadmask()` call to do this.

Watch out for common file descriptors. This means that if you treat the FD as a global variable, then multiple threads can easily access the same file and can possibly cause contention unless the file descriptor is protected by a mutex or something equivalent. If the

open() call happens in the thread, then chances are that the other threads are not aware of the file descriptor.

Review - Match the Terms to Their Description

Thread Safe	The data that is global only to a unique thread.
Reentrant	The data that is found on the thread unique stack that goes away when the function is done.
Thread Efficient	The function that contains no local static variables.
Mutex	The object used to lock global variables.
Thread Specific Data	The object that contains characteristics describing another object.
Object	The function that locks global variables before accessing them.
Attribute	The program that was designed from the beginning with parallelization and speed.
Local Auto Variable	The entity that typically has hidden information that operates according to set attributes.

Figure 13-10. Review - Match the Terms to Their Description

AU253.0

Notes:

Match the Terms to Their Description-Answers

Thread Safe	The function that locks global variables before accessing them.
Reentrant	The function that contains no local static variables.
Thread Efficient	The program that was designed from the beginning with parallelization and speed.
Mutex	The object used to lock global variables.
Thread Specific Data	The data that is global only to a unique thread.
Object	The entity that typically has hidden information that operates according to set attributes.
Attribute	The object that contains characteristics describing another object.
Local Variable Data	The data that is found on the thread unique stack that goes away when the function is done.

Figure 13-11. Match the Terms to Their Description - Answers

AU253.0

Notes:

13.2 Producing a Simple Threads Program

Call Comparison

PROCESSES	THREADS
fork	pthread_create
exit	pthread_exit
wait	pthread_join
kill	pthread_cancel
semop	pthread_mutex_lock
	pthread_cond_wait
	pthread_cond_timedwait

Figure 13-12. Call Comparison

AU253.0

Notes:

There really is not a one-to-one correspondence to the normal process calls, but they are close enough for comparison purposes.

pthread_cond_wait() and pthread_cond_timed_wait() have no real equivalent in process terminology. Probably their closest comparison is to the GETVAL command on the semctl() call, if it had a more sophisticated wait ability.

Thread Creation

```
int pthread_create(thread, attr, function, arg )
```

- Initiates an additional thread beyond the main thread.
- *thread* is a pthread_t pointer identifying the Thread ID being created
- *attr* is a pthread_attr_t pointer that describes the type of thread being created.
- *function* is a void pointer to a function that will start running when the thread is initially dispatched.
- *arg* is a void pointer to an argument to be passed to the function listed above when it is dispatched.
- Includes pthread.h header file.

Figure 13-13. Thread Creation

AU253.0

Notes:

Syntax:

```
int pthread_create(pthread_t thread, const pthread_attr_t * attr,  
(void *) (* function)(void), void * arg)
```

The main reason this call would fail would be due to a shortage of resources. For example, each process is limited to a maximum of 512 threads. This is defined by PTHREAD_THREADS_MAX in pthread.h

Note that no attributes were used. This means a "default" thread is created. We will discuss what this means soon.

Note that just a single argument was passed to the function this time. Because the type is void on the argument being passed, you could pass anything, even a large structure with many arguments inside. Just remember to cast it as type void on the pthread_create() call, and uncast it back when you get into the "funca" function.

The new thread inherits its creating thread's signal mask, but any pending signal of the creating thread will be cleared for the new thread.

For example:

```
#include <pthread.h>
main()
{
    pthread_t thd1;
    short rc;
    int arg1=1;
    rc=pthread_create(&thd1,NULL,funca,(void *) &arg1);
    ...
}
```

Thread Deletion

```
void pthread_exit ( void * return_data )
```

- Ends this thread's activity.
- *return_data* is a void pointer to data that is to be returned to the joining (waiting) thread
- Includes pthread.h header file.

Figure 13-14. Thread Deletion

AU253.0

Notes:

If an `exit()` call is used instead of `pthread_exit()`, then this thread will terminate the entire process. This is true even for the main function.

A `return()` call can be used instead of `pthread_exit()` unless you are the main thread. In the main thread, terminate the entire process with either a `return()` or `exit()` call.

Because of the void pointer used on the `pthread_exit()` call, you can pass all kinds of information back to the joining thread. Use the same principle here as the argument passing concept on the `pthread_create()` call discussed on the previous foil.

Note that you can only give exit code information from a thread that is in the UNDETACHED state. We'll talk about this after showing you a thread example.

For example:

```
#include <pthread.h>
void * funca(void * notused)
{
    int rc;
```

```
rc=some_operation;

pthread_exit((void *) rc);
}
```

Basic Thread Example

```
/* Program name - thrd1.c */

#include <pthread.h> /* for pthread defns and calls */
#include <stdio.h>   /* for printf */

void * funca(void * arg) /* Thread thd1 = T2 */
{
    printf("Thread says: %s\n", (char *) arg);
    pthread_exit(NULL);
}

main() /* main thread - T1 */
{
    pthread_t thd1;
    int rc;
    char buffer[] = "Hi there!";

    rc=pthread_create(&thd1, NULL, funca, (void *) buffer);

    if (rc){
        /* error processing */
    }
    pthread_exit(NULL);
}
```

Figure 13-15. Basic Thread Example

AU253.0

Notes:

The example program creates a thread and passes it a buffer argument to be printed.

The output of the program is:

Thread says: Hi There!

Thread Compilation

Use:

```
cc_r -o basic basic.c
```

Assumes /etc/vac.cfg:

```
cc_r:
```

```
libraries = -L/usr/lib/threads, -lc_r, -lpthreads, -lc  
options   = -D_THREAD_SAFE
```

Figure 13-16. Thread Compilation

AU253.0

Notes:

All thread programs have to be compiled using the `cc_r` compiler. This is a new option that has been available since AIX Version 4 to handle threads. It is one of the many new compiler options to handle thread activity. These options come shipped with the C for AIX compiler. The options for the `cc_r` compiler are the same as for the `cc` compiler.

The most important library is the `pthread` library. This is where all the `pthread` calls are found. However, some reentrant versions of the C library functions may also be needed (such as `strtok_r()` and `ctime_r()`) and are found in `libc_r.a`. Not many functions fit into this category. Typically, those functions that deal with time or hold other constant data are suspects. Whenever you find a counterpart call in this library, always use it. An easy way to do this is to issue the `nm` command against the `libc_r.a` library and see if your function exists with a `_r` suffix. If so, you need to use it. For further information, refer to the AIX online documentation.

The `_THREAD_SAFE` macro causes the preprocessor to pick up the right prototypes and definitions from the header files to handle the new reentrant functions and other miscellaneous features, such as the new thread specific `errno` variable change.

Getting the Thread Return_Data (1 of 2)

```
int pthread_join( pthread_t thread, void ** return_data )
```

- Waits on the given thread to finish and stores its return_data.
- *thread* is a pthread_t data type representing the exiting thread of interest.
- *return_data* is a void double pointer (void **) to where the data_returned should be stored.
- Includes pthread.h header file.

Figure 13-17. Getting the Thread Return_Data (1 of 2)

AU253.0

Notes:

Any thread can wait on any other thread. However, only one thread should wait on another thread. Anything beyond that is considered non-portable. Of course, all this implies that the thread was in the undetached state when it exited. This is not the default, so other calls must be used to ensure that the terminating thread is set up to finish in an undetached state.

For example:

```
#include <pthread.h>
int thd1_rc;
pthread_join(thd1, (void **)&thd1_rc);
```

Getting the Thread Return_Data (2 of 2)

```
int pthread_attr_init( pthread_attr_t * attr )
```

- Initializes a pthread attribute to default values.
- *attr* is a pthread_attr_t pointer to the attribute to initialize.
- Includes pthread.h header file.

Example: pthread_attr_init(&attr);

```
int pthread_attr_setdetachstate( pthread_attr_t * attr, int state)
```

- Sets the detach state on the pthread attribute.
- *attr* is a pthread_attr_t pointer to the attribute to change.
- *state* is an int field that describes what detach state to leave the thread in when the thread is finished. Default is detached. The PTHREAD_CREATE_UNDETACHED macro leaves the thread in an undetached state.
- Includes pthread.h header file.

Example:
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_UNDETACHED);

Figure 13-18. Getting the Thread Return_Data (2 of 2)

AU253.0

Notes:

In order to get the return data from a thread, the thread must be left in an undetached state. Hence, a pthread attribute must be created to produce the detached state effect. This attribute will then be used on the pthread_create() call.

The detach state is the only commonly used pthread attribute. Please refer to AIX online documentation for more details.

Getting the Thread Return_Data Example

```

/* Program name - thrd2.c */

#include <pthread.h> /* for pthread defns and calls */
#include <stdio.h>   /* for printf */

void * funca(void * arg) /* Thread thd1 = T2 */
{
    int rc;
    rc=printf("Main thread says: %s\n", (char *) arg);
    pthread_exit((void *) rc);
}

main() /* main thread - T1 */
{
    pthread_t thd1;
    pthread_attr_t attr;
    int rc;
    char buffer[] = "Hi there!";
    int thd1_rc;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_UNDETACHED);
    rc=pthread_create(&thd1, &attr, funca, (void *) buffer);

    if (rc) {
        /* error processing */
    }

    pthread_join(thd1, (void **) &thd1_rc);
    printf("Thread 1's rc is %d\n", thd1_rc);
    pthread_exit((void *) 0);
}

```

Figure 13-19. Getting the Thread Return_Data Example

AU253.0

Notes:

Output:

```

Main thread says: Hi There!
Thread 1's rc is <num>

```

<num> will be the return value of the printf statement in the funca function.

Unit Summary

- Introduced Thread Concepts
- Produced a Simple POSIX Threads Program

Figure 13-20. Unit Summary

AU253.0

Notes:

Unit 14. Threads: Worrying About Your Neighbor

What This Unit is About

This lecture introduces the concepts and more specific calls to handle the real world of POSIX threads programming. In the real world, a programmer needs to worry about protecting data resources from simultaneous thread updates. A programmer may also be interested in controlling when a given thread executes and how fast it executes.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Protect data resources from multi-thread usage.
- Control thread execution.

How You Will Check Your Progress

You can check your progress by completing

- Lab exercise 12

References

- AIX 5L online documentation

Unit Objectives

After completing this unit, you should be able to:

- Protect data resources from multithread usage.
- Control thread execution.

Figure 14-1. Unit Objectives

AU253.0

Notes:

14.1 Protecting Data Resources

Protecting Data Resources

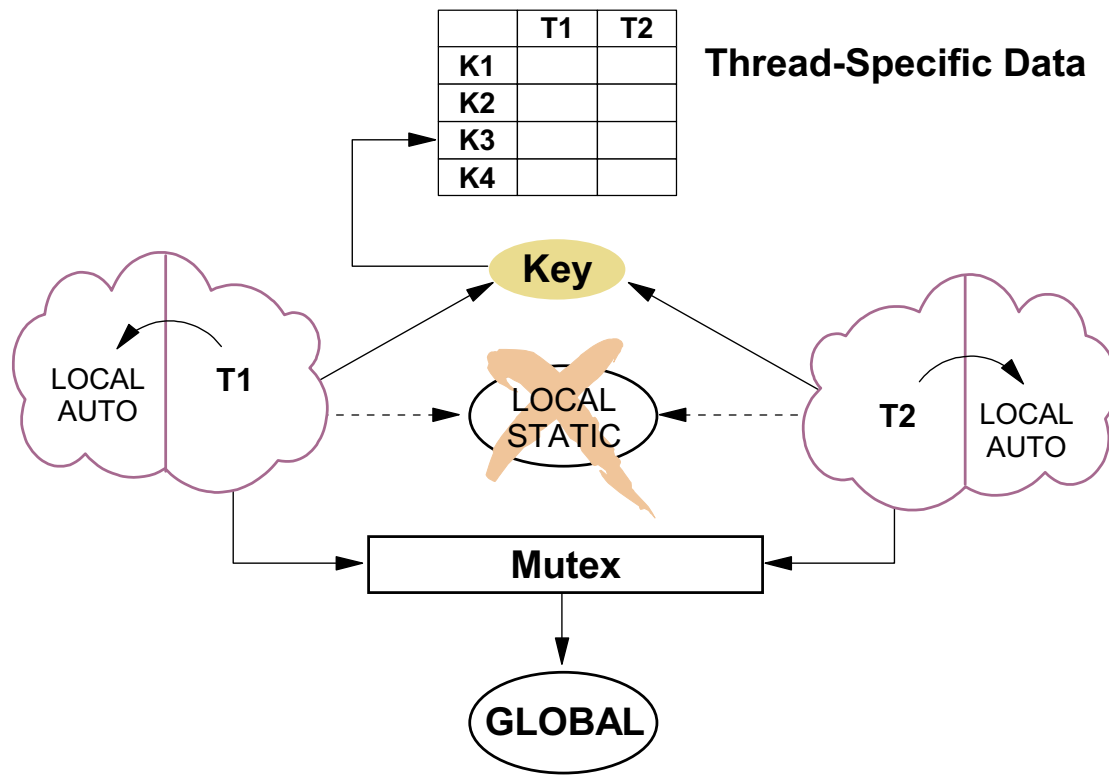


Figure 14-2. Protecting Data Resources

AU253.0

Notes:

Figure 14-2 reviews the data resources we need to protect when writing a multithreaded program, and the techniques generally used to protect them. The only data we do not have to worry about are the local auto variables. They are kept on the local stack. We will discuss how to handle the protection of the remaining data types: global data, thread specific data, and local static data.

