Applied Computer Science:CSI 4101

SYSTEMS PROGRAMMING

Dr.Godfrey Justo

Foreword

The African Virtual University (AVU) is proud to participate in increasing access to education in African countries through the production of quality learning materials. We are also proud to contribute to global knowledge as our Open Educational Resources are mostly accessed from outside the African continent.

This module was developed as part of a diploma and degree program in Applied Computer Science, in collaboration with 18 African partner institutions from 16 countries. A total of 156 modules were developed or translated to ensure availability in English, French and Portuguese. These modules have also been made available as open education resources (OER) on oer.avu.org.

On behalf of the African Virtual University and our patron, our partner institutions, the African Development Bank, I invite you to use this module in your institution, for your own education, to share it as widely as possible and to participate actively in the AVU communities of practice of your interest. We are committed to be on the frontline of developing and sharing Open Educational Resources.

The African Virtual University (AVU) is a Pan African Intergovernmental Organization established by charter with the mandate of significantly increasing access to quality higher education and training through the innovative use of information communication technologies. A Charter, establishing the AVU as an Intergovernmental Organization, has been signed so far by nineteen (19) African Governments - Kenya, Senegal, Mauritania, Mali, Cote d'Ivoire, Tanzania, Mozambique, Democratic Republic of Congo, Benin, Ghana, Republic of Guinea, Burkina Faso, Niger, South Sudan, Sudan, The Gambia, Guinea-Bissau, Ethiopia and Cape Verde.

The following institutions participated in the Applied Computer Science Program: (1) Université d'Abomey Calavi in Benin; (2) Université de Ougagadougou in Burkina Faso; (3) Université Lumière de Bujumbura in Burundi; (4) Université de Douala in Cameroon; (5) Université de Nouakchott in Mauritania; (6) Université Gaston Berger in Senegal; (7) Université des Sciences, des Techniques et Technologies de Bamako in Mali (8) Ghana Institute of Management and Public Administration; (9) Kwame Nkrumah University of Science and Technology in Ghana; (10) Kenyatta University in Kenya; (11) Egerton University in Kenya; (12) Addis Ababa University in Ethiopia (13) University of Rwanda; (14) University of Dar es Salaam in Tanzania; (15) Universite Abdou Moumouni de Niamey in Niger; (16) Université Cheikh Anta Diop in Senegal; (17) Universidade Pedagógica in Mozambique; and (18) The University of the Gambia in The Gambia.

Bakary Diallo

The Rector

African Virtual University

Production Credits

Author

Godfrey Justo

Peer Reviewer

Dessalegn Mequanint

AVU - Academic Coordination

Dr. Marilena Cabral

Overall Coordinator Applied Computer Science Program

Prof Tim Mwololo Waema

Module Coordinator

Jules Degila

Instructional Designers

Elizabeth Mbasu

Benta Ochola

Diana Tuel

Media Team

Sidney McGregor

Barry Savala

Edwin Kiprono

Kelvin Muriithi

Victor Oluoch Otieno

Michal Abigael Koyier

Mercy Tabi Ojwang

Josiah Mutsogu

Kefa Murimi

Gerisson Mulongo

Copyright Notice

This document is published under the conditions of the Creative Commons
http://en.wikipedia.org/wiki/Creative_Commons

Attribution http://creativecommons.org/licenses/by/2.5/

Module Template is copyright African Virtual University licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. CC-BY, SA

Supported By

AVU Multinational Project II funded by the African Development Bank.

Table of Contents

Foreword 2

Production Credits 3

Copyright Notice 4

Supported By 4

Course Overview 10

- Welcome to Systems Programming Module . . . 10
- Prerequisites . . . 10
- Materials . . . 10

Units 11

- Unit 0: Pre-Assessment . . . 11
- Unit 1: The C Library and I/O System Calls . . . 11
- Unit 2: Shell Programming and Embedding Assembly in C . . . 11
- Unit 3: Processes, Threads and Memory management . . . 11
- Unit 3: Processes, Threads and Memory management . . . 12
- Unit 4: Inter Process Communication . . . 12
  - Assessment . . . 12

Schedule 13

Readings and Other Resources 13

- Unit 0 . . . 13
- Unit 1 . . . 14
- Unit 2 . . . 14
- Unit 3 . . . 14
- Unit 4 . . . 15

Unit 0. Pre-Assessment 16

- Unit Introduction . . . 16
- Unit Objectives . . . 17
- Activity 1 – Getting Started with C Programming . . . 17

Introduction 17
Activity details 18
Conclusion 28
Answers 28
Unit Readings and Other Resources 28
Assessment 28
Unit Assessment. 28
Instructions 28
Unit .1 The C Library and I/O System Call 29
Unit Introduction. 29
Unit Objectives 29
Learning Activities 30
Activity 1- The GNU C Library Overview. 30
Introduction 30
Activity Details. 30
Standards and Portability 30
Using the Library. 32
The C Library for Allocation of Storage for Program Data 40
Conclusion 43
Assessment 44
Activity 2 - File, Directories and Links 44
Introduction 44
Activity Details. 44
Examining or Modifying Directories 44
Conclusion 52
Assessment 53
Activity 3 - File input/output system calls 55
Introduction 55
Activity Details. 56

Activity Details . . . 61
Assessment . . . 63
Unit Assessment . . . 65
Grading Scheme . . . 66
Answers 66
Unit Readings and Other Resources . . . 66
Unit 2.Shell Programming and Embedding Assembly in C 67
Unit Introduction . . . 67
Learning Activities . . . 69
Activity 1 - Shell Basics. . . . 69
Introduction 69
Key Terms . . . 69
Activity Details . . . 71
Conclusion . . . 81
Activity 2 – Shell Programming . . . 81
Introduction 81
Assessment . . . 81
Activity Details . . . 82
Getting started with Scripting . . . 82
Conclusion . . . 94
Activity 3 - Inline Assembly Code . . . 94
Introduction 94
Activity Details . . . 95
Assessment . . . 102
Unit Summary . . . 102
Conclusion . . . 102
Unit Assessment . . . 103
Instructions 103
Grading Scheme . . . 106

Unit Readings and Other Resources 106

Unit 3. Processes,Threads and Memory management 107

Unit Introduction. 107

Key Terms 108

Learning Activities 108

Activity 1 - Processes 108

Introduction 108

Activity Details 109

Assessment. 124

Conclusion 124

Activity 2 - Threads 124

Introduction 124

Activity Details 126

Conclusion 132

Activity 3 - Memory Management 133

Introduction 133

Activity details 134

Assessment 142

Conclusion 142

Unit Summary. 144

Unit Assessment 144

Instructions 144

Grading Scheme 147

Answers 147

Unit Readings and Other Resources 147

Unit 4. Inter Process Communication 148

Unit Introduction. 148

Unit Objectives 149

Learning Activities 149

Activity 1 - Pipes **149**

Introduction 149

Activity Details **150**

Conclusion **155**

Assessment **155**

Activity 2 - FIFOs **156**

Introduction 156

Activity Details **156**

Conclusion **157**

Activity 3 - Sockets **157**

Introduction 157

Assessment **157**

Activity Details **158**

Conclusion **168**

Assessment **168**

Unit Summary **169**

Unit Assessment **169**

Instructions 169

Course Overview

Welcome to Systems Programming Module

The basic objective of coding activity is to produce programs that are easy to understand. It has been argued by many that systems programming practice helps develop programs that are easy to understand. The System Programming module is about advance low level programming topics in C, thus assume that you are already familiar with the C programming language and that you know how to use the standard C library functions in your programs. The C language is the most widely used language for developing System software; most of the commands and libraries that we discuss in this module, and most of the Linux kernel itself, are written in C. Consolidate the programming skills from the previous core courses. The System Programming course concentrates on how programs run in user space and how the interact with the OS. It does not cover OS internals. That will be covered in the Operating Systems CourseIt solidifies the programming skills by having the students write large programs (>1000 lines). The students will use tools like IDEs, debuggers, profilers, and source control to help them write good and maintainable code. The students will learn how to work on teams. The Module Intends to apply use of Scripting Languages. The students will learn to write multi-process and multi-threaded programs. This Module will cover The C Library and I/O System Calls, Shell Programming and Embedding Assembly in C, Processes, Threads and Memory management and Inter Process Communication

Prerequisites

- Introduction to operating systems
- Introduction to structured programming
- Unix/Linux Operating system basics
- Programming in C

Materials

The materials required to complete this course are:

- Linux,
- Computer with internet
- C programming language

Course Goals

Upon completion of this course the learner should be able to

- write low-level programs using C programming language;
- write shell programs to access the kernel and Kernel APIs. basic operating system command files
- Program in low-level Unix/Linux

Units

Unit 0: Pre-Assessment

This unit provides an overview to the systems programming module. It reviews the basics of programming with C Language and describe the organization of modules contents, activities and assessments.