Handling Global Data: Mutex

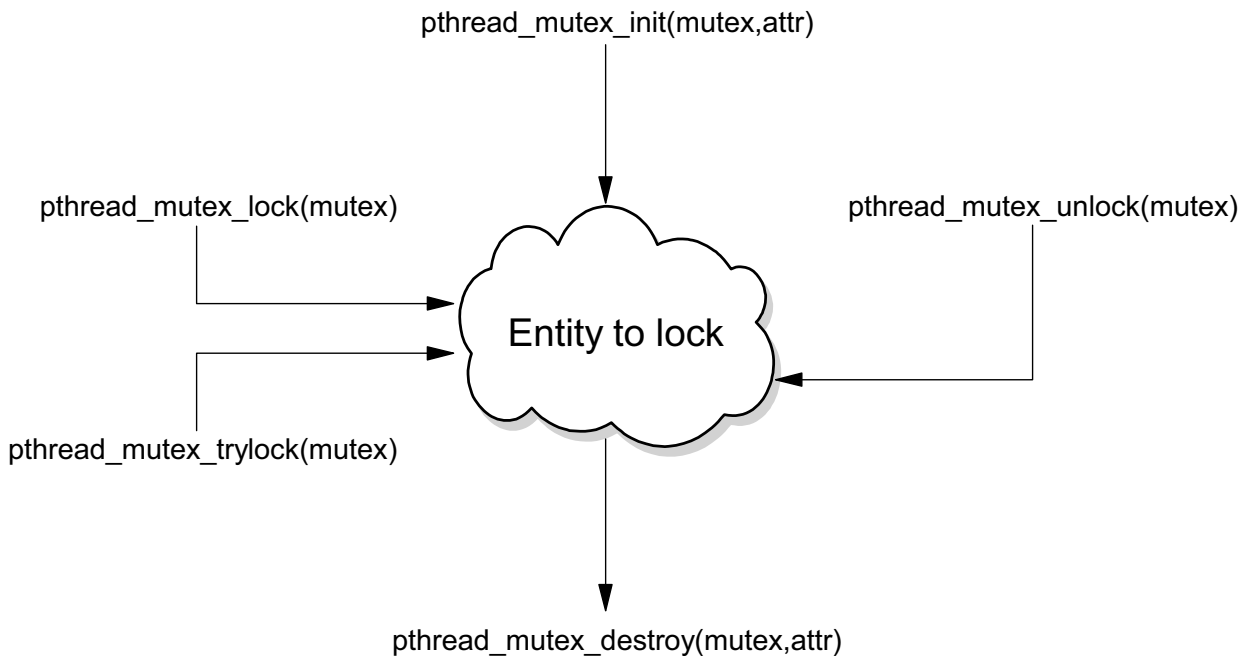


Figure 14-3. Handling Global Data: Mutex

AU253.0

Notes:

mutex is `pthread_mutex_t` pointer to the mutex.

attr is `pthread_mutexattr_t` pointer to the attribute object describing the mutex.

This data is common to all threads in the process. This can be the replacement for the old shared memory. But just like shared memory, we also need a locking mechanism. In this case, it is a mutex.

Mutexes work very similar to semaphores in that they can represent a lock on an undesignated amount of data. The locking portion must be designated up-front.

Mutexes are typically initialized and destroyed in main. This keeps them from accomplishing these tasks accidentally more than once. However, if it must be done from another thread, be sure to use the `pthread_once()` call, as discussed later in the lecture.

Mutexes cannot be destroyed until the lock is cleared.

Optionally, one can initialize and set up an attribute to affect the mutex's performance. The main attribute one might set is the priority of the mutex. Setting the priority can help prevent deadlocks.

The main difference between `pthread_mutex_lock()` and `pthread_mutex_trylock()` is that `pthread_mutex_lock()` waits until a lock can be achieved, whereas `pthread_mutex_trylock()` comes back on an error if the mutex is already locked.

You can produce a timed lock call by implementing mutexes in combination with condition variables. Refer to the AIX online documentation for more details.

Handling Global Data: Example (1 of 2)

```
/* global.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int g_count;
int g_done;
pthread_mutex_t g_count_mutex;

void * callFunc(void * notused)
{
    int i;

    printf("I'm in the child thread now....\n");
    for(i=0;i<10;i++)
    {
        pthread_mutex_lock(&g_count_mutex);
        g_count++;
        pthread_mutex_unlock(&g_count_mutex);
        printf("Count in CHILD is %d\n", g_count);
        sleep(1);
    }
    g_done = 1; /* Child is done with the mutex */
    pthread_exit(NULL);
}
```

Figure 14-4. Handling Global Data: Example (1 of 2)

AU253.0

Notes:

Figure 14-4 demonstrates the use of mutex to safely access the same data between two threads. callFunc() is the entry point for the child thread.

Handling Global Data: Example (2 of 2)

```
main()
{
    pthread_t callThd;
    int ret;
    int i;

    g_count=0;
    g_done=0;
    pthread_mutex_init(&g_count_mutex,NULL);
    ret=pthread_create(&callThd, NULL, callFunc, NULL);
    if(ret) {
        printf("problems on creating thread\n");
        exit(EXIT_FAILURE);
    }
    for(i=0;i<10;i++)
    {
        pthread_mutex_lock(&g_count_mutex);
        g_count++;
        pthread_mutex_unlock(&g_count_mutex);
        printf("Count in PARENT is %d\n", g_count);
        sleep(1);
    }
    while (!g_done)
        sleep(1);
    pthread_mutex_destroy(&g_count_mutex);
    pthread_exit(NULL);
}
```

Figure 14-5. Handling Global Data: Example (2 of 2)

AU253.0

Notes:

Both the threads try to modify the variable `g_count`. Notice the use of mutex to protect the variable `g_count`.

The output from the example is:

```
Count in PARENT is 1
I'm in the child thread now....
Count in CHILD is 2
Count in PARENT is 3
Count in CHILD is 4
Count in PARENT is 5
Count in CHILD is 6
Count in PARENT is 7
Count in CHILD is 8
Count in PARENT is 9
Count in CHILD is 10
Count in PARENT is 11
```

Count in CHILD is 12
Count in PARENT is 13
Count in CHILD is 14
Count in PARENT is 15
Count in CHILD is 16
Count in PARENT is 17
Count in CHILD is 18
Count in PARENT is 19
Count in CHILD is 20

Thread Specific Data: Initialization

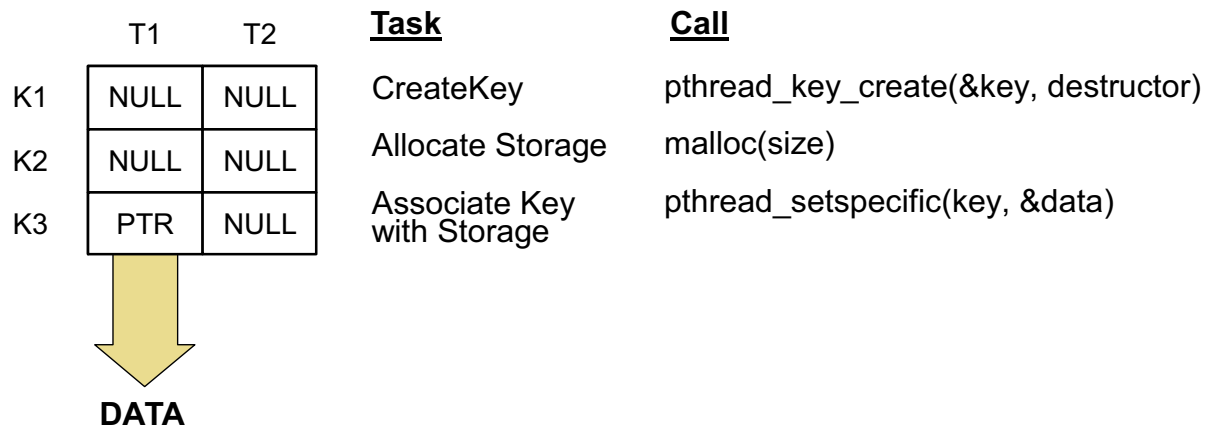


Figure 14-6. Thread Specific Data: Initialization

AU253.0

Notes:

key is a `pthread_key_t` global variable that is used to retrieve thread-specific data.

destructor is a function that is called to clean up the thread-specific data when the thread terminates.

size is the size in bytes that you plan on using for your thread-specific data.

data is your thread-specific data that you wish to associate with the key.

Thread specific data is global only to a particular thread, yet its retrieval is by a global key. The key will point to an address in memory where the data for the *current* thread is actually located. Once a function has retrieved the address of the data, it can manipulate it at will using that address.

The creation of a key creates another row on thread-specific data. This key is process wide, but the data is thread wide. For each thread that gets started up, another column is added to the table.

When the key is made, it fills in the table with NULLS. Only after the `pthread_setspecific()` call does an address get filled in (PTR in the picture). This data is found on the heap, where all functions can access it.

Thread-specific data must be initialized, used, and then destroyed. This page describes the initialization section.

The key must be a global variable that all functions can access. The global variable will be treated as a key as soon as the `pthread_key_create()` call is issued. At this point, no data is assigned to it. In other words, it contains NULL pointers for every currently running thread.

A function must initialize a data area and associate it with the thread. Typically, this is done by testing the data with the `pthread_getspecific()` call. If it returns a NULL pointer, then data has not been assigned to the key. To do this, the function can issue the `malloc` and `pthread_setspecific()` calls.

Thread-Specific Data: Usage and Destruction

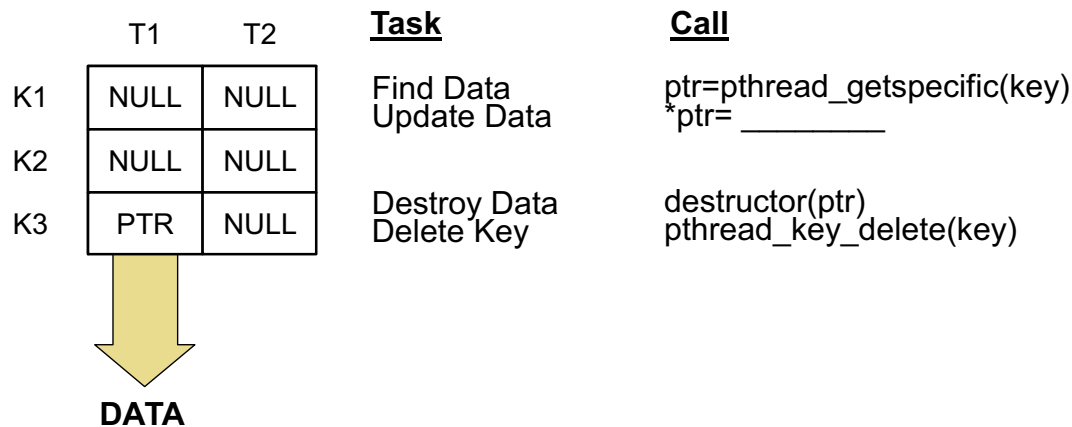


Figure 14-7. Thread-Specific Data: Usage and Destruction

AU253.0

Notes:

key is the `pthread_key_t` global variable that is used to retrieve thread-specific data.

ptr is a void pointer to the thread-specific data to be destroyed.

Once the data is associated with the key, then any function can get access to the data by issuing the `pthread_getspecific()` call. Once the *ptr* is retrieved, the function can use that pointer to both read and write data from/to that location directly. Remember that each function must do this, or it must retrieve the pointer from an argument passed to the function.

Once the thread terminates, the destructor routine is called to destroy both the data, and the pointer in the thread-specific data table. This is typically done by the `free()` call. This destructor routine is optional, but remember that all pointers in the thread-specific data table must be NULL before the key can be deleted.

Once all threads have destroyed their data, then the key can be deleted. This would probably end up in a shutdown routine. Note that this does not delete the global variable, but would destroy the thread-specific data row associated with this key.

Thread-Specific Data: Example (1 of 2)

```

/* tsd.c */
#include <pthread.h>
pthread_key_t count_key;

void * thread_starter(void * TID)
{
    int * ptr;
    int ThreadID;
    ThreadID=(int *)TID;

    ptr=(int *) malloc(sizeof(int));
    pthread_setspecific(count_key, ptr);
    *ptr=0;

    (*ptr)++;
    subfunc();
    (*ptr)++;
    printf("Thread %d's count should be 3 now. The pointer says:%d\n",
        ThreadID, *ptr);
    *(int *)TID = 0;
}
subfunc(void)
{
    int * local_p;
    local_p=pthread_getspecific(count_key);
    (*local_p)++;
}

```

Figure 14-8. Thread-Specific Data: Example (1 of 2)

AU253.0

Notes:

Figure 14-8 shows how to use thread-specific data.

`thread_starter()` is the entry function for the thread that would be created by the main thread of the program.

The `pthread_setspecific()` and `pthread_getspecific()` calls are used to create and use thread-specific data.

Thread-Specific Data: Example (2 of 2)

```
void count_key_destructor(void * Data)
{
    free(Data);
}

main()
{
    pthread_t thd1;
    pthread_t thd2;
    int tid1=1, tid2=2;

    pthread_key_create(&count_key, count_key_destructor);

    pthread_create(&thd1, NULL, thread_starter, (void *)&tid1);
    pthread_create(&thd2, NULL, thread_starter, (void *)&tid2);
    while (tid1 != 0 || tid2 != 0)
        sleep(1);
    while(pthread_key_delete(count_key) != 0)
        sleep(1);

    pthread_exit(NULL);
}
```

Figure 14-9. Thread-Specific Data: Example (2 of 2)

AU253.0

Notes:

It is assumed that the threadstarter() would always initialize the data. If you are not sure who will initialize the data, then the initialization code in threadstarter() should permeate each function.

Data in the destructor routine is the data that was associated with the key on the pthread_setspecific() call. Note that this is assumed and that there is nothing you have to code.

The order of manipulation in each function is to first do the pthread_getspecific() call, and then use the pointer to read and write the data. An alternative could have been to pass the pointer to the subfunc call directly.

You need to use the libpthread.a library while linking this example.

The output from the example is as follows:

Thread 1's count should be 3 now. The pointer says: 3

Thread 2's count should be 3 now. The pointer says: 3

Handling Local Static Data

BEFORE:

```
funca()
{
    static int count = 0;
    count++;
    printf("count is %d\n", count);
}

main()
{
    funca();
    funca();
}
```

AFTER:

```
funca(short funca_count)
{
    printf("count is %d\n", funca_count);
}

main()
{
    short count = 1;
    funca(count);
    count++;
    funca(count);
}
```

Figure 14-10. Handling Local Static Data

AU253.0

Notes:

Local static variables are no longer allowed. Because local static variables are kept in the data area, they are accessible to multiple threads. This causes a problem, because the data needs to be unique to each operating environment, which in this case is a thread. To get beyond this problem and make your function reentrant, it must no longer use local static variables or pass pointers to local static variables as a return value on a call. One must now pass the variable information to the function from a thread-specific location. The most likely way to accomplish this is to pass the data from the local auto variable section of the function calling this function. It now becomes the responsibility of the calling function to maintain the status of the variable rather than the called function. An example of this is shown in the "AFTER" section.

All functions must be rewritten to take them out and pass them as an argument to the function of interest. This will make the function reentrant.

The C library has been redesigned as reentrant. Certain functions that had static variables before or returned a pointer to the static variable have now been redesigned. Their alternative is found in the `libc_r.a` library. A common example is the `strtok` function.

14.2 Controlling Thread Execution

Controlling Execution

- Create
- Exit
- Wait/Interrupt
- Cancel
- Schedule
- Other

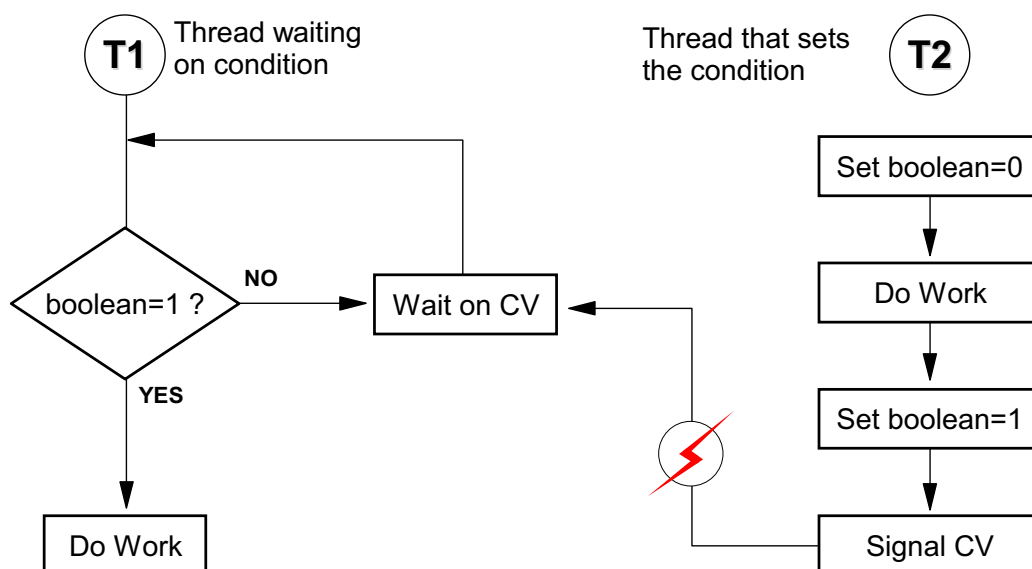
Figure 14-11. Controlling Execution

AU253.0

Notes:

We have already discussed creating and exiting a thread via the `pthread_create()` and `pthread_exit()` calls. In this section, we will deal with waiting (condition variables and signals), cancelling a thread, changing the scheduling of a thread (priority and/or policy), and other miscellaneous items, such as one-time execution and forking considerations.

Condition Variable Logic



Legend
CV: Conditional Variable

Figure 14-12. Condition Variable Logic

AU253.0

Notes:

This logic is NOT complete. It is merely an introduction to what condition variables are about: a mechanism used to wait on a condition.

The logic needs at least two key components: the boolean and the condition variable. (We will see later that a mutex is also needed.) The boolean represents the condition. The condition variable is used to wait on the boolean if it is not at the right value.

In our scenario, T1 represents the waiting thread while T2 represents the thread setting the condition. When T2 is ready, it signals one or more threads waiting on the condition.

A thread must still check the boolean after it has waited, because more than one thread could have been released at one time. Another "waiting" thread may have reset the boolean back before the current thread could reach it.

Condition Variable Calls

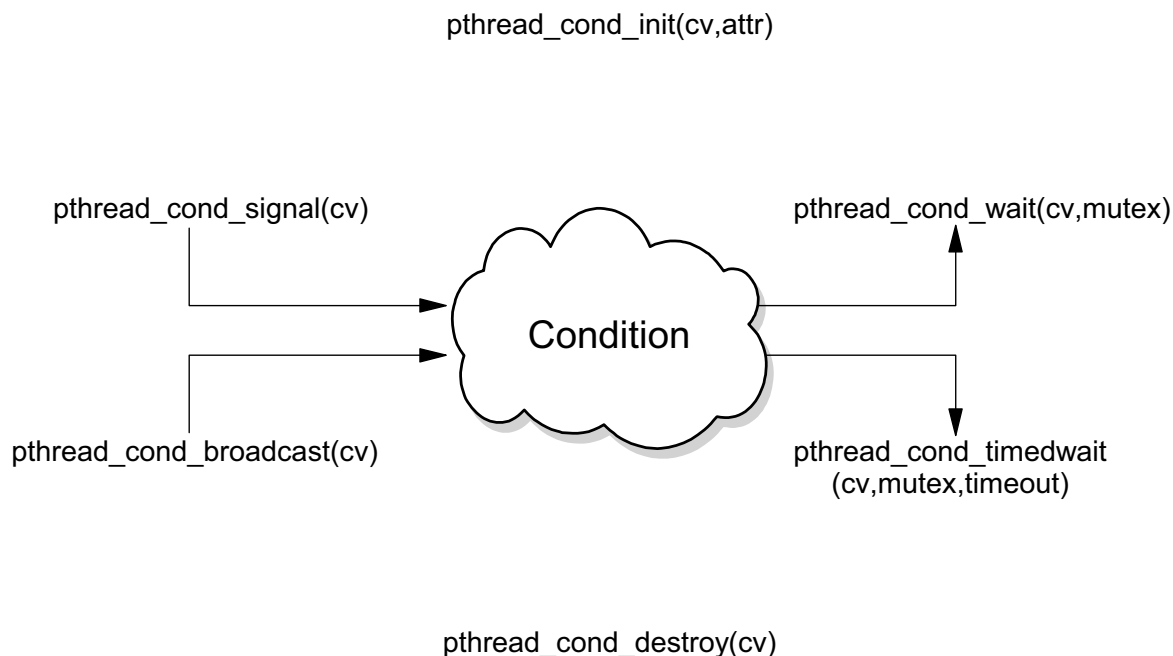


Figure 14-13. Condition Variable Calls

AU253.0

Notes:

cv is a `pthread_cond_t` pointer to the condition variable.

attr is a `pthread_condattr_t` pointer to the attributes of the condition variable.

mutex is a `pthread_mutex_t` pointer to the mutex protecting the condition variable.

timeout is a constant struct `timespec` pointer to the time when the thread no longer chooses to wait.

The initialize and destroy routines are run once, typically from main. Also, the mutex initialize and destroy routines should be done at the same time.

`pthread_cond_signal()` will signal one or more threads, while `pthread_cond_broadcast()` signals all threads.

`pthread_cond_wait()` will wait until it is signaled by the `pthread_cond_signal()` or `pthread_cond_broadcast()` call. The `pthread_cond_timedwait()` will wait until either the time specified or a condition variable signal is received.

Because the `pthread_cond_timedwait()` call is not shown in our examples, we have included the following section of code to show a way to implement the time value on the `pthread_cond_timedwait()` call:

```
pthread_mutex_t m;  
struct timespec time;  
struct timeval tm;  
  
gettimeofday(&tm, 0);  
time.tv_sec = tm.tv_sec +5;  
time.tv_nsec = 0;  
  
pthread_cond_timedwait(&cv, &m, &time);
```

Condition Wait Implementation

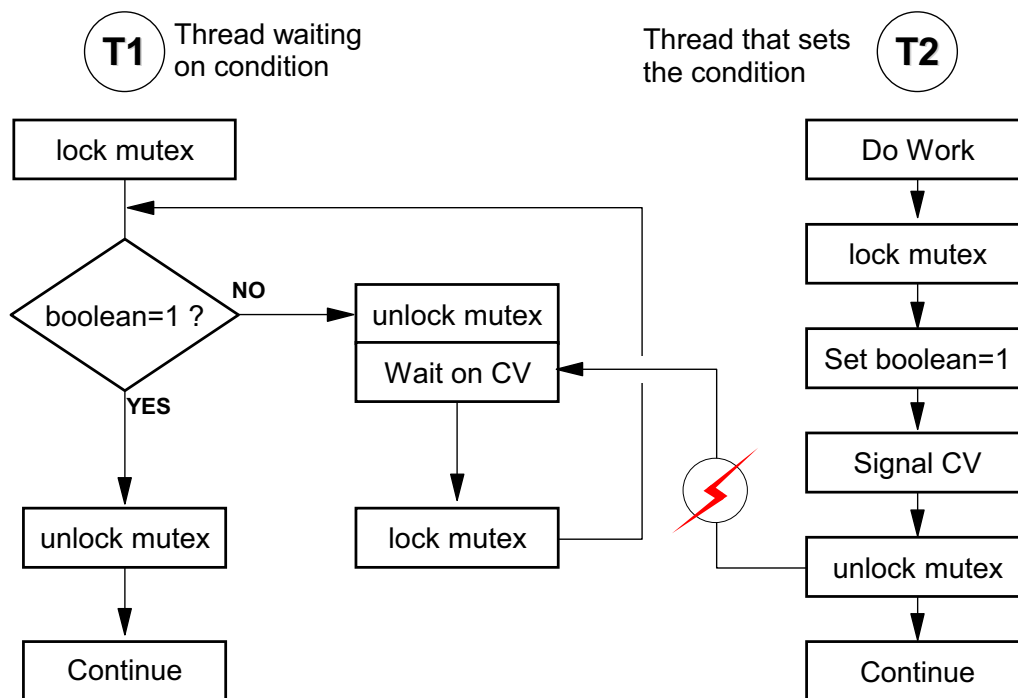


Figure 14-14. Condition Wait Implementation

AU253.0

Notes:

In this scenario, T1 (which may represent many threads) waits on a condition to be set by the T2 thread.

The T1 thread waits on a condition set by the T2 thread. This might be a variable that is updated and ready to go, a file that is finished, or virtually anything you can think of that needs to be complete before others can start working on it.

Condition Wait Example (1 of 2)

```

/* condwait.c */
#include <signal.h>
#include <stdio.h>
#include <pthread.h>

int Ready=0;
pthread_mutex_t m;
pthread_cond_t cv;

void * slave(void * threadID)
{
    int threadNum = *(int *) threadID;
    pthread_mutex_lock(&m);
    while(Ready != 1)
        pthread_cond_wait(&cv, &m);
    pthread_mutex_unlock(&m);
    printf("Thread %d continuing.... \n",threadNum);
    *(int *)threadID = 0;
    pthread_exit(NULL);
}

main()
{
    pthread_t thd1;
    pthread_t thd2;
    pthread_t thd3;

    int j1=1, j2=2, j3=3;

    pthread_mutex_init(&m,NULL);
    pthread_cond_init(&cv,NULL);

    pthread_create(&thd1, NULL, slave, (void *) &j1);
    pthread_create(&thd2, NULL, slave, (void *) &j2);
    pthread_create(&thd3, NULL, slave, (void *) &j3);
}

```

Figure 14-15. Condition Wait Example (1 of 2)

AU253.0

Notes:

This is an implementation of the logic shown in Figure 14-14.