Unit 1: The C Library and I/O System Calls

This unit provides an overview of the GNU C Library, including the library Standards and Portability, the basics of using the library, and the case of C library functions for allocation of storage for program data. The unit further explores the GNU C library's functions for manipulating files and directories. The unit also discusses the file input/output system calls in GNU/LINUX platform and highlights the kernel's I/O calls relation to Standard C Library I/O functions.

Unit 2: Shell Programming and Embedding Assembly in C

Shell program provided by the Operating Systems accepts user instruction or commands and pass to the kernel, similarly, programs written in Higher-level languages such as C also provide a way to instruct the hardware to perform some user task. For occasions when high-level language programmers need to use assembly instructions in their programs, the GNU Compiler Collection permits programmers to add architecture-dependent assembly language instructions to their programs. This unit explores both programming context, namely, the shell programming and inlining assembly in C.

Unit 3: Processes, Threads and Memory management

Program must be brought into memory and placed within a process for it to be run. A running instance of a program is called a process, and threads like processes, are a mechanism to allow a program to do more than one thing at a time. In this unit we shall describe the process, thread and memory manipulation functions in Linux systems most of whose are similar to those on other UNIX systems.

Unit 3: Processes, Threads and Memory management

Program must be brought into memory and placed within a process for it to be run. A running instance of a program is called a process, and threads like processes, are a mechanism to allow a program to do more than one thing at a time. In this unit we shall describe the process, thread and memory manipulation functions in Linux systems most of whose are similar to those on other UNIX systems.

Unit 4: Inter Process Communication

Inter-process communication is the transfer of data among processes. In this unit we present various ways for communicating between parents and children, between "unrelated" processes, and even between processes on different machines. Three types of inter-process communication are discussed:

i. Pipes - permit sequential communication from one process to a related process;

ii. FIFOs - are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem ;

iii. Sockets - support communication between unrelated processes even on different computers.

Assessment

Formative assessments, used to check learner progress, are included in each unit.

Summative assessments, such as final tests and assignments, are provided at the end of each module and cover knowledge and skills from the entire module.

Summative assessments are administered at the discretion of the institution offering the course. The suggested assessment plan is as follows:

1	Formative evaluation	60
2	Summative evaluation	40
3	Total	100

Schedule

Unit	Activities	Estimated
Unit 0: Pre-Assessment	Activities+ Questions	20 Hours
Unit 1: The C Library and I/O System Calls	Activities + Questions	25 Hours
Unit 2: Shell Programming and Embedding Assembly in C	Activities + Questions	25 Hours
Unit 3: Processes, Threads and Memory management	Activities+ Questions	25 Hours
Unit 4: Inter Process Communication	Activities + Questions	25 Hours

Readings and Other Resources

The readings and other resources in this course are:

Unit 0

Required readings and other resources:

1. Advanced Linux Programming, by Mark Mitchell, Jeffrey Oldham, and Alex Samuel, New Riders Publishing, FIRST EDITION: June, 2001, pp 3-15

Optional readings and other resources:

1. 1. Linux System Programming: Talking Directly to the Kernel and C Library By Robert Love
2. UNIX Systems Programming: Communication, Concurrency, and Threads. By Kay A. Robbins, Steven Robbins

Unit 1

Required readings and other resources:

1. The GNU C Library Reference Manual, Sandra Loosemore With Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper for version 2.21, Copyright c 1993–2014 Free Software Foundation, Inc. (http://www.gnu.org/software/libc/manual/pdf/libc.pdf)
2. Advanced Linux Programming, by Mark Mitchell, Jeffrey Oldham, and Alex Samuel, New Riders Publishing, FIRST EDITION: June, 2001, pp 281-300

Optional readings and other resources:

1. Linux System Programming: Talking Directly to the Kernel and C Library By Robert Love
2. UNIX Systems Programming: Communication, Concurrency, and Threads. By Kay A. Robbins, Steven Robbins
3. http://www.acm.uiuc.edu/webmonkeys/book/c_guide/: The C Library Reference Guide
4. http://www.delorie.com/gnu/docs/glibc/libc_toc.html: The GNU C Library

Unit 2

Required readings and other resources:

1. Learning the bash Shell: Unix Shell Programming By Cameron Newham and Bill Rosenblatt,Copyright 2005, O'Reilly Media, USA.

Optional readings and other resources:

1. Learning the bash Shell: Unix Shell Programming (In a Nutshell (O'Reilly)),
2. Kindle Edition, Cameron Newham (Author)
3. Advanced Linux Programming, by Mark Mitchell, Jeffrey Oldham, and Alex
4. Samuel, New Riders Publishing, FIRST EDITION: June, 2001

Unit 3

Required readings and other resources:

1. Advanced Linux Programming, by Mark Mitchell, Jeffrey Oldham, and Alex Samuel, New Riders Publishing, FIRST EDITION: June, 2001, pp 3-15

Optional readings and other resources:

1. Linux System Programming: Talking Directly to the Kernel and C Library By Robert Love
2. Linux Kernel Development, Robert Love, Pearson Education, 22 Jun 2010
3. UNIX Systems Programming: Communication, Concurrency, and Threads. By Kay A. Robbins, Steven Robbins

Unit 4

Required readings and other resources:

1. Advanced Linux Programming, by Mark Mitchell, Jeffrey Oldham, and Alex Samuel, New Riders Publishing, FIRST EDITION: June, 2001
2. Optional readings and other resources:
3. UNIX Systems Programming: Communication, Concurrency, and Threads, By Kay A. Robbins, Steven Robbins, Prentice Hall Professional, 2003
4. Interprocess Communications in Linux, By John Shapley Gray, Prentice Hall Professional, 2003
5. UNIX Network Programming: Interprocess communications, Volume 2 , W. Richard Stevens Prentice Hall PTR, 1999
6. Linux System Programming: Talking Directly to the Kernel and C Library By Robert Love

Unit 0. Pre-Assessment

Unit Introduction

The System Programming module is about advanced programming topics in C, thus assume that you are already familiar with the C programming language and that you know how to use the standard C library functions in your programs. The C language is the most widely used language for developing System software; most of the commands and libraries that we discuss in this module, and most of the Linux kernel itself, are written in C.

The information in this module is equally applicable to C++ programs because the language is roughly a superset of C. If you've programmed on another UNIX-like system platform before, chances are good that you already know your way around Linux's low-level I/O functions (open,read, stat, and so on). These are different from the standard C library's I/O functions (fopen, fprintf, fscanf, and so on). Both are useful in System programming, and we shall learn and use both sets of I/O functions in this module. If you're not familiar with the low-level I/O functions, relax and rest assured to be familiar soon since in Unit 1 we shall deal with "Low-Level I/O".

This module does not provide a general introduction to GNU/Linux systems. It is assumed that you already have a basic knowledge of how to interact with a GNU/Linux system and perform basic operations in graphical and command-line environments (If you are wondering how you can use GCC on Windows, you can just download Cygwin from www.cygwin.com). The following basic conventions shall be used.

- When we show interactions with a command shell, we use $ as the shell prompt (your shell is probably configured to use a different prompt). Everything after the prompt is what you type, while other lines of text are the system's response. For example, in this interaction

  $uname

  Linux

  the system prompted you with $. You entered the uname command. The system responded by printing Linux.

- The source code examples includes a filename in double quotation marks. If you copy in the code example, save it to a file by this name. The code examples were written and tested using the Red Hat 6.2 distribution of GNU/Linux. This distribution incorporates release 2.2.14 of the Linux kernel, release 2.1.3 of the GNU C library, and the EGCS 1.1.2 release of the GNU C compiler. The information andprograms provided in this module should generally be applicable to other versions and distributions of GNU/Linux as well, including 2.4 releases of the Linux kernel and 2.2 releases of the GNU C library.

- Release 2.2.14 of the Linux kernel, release 2.1.3 of the GNU C library, and the EGCS 1.1.2 release of the GNU C compiler. The information and programs provided in this module should generally be applicable to other versions and distributions of GNU/Linux as well, including 2.4 releases of the Linux kernel and 2.2 releases of the GNU C library.

Unit Objectives

Upon completion of this unit you should be able to:

- Create and open C/C++ source file
- Compile with GCC single/multiple C/C++ source files
- Link object files
- Automate the compilation and link Process with GNU Make
- Debug C/C++ programs with GNU Debugger
- Work with header files in C/C++ programs
- Search for help information from Linux sources

Key Terms

Compiler: Turns human-readable source code into machine-readable object code that can actually run

Linker: A computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file

Debugger: The program that you use to figure out why your program isn't behaving the way you think it should

GCC: The GNU Compiler Collection

GNU:GNU's Not Unix!

Activity 1 – Getting Started with C Programming

Introduction

This module reviews the basic steps required to create a C or C++ Linux program. In particular, we describe how to create and modify C and C++ source code, compile that code, and debug the result.

Activity details

Creating/Opening a C or C++ Source File

An editor is the program that you use to edit source code. Lots of different editors are available for Linux, but the most popular and full-featured editor is probably GNU Emacs. You can start Emacs or any of your favourite editor by typing its name in your terminal window and pressing the Return key, e.g. type emacs to invoke the Emacs editor. When your editor has been started, you can use the menus at the top to create a new source file.