The mutex is used to protect the condition variable. Note that the mutex will automatically unlock when you call `pthread_cond_wait()`. This allows other threads to come into the waiting area. When T2 signals T1, one or more threads will be woken up, but only one can get the mutex and proceed. The others will be in a holding pattern one-by-one until it is their turn to grab the mutex and proceed. In our example here, we probably do not care whether each T1 thread executes the "continue" statements sequentially, and thus the mutex is released immediately.

Condition Wait Example (2 of 2)

```
printf("Main doing work....\n");
sleep(5);          /* Simulate work */
pthread_mutex_lock(&m);
Ready=1;
pthread_cond_broadcast(&cv);
pthread_mutex_unlock(&m);
printf("Main ready for others and continuing on.... \n");
sleep(5);

/* Wait for children to terminate
 * (using pthread_join with undetached
 * children would be better)
 */

while ( j1 != 0 || j2 != 0 || j3 != 0 )
    sleep(1);
pthread_mutex_destroy(&m);
pthread_cond_destroy(&cv);
pthread_exit((void *) 0);
}
```

Figure 14-16. Condition Wait Example (2 of 2)

AU253.0

Notes:

This implementation assumes that everyone can start at once on this "condition" once T2 signals. We use a `pthread_cond_broadcast()` in this case. If you wanted only one thread at a time to work on this condition, then the T1 flow would have to add in a step to decrease the boolean variable. Then, it could perform its work, then go back and play T2's role so others could continue on when it was done.

The output from the example is:

```
Main doing work....
Main ready for others and continuing on....
Thread 3 continuing....
Thread 2 continuing....
Thread 1 continuing....
```

Master/Slave Implementation

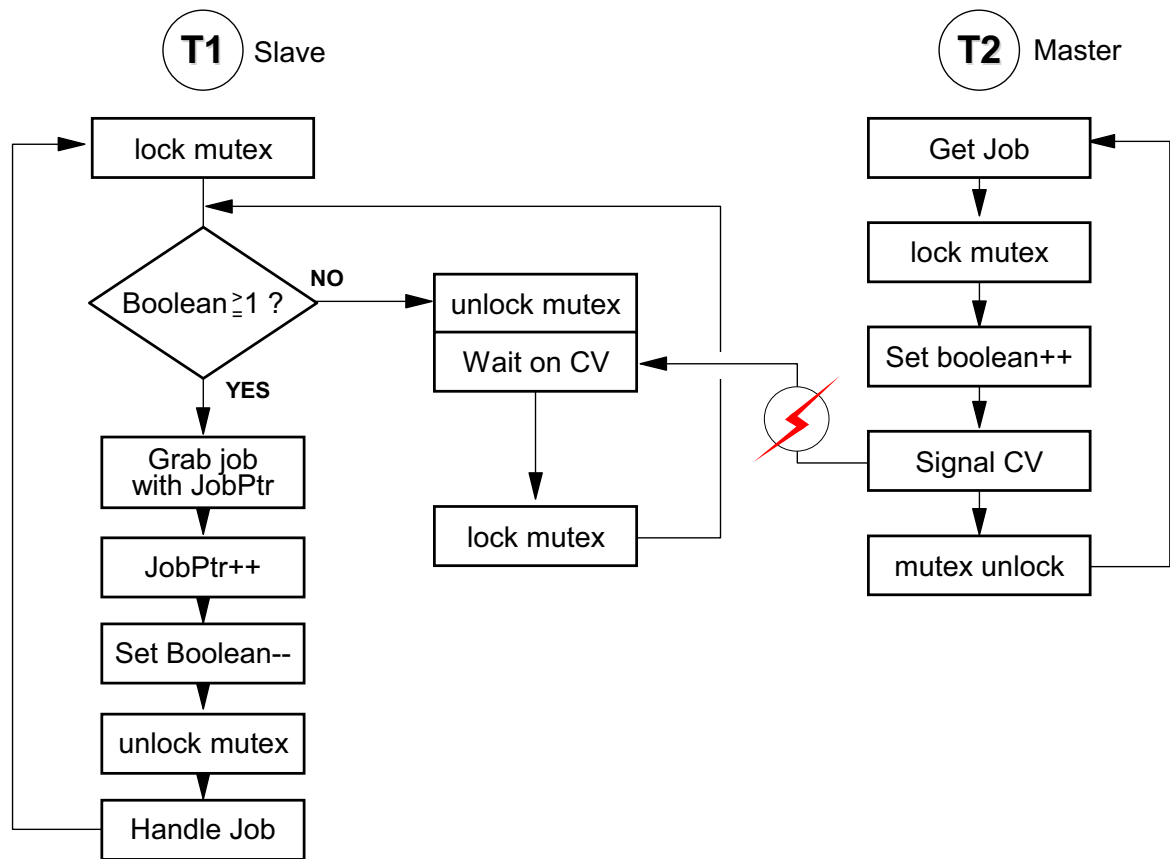


Figure 14-17. Master/Slave Implementation

AU253.0

Notes:

The main difference between this and the last implementation is the looping, and the idea that the boolean goes both up and down as jobs are received and handled. A `JobPtr` is used to pass data about the job to the slave T1 thread.

In this scenario, T1 (which may represent many threads) represents the slaves doing T2's work. As each job comes in to T2, the boolean count (representing more work) is incremented. As each T1 slave accepts a job, it decrements the boolean count.

In addition, we should use the `pthread_cond_signal()` `pthread_cond_wait()` rather than the `pthread_cond_broadcast()` call because we want to wake up one thread.

Note that there is a lot more going on in the T1 flow before releasing the mutex. This is because we are grabbing information that we want exclusive access to before other threads are released.

Master/Slave Example (1 of 2)

```
/* master.c */
#include <signal.h>
#include <stdio.h>
#include <pthread.h>

int jobInfo[25];          /* job data passed to thread */
int jobAvail = 0;         /* Boolean - # jobs available */
int jobPtr=0;             /* job index for slave thread */

pthread_mutex_t m;        /* used to protect cv */
pthread_cond_t cv;        /* causes slave to wait for new job */
pthread_key_t key;        /* provides TSD data for global jobInfo */

void * slave(void * threadID)
{
    int threadNum=(int *) threadID;
    int * local_job;       /* will store jobInfo data */
    for(;;)
    {
        pthread_mutex_lock(&m);
        while(jobAvail < 1 && jobPtr < 25)
            pthread_cond_wait(&cv, &m);

        if (jobPtr == 25) {
            pthread_mutex_unlock(&m); /* We're done */
            break;
        }
        local_job = (int *)malloc(sizeof(int)); /* allocate room for jobInfo */
        pthread_setspecific(key, (void *)local_job);
        *local_job = jobInfo[jobPtr]; /*global to TSD*/
        jobPtr++; /* job handled count goes up */
        jobAvail--; /* one less job left */
        pthread_mutex_unlock(&m); /* Ready for others to continue */

        printf("Thread %d handling job with local_job info of %d\n",
            threadNum, *local_job);
    }
    *(int *)threadID = 0;
    pthread_exit(NULL);
}
```

Figure 14-18. Master/Slave Example (1 of 2)

AU253.0

Notes:

This is an implementation of the logic shown in Figure 14-17.

Master/Slave Example (2 of 2)

```

main()
{
    pthread_t thd1;
    pthread_t thd2;
    pthread_t thd3;
    int j1=1,j2=2,j3=3;          /* thread IDs passed to threads */
    int i;                       /* job index for master thread */

    pthread_mutex_init(&m,NULL);
    pthread_cond_init(&cv,NULL);

    pthread_create(&thd1, NULL, slave, (void * )&j1);
    pthread_create(&thd2, NULL, slave, (void * )&j2);
    pthread_create(&thd3, NULL, slave, (void * )&j3);

    for(i=0;i< 25;i++) /* create 25 jobs */
    {
        printf("Getting a new job\n");          /*simulates a new job created */
        jobInfo[i] = i + 100;                  /* simulates making data */

        pthread_mutex_lock(&m);
        jobAvail++;                            /* new job available */
        pthread_cond_signal(&cv);
        pthread_mutex_unlock(&m);
    }

    /* Help others finish */
    /* (some might be blocked on the pthread_cond_wait call) */

    while ( j1 != 0 || j2 != 0 || j3 != 0 ) {
        pthread_mutex_lock(&m);
        pthread_cond_signal(&cv);
        pthread_mutex_unlock(&m);
        sleep(1);
    }

    pthread_mutex_destroy(&m);
    pthread_cond_destroy(&cv);
    exit(0);
}

```

Figure 14-19. Master/Slave Example (2 of 2)

AU253.0

Notes:

The output from the example is as follows:

```

Getting a new job (repeated 25 times)
Thread 3 handling job with local_job info of 100
Thread 3 handling job with local_job info of 101
Thread 3 handling job with local_job info of 102
Thread 3 handling job with local_job info of 103
Thread 3 handling job with local_job info of 104
Thread 3 handling job with local_job info of 105
Thread 3 handling job with local_job info of 106
Thread 3 handling job with local_job info of 107
Thread 3 handling job with local_job info of 108
Thread 3 handling job with local_job info of 109
Thread 3 handling job with local_job info of 110
Thread 3 handling job with local_job info of 111
Thread 3 handling job with local_job info of 112
Thread 3 handling job with local_job info of 115

```

```
Thread 3 handling job with local_job info of 116
Thread 3 handling job with local_job info of 117
Thread 3 handling job with local_job info of 118
Thread 3 handling job with local_job info of 119
Thread 3 handling job with local_job info of 120
Thread 3 handling job with local_job info of 121
Thread 3 handling job with local_job info of 122
Thread 3 handling job with local_job info of 123
Thread 3 handling job with local_job info of 124
Thread 2 handling job with local_job info of 113
Thread 1 handling job with local_job info of 114
```

Note that in the example run, thread 3 does practically all of the work. This is because the actual amount of work is so small that thread 3 did almost all of it before it was forced to give up the CPU by the kernel. If the work per job was larger, then the sharing would probably be much more equitable. If you run this program ten times, the sharing of the work will probably be different each time.

Signal Management

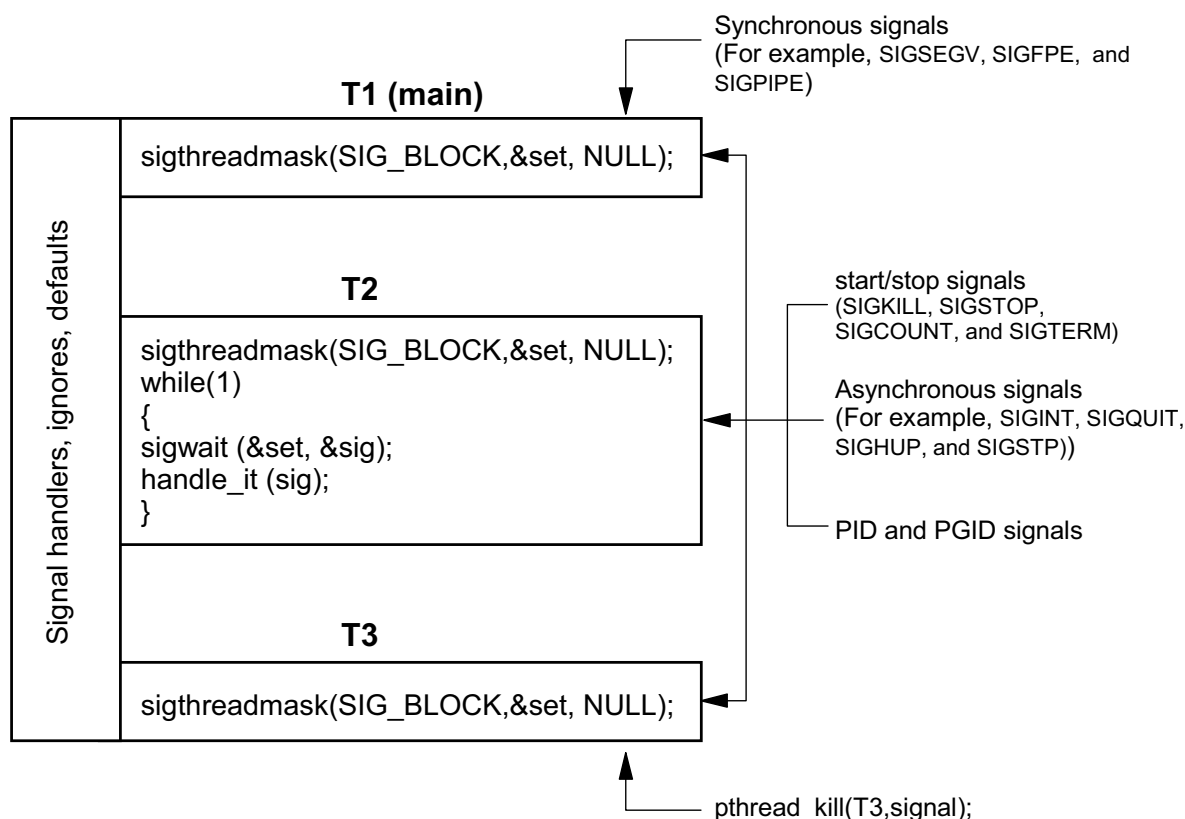


Figure 14-20. Signal Management

AU253.0

Notes:

`set` is a `sigset_t` type of data used to represent the signals to either block or wait (depending upon the call).