If you want to create a C source file, use a filename that ends in .c or .h. If you want to create a C++ source file, use a filename that ends in .cpp, .hpp, .cxx, .hxx, .C, or .H. When the file is open, you can type as you would in any ordinary word-processing program.

Compiling with GCC

A compiler turns human-readable source code into machine-readable object code that can actually run. The compilers of choice on Linux systems are all part of the GNU Compiler Collection, usually known as GCC. GCC also include compilers for C, C++, Java, Objective-C, Fortran, and Chill. This module focuses mostly on C programming. Suppose that you have a project with one C++ source file "reciprocal.cpp" and one C source file "main.c" as shown below. These two files are supposed to be compiled and then linked together to produce a program called reciprocal (Note: In Windows, executables usually have names that end in .exe. Linux programs, on the other hand, usually have no extension. So, the Windows equivalent of this program would probably be called reciprocal.exe; the Linux version is just plain reciprocal.). This program computes the reciprocal of an integer.

```
//Program "main.c" -  C source file—main.c

#include <stdio.h>

#include "reciprocal.hpp"

int main (int argc, char **argv)

{

int i;

i = atoi (argv[1]);

printf ("The reciprocal of %d is %g\n", i, reciprocal (i));

return 0;

}

//Program "reciprocal.cpp" -  C++ source file—reciprocal.cpp

#include <cassert>

#include "reciprocal.hpp"
```

```
double reciprocal (int i) {

// I should be non-zero.

assert (i != 0);

return 1.0/i;

}
```

There is also one header file called "reciprocal.hpp", as given below.

```
//Header file "reciprocal.hpp" - Header file—reciprocal.hpp

#ifdef __cplusplus

extern "C" {

#endif

extern double reciprocal (int i);

#ifdef __cplusplus

}

#endif
```

The first step is to turn the C and C++ source code into object code.

Compiling a Single Source File

The name of the C compiler is gcc. To compile a C source file, you use the –c option. So, for example, entering this at the command prompt compiles the main.c source file:

```
$ gcc -c main.c
```

The resulting object file is named main.o.

The C++ compiler is called g++. Its operation is very similar to gcc; compiling reciprocal.cpp is accomplished by entering the following:

```
$ g++ -c reciprocal.cpp
```

The -c option tells g++ to compile the program to an object file only; without it, g++ will attempt to link the program to produce an executable. After you've typed this command, you'll have an object file called reciprocal.o.You'll probably need a couple other options to build any reasonably large program. The -I option is used to tell GCC where to search for header files. By default, GCC looks in the current directory and in the directories where headers for the standard libraries are installed. If you need to include header files from somewhere else, you'll need the -I option. For example, suppose that your project has one directory called src, for source files, and another called include.You would compile reciprocal.cpp like this to indicate that g++ should use the ../include directory in addition to find reciprocal.hpp:

```
$ g++ -c -I ../include reciprocal.cpp
```

Sometimes you'll want to define macros on the command line. For example, in production code, you don't want the overhead of the assertion check present in reciprocal.cpp; that's only there to help you debug the program. You turn off the check by defining the macro NDEBUG. You could add an explicit #define to reciprocal.cpp, but that would require changing the source itself. It's easier to simply define NDEBUG on the command line, like this:

```
$ g++ -c -D NDEBUG reciprocal.cpp
```

If you had wanted to define NDEBUG to some particular value, you could have done something like this:

```
$ g++ -c -D NDEBUG=3 reciprocal.cpp
```

If you're really building production code, you probably want to have GCC optimize the code so that it runs as quickly as possible. You can do this by using the -O2 command-line option. (GCC has several different levels of optimization; the second level is appropriate for most programs.)

For example, the following compiles reciprocal.cpp with optimization turned on:

```
$ g++ -c -O2 reciprocal.cpp
```

Note that compiling with optimization can make your program more difficult to debug with a debugger. Also, in certain instances, compiling with optimization can uncover bugs in your program that did not manifest themselves previously. You can pass lots of other options to gcc and g++. The best way to get a complete list is to view the online documentation. You can do this by typing the following at your command prompt:

```
$ info gcc
```

Linking Object Files

Now that you've compiled main.c and reciproca.cpp, you'll want to link them. You should always use g++ to link a program that contains C++ code, even if it also contains C code. If your program contains only C code, you should use gcc instead. Because this program contains both C and C++, you should use g++, like this:

```
$ g++ -o reciprocal main.o reciprocal.o
```

The -o option gives the name of the file to generate as output from the link step. Now you can run reciprocal like this:

```
$ ./reciprocal 7

The reciprocal of 7 is 0.142857
```

As you can see, g++ has automatically linked in the standard C runtime library containing the implementation of printf. If you had needed to link in another library (such as a graphical user interface toolkit), you would have specified the library with the -l option. In Linux, library names almost always start with lib. For example, the Pluggable Authentication Module (PAM) library is called libpam.a. To link in libpam.a, you use a command like this:

```
$ g++ -o reciprocal main.o reciprocal.o –lpam
```

The compiler automatically adds the lib prefix and the .a suffix. As with header files, the linker looks for libraries in some standard places, including the /lib and /usr/lib directories that contain the standard system libraries. If you want the linker to search other directories as well, you should use the -L option, which is the parallel of the -I option discussed earlier. You can use this line to instruct the linker to look for libraries in the /usr/local/lib/pam directory before looking in the usual places:

```
$ g++ -o reciprocal main.o reciprocal.o -L/usr/local/lib/pam –lpam
```

Although you don't have to use the -I option to get the preprocessor to search the current directory, you do have to use the -L option to get the linker to search the current directory. In particular, you could use the following to instruct the linker to find the test library in the current directory:

```
$ gcc -o app app.o -L. –ltest
```

Automating the Process with GNU Make

If you're accustomed to programming for the Windows operating system, you're probably accustomed to working with an Integrated Development Environment (IDE).You add sources files to your project, and then the IDE builds your project automatically. Although IDEs are available for Linux, this module does not discuss them. Instead, this module shows you how to use GNU Make to automatically recompile your code, which is what most Linux programmers actually do.

The basic idea behind make is simple. You tell make what targets you want to build and then give rules explaining how to build them. You also specify dependencies that indicate when a particular target should be rebuilt. In our sample reciprocal project, there are three obvious targets: reciprocal.o, main.o, and the reciprocal itself. You already have rules in mind for building these targets in the form of the command lines given previously. The dependencies require a little bit of thought. Clearly, reciprocal depends on reciprocal.o and main.o because

you can't link the complete program until you have built each of the object files. The object files should be rebuilt whenever the corresponding source files change. There's one more twist in that a change to reciprocal.hpp also should cause both of the object files to be rebuilt because both source files include that header file.

In addition to the obvious targets, there should always be a clean target. This target removes all the generated object files and programs so that you can start fresh. The rule for this target uses the rm command to remove the files

You can convey all that information to make by putting the information in a file named Makefile. Here's what Makefile contains:

```
reciprocal: main.o reciprocal.o

g++ $(CFLAGS) -o reciprocal main.o reciprocal.o

main.o: main.c reciprocal.hpp

gcc $(CFLAGS) -c main.c
```

```
reciprocal.o: reciprocal.cpp reciprocal.hpp

g++ $(CFLAGS) -c reciprocal.cpp

clean:

rm -f *.o reciprocal
```

You can see that targets are listed on the left, followed by a colon and then any dependencies. The rule to build that target is on the next line (Ignore the $(CFLAGS) bit for the moment). The line with the rule on it must start with a Tab character, or make will get confused. If you edit your Makefile in Emacs, Emacs will help you with the formatting. If you remove the object files that you've already built, and just type

```
$ make

on the command-line, you'll see the following:

$ make

gcc -c main.c

g++ -c reciprocal.cpp

g++ -o reciprocal main.o reciprocal.o
```

You can see that make has automatically built the object files and then linked them. If you now change main.c in some trivial way and type make again, you'll see the following:

```
$ make

gcc -c main.c

g++ -o reciprocal main.o reciprocal.o
```

You can see that make knew to rebuild main.o and to re-link the program, but it didn't bother to recompile reciprocal.cpp because none of the dependencies for reciprocal.o had changed. The $(CFLAGS) is a make variable. You can define this variable either in the Makefile itself or on the command line. GNU make will substitute the value of the variable when it executes the rule. So, for example, to recompile with optimization enabled, you would do this:

```
$ make clean

rm -f *.o reciprocal

$ make CFLAGS=-O2

gcc -O2 -c main.c

g++ -O2 -c reciprocal.cpp

g++ -O2 -o reciprocal main.o reciprocal.o.
```

Note that the -O2 flag was inserted in place of $(CFLAGS) in the rules. This section, has presented only the most basic capabilities of make. You can find out more by typing this:

```
$ info make
```

In that manual, you'll find information about how to make maintaining a Makefile easier, how to reduce the number of rules that you need to write, and how to automatically compute dependencies. You can also find more information in GNU, Autoconf,Automake, and Libtool by Gary V.Vaughan, Ben Elliston,Tom Tromey, and Ian Lance Taylor (New Riders Publishing, 2000).

Debugging with GNU Debugger (GDB)

The debugger is the program that you use to figure out why your program isn't behaving the way you think it should. You'll be doing this a lot unless your programs always work the first time. The GNU Debugger (GDB) is the debugger used by most Linux programmers. You can use GDB to step through your code, set breakpoints, and examine the value of local variables.

Compiling with Debugging Information

To use GDB, you'll have to compile with debugging information enabled. Do this by adding the -g switch on the compilation command line. If you're using a Makefile as described previously, you can just set CFLAGS equal to -g when you run make, as shown here:

```
$ make CFLAGS=-g

gcc -g -c main.c

g++ -g -c reciprocal.cpp

g++ -g -o reciprocal main.o reciprocal.o
```

When you compile with -g, the compiler includes extra information in the object files and executables. The debugger uses this information to figure out which addresses correspond to which lines in which source files, how to print out local variables, and so forth.

Running GDB

You can start up gdb by typing:

```
$ gdb reciprocal
```

When gdb starts up, you should see the GDB prompt:

```
(gdb)
```

The first step is to run your program inside the debugger. Just enter the command run and any program arguments. Try running the program without any arguments, like this:

```
(gdb) run
Starting program: reciprocal
Program received signal SIGSEGV, Segmentation fault.
 __strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
287 strtol.c: No such file or directory.
(gdb)
```

The problem is that there is no error-checking code in main. The program expects one argument, but in this case the program was run with no arguments. The SIGSEGV message indicates a program crash. GDB knows that the actual crash happened in a function called __strtol_internal. That function is in the standard library, and the source isn't installed, which explains the "No such file or directory" message. You can see the stack by using the where command:

```
(gdb) where
#0 __strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
#1 0x40096fb6 in atoi (nptr=0x0) at ../stdlib/stdlib.h:251
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
```

You can see from this display that main called the atoi function with a NULL pointer, which is the source of the trouble. You can go up two levels in the stack until you reach main by using the up command:

```
(gdb) up 2
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
8 i = atoi (argv[1]);
```

Note that gdb is capable of finding the source for main.c, and it shows the line where the erroneous function call occurred. You can view the value of variables using the print command:

```
(gdb) print argv[1]
$2 = 0x0
```

That confirms that the problem is indeed a NULL pointer passed into atoi. You can set a breakpoint by using the break command:

```
(gdb) break main

Breakpoint 1 at 0x804862e: file main.c, line 8.
```

This command sets a breakpoint on the first line of main (Some people have commented that saying break main is a little bit funny because usually you want to do this only when main is already broken). Now try rerunning the program with an argument, like this:

```
(gdb) run 7

Starting program: reciprocal 7

Breakpoint 1, main (argc=2, argv=0xbffff5e4) at main.c:8

8 i = atoi (argv[1]);
```

You can see that the debugger has stopped at the breakpoint. You can step over the call to atoi using the next command:

```
(gdb) next

9 printf ("The reciprocal of %d is %g\n", i, reciprocal (i));
```

If you want to see what's going on inside reciprocal, use the step command like this:

```
(gdb) step

reciprocal (i=7) at reciprocal.cpp:6

6 assert (i != 0);
```

You're now in the body of the reciprocal function. You might find it more convenient to run gdb from within Emacs rather than using gdb directly from the command line. Use the command M-x gdb to start up gdb in an Emacs window. If you are stopped at a breakpoint, Emacs automatically pulls up the appropriate source file. It's easier to figure out what's going on when you're looking at the whole file rather than just one line of text.

Finding More Information

Nearly every Linux distribution comes with a great deal of useful documentation. You can learn most of what has been presented in this module by reading documentation in your Linux distribution (although it would probably take you much longer). The documentation isn't always well-organized, though, so the tricky part is finding what you need. Documentation is also sometimes out-of-date, so take everything that you read with a grain of salt. If the system doesn't behave the way a man page (manual pages) says it should, for instance, it may be that the man page is outdated.To help you navigate, here are the most useful sources of information about advanced Linux programming.

Man Pages

Linux distributions include man pages for most standard commands, system calls, and standard library functions. The man pages are divided into numbered sections; for programmers, the most important are these:

(1) User commands

(2) System calls

(3) Standard library functions

(8) System/administrative commands

The numbers denote man page sections. Linux's man pages come installed on your system; use the man command to access them. To look up a man page, simply invoke man name, where name is a command or function name. In a few cases, the same name occurs in more than one section; you can specify the section explicitly by placing the section number before the name. For example, if you type the following, you'll get the man page for the sleep command (in section 1 of the Linux man pages):

```
$ man sleep
```

To see the man page for the sleep library function, use this command:

```
$ man 3 sleep
```

Each man page includes a one-line summary of the command or function. The whatis name command displays all man pages (in all sections) for a command or function matching name. If you're not sure which command or function you want, you can perform a keyword search on the summary lines, using man -k keyword.

Man pages include a lot of very useful information and should be the first place you turn for help. The man page for a command describes command-line options and arguments, input and output, error codes, configuration, and the like. The man page for a system call or library function describes parameters and return values, lists error codes and side effects, and specifies which include file to use if you call the function.

Info

The Info documentation system contains more detailed documentation for many core components of the GNU/Linux system, plus several other programs. Info pages are hypertext documents, similar to Web pages. To launch the text-based Info browser, just type info in a shell window. You'll be presented with a menu of Info documents installed on your system. (Press Control+H to display the keys for navigating an Info document.)

Among the most useful Info documents are these:

- gcc—The gcc compiler
- libc—The GNU C library, including many system calls
- gdb—The GNU debugger
- emacs—The Emacs text editor
- info—The Info system itself

Almost all the standard Linux programming tools (including ld, the linker; as, the assembler; and gprof, the profiler) come with useful Info pages. You can jump directly to a particular Info document by specifying the page name on the command line:

```
$ info libc
```

If you do most of your programming in Emacs, you can access the built-in Info browser by typing M-x info or C-h i.

Header Files

You can learn a lot about the system functions that are available and how to use them by looking at the system header files. These reside in /usr/include and /usr/include/sys. If you are getting compile errors from using a system call, for instance, take a look in the corresponding header file to verify that the function's signature is the same as what's listed in the man page. On Linux systems, a lot of the nitty-gritty details of how the system calls work are reflected in header files in the directories /usr/include/bits, /usr/include/asm, and /usr/include/linux. For instance, the numerical values of signals are defined in /usr/include/bits/signum.h. These header files make good reading for inquiring minds. Don't include them directly in your programs, though; always use the header files in /usr/include or as mentioned in the man page for the function you're using.

Source Code

This is Open Source materials, right? The final arbiter of how the system works is the system source code itself, and luckily for Linux programmers, that source code is freely available. Chances are, your Linux distribution includes full source code for the entire system and all programs included with it; if not, you're entitled under the terms of the GNU General Public License to request it from the distributor (The source code might not be installed on your disk, though. See your distribution's documentation for instructions on installing it).

The source code for the Linux kernel itself is usually stored under /usr/src/linux. If this module leaves you thirsting for details of how processes, shared memory, and system devices work, you can always learn straight from the source code. Most of the system functions described in this module are implemented in the GNU C library; check your distribution's documentation for the location of the C library source code.

Conclusion

In this unit we presented the basics of programming with the C/C++ language.

Assessment

1. Practise the example code given in this unit.

Unit Summary

This unit presented the fundamentals of working with C/C++ source files, compiling, linking and debugging.

Unit Assessment

Check your understanding!

Miscellaneous exercises

Instructions

What is the purpose of a Makefile?

1. Inspect the source code given in the Linux filesystem folder /usr/src/linux

Grading Scheme

As guided by the offering Institution Grading Regulations

Answers

mailto:njulumi@gmail.com

Unit Readings and Other Resources

1. Mark Mitchell, Jeffrey Oldham, and Alex Samuel; Advanced Linux Programming; Copyright © 2001 by New Riders Publishing; FIRST EDITION: June, 2001

Unit .1 The C Library and I/O System Call

Unit Introduction

The C standard library provides macros, type definitions, and functions for tasks like string handling, mathematical computations, input/output processing, memory allocation and several other operating system services. This unit will make use of the GNU C library, and assume that you are at least somewhat familiar with the C programming language and basic programming concepts. Specifically, familiarity with ISO standard, rather than "traditional" pre-ISO C dialects is assumed.

The GNU C library includes several header files, each of which provides definitions and declarations for a group of related facilities; this information is used by the C compiler when processing your program. For example, the header file `stdio.h' declares facilities for performing input and output, and the header file `string.h' declares string processing utilities.

There are a lot of functions in the GNU C library and it's not realistic to expect that you will be able to remember exactly how to use each and every one of them. It's more important to become generally familiar with the kinds of facilities that the library provides, so that when you are writing your programs you can recognize when to make use of library functions. The purpose of this unit is to introduce learners how to use the various features of the GNU library. Specifically, the unit shall discuss the commonly used file I/O functions and system calls and facilities for file, directory and links manipulations.