`sig` is an `int` pointer to where the signal received is to be stored.

Synchronous signals (that is, hardware signals) and `pthread_kill` signals are delivered to the specifically affected thread. Start/Stop type of signals and asynchronous signals (that is, terminal sent signals) are sent to the entire process.

Signal handlers are common to all threads, as are `SIG_IGNs` and `SIG_DFLs`. The only thing a thread can do as a unique setting is to block incoming signals by using the `sigthreadmask()` call.

It is common to have one thread set up to handle incoming async signals via the `sigwait()` call. Once received, it can react to it and/or set up appropriate flags for other threads to see, or it can choose to use condition variables to broadcast the signal to multiple other threads.

Signal Management Example (1 of 2)

```
/* sigmgmt.c */
#include <pthread.h>
#include <signal.h>

void *threadfunc(void * notused)
{
    struct sigset_t set, oset;

    sigemptyset(&set);
    sigaddset(&set, SIGQUIT);
    sigthreadmask(SIG_BLOCK, &set, NULL);

    for(;;); /* simulate action */
}

void * sigwaiter_thread(void *notused)
{
    int sig;
    int ret;
    struct sigset_t set, oset;
    sigemptyset(&set);
    sigaddset(&set, SIGQUIT);
    sigthreadmask(SIG_BLOCK, &set, NULL);

    for(;;)
    {
        sigwait(&set,&sig);
        printf("sigwait() returned signal = %d\n", sig);
    }
}
```

Figure 14-21. Signal Management Example (1 of 2)

AU253.0

Notes:

threadfunc() is the entry point of the thread that does not handle the SIGQUIT signal.

sigwaiter_thread() is the entry point of the thread that handles the SIGQUIT signal.

The sigemptyset and sigaddset subroutines manipulate sets of signals.

The sigemptyset() subroutine initializes the signal set such that all signals are excluded:

```
int sigemptyset (sigset_t *Set)
```

The sigaddset() subroutine adds the specified signal to the signal set:

```
int sigaddset (sigset_t *Set, int SignalNumber)
```

Signal Management Example (2 of 2)

```
main()
{
    pthread_t waiterthd, thd;
    int status;
    struct sigset_t set, oset;
    struct sigaction mysig;
    pthread_attr_t attr;

    /* Block the signals. */
    sigemptyset(&set);
    sigaddset(&set, SIGQUIT);
    sigthreadmask(SIG_BLOCK, &set, NULL);

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    pthread_create(&waiterthd, &attr, sigwaiter_thread, NULL);
    pthread_create(&thd, &attr, threadfunc, NULL);

    sleep(2);

    pthread_join(thd, (void **)&status);
    printf("Main- check-\n");
}
```

Figure 14-22. Signal Management Example (2 of 2)

AU253.0

Notes:

Each thread must block its own signals. Even the `sigwait()` thread must block incoming signals to make the call function correctly.

The output from this example is:

```
(when ctrl-\ hit at keyboard)
sigwait() returned signal = 3
```

Cancellation Choices

Call	Thread Only	Thread Cleanup	Thread Deferred Cancellation
exit	N	N	N
kill	N	N	N
pthread_kill	Y	N	N
pthread_exit	Y	Y	N
pthread_cancel	Y	Y	Y

Figure 14-23. Cancellation Choices

AU253.0

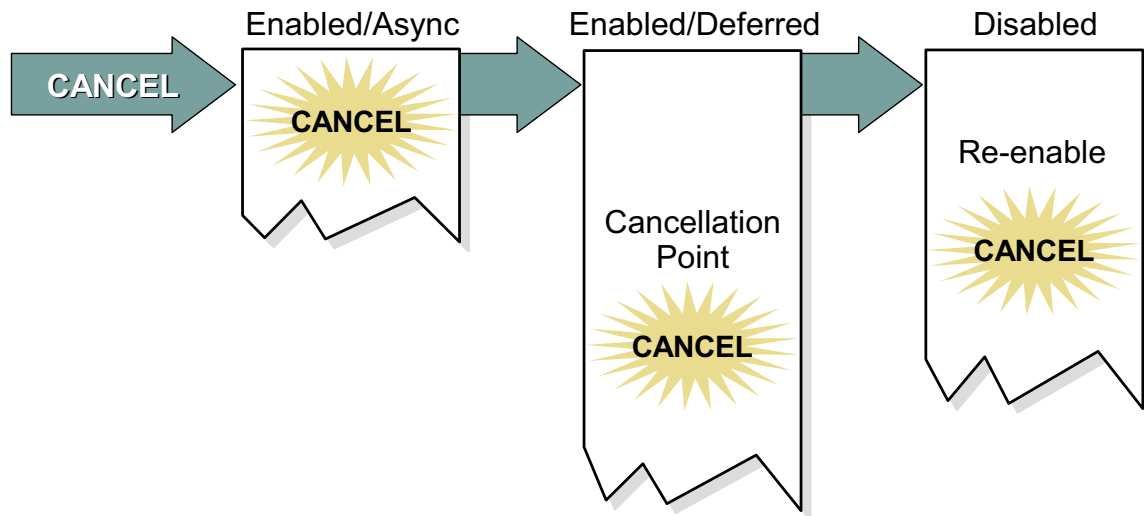
Notes:

The `pthread_cancel()` series of calls have the most complex series of options for the safest termination of the thread.

The `kill` call could be thread-specific with the right setup. All threads but one would have to block the incoming signal.

The `exit` and `kill` calls can also provide some type of cleanup, but it is not thread-specific, and would not clean up mutexes of interest. (Plus the `exit` call would exit the entire process.)

Cancellation Points



```
int pthread_cancel(pthread_t thread)
int pthread_testcancel(void)
int pthread_setcancelstate(int state, int * oldstate)
int pthread_setcanceltype(int type, int * oldtype)
```

Figure 14-24. Cancellation Points

AU253.0

Notes:

The system calls are as follows:

- `pthread_cancel()` cancels the thread shown.
- `pthread_testcancel` establishes a cancellation point.
- `pthread_setcancelstate()` changes the state to either `PTHREAD_CANCEL_ENABLED` or `PTHREAD_CANCEL_DISABLED`, and optionally records the prior setting in `oldstate`.
- `pthread_setcanceltype()` `pthread_cleanup_push()` changes the type to either `PTHREAD_CANCEL_ASYNCHRONOUS` or `PTHREAD_CANCEL_DEFERRED`, and optionally records the prior setting in `oldtype`.

The parameters used in the above system calls are described below:

- *thread* is a `pthread_t` thread ID.
- *state* specifies whether cancellation is allowed during this timeframe or not (enabled versus disabled)
- *oldstate* holds the previous cancellation state.

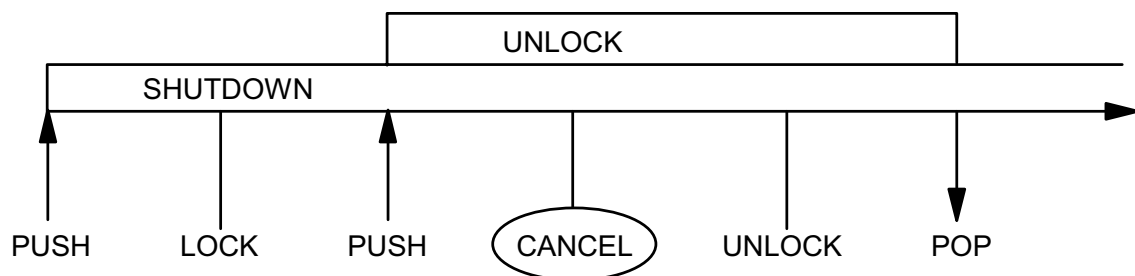
- *type* specifies how cancellation will be handled if it is enabled (async or deferred).
- *oldtype* holds the previous cancellation type.

Cancellation of the thread is merely requested when the `pthread_cancel()` call is issued. There are three basic ways to handle a cancellation request: enabled/async, enabled/deferred, or disabled. The middle method is the default. With this method, the thread can be cancelled by the `pthread_cancel()` call as soon as a cancellation point is reached. A cancellation point is reached when either the `pthread_testcancel` call is issued, or a cancellation point call is issued (`pthread_join`, `pthread_cond_wait`, or `pthread_cond_timedwait`).

An enabled/async cancellation setting would cause the thread to terminate immediately. This is typically not recommended because the program is left in an unknown state.

A disabled cancellation setting might be useful if the thread is doing complex work, such as I/O transaction calls, and does not want to accidentally run across a cancellation point too early.

Cancellation Cleanup



```
pthread_cleanup_push(routine, arg)
```

```
pthread_cleanup_pop(execute)
```

Figure 14-25. Cancellation Cleanup

AU253.0

Notes:

The system call `pthread_cleanup_push()` establishes cleanup operations to be performed upon cancellation of the thread. The parameters used in this system call are described below:

- *routine* is the function that is called that has instructions to place on the cleanup stack.
- *arg* is a void pointer to an argument that is passed to the cleanup routine.

The system call `pthread_cleanup_pop()` removes the instructions from the cleanup stack and optionally executes them. The `execute` argument is an `int` that indicates whether execution of the stack should take place.

When cancellation occurs, the cleanup routines are issued, followed by cleanup of thread specific data.

The cleanup routines are pushed onto a stack with the `pthread_cleanup_push()` routine and are executed when the thread is cancelled or when the `pthread_cleanup_pop()` routine is issued with the `execute` argument turned on. Typically, the `cleanup_push` routine is used when thread specific resources (such as mutexes) need to be released if the thread is

cancelled. The `cleanup_pop` routine is used to "undo" the push at the end of the code if the thread was not cancelled.

The horizontal arrow represents a timeline. If you lock a resource, then you need to be prepared for a cancellation. Hence, push the routine name that has the unlock call in it onto the stack. If the thread is cancelled, the unlock will occur automatically, because the cancellation handler was on the stack. If the thread is not cancelled, you can unlock it yourself, and get it off of the stack by using the `pop` call.

When the program begins, it is probably a good idea to put your shutdown routine onto the stack. Then, regardless of when you get cancelled, your shutdown routine should run.

Debuggers can get information about all active cancellation cleanup handlers in a process. This feature has been introduced in AIX 5L.

Cancellation Example (1 of 2)

```
/* pushpop.c */
#include <pthread.h>
pthread_mutex_t m;

void cleanup_m(void * m)
{
    printf("In cleanup_m handler.\n");
    pthread_mutex_unlock((pthread_mutex_t *)m);
}

void * thread_starter(void * notused)
{
    printf("In thd1.\n");
    pthread_mutex_lock(&m);
    pthread_cleanup_push(cleanup_m, (void *)&m);
    sleep(1);
    pthread_testcancel(); /* Cancellation point */
    printf("If I reached here, I didn't get canceled.\n");
    pthread_mutex_unlock(&m);
    pthread_cleanup_pop(0);
    pthread_exit(NULL);
}
```

Figure 14-26. Cancellation Example (1 of 2)

AU253.0

Notes:

This part of the example contains the thread entry point `thread_starter()` and a cancellation handler `cleanup_m()`.

Cancellation Example (2 of 2)

```
main()
{
    pthread_t thd1;

    pthread_mutex_init(&m, NULL);
    pthread_create(&thd1, NULL, thread_starter, NULL);

    pthread_cancel(thd1);

    pthread_mutex_destroy(&m);
    pthread_exit(NULL);
}
```

Figure 14-27. Cancellation Example (2 of 2)

AU253.0

Notes:

The default of the "deferred/enabled" cancellation is used. Because of this situation, we must enable the thread to be cancelled at certain intervals by issuing the `pthread_testcancel` call (or wait for the rare other calls that also set cancellation points).

Once cancellation occurs, the `cleanup_m` routine is called. This allows us to release the "m" mutex that we were holding. If the mutex had not been cancelled by the main thread, then the `mutex_unlock` call would have released the mutex and the `pthread_cleanup_pop()` routine would have taken the `cleanup_m` routine off of the stack.

The output from this example is:

In `thd1`.

In `cleanup_m` handler.

Scheduling

Default: Inherit from parent thread

How to Set: Set attr on pthread_create call
or
Use pthread_setschedparam

What to Set:

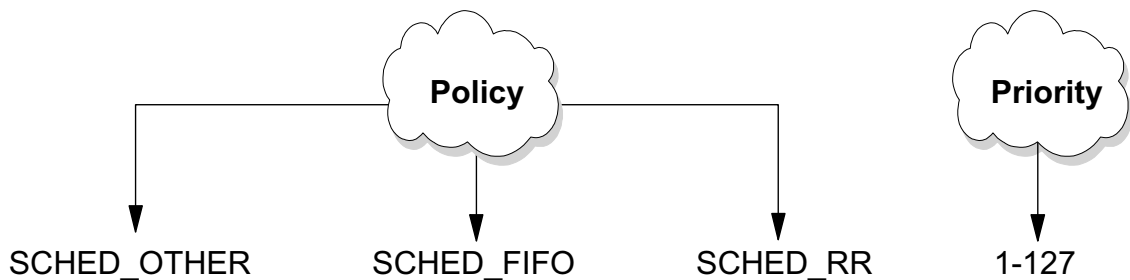


Figure 14-28. Scheduling

AU253.0

Notes:

Priorities are the opposite of native AIX priorities. Here, 1 is the least favored priority, and 127 the most favored.