Unit Objectives

Upon completion of this unit you should be able to:

- apply different features of the C library in C programs
- use I/O functions and System calls in writing programs
- add files, directories and links in C programs

Key Terms

ISO: International Standard Organization

GNU: Goose Not Unix

POSIX: Portable Operating System Interface Extension

C Library: Collection of pre-compiled program routines available for re-use in an ordinary C program

Directory (folder): a file system structure in which to store computer files

Link (symlink): A special type of file that contains a reference to another file or directory in the form of an absolute or relative path and that affects pathname resolution.

File:A logical unit of data storage in Operating Systems

Input/output (I/O): Input/output (also "I/O") refers to the activity of connecting to an input or output device for reading or writing data respectively.

System call: a request for the operating system to do something on behalf of the user's program

Learning Activities

Activity 1- The GNU C Library Overview

Introduction

The C language provides no built-in facilities for performing such common operations as input/output, memory management, string manipulation, and the like. Instead, these facilities are defined in a standard library, which you compile and link with your programs.

The GNU C library, described in this unit, defines all of the library functions that are specified by the ISO C standard, as well as additional features specific to POSIX and other derivatives of the Unix operating system, and extensions specific to the GNU system.

Activity Details

Standards and Portability

This section discusses the various standards and other sources that the GNU C library is based upon. These sources include the ISO C and POSIX standards, and the System V and Berkeley Unix implementations. This section gives you an overview of these standards, so that you will know what they are when they are mentioned in other parts of the unit.

ISO C - The international standard for the C programming

The GNU C library is compatible with the C standard adopted by the American National Standards Institute (ANSI): American National Standard X3.159-1989---"ANSI C" and later by the International Standardization Organization (ISO): ISO/IEC 9899:1990, "Programming languages--C". We here refer to the standard as ISO C since this is the more general standard in respect of ratification. The header files and library facilities that make up the GNU library are a superset of those specified by the ISO C standard.

If you are concerned about strict adherence to the ISO C standard, you should use the `-ansi' option when you compile your programs with the GNU C compiler. This tells the compiler to define only ISO standard features from the library header files, unless you explicitly ask for additional features.

Being able to restrict the library to include only ISO C features is important because ISO C puts limitations on what names can be defined by the library implementation, and the GNU extensions don't fit these limitations.

This unit does not attempt to give you complete details on the differences between ISO C and older dialects. It gives advice on how to write programs to work portably under multiple C dialects, but does not aim for completeness.

POSIX (The Portable Operating System Interface) - The ISO/IEC 9945 (aka IEEE 1003) standards for operating systems

The GNU library is also compatible with the ISO POSIX family of standards, known more formally as the Portable Operating System Interface for Computer Environments (ISO/IEC 9945). They were also published as ANSI/IEEE Std 1003. POSIX is derived mostly from various versions of the Unix operating system.

The library facilities specified by the POSIX standards are a superset of those required by ISO C; POSIX specifies additional features for ISO C functions, as well as specifying new additional functions. In general, the additional requirements and functionality defined by the POSIX standards are aimed at providing lower-level support for a particular kind of operating system environment, rather than general programming language support which can run in many diverse operating system environments.

The GNU C library implements all of the functions specified in ISO/IEC 9945-1:1996, the POSIX System Application Program Interface, commonly referred to as POSIX.1. The primary extensions to the ISO C facilities specified by this standard include file system interface primitives, device-specific terminal control functions, and process control functions.

Some facilities from ISO/IEC 9945-2:1993, the POSIX Shell and Utilities standard (POSIX.2) are also implemented in the GNU library. These include utilities for dealing with regular expressions and other pattern matching facilities.

Berkeley Unix - BSD and SunOS

The GNU C library defines facilities from some versions of Unix which are not formally standardized, specifically from the 4.2 BSD, 4.3 BSD, and 4.4 BSD Unix systems (also known as Berkeley Unix) and from SunOS (a popular 4.2 BSD derivative that includes some Unix System V functionality). These systems support most of the ISO C and POSIX facilities, and 4.4 BSD and newer releases of SunOS in fact support them all. The BSD facilities include symbolic links, the select function, the BSD signal functions, and sockets.

SVID (The System V Interface Description) - The System V Interface Description

The System V Interface Description (SVID) is a document describing the AT&T Unix System V operating system. It is to some extent a superset of the POSIX standard.

The GNU C library defines most of the facilities required by the SVID that are not also required by the ISO C or POSIX standards, for compatibility with System V Unix and other Unix systems (such as SunOS) which include these facilities. However, many of the more obscure and less generally useful facilities required by the SVID are not included. (In fact, Unix System V itself does not provide them all.The supported facilities from System V include the methods for inter-process communication and shared memory, the hsearch and drand48 families of functions, fmtmsg and several of the mathematical functions.

XPG (The X/Open Portability Guide) - The X/Open Portability Guide

The X/Open Portability Guide, published by the X/Open Company, Ltd., is a more general standard than POSIX. X/Open owns the Unix copyright and the XPG specifies the requirements for systems which are intended to be a Unix system.

The GNU C library complies to the X/Open Portability Guide, Issue 4.2, with all extensions common to XSI (X/Open System Interface) compliant systems and also all X/Open UNIX extensions. The additions on top of POSIX are mainly derived from functionality available in System V and BSD systems. Some of the really bad mistakes in System V systems were corrected, though. Since fulfilling the XPG standard with the Unix extensions is a precondition for getting the Unix brand chances are good that the functionality is available on commercial systems.

Using the Library

This section describes some of the practical issues involved in using the GNU C library.

Header Files

Libraries for use by C programs really consist of two parts: header files that define types and macros and declare variables and functions; and the actual library or archive that contains the definitions of the variables and functions. (Recall that in C, a declaration merely provides information that a function or variable exists and gives its type. For a function declaration, information about the types of its arguments might be provided as well. The purpose of declarations is to allow the compiler to correctly process references to the declared variables and functions. A definition, on the other hand, actually allocates storage for a variable or says what a function does.)

In order to use the facilities in the GNU C library, you should be sure that your program source files include the appropriate header files. This is so that the compiler has declarations of these facilities available and can correctly process references to them. Once your program has been compiled, the linker resolves these references to the actual definitions provided in the archive file.

Header files are included into a program source file by the `#include' preprocessor directive. The C language supports two forms of this directive; the first,

```
#include "header"
```

is typically used to include a header file 'header' that you write yourself; this would contain definitions and declarations describing the interfaces between the different parts of your particular application. By contrast,

```
#include <file.h>
```

is typically used to include a header file `file.h' that contains definitions and declarations for a standard library. This file would normally be installed in a standard place by your system administrator. You should use this second form for the C library header files.

Typically, `#include' directives are placed at the top of the C source file, before any other code. If you begin your source files with some comments explaining what the code in the file does (a good idea), put the `#include' directives immediately afterwards, following the feature test macro definition.

The GNU C library provides several header files, each of which contains the type and macro definitions and variable and function declarations for a group of related facilities. This means that your programs may need to include several header files, depending on exactly which facilities you are using.

Some library header files include other library header files automatically. However, as a matter of programming style, you should not rely on this; it is better to explicitly include all the header files required for the library facilities you are using. The GNU C library header files have been written in such a way that it doesn't matter if a header file is accidentally included more than once; including a header file a second time has no effect. Likewise, if your program needs to include multiple header files, the order in which they are included doesn't matter.

Compatibility Note: Inclusion of standard header files in any order and any number of times works in any ISO C implementation. However, this has traditionally not been the case in many older C implementations.

Strictly speaking, you don't have to include a header file to use a function it declares; you could declare the function explicitly yourself. But it is usually better to include the header file because it may define types and macros that are not otherwise available and because it may define more efficient macro replacements for some functions. It is also a sure way to have the correct declaration.

Macro Definitions of Functions

If we describe something as a function, it may have a macro definition as well. This normally has no effect on how your program runs--the macro definition does the same thing as the function would. In particular, macro equivalents for library functions evaluate arguments exactly once, in the same way that a function call would. The main reason for these macro definitions is that sometimes they can produce an inline expansion that is considerably faster than an actual function call.

Taking the address of a library function works even if it is also defined as a macro. This is because, in this context, the name of the function isn't followed by the left parenthesis that is syntactically necessary to recognize a macro call.

You might occasionally want to avoid using the macro definition of a function--perhaps to make your program easier to debug. There are two ways you can do this:

- You can avoid a macro definition in a specific use by enclosing the name of the function in parentheses. This works because the name of the function doesn't appear in a syntactic context where it is recognizable as a macro call.
- You can suppress any macro definition for a whole source file by using the `#undef' preprocessor directive, unless otherwise stated explicitly in the description of that facility.

For example, suppose the header file `stdlib.h' declares a function named abs with

```
extern int abs (int);
```

and also provides a macro definition for abs. Then, in:

```
#include <stdlib.h>

int f (int *i) { return abs (++*i); }
```

the reference to abs might refer to either a macro or a function. On the other hand, in each of the following examples the reference is to a function and not a macro.

```
#include <stdlib.h>

int g (int *i) { return (abs) (++*i); }

#undef abs

int h (int *i) { return abs (++*i); }
```

Since macro definitions that double for a function behave in exactly the same way as the actual function version, there is usually no need for any of these methods. In fact, removing macro definitions usually just makes your program slower.

Reserved Names

The names of all library types, macros, variables and functions that come from the ISO C standard are reserved unconditionally; your program may not redefine these names. All other library names are reserved if your program explicitly includes the header file that defines or declares them. There are several reasons for these restrictions:

- Other people reading your code could get very confused if you were using a function named exit to do something completely different from what the standard exit function does, for example. Preventing this situation helps to make your programs easier to understand and contributes to modularity and maintainability.
- It avoids the possibility of a user accidentally redefining a library function that is called by other library functions. If redefinition were allowed, those other functions would not work properly.
- It allows the compiler to do whatever special optimizations it pleases on calls to these functions, without the possibility that they may have been redefined by the user. Some library facilities, such as those for dealing with variadic arguments and non-local exits, actually require a considerable amount of cooperation on the part of the C compiler, and with respect to the implementation, it might be easier for the compiler to treat these as built-in parts of the language.

In addition to the names documented GNU Library, reserved names include all external identifiers (global functions and variables) that begin with an underscore (`_') and all identifiers regardless of use that begin with either two underscores or an underscore followed by a capital letter are reserved names. This is so that the library and header files can define functions, variables, and macros for internal purposes without risk of conflict with names in user programs.

Some additional classes of identifier names are reserved for future extensions to the C language or the POSIX.1 environment. While using these names for your own purposes right now might not cause a problem, they do raise the possibility of conflict with future versions of the C or POSIX standards, so you should avoid these names.

- Names beginning with a capital `E' followed a digit or uppercase letter may be used for additional error code names.
- Names that begin with either `is' or `to' followed by a lowercase letter may be used for additional character testing and conversion functions.
- Names that begin with `LC_' followed by an uppercase letter may be used for additional macros specifying locale attributes.
- Names of all existing mathematics functions suffixed with `f' or `l' are reserved for corresponding functions that operate on float and long double arguments, respectively.
- Names that begin with `SIG' followed by an uppercase letter are reserved for additional signal names.

- Names that begin with `SIG_' followed by an uppercase letter are reserved for additional signal actions.
- Names beginning with `str', `mem', or `wcs' followed by a lowercase letter are reserved for additional string and array functions.
- Names that end with `_t' are reserved for additional type names.

In addition, some individual header files reserve names beyond those that they actually define. You only need to worry about these restrictions if your program includes that particular header file.

- The header file `dirent.h' reserves names prefixed with `d_'.
- The header file `fcntl.h' reserves names prefixed with `l_', `F_', `O_', and `S_'.
- The header file `grp.h' reserves names prefixed with `gr_'.
- The header file `limits.h' reserves names suffixed with `_MAX'.
- The header file `pwd.h' reserves names prefixed with `pw_'.
- The header file `signal.h' reserves names prefixed with `sa_' and `SA_'.
- The header file `sys/stat.h' reserves names prefixed with `st_' and `S_'.
- The header file `sys/times.h' reserves names prefixed with `tms_'.
- The header file `termios.h' reserves names prefixed with `c_', `V', `I', `O', and `TC'; and names prefixed with `B' followed by a digit.

Feature Test Macros

The exact set of features available when you compile a source file is controlled by which feature test macros you define.

If you compile your programs using `gcc -ansi', you get only the ISO C library features, unless you explicitly request additional features by defining one or more of the feature macros. See section `GNU CC Command Options' in The GNU CC Manual, for more information about GCC options.

You should define these macros by using `#define' preprocessor directives at the top of your source code files. These directives must come before any #include of a system header file. It is best to make them the very first thing in the file, preceded only by comments. You could also use the `-D' option to GCC, but it's better if you make the source files indicate their own meaning in a self-contained way.

This system exists to allow the library to conform to multiple standards. Although the different standards are often described as supersets of each other, they are usually incompatible because larger standards require functions with names that smaller ones reserve to the user program. This is not mere pedantry -- it has been a problem in practice. For instance, some non-GNU programs define functions named getline that have nothing to do with this library's getline. They would not be compilable if all features were enabled indiscriminately.

This should not be used to verify that a program conforms to a limited standard. It is insufficient for this purpose, as it will not protect you from including header files outside the standard, or relying on semantics undefined within the standard.

Macro: **_POSIX_SOURCE**

If you define this macro, then the functionality from the POSIX.1 standard (IEEE Standard 1003.1) is available, as well as all of the ISO C facilities.

The state of _POSIX_SOURCE is irrelevant if you define the macro _POSIX_C_SOURCE to a positive integer.

Macro: **_POSIX_C_SOURCE**

Define this macro to a positive integer to control which POSIX functionality is made available. The greater the value of this macro, the more functionality is made available.

If you define this macro to a value greater than or equal to 1, then the functionality from the 1990 edition of the POSIX.1 standard (IEEE Standard 1003.1-1990) is made available.

If you define this macro to a value greater than or equal to 2, then the functionality from the 1992 edition of the POSIX.2 standard (IEEE Standard 1003.2-1992) is made available.

If you define this macro to a value greater than or equal to 199309L, then the functionality from the 1993 edition of the POSIX.1b standard (IEEE Standard 1003.1b-1993) is made available.

Greater values for _POSIX_C_SOURCE will enable future extensions. The POSIX standards process will define these values as necessary, and the GNU C Library should support them some time after they become standardized. The 1996 edition of POSIX.1 (ISO/IEC 9945-1: 1996) states that if you define _POSIX_C_SOURCE to a value greater than or equal to 199506L, then the functionality from the 1996 edition is made available.

Macro: **_BSD_SOURCE**

If you define this macro, functionality derived from 4.3 BSD Unix is included as well as the ISO C, POSIX.1, and POSIX.2 material.

Some of the features derived from 4.3 BSD Unix conflict with the corresponding features specified by the POSIX.1 standard. If this macro is defined, the 4.3 BSD definitions take precedence over the POSIX definitions.

Due to the nature of some of the conflicts between 4.3 BSD and POSIX.1, you need to use a special BSD compatibility library when linking programs compiled for BSD compatibility. This is because some functions must be defined in two different ways, one of them in the normal C library, and one of them in the compatibility library. If your program defines _BSD_SOURCE, you must give the option `-lbsd-compat' to the compiler or linker when linking the program, to tell it to find functions in this special compatibility library before looking for them in the normal C library.

Macro: **_SVID_SOURCE**

If you define this macro, functionality derived from SVID is included as well as the ISO C,

POSIX.1, POSIX.2, and X/Open material.

Macro: **_XOPEN_SOURCE**

Macro: **_XOPEN_SOURCE_EXTENDED**

If you define this macro, functionality described in the X/Open Portability Guide is included. This is a superset of the POSIX.1 and POSIX.2 functionality and in fact _POSIX_SOURCE and _POSIX_C_SOURCE are automatically defined.

As the unification of all Unices, functionality only available in BSD and SVID is also included.

If the macro _XOPEN_SOURCE_EXTENDED is also defined, even more functionality is available. The extra functions will make all functions available which are necessary for the X/Open Unix brand.

If the macro _XOPEN_SOURCE has the value 500 this includes all functionality described so far plus some new definitions from the Single Unix Specification, version 2.

Macro: **_LARGEFILE_SOURCE**

If this macro is defined some extra functions are available which rectify a few shortcomings in all previous standards. Specifically, the functions fseeko and ftello are available. Without these functions the difference between the ISO C interface (fseek, ftell) and the low-level POSIX interface (lseek) would lead to problems.

This macro was introduced as part of the Large File Support extension (LFS).

Macro: **_LARGEFILE64_SOURCE**

If you define this macro an additional set of functions is made available which enables 32 bit systems to use files of sizes beyond the usual limit of 2GB. This interface is not available if the system does not support files that large. On systems where the natural file size limit is greater than 2GB (i.e., on 64 bit systems) the new functions are identical to the replaced functions.

The new functionality is made available by a new set of types and functions which replace the existing ones. The names of these new objects contain 64 to indicate the intention, e.g., off_t vs. off64_t and fseeko vs. fseeko64.

This macro was introduced as part of the Large File Support extension (LFS). It is a transition interface for the period when 64 bit offsets are not generally used (see _FILE_OFFSET_BITS).

Macro: **_FILE_OFFSET_BITS**

This macro determines which file system interface shall be used, one replacing the other. Whereas _LARGEFILE64_SOURCE makes the 64 bit interface available as an additional interface, _FILE_OFFSET_BITS allows the 64 bit interface to replace the old interface.

If _FILE_OFFSET_BITS is undefined, or if it is defined to the value 32, nothing changes. The 32 bit interface is used and types like off_t have a size of 32 bits on 32 bit systems.