SCHED_OTHER is the default scheduling policy. This uses the standard AIX scheduling technique: a floating CPU algorithm that penalizes CPU intensive applications. Only this time, it is on a thread basis versus a process basis.

SCHED_FIFO is first-in-first-out. This scheduling technique should be highly discouraged. If multiple threads are at the same priority, the first thread runs to completion before another begins. This almost defeats the purpose of threads.

SCHED_RR is a round-robin technique. It ensures that all threads at a given priority get an equal time slice. It is used to handle fixed priority threads.

Calls associated with changing or looking at scheduling and priorities are: pthread_setschedparam, pthread_getschedparam, pthread_attr_getschedpolicy, pthread_attr_setschedpolicy, pthread_attr_setinheritsched, and pthread_attr_getinheritsched.

Scheduling Example (1 of 2)

```
/* thrdsched.c */
#include <pthread.h>

void *func1( void *notused)
{
    int priority;
    struct sched_param sched;

    pthread_getschedparam(pthread_self(), &priority, &sched);
    printf("Child thread policy is %d and priority is %d\n",
           sched.sched_policy, sched.sched_priority);
}
```

Figure 14-29. Scheduling Example (1 of 2)

AU253.0

Notes:

func1() is the entry point of the child thread. This thread prints its current priority and current policy.

Scheduling Example (2 of 2)

```
main()
{
    struct sched_param sched;

    pthread_attr_t attr;
    pthread_t thd;

    printf("Legend:\nPolicy 0 is SCHED_OTHER.\n");
    printf("Policy 1 is SCHED_FIFO.\n");
    printf("Policy 2 is SCHED_RR.\n\n");

    pthread_create( &thd, NULL, func1, NULL);
    sleep(1);

    printf("Changing policy of child thread to RR\n");
    printf("Changing priority of child thread to 80\n");

    pthread_attr_init(&attr);
    sched.sched_policy = SCHED_RR;
    sched.sched_priority= 80;
    pthread_attr_setschedparam(&attr, &sched);
    pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

    pthread_create( &thd, &attr, func1, NULL);

    pthread_exit(NULL);
}
```

Figure 14-30. Scheduling Example (2 of 2)

AU253.0

Notes:

This example creates two threads. It creates the first thread with the default priority and policy. The second thread is created at a priority of 80 using the SCHED_RR scheduling technique.

The output from this example is:

Legend:

Policy 0 is SCHED_OTHER.

Policy 1 is SCHED_FIFO.

Policy 2 is SCHED_RR.

Child thread policy is 0 and priority is 1

Changing policy of child thread to RR

Changing priority of child thread to 80

Child thread policy is 2 and priority is 80

Once Only Initialization

Initialize **variable=PTHREAD_ONCE_INIT**

Even if many threads call `pthread_once(&variable, routine)`, `routine()` is called only once.

```
pthread_once(&variable, routine)
is equivalent to :
if(variable equals PTHREAD_ONCE_INIT)
{
    routine()
    change variable
}
```

Figure 14-31. Once Only Initialization

AU253.0

Notes:

variable is a `pthread_once_t` variable that is initially set to the condition `PTHREAD_ONCE_INIT`.

routine is the routine to be called when `pthread_once()` is issued if the `PTHREAD_ONCE_INIT` condition still exists.

This procedure is used when a series of instructions should only happen once. An easy way to do this is to have only main run the series of instructions. In reality, this may not always be the case. If not, we can rely on the `pthread_once()` call. It checks the variable indicated. If the variable is still set at the `PTHREAD_ONCE_INIT` setting, it will proceed with the routine mentioned. The routine should include the "once only" instructions.

Once Only Example (1 of 2)

```
/* once.c */
#include <pthread.h>
#include <unistd.h>

pthread_t thd[2];
static pthread_once_t once = PTHREAD_ONCE_INIT;

void init()
{
    pthread_t self;
    /* Do your initializations here */

    /* Check which thread is here */
    self = pthread_self();
    if(pthread_equal(self, thd[0]))
        printf("init() - invoked by Thread 1\n");
    if(pthread_equal(self, thd[1]))
        printf("init() - invoked by Thread 2\n");
}
```

Figure 14-32. Once Only Example (1 of 2)

AU253.0

Notes:

In this example, `init()` is the once only initialization routine.

The `pthread_once()` call is used to set `init()` as the once only initialization routine. The variable `once` is used as the flag. It is initialized to `PTHREAD_ONCE_INIT`.

Once Only Example (2 of 2)

```
void * func(void *arg)
{
    pthread_t self;

    /* init() should be called only once */
    (void)pthread_once(&once, init);

    self = pthread_self();
    if(pthread_equal(self, thd[0]))
        printf("In Thread 1\n");
    if(pthread_equal(self, thd[1]))
        printf("In Thread 2\n");
    pthread_exit((void *)1);
}

main()
{
    pthread_create(&thd[0], NULL, func, 0);
    pthread_create(&thd[1], NULL, func, 0);
    pthread_exit((void **)1);
}
```

Figure 14-33. Once Only Example (2 of 2)

AU253.0

Notes:

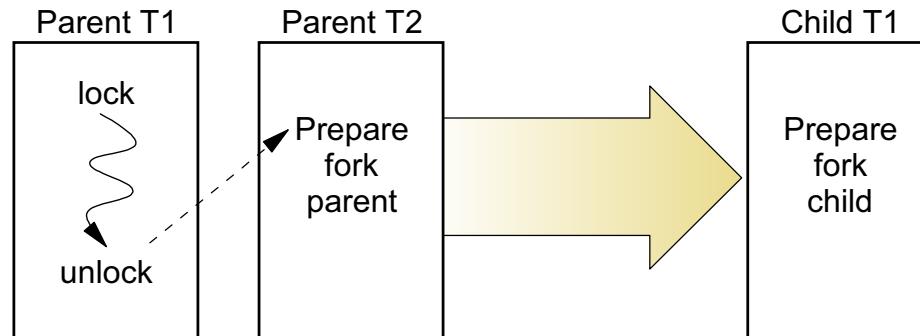
Figure 14-33 shows the use of `pthread_once()` call to invoke a initialization routine only once. In this example, two threads are created. Neither thread knows which thread actually calls the `init()` function. So the `pthread_once()` call is used to let just the first thread call `init()`. The `pthread_once()` call only runs the `init` routine when the `once` variable is at the `PTHREAD_ONCE_INIT` value. When the `init` function is run once, this "once" variable changes value.

Note also the `pthread_equal()` call used to identify who has what role.

The output from this example is:

```
init() - invoked by Thread 1
In Thread 1
In Thread 2
```


Forking Considerations



pthread_atfork(prepare, parent, child)

Figure 14-34. Forking Considerations

AU253.0

Notes:

`pthread_atfork()` determines which routines will run before and after a fork occurs.

- *prepare* is the routine that runs before the fork occurs.
- *parent* is the routine that runs in the parent after the fork occurs.
- *child* is the routine that runs in the child after the fork occurs.

A fork will copy the current thread only to the child. Because of this, deadlock conditions could occur. For example, the lock in the parent's T1 thread in Figure 14-34 would leave the child in a hung condition if it attempts to lock that same resource, because it does not have a copy of the T1 thread to eventually unlock that resource. This is why the `pthread_atfork()` call should always be run before a fork.

Debuggers can get information about all active `pthread_atfork()` handlers in a process. This feature has been introduced in AIX 5L.

It is possible to unregister the `pthread_atfork()` handlers. This can be accomplished using the `pthread_atfork_unregister_np()` call.

Forking Example (1 of 2)

```
/* atfork.c */
#include <pthread.h>
#include <sys/types.h>
pthread_mutex_t m;

void prefork_prepare(void)
{
    printf("prepare\n");
    pthread_mutex_lock(&m);
}
void postfork_parent(void)
{
    printf("parent\n");
    pthread_mutex_unlock(&m);
}
void postfork_child(void)
{
    printf("Child\n");
    pthread_mutex_unlock(&m);
}
void * func(void * arg)
{
    pthread_mutex_lock(&m);
    sleep(1);
    printf("In func before unlock\n");
    pthread_mutex_unlock(&m);
    pthread_exit(NULL);
}
```

Figure 14-35. Forking Example (1 of 2)

AU253.0

Notes:

This part of the example contains the necessary `pthread_atfork()` handlers.

Forking Example (2 of 2)

```
main()
{
    pthread_t thd;
    int status;

    pthread_mutex_init(&m, NULL);

    pthread_create(&thd, NULL, func, (void *)func);

    pthread_atfork(prefork_prepare, postfork_parent, postfork_child);

    if(fork() == 0) {
        printf("Child Process - Do lock\n");
        pthread_mutex_lock(&m);
        sleep(2);
        printf("Child is exiting\n");
        exit(0);
    }
    wait(&status);
    pthread_exit(NULL);
}
```

Figure 14-36. Forking Example (2 of 2)

AU253.0

Notes:

The atfork handlers are setup by the main() routine before a child process is created using the fork() system call.

By executing this example, you can look at the order in which these handlers are invoked.

The output from this example is:

```
prepare
In func before unlock
Child
Child Process- Do lock
parent
Child is exiting
```

Unit Summary

- Protect data resources
- Control thread execution

Figure 14-37. Unit Summary

AU253.0

Notes:

Appendix A. Handling Core Dumps

There are two main issues in handling core dumps we have not discussed yet. One is how to produce a full core dump with data in it (that is, variable contents to view). The other is how to retrieve information from a core dump stripped from all debugging symbols.

Producing a Data Full Core Dump

To get a full core dump with data, you must request it via the SA_FULLDUMP flag on the sigaction call of the signal that is received. If you do this literally, you can receive a good dump, but you will have a poorly controlled shutdown. An alternative to this is to capture the signal coming in and have that signal cleanup and eventually call another signal that will produce the full core dump.

In the example below, we assumed that SIGSEGV was the signal coming in (via a poorly managed pointer) that would typically have given us the core dump. We had that signal instead invoke a handler routine that did the cleanup, and then called SIGABRT to produce the full core dump. Note that SIGABRT was the signal set up with the SA_FULLDUMP flag.

```
/* fullcore.c */
#include <sys/signal.h>
#include <fcntl.h>
#include <stdio.h>

#define RECSIZE 80

void shutdown(int signo);
int g_fd;

main()
{
    char * ptr;

    struct sigaction sigstd;    /* set up shutdown routine */
    sigstd.sa_handler=shutdown;
    sigfillset(&sigstd.sa_mask );
    sigaction(SIGSEGV, &sigstd,NULL);

    sigstd.sa_handler=SIG_DFL; /* set up SIGABRT to produce full core dump */
    sigstd.sa_flags = SA_FULLDUMP;
    sigaction(SIGABRT, &sigstd,NULL);

    g_fd=open("tmpfile", 0_WRONLY- O-CREAT,O600);
```

```
for(;;)

    printf("Trying to use   pointer incorrectly....\n");
    *ptr='a';
    write(g_fd, ptr, 1);
    }
}

void shutdown(int signo)
{
    fprintf(stderr,"Signal %d received. Shutting down the server.-n",signo);
    fprintf(stderr, "Core dump will be produced in current directory.\n");
    close(g_fd);
    kill(0, SIGABRT);    /* cause full core dump to be produced */
}
```

Debugging Core Dumps from a Stripped/Optimized Module

Another problem people have is trying to debug a core dump that does not have the nice debugging symbols in it. Typically, we could just feed the core dump into the dbx debugger and find out all kinds of information. Now we will have to go the extra mile. Here are some suggested steps to get you around this difficult problem.

1. Identify the executable that caused the core dump
 - If not known already, try this: "dbx core". Then look for the name in parentheses. That is the executable that caused the core dump. In our case, it was called "fullcore".
2. Retrieve the source code modules that produced that executable
 - Use **what** "executable name"? In our case, it was "what fullcore".
 - This should tell you what source code modules and release levels produced this executable. See the **what** command for more information.
3. Recompile the source code using the following technique:
 - **cc -O -s -qlist -bR:map fullcore.c**
4. Run dbx against it and retrieve the hexadecimal address that caused the error, and the function that caused it.
 - **dbx fullcore**
 - **(dbx) where**
 - Look after the abort line. It will tell you at what hexadecimal address the error occurred. (that is, main() at 0x10000308). It also tells you which function it occurred in.
 - Optionally, while in dbx, you can type in "map" to find out more information.
5. Retrieve the starting location of your function to figure out the offset of the instruction that caused the problem.
 - **pg map** (remember your -bR:map option on your compile?)
 - Look at the line where your function is listed; in our case, it is ".main". Notice the address at the far left; in our case, it says 00000278. This is the starting location of main.
6. Subtract the starting location of your function from the error address.
 - In our case, it would be $0x308 - 0x278 = 0x90$.
7. Now retrieve the source code line that produced the error
 - **pg fullcore.lst**
 - Look for the OBJECT SECTION. This is a mapping of the optimized machine code to the assembler language call and source code line.
 - Look for your function section. Ours is labeled main.