If the macro is defined to the value 64, the large file interface replaces the old interface. I.e., the functions are not made available under different names (as they are with _LARGEFILE64_SOURCE). Instead the old function names now reference the new functions, e.g., a call to fseeko now indeed calls fseeko64.

This macro should only be selected if the system provides mechanisms for handling large files. On 64 bit systems this macro has no effect since the *64 functions are identical to the normal functions.

This macro was introduced as part of the Large File Support extension (LFS).

Macro: **_ISOC99_SOURCE**

Until the revised ISO C standard is widely adopted the new features are not automatically enabled. The GNU libc nevertheless has a complete implementation of the new standard and to enable the new features the macro _ISOC99_SOURCE should be defined.

Macro: **_GNU_SOURCE**

If you define this macro, everything is included: ISO C89, ISO C99, POSIX.1, POSIX.2, BSD, SVID, X/Open, LFS, and GNU extensions. In the cases where POSIX.1 conflicts with BSD, the POSIX definitions take precedence.

If you want to get the full effect of _GNU_SOURCE but make the BSD definitions take precedence over the POSIX definitions, use this sequence of definitions:

```
    #define _GNU_SOURCE
#define _BSD_SOURCE
#define _SVID_SOURCE
```

Note that if you do this, you must link your program with the BSD compatibility library by passing the `-lbsd-compat' option to the compiler or linker. Note: If you forget to do this, you may get very strange errors at run time.

Macro: **_REENTRANT**

Macro: **_THREAD_SAFE**

If you define one of these macros, reentrant versions of several functions get declared. Some of the functions are specified in POSIX.1c but many others are only available on a few other systems or are unique to GNU libc. The problem is the delay in the standardization of the thread safe C library interface.

Unlike on some other systems, no special version of the C library must be used for linking. There is only one version but while compiling this it must have been specified to compile as thread safe.

We recommend you use _GNU_SOURCE in new programs. If you don't specify the `-ansi' option to GCC and don't define any of these macros explicitly, the effect is the same as defining _POSIX_C_SOURCE to 2 and _POSIX_SOURCE, _SVID_SOURCE, and _BSD_SOURCE to 1.

When you define a feature test macro to request a larger class of features, it is harmless to define in addition a feature test macro for a subset of those features. For example, if you define _POSIX_C_SOURCE, then defining _POSIX_SOURCE as well has no effect. Likewise, if you define _GNU_SOURCE, then defining either _POSIX_SOURCE or _POSIX_C_SOURCE or _SVID_SOURCE as well has no effect.

Note, however, that the features of _BSD_SOURCE are not a subset of any of the other feature test macros supported. This is because it defines BSD features that take precedence over the POSIX features that are requested by the other macros. For this reason, defining _BSD_SOURCE in addition to the other feature test macros does have an effect: it causes the BSD features to take priority over the conflicting POSIX features.

The C Library for Allocation of Storage for Program Data

The C language supports two kinds of memory allocation through the variables in C programs:

- Static allocation is what happens when you declare a static or global variable.
- Each static or global variable defines one block of space, of a fixed size. The space is allocated once, when your program is started (part of the exec operation), and is never freed.
- Automatic allocation happens when you declare an automatic variable, such as a function argument or a local variable. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when that compound statement is exited.

A third important kind of memory allocation, dynamic allocation, is not supported by C variables but is available via GNU C Library functions.

Dynamic Memory Allocation

Dynamic memory allocation is a technique in which programs determine as they are running where to store some information. You need dynamic allocation when the amount of memory you need, or how long you continue to need it, depends on factors that are not known before the program runs.

For example, you may need a block to store a line read from an input file; since there is no limit to how long a line can be, you must allocate the memory dynamically and make it dynamically larger as you read more of the line. Or, you may need a block for each record or each definition in the input data; since you can't know in advance how many there will be, you must allocate a new block for each record or definition as you read it.

When you use dynamic allocation, the allocation of a block of memory is an action that the program requests explicitly. You call a function or macro when you want to allocate space, and specify the size with an argument. If you want to free the space, you do so by calling another function or macro. You can do these things whenever you want, as often as you want.

Dynamic allocation is not supported by C variables; there is no storage class dynamic, and there can never be a C variable whose value is stored in dynamically allocated space. The only way to get dynamically allocated memory is via a system call (which is generally via a GNU C Library function call), and the only way to refer to dynamically allocated space is through a pointer.

Basic Memory Allocation

The most general dynamic allocation facility is malloc. It allows you to allocate blocks of memory of any size at any time, make them bigger or smaller at any time, and free the blocks individually at any time (or never). To allocate a block of memory, call malloc. The prototype for this function is in 'stdlib.h'.

Because it is less convenient, and because the actual process of dynamic allocation requires more computation time, programmers generally use dynamic allocation only when neither static nor automatic allocation will serve. For example, if you want to allocate dynamically some space to hold a struct foobar, you cannot declare a variable of type struct foobar whose contents are the dynamically allocated space. But you can declare a variable of pointer type struct foobar * and assign it the address of the space. Then you can use the operators '*' and '->' on this pointer variable to refer to the contents of the space:

```
{
struct foobar *ptr = (struct foobar *) malloc (sizeof (struct foobar));
ptr->name = x;
ptr->next = current_foobar;
current_foobar = ptr;
}
void * malloc (size_t size) Function
```

This function returns a pointer to newly allocated block size bytes long or a null pointer if the block could not be allocated. The contents of the block are undefined; you must initialize it yourself. Normally you would cast the value as a pointer to the kind of object that you want to store in the block. See an example of doing so, and of initializing the space with zeros using the library function memset, below.

```
struct foo *ptr;
...
ptr = (struct foo *) malloc (sizeof (struct foo));
if (ptr == 0) abort ();
memset (ptr, 0, sizeof (struct foo));
```

You can store the result of malloc into any pointer variable without a cast, because ISO C automatically converts the type void * to another type of pointer when necessary. But the cast is necessary in contexts other than assignment operators or if you might want your code to run in traditional C.

Remember that when allocating space for a string, the argument to malloc must be one plus the length of the string. This is because a string is terminated with a null character that doesn't count in the "length" of the string but does need space. For example:

```
char *ptr;
...
ptr = (char *) malloc (length + 1);
```

The block that malloc gives you is guaranteed to be aligned so that it can hold any type of data. In the GNU system, the address is always a multiple of eight on most systems, and a multiple of sixteen on 64-bit systems.

Freeing Memory Allocated with malloc

When you no longer need a block that you got with malloc, use the function free to make the block available to be allocated again. The prototype for this function is in 'stdlib.h'.

```
void free (void *ptr) Function
```

The free function deallocates the block of memory pointed at by ptr.

```
void cfree (void *ptr) Function
```

This function does the same thing as free. It's provided for backward compatibility with SunOS; you should use free instead.

Freeing a block alters the contents of the block. Do not expect to find any data (such as a pointer to the next block in a chain of blocks) in the block after freeing it. Copy whatever you need out of the block before freeing it! Here is an example of the proper way to free all the blocks in a chain, and the strings that they point to:

```
struct chain
   {
struct chain *next;
char *name;
   }
void  free_chain (struct chain *chain)
  {
while (chain != 0)
```

```
  {
struct chain *next = chain->next;
free (chain->name);
free (chain);
chain = next;
 }
 }
```

Occasionally, free can actually return memory to the operating system and make the process smaller. Usually, all it can do is allow a later call to malloc to reuse the space. In the meantime, the space remains in your program as part of a free-list used internally by malloc.

There is no point in freeing blocks at the end of a program, because all of the program's space is given back to the system when the process terminates.

Other related Facilities

```
void * realloc (void *ptr, size_t newsize) Function
```

The realloc function changes the size of the block whose address is ptr to be newsize.

```
void * calloc (size_t count, size_t eltsize) Function
```

This function allocates a block long enough to contain a vector of count elements, each of size eltsize. Its contents are cleared to zero before calloc returns.

Conclusion

This unit provided an overview of the GNU C Library. A number of issues were presented including the library Standards and Portability, the basics of using the library, and the case of C library functions for allocation of storage for program data.

Assessment

If no more space is available, malloc returns a null pointer. You should check the value of every call to malloc. Instead, it is useful to write a subroutine that calls malloc and reports an error if the value is a null pointer, returning only if the value is nonzero. This function is conventionally called xmalloc (see Unit_2_Demos\xmalloc.c). Study this program.

Demonstrate use of malloc by writing a complete C program based on the function provided in Unit_2_Demos\savestring.c which copy a sequence of characters into a newly allocated nullterminated string.

Write a subroutine, called realloc, which takes care of the error message as xmalloc does for malloc(Refere to question No. 1 above)

Define the function calloc by using malloc.

Activity 2 - File, Directories and Links

Introduction

In this section we present the GNU C library's functions for manipulating files. These functions are concerned with operating on the files themselves, rather than on their contents. Among the facilities described in this section are functions for examining or modifying directories, functions for renaming and deleting files, and functions for examining and setting file attributes such as access permissions and modification times.