- Locate the second column (the six digit number following the vertical line). This is the offset number from the beginning of main. In our case, we need to find the number 000090.
- Now retrieve the number to the left of this number. This is your source code line that caused the error. In our case, it was line 28.

Glossary

access mode A matrix of protection information stored with each file specifying who may do what to a file. Three classes of users (owner, group, all others) are allowed or denied three levels of access (read, write, execute).

access permission See **access mode**.

access privilege See **access mode**.

address space The address space of a process is the range of addresses available to it for code and data. The relationship between real and perceived space depends on the system and support hardware.

AIX Advanced Interactive Executive. IBM's implementation of the UNIX Operating System.

AIX Family Definition IBM's definition for the common operating system environment for all members of the AIX family. The AIX Family Definition includes specifications for the AIX Base System, User Interface, Programming Interface, Communications Support, Distributed Processing, and Applications.

alias The command and process of assigning a new name to a command.

ANSI American National Standards Institute. A standards organization. The United States liaison to the International Standards Organization (ISO).

application program A program used to perform an application or part of an application.

argument An item of information following a command. It may, for example, modify the command or identify a file to be affected.

ASCII American Standard Code for Information Interchange. A collection of public domain character sets considered standard throughout the computer industry.

attribute A set of characteristics about an object (how it is to function etc.). In layman's terms, it might be thought of as a structure of information hidden from our view that the object refers to when processing a request. In AIX, the attributes themselves are treated as an object. See **object**.

awk An interpreter, included in most UNIX operating systems, that performs sophisticated text pattern matching. In combination with shell scripts, awk can be used to prototype or implement applications far more quickly than traditional programming methods.

background (process) A process is "in the background" when it is running independently of the initiating terminal. It is specified by ending the ordinary command with an ampersand (&). The parent of the background process does not wait for its "death".

backup diskette A diskette containing information copied from another diskette. It is used in case the original information is unintentionally destroyed.

Berkeley Software Distribution Disseminating arm of the UNIX operating system community at the University of California at Berkeley; commonly abbreviated "BSD". Complete versions of the UNIX operating system have been released by BSD for a number of years; the latest is numbered 4.3. The phrase "Berkeley extensions" refers to features and functions, such as the C shell, that originated or were refined at UC Berkeley and that are now considered a necessary part of any fully configured version of the UNIX operating system.

bit bucket The AIX file "/dev/null" is a special file which will absorb all input written to it and return no data (null or end of file) when read.

block A group of records that is recorded or processed as a unit.

block device A device that transfers data in fixed size blocks. In AIX, normally 512 or 1024 bytes.

block special file An interface to a device capable of supporting a file system.

booting Starting the computer from scratch (power off or system reset).

break key The terminal key used to unequivocally interrupt the foreground process.

BSD Berkeley Software Distribution.

- BSD 2.x - PDP-11 Research
- BSD 4.x - VAX Research
- BSD 4.3 - Current popular VAX version of UNIX.

button

1. A word, number, symbol, or picture on the screen that can be selected. A button may represent a command, file, window, or value, for example.
2. A key on a mouse that is used to select buttons on the display screen or to scroll the display image.

byte The amount of storage required to represent one character; a byte is 8 bits.

C The programming language in which the UNIX operating system and most UNIX application programs are written. The portability attributed to UNIX operating systems is largely due to the fact that C, unlike other higher level languages, permits programmers to write systems-level code that will work on any computer with a standard C compiler.

change mode The **chmod** command will change the access rights to your own files only, for yourself, your group or all others.

character I/O The transfer of data byte by byte; normally used with slower, low volume devices such as terminals or printers.

character special file An interface to devices not capable of supporting a file system; a byte oriented device.

child The process emerging from a **fork** command with a zero return code, as distinguished from the parent which gets the process id of the child.

client User of a network service. In the client/server model, network elements are defined as either using (client) or providing (server) network resources.

command A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command file A data file containing shell commands. See **shell file**, or **shell script**.

command interpreter The part of the operating system that translates your commands into instructions that the operating system understands.

command or **previous command** key.

concatenate The process of forming one character string or file from several. The degenerate case is one file from one file just to display the result using the **cat** command.

condition variable A object used to allow threads to wait until some event or condition has occurred. It is similar to event semaphores in os/2, and using the GETVAL command on the semctl call in AIX.

console The only terminal known explicitly to the Kernel. It is used during booting and it is the destination of serious system messages.

context The hardware environment of a process, including:

- CPU registers
- Program address
- Stack
- I/O status.

The entire context must be saved during a process swap.

control character Codes formed by pressing and holding the **control** key and then some other key; used to form special functions like **End Of File**.

control-d See **eof** character.

cooked input Data from a character device from which backspace, line kill, and interrupt characters have been removed (processed). See **raw input**.

current directory The currently active directory. When you specify a file name without specifying a directory, the system assumes that the file is in your current directory.

current subtree Files or directories attached to the current directory.

curses A C subroutine library providing flexible screen handling. See **Termlib** and **Termcap**.

cursor A movable symbol (such as an underline) on a display, usually used to indicate to the operator where to type the next character.

customize To describe (to the system) the devices, programs, users, and user defaults for a particular data processing system.

DASD Direct Access Storage Device. IBM's term for a hard disk.

device driver A program that operates a specific device, such as a printer, disk drive, or display.

device special file A file which passes data directly to/from the device.

directory A type of file containing the names and controlling information for other files or other directories.

directory pathname The complete and unique external description of a file giving the sequence of connection from the root directory to the specified directory or file.

diskette A thin, flexible magnetic plate that is permanently sealed in a protective cover. It can be used to store information copied from the disk.

diskette drive The mechanism used to read and write information on diskettes.

display device An output unit that gives a visual representation of data.

display screen The part of the display device that displays information visually.

echo To simply report a stream of characters, either as a message to the operator or a debugging tool to see what the file name generation process is doing.

editor A program used to enter and modify programs, text, and other types of documents.

environment A collection of values passed either to a C program or a shell script file inherited from the invoking process.

escape The backslash "\" character specifies that the single next character in a command is ordinary text without special meaning.

Ethernet A baseband protocol, invented by the XEROX Corporation, in common use as the local area network for UNIX operating systems interconnected via TCP/IP.

event One of the previous lines of input from the terminal. Events are stored in the (Berkeley) History file.

event identifier A code used to identify a specific event.

execution permission For a file, the permission to execute (run) code in the file. A text file must have execute permission to be a shell script. For a directory, the permission to search the directory.

field A contiguous group of characters delimited by blanks. A field is the normal unit of text processed by text processes like **sort**.

field separator The character used to separate one field from the next; normally a blank or tab.

FIFO "First In, First Out". In AIX, a FIFO is a permanent, named pipe which allows two unrelated processes to communicate. Only related processes can use normal pipes.

file A collection of related data that is stored and retrieved by an assigned name. In AIX, files are grouped by directories.

file index Sixty-four bytes of information describing a file. Information such as the type and size of the file and the location on the physical device on which the data in the file is stored is kept in the file index. This index is the same as the AIX Operating System i-node.

filename expansion or generation A procedure used by the shell to generate a set of filenames based on a specification using metacharacters, which define a set of textual substitutions.

file system The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

filter Data-manipulation commands (which, in UNIX operating systems, amount to the to small programs) that take input from one process and perform an operation yielding new output. Filters include editors, pattern-searchers, and commands that sort or differentiate files, among others.

fixed disk A storage device made of one or more flat, circular plates with magnetic surfaces on which information can be stored.

fixed disk drive The mechanism used to read and write information on a fixed disk.

flag See **Options**.

foreground (process) An AIX process which interacts with the terminal. Its invocation is not followed by an ampersand.

formatting The act of arranging text in a form suitable for reading. The publishing equivalent to compiling a program.

fsck A utility to check and repair a damaged file structure. This normally results from a power failure or hardware malfunction. It looks for blocks not assigned to a file or the free list and puts them in the free list. (The use of blocks not pointed at cannot be identified.)

free list The set of all blocks not assigned to a file.

full path name The name of any directory or file expressed as a string of directories and files beginning with the root directory.

gateway A device that acts as a connector between two physically separate networks. It has interfaces to more than one network and can translate the packets of one network to another, possibly dissimilar network.

global Applying to all entities of a set. For example:

- A global search - look everywhere
- A global replace - replace all occurrences
- A global symbol - defined everywhere.

grep An AIX command which searches for strings specified by a regular expression. (Global Regular Expression and Print.)

group A collection of AIX users who share a set of files. Members of the group have access privileges exceeding those of other users.

hardware The equipment, as opposed to the programming, of a system.

header A record at the beginning of the file specifying internal details about the file.

heterogeneous Descriptor applied to networks composed of products from multiple vendors.

hierarchy A system of objects in which each object belongs to a group. Groups belong to other groups. Only the "head" does not belong to another group. In AIX this object is called the "Root Directory".

highlight To emphasize an area on the display screen by any of several methods, such as brightening the area or reversing the color of characters within the area.

history A list of recently executed commands.

home (directory)

1. A directory associated with an individual user.
2. Your current directory on login or after issuing the **cd** command with no argument.

homogeneous Descriptor applied to networks composed of products from a single vendor.

hypertext Term for on-line interactive documentation of computer software; to be included with AIX.

IEEE Institute of Electrical and Electronics Engineers. A professional society active in standards work, the IEEE is the official body for work on the POSIX (Portable Operating System for Computer Environments) open system interface definition.

index See **file index**.

indirect block A file element which points at data sectors or other indirect blocks.

init The initialization process of AIX. The ancestor of all processes.

initial program load The process of loading the system programs and preparing the system to run jobs.

i-node A collection of logical information about a file including owner, mode, type and location.

i number The internal index or identification of an i-node.

input field An area into which you can type data.

input redirection The accessing of input data from other than standard input (the keyboard or a pipe).

interoperability The ability of different kinds of computers to work well together.

interpreter A program which "interprets" program statements directly from a text (or equivalent) file.

Distinguished from a compiler which creates computer instructions for later direct execution.

interrupt A signal that the operating system must reevaluate its selection of which process should be running. Usually to service I/O devices but also to signal from one process to another.

IP Internet Protocol.

ipl See **initial program load**.

ISO International Standards Organization. A United Nations agency that provides for creation and administration of worldwide standards.

job A collection of activities.

job number An identifying number for a collection of processes devolving from a terminal command.

kernel The part of an operating system that contains programs that control how the computer does its work, such as input/output, management and control of hardware, and the scheduling of user tasks.

keyboard An input device consisting of various keys allowing the user to input data, control cursor and pointer locations, and to control the user/work station dialogue.

kill To prematurely terminate a process.

kill character. The character which erases an entire line (usually @).

LAN Local Area Network. A facility, usually a combination of wiring, transducers, adapter boards, and software protocols, which interconnects workstations and other computers located within a department, building, or neighborhood. Token-Ring and Ethernet are local area network products.

libc A basic set of C callable routines.

library In UNIX operating systems, a collection of existing subroutines that allows programmers to make use of work already done by other programmers. UNIX operating systems often include separate libraries for communications, window management, string handling, math, etc.

line editor An editor which processes one line at a time by the issuing of a command. Usually associated with sequential only terminals such as a teletype.

link An entry in an AIX directory specifying a data file or directory and its name. Note that files and directories are named solely by virtue of links. A name is not an intrinsic property of a file. A file is uniquely identified only by a system generated identification number.

lint A program for removing "fuzz" from C code. Stricter than most compilers. Helps former Pascal programmers sleep at night.

Local Area Network (LAN) A facility, usually a combination of wiring, transducers, adapter boards, and software protocols, which interconnects workstations and other computers located within a department, building, or neighborhood. Token-Ring and Ethernet are local area network products.

login Identifying oneself to the system to gain access.

login directory See **home directory**.

login name The name by which a user is identified to the system.

logout Informing the system that you are through using it.

mail The process of sending or receiving an electronically delivered message within an AIX system. The message or data so delivered.

make Programming tool included in most UNIX operating systems that helps "make" a new program out of a collection of existing subroutines and utilities, by controlling the order in which those programs are linked, compiled, and executed.

map The process of reassigning the meaning of a terminal key. In general, the process of reassigning the meaning of any key.

memory Storage on electronic memory such as random access memory, read only memory, or registers. See **storage**.

message Information displayed about an error or system condition that may or may not require a user response.

motd "Message of the day". The login "billboard" message.

Motif The graphical user interface for OSF, incorporating the X Window System. Behavior of this nterface is compatible with the IBM/Microsoft Presentation Manager user interface for OS/2. Also called OSF/Motif.

mount A logical (i.e., not physical) attachment of one file directory to another. "remote mounting" allows files and directories that reside on physically separate computer systems to be attached to a local system.

mouse A device that allows you to select objects and scroll the display screen by means of buttons.

move Relinking a file or directory to a different or additional directory. The data (if any) is not moved, only the links.

multiprogramming Allocation of computer resources among many programs. Used to allow many users to operate simultaneously and to keep the system busy during delays occasioned by I/O mechanical operations.

multitasking Capability of performing two or more computing tasks, such as interactive editing and complex numeric calculations, at the same time. AIX and OS/2 are multi-tasking operating systems; DOS, in contrast, is a single-tasking system.

multiuser A computer system which allows many people to run programs "simultaneously" using multiprogramming techniques.

mutex An object used to lock a resource. It is similar to mutex semaphores in OS/2 and binary semaphores in AIX.

named pipe See **FIFO**.

Network File System (NFS) A program developed by SUN Microsystems, Inc. for sharing files among

systems connected via TCP/IP. IBM's AIX, VM, and MVS operating systems support NFS.

NFS See Network File System.

NIST National Institute of Science and Technology (formerly the National Bureau of Standards).

node An element within a communication network.

- Computer
- Terminal
- Control Unit.

null A term denoting emptiness or nonexistence.

null device A device used to obtain empty files or dispose of unwanted data.

null string A character string containing zero characters.

object An entity within a program that has attributes (characteristics) in how it behaves. Often it can not be looked at directly, but must be manipulated with calls to alter its characteristics. This is done for portability purposes, and for enhanced reuse of the object in other programs. (See Object Oriented Programming). Common examples of objects are threads and mutexes. Sometimes objects are referred to as opaque objects, which further emphasizes that you don't know what's inside.

object-oriented programming Method of programming in which sections of program code and data are represented, used, and edited in the form of "objects", such as graphical elements, window components, etc., rather than as strict computer code. Through object-oriented programming techniques, toolkits can be designed that make programming much easier. Examples of object-oriented programming languages include Pareplace Systems, Inc.'s Smalltalk-80™, AT C++™, and Stepstone Inc.'s Objective-C®.

oem original equipment manufacturer. In the context of AIX, OEM systems refer to the processors of a heterogeneous computer network that are not made or provided by IBM.

Open Software Foundation (OSF) A non-profit consortium of private companies, universities, and research institutions formed to conduct open technological evaluations of available components of UNIX operating systems, for the purpose of assembling selected elements into a complete version of the UNIX operating system available to those who wish to license it. IBM is a founding sponsor and member of OSF.

operating system The programs and procedures designed to cause a computer to function, enabling the user to interact with the system.

option A command argument used to specify the details of an operation. In AIX an option is normally preceded by a hyphen.

ordinary file Files containing text, programs, or other data, but not directories.

OSF See Open Software Foundation.

output redirection Passing a programs standard output to a file.

owner The person who created the file or his subsequent designee.

packet switching The transmission of data in small, discrete switching "packets" rather than in streams, for the purpose of making more efficient use of the physical data channels. Employed in some UNIX system communications.

page To move forward or backward on screen full of data through a file usually referring to an editor function.

parallel processing A computing strategy in which a single large task is separated into parts, each of which then runs in parallel on separate processors.

parent The process emerging from a Fork with a non-zero return code (the process ID of the child process). A directory which points at a specified directory.

password A secret character string used to verify user identification during login.

PATH A variable which specifies which directories are to be searched for programs and shell files.

path name A complete file name specifying all directories leading to that file.

pattern-matching character Special characters such as * or ? that can be used in a file specification to match one or more characters. For example, placing a ? in a file specification means that any character can be in that position.

permission The composite of all modes associated with a file.

pipes UNIX operating system routines that connect the standard output of one process with the standard input of another process. Pipes are central to the function of UNIX operating systems, which generally consist of numerous small programs linked together into larger routines by pipes. The "piping" of the list directory command to the word count command is **ls | wc**. The passing of data by a pipe does not (necessarily) involve a file. When the first program generates enough data for the second program to process, it is suspended and the second program runs. When the second program runs out of data it is suspended and the first one runs.

pipe fitting Connecting two programs with a pipe.

pipeline A sequence of programs or commands connected with pipes.

portability Desirable feature of computer systems and applications, referring to users' freedom to run application programs on computers from many vendors without rewriting the program's code. Also known as "applications portability", "machine-independence", and "hardware-independence"; often cited as a cause of the recent surge in popularity of UNIX operating systems.

port A physical I/O interface into a computer.

POSIX "Portable Operating Systems for Computer Environments". A set of open standards for an

operating system environment being developed under the aegis of the IEEE.

preprocessor The macro generator preceding the C compiler.

process A unit of activity known to the AIX system, usually a program.

process 0 (zero) The scheduler. Started by the "boot" and permanent. See **init**.

process id A unique number (at any given time) identifying a process to the system.

process status The process's current activity.

- Non existent
- Sleeping
- Waiting
- Running
- Intermediate
- Terminated
- Stopped.

profile A file in the users home directory which is executed at login to customize the environment. The name is **.profile**.

prompt A displayed request for information or operator action.

protection The opposite of permission, denying access to a file.

quotation Temporarily cancelling the meaning of a metacharacter to be used as a ordinary text character. A backslash (\) "quotes" the next character only.

raw I/O I/O conducted at a "physical" level.

read permission Allows reading (not execution or writing) of a file.

recursive A recursive program calls itself or is called by a subroutine which it calls.

redirection The use of other than standard input (keyboard or pipe output) or standard output (terminal display or pipe). Usually a file.

reentrant A function that does not hold static data over successive calls, nor does it return a pointer to static data. Also, a reentrant function must not call non-reentrant functions. You can typically tell a reentrant function because it has removed its static data and instead chooses to receive that data as an argument to the function. The argument will be coming from a thread specific location, such as local variables or thread specific data.

regular expression An expression which specifies a set of character strings using metacharacters.

relative path name The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory.

RISC Reduced Instruction Set Computer. A class of computer architectures, pioneered by IBM's John

Cocke, that improves price-performance by minimizing the number and complexity of the operations required in the instruction set of a computer. In this class of architecture, advanced compiler technology is used to provide operations, such as multiplication, that are infrequently used in practice.

root directory The directory that contains all other directories in the RT file system.

scalability Desirable feature of computer systems and applications. Refers to the capability to use the same environment on many classes of computers, from personal computers to supercomputers, to accommodate growth or divergent environments, without rewriting code or losing functionality.

SCCS Source Code Control System. A set of programs for maintaining multiple versions of a file using only edit commands to specify alternate versions.

scope The field of an operation or definition. Global scope means all objects in a set. Local scope means a restriction to a subset of the objects.

screen See **display screen**.

scroll To move information vertically or horizontally to bring into view information that is outside the display screen or pane boundaries.

search and replace The act of finding a match to a given character string and replacing each occurrence with some other string.

search string The pattern used for matching in a search operation.

sed Non-interactive stream editor used to do "batch" editing. Often used as a tool within shell scripts.

server A provider of a service in a computer network; for example, a mainframe computer with large storage capacity may play the role of database server for interactive terminals. See **client**.

setuid A permission which allows the access rights of a program owner to control the access to a file. The program can act as a filter for user data requests.

shell The outermost (user interface) layer of UNIX operating systems. Shell commands start and control other processes, such as editors and compilers; shells can be textual or visual. A series of system commands can be collected together into a "shell script" that executes like a batch (.BAT) file in DOS.

shell program A program consisting of a sequence of shell commands stored in an ordinary text file which has execution permission. It is invoked by simply naming the file as a shell command.

shell script See **shell program**.

single user (mode) A temporary mode used during "booting" of the AIX system.

signal A software generated interrupt to another process. See **kill**.

sockets Destination points for communication in many versions of the UNIX operating system, much as electrical sockets are destination points for

electrical plugs. Sockets, associated primarily with 4.3 BSD, can be customized to facilitate communication between separate processes or between UNIX operating systems.

software Programs.

special character See **metacharacter**.

special file A technique used to access I/O devices in which "pseudo files" are used as the interface for commands and data.

standard error The standard device at which errors are reported, normally the terminal. Error messages may be directed to a file.

standard input The source of data for a filter, which is by default obtained from the terminal, but which may be obtained from a file or the standard output of another filter through a pipe.

standard output The output of a filter which normally is by default directed to the terminal, but which may be sent to a file or the standard input of another filter through a pipe.

stdio A "Standard I/O" package of C routines.

sticky bit A flag which keeps commonly used programs "stick" to the swapping disk for performance.

stopped job A job that has been halted temporarily by the user and which can be resumed at his command.

storage In contrast to memory, the saving of information on physical devices such as fixed disk or tape. See **memory**.

store To place information in memory or onto a diskette, fixed disk, or tape so that it is available for retrieval and updating.

streams Similar to sockets, streams are destination points for communications in UNIX operating systems. Associated primarily with UNIX System V, streams are considered by some to be more elegant than sockets, particularly for interprocess communication.

string A linear collection of characters treated as a unit.

subdirectory A directory which is subordinate to another directory.

subtree That portion of an AIX file system accessible from a given directory below the root.

suffix A character string attached to a file name that helps identify its file type.

superblock Primary information repository of a file system (location of i-nodes, free list, etc.).

superuser The system administration; a user with unique privileges such as upgrading execution priority and write access to all files and directories.

superuser authority The unrestricted ability to access and modify any part of the Operating System. This authority is associated with the user who manages the system.

SVID System V Interface Definition. An AT document defining the standard interfaces to be

used by UNIX System V application programmers and users.

swap space (disk) That space on an I/O device used to store processes which have been swapping out to make room for other processes.

swapping The process of moving processes between main storage and the "swapping device", usually a disk.

symbolic debugger Program for debugging other programs at the source code level. Common symbolic debuggers include sdb, dbx, and xdbx.

sync A command which copies all modified blocks from RAM to the disk.

system The computer and its associated devices and programs.

system unit The part of the system that contains the processing unit, the disk drive and the disk, and the diskette drive.

System V ATC recent releases of its UNIX operating system are numbered as releases of "UNIX System V".

TCP Transmission Control Protocol. A facility for the creation of reliable bytestreams (byte-by-byte, end-to-end transmission) on top of unreliable datagrams. The transmission layer of TCP/IP is used to interconnect applications, such as FTP, so that issues of re-transmission and blocking can be subordinated in a standard way. See TCP/IP.

TCP/IP Transmission Control Protocol/Internet Protocol. Pair of communications protocol considered defacto standard in UNIX operating system environments. IBM TCP/IP for VM and IBM TCP/IP for MVS are licensed programs that provide VM an MVS users with the capability of participating in networks using the TCP/IP protocol suite.

termcap A file containing the description of several hundred terminals. For use in determining communication protocol and available function.

termlib A set of C programs for using **termcap**.

thread A portion of a process that is schedulable and contains many of its own resources. It provides a technique to handle multiple parallel tasks within one process, thus saving on the overhead of multi-process data sharing and management.

thread efficient A program that has been designed from the beginning to efficiently handle multiple concurrent threads.

thread safe A function that protects shared resources from concurrent access by using locks. The most common example of this is to use mutexes to protect global variables in your program. This may not necessarily imply the code is thread efficient. Also see **reenrance**.

thread specific data Data that is global to the thread, but not global to the entire process. It is retrieved by a key that is common to all threads within the process.

tools Compact, well designed programs to perform specific tasks. More complex processes are

performed by sequences of tools, often in the form of pipelines which avoid the need for temporary files.

two-digit display Two seven-segment light-emitting diodes (LEDs) on the operating panel used to track the progress of power-on self-tests (POSTs).

UNIX® Operating System A multi-user, multi-tasking interactive operating system created at AT Bell Laboratories that has been widely used and developed by universities, and that now is becoming increasingly popular in a wide range of commercial applications. See Kernel, Shell, Library, Pipes, Filters.

user interface The component of the AIX Family Definition that describes common user interface functions for the AIX PS/2, AIX/RT, and AIX/370 operating systems.

/usr/grp One of the oldest, and still active, user groups for the UNIX operating systems. IBM is a member of /usr/grp.

uucp A set of AIX utilities allowing

- Autodial of remote systems
- Transfer of files
- Execution of commands on the remote system
- Reasonable security.

vi Visual editor. A character editor with a very powerful collection of editing commands optimized for ASCII terminals; associated with BSD versions of the UNIX operating system.

visual editor An optional editor provided with AIX in which changes are made by modifying an image of the file on the screen, rather than through the exclusive use of commands.

wild card A metacharacter used to specify a set of replacement characters and thus a set of file names. For example "*" is any zero or more characters and "?" is any one character.

window A rectangular area of the screen in which the dialog between you and a given application is displayed.

working directory The directory from which file searches are begun if a complete pathname is not specified. Controlled by the **cd** (change directory) command.

workstation A device that includes a keyboard from which an operator can send information to the system, and a display screen on which an operator can see the information sent to or received from the computer.

write Sending data to an I/O device.

write permission Permission to modify a file or directory.

X/Open An international consortium, including many suppliers of computer systems, concerned with the selection and adoption of open system standards for computing applications. IBM is a corporate sponsor of X/Open. See Common Application Environment.

X Windows IBM's implementation of the X Window System developed at the Massachusetts Institute of Technology with the support of IBM and DEC™, that gives users "windows" into applications and processes not located only or specifically on their own console or computer system. X-Windows is a powerful vehicle for distributing applications among users on heterogeneous networks.

yacc "Yet Another Compiler-Compiler". For producing new command interfaces.

zeroeth argument The command name; the argument before the first.