Activity Details

Examining or Modifying Directories

Manipulating the working directory

Each process has associated with it a directory, called its current working directory or simply working directory that is used in the resolution of relative file names. When you log in and begin a new session, your working directory is initially set to the home directory associated with your login account in the system user database. You can find any user's home directory using the getpwuid or getpwnam functions.

Users can change the working directory using shell commands like cd. The functions described in this section are the primitives used by those commands and by other programs for examining and changing the working directory.

Prototypes for these functions are declared in the header file `unistd.h'.

```
Function: char * getcwd (char *buffer, size_t size)
```

The getcwd function returns an absolute file name representing the current working directory, storing it in the character array buffer that you provide. The size argument is how you tell the system the allocation size of buffer.

The GNU library version of this function also permits you to specify a null pointer for the buffer argument. Then getcwd allocates a buffer automatically, as with malloc. If the size is greater than zero, then the buffer is that large; otherwise, the buffer is as large as necessary to hold the result.

The return value is buffer on success and a null pointer on failure. Table 1 defines the errno error conditions for the getcwd function.

Table 1. Error condition description for the getcwd function

Error condition	Meaning
EINVAL	The size argument is zero and buffer is not a null pointer.
ERANGE	The size argument is less than the length of the working directory name. You need to allocate a bigger array and try again.
EACCES	Permission to read or search a component of the file name was denied.

```
Function: char * getwd (char *buffer)
```

This is similar to getcwd, but has no way to specify the size of the buffer. The GNU library provides getwd only for backwards compatibility with BSD.

The buffer argument should be a pointer to an array at least PATH_MAX bytes long. In the GNU system there is no limit to the size of a file name, so this is not necessarily enough space to contain the directory name. That is why this function is deprecated.

```
Function: int chdir (const char *filename)
```

This function is used to set the process's working directory to filename.

The normal, successful return value from chdir is 0. A value of -1 is returned to indicate an error. The errno error conditions defined for this function are the usual file name syntax errors, plus ENOTDIR if the file filename is not a directory.

Accessing Directories

This section presents the facilities that let you read the contents of a directory file. This is useful if you want your program to list all the files in a directory, perhaps as part of a menu.

The opendir function opens a directory stream whose elements are directory entries. You use the readdir function on the directory stream to retrieve these entries, represented as struct dirent objects. The name of the file for each entry is stored in the d_name member of this structure. Table 2 presents what you find in a single directory entry, as you might obtain it from a directory stream. All the symbols are declared in the header file `dirent.h'.

Table 2: Format of a directory entry as obtained from a directory stream

Data Type: struct dirent	
This is a structure type used to return information about directory entries. It contains the following fields:	
Field name	Meaning
char d_name[]	This is the null-terminated file name component. This is the only field you can count on in all POSIX systems.
ino_t d_fileno	This is the file serial number. For BSD compatibility, you can also refer to this member as d_ino. In the GNU system and most POSIX systems, for most files this is the same as the st_ino member that stat will return for the file.
unsigned char d_namlen	This is the length of the file name, not including the terminating null character. Its type is unsigned char because that is the integer type of the appropriate size

<table>
<tr><td rowspan="12">unsigned char d_type</td><td colspan="2">This is the type of the file, possibly unknown. The following constants are defined for its value:</td></tr>
<tr><td>Constant</td><td>Meaning</td></tr>
<tr><td>DT_UNKNOWN</td><td>The type is unknown. On some systems this is the only value returned</td></tr>
<tr><td>DT_REG</td><td>A regular file.</td></tr>
<tr><td>DT_DIR</td><td>A directory.</td></tr>
<tr><td>DT_FIFO</td><td>A named pipe, or FIFO</td></tr>
<tr><td>DT_SOCK</td><td>A local-domain socket.</td></tr>
<tr><td>DT_CHR</td><td>A character device.</td></tr>
<tr><td>DT_BLK</td><td>A block device.</td></tr>
<tr><td colspan="2">This member is a BSD extension. Each value except DT_UNKNOWN corresponds to the file type bits in the st_mode member of struct statbuf. These two macros convert between d_type values and st_mode values:</td></tr>
<tr><td colspan="2">Function: int IFTODT (mode_t mode) : This returns the d_type value corresponding to mode.</td></tr>
<tr><td colspan="2">Function: mode_t DTTOIF (int dirtype) : This returns the st_mode value corresponding to dirtype.</td></tr>
<tr><td colspan="3">This structure may contain additional members in the future. When a file has multiple names, each name has its own directory entry. The only way you can tell that the directory entries belong to a single file is that they have the same value for the d_fileno field.</td></tr>
<tr><td colspan="3">File attributes such as size, modification times, and the like are part of the file itself, not any particular directory entry.</td></tr>
</table>

Opening a Directory Stream

This section describes how to open a directory stream. All the symbols are declared in the header file `dirent.h’.

Data Type: DIR

The DIR data type represents a directory stream.

You shouldn’t ever allocate objects of the struct dirent or DIR data types, since the directory access functions do that for you. Instead, you refer to these objects using the pointers returned by the function specified in Table 3.

Table 3: The opendir function

Function: DIR * opendir (const char *dirname) The opendir function opens and returns a directory stream for reading the directory whose file name is dirname. The stream has type DIR *. If unsuccessful, opendir returns a null pointer. In addition to the usual file name syntax errors, the following errno error conditions are defined for this function:	
Error condition	Meaning
EACCES	Read permission is denied for the directory named by dirname.
EMFILE	The process has too many files open.
ENFILE	The entire system, or perhaps the file system which contains the directory, cannot support any additional open files at the moment. (This problem cannot happen on the GNU system.)

The DIR type is typically implemented using a file descriptor, and the opendir function in terms of the open function. Directory streams and the underlying file descriptors are closed on exec.

Reading and Closing a Directory Stream

Table 4 presents the functions used to read directory entries from a directory stream, and close the stream when you are done with it. All the symbols are declared in the header file `dirent.h'.

Table 4: Functions used to read directory entries from a directory stream

Function: struct dirent * readdir (DIR *dirstream) This function reads the next entry from the directory. It normally returns a pointer to a structure containing information about the file. This structure is statically allocated and can be rewritten by a subsequent call. Portability Note: On some systems, readdir may not return entries for `.' and `..', even though these are always valid file names in any directory. If there are no more entries in the directory or an error is detected, readdir returns a null pointer. The following errno error conditions are defined for this function:		
	Error condition	Meaning
	EBADF	The dirstream argument is not valid.

Function: int closedir (DIR *dirstream) This function closes the directory stream dirstream. It returns 0 on success and -1 on failure. The following errno error conditions are defined for this function:		
The following errno error conditions are defined for this function:		
	Error condition	Meaning
	EBADF	The dirstream argument is not valid.

Random Access in a Directory Stream

Table 5 presents the functions used to reread parts of a directory that you have already read from an open directory stream. All the symbols are declared in the header file `dirent.h'.

Table 5: Functions used to reread parts of a directory that you have already read from an open directory stream.

Function: void rewinddir (DIR *dirstream) The rewinddir function is used to reinitialize the directory stream dirstream, so that if you call readdir it returns information about the first entry in the directory again. This function also notices if files have been added or removed to the directory since it was opened with opendir. (Entries for these files might or might not be returned by readdir if they were added or removed since you last called opendir or rewinddir.)
Function: off_t telldir (DIR *dirstream) The telldir function returns the file position of the directory stream dirstream. You can use this value with seekdir to restore the directory stream to that position.
Function: void seekdir (DIR *dirstream, off_t pos) The seekdir function sets the file position of the directory stream dirstream to pos. The value pos must be the result of a previous call to telldir on this particular stream; closing and reopening the directory can invalidate values returned by telldir.

Working with Multiple file names

In POSIX systems, one file can have many names at the same time. The additional name to the existing name (names) is called a hard link to the file. All of the names are equally real, and no one of them is preferred to the others. One file can have names in several directories, so the organization of the file system is not a strict hierarchy or tree. In most implementations, it is not possible to have hard links to the same file in multiple file systems.

The GNU system supports soft links or symbolic links. This is a kind of "file" that is essentially a pointer to another file name. Unlike hard links, symbolic links can be made to directories or across file systems with no restrictions. You can also make a symbolic link to a name which is not the name of any file (Opening this link will fail until a file by that name is created). Likewise, if the symbolic link points to an existing file which is later deleted, the symbolic link continues to point to the same file name even though the name no longer names any file.

The reason symbolic links work the way they do is that special things happen when you try to open the link. The open function realizes you have specified the name of a link, reads the file name contained in the link, and opens that file name instead. The stat function likewise operates on the file that the symbolic link points to, instead of on the link itself.

By contrast, other operations such as deleting or renaming the file operate on the link itself. The functions readlink and lstat also refrain from following symbolic links, because their purpose is to obtain information about the link. So does link, the function that makes a hard link--it makes a hard link to the symbolic link, which one rarely wants.

Creating Hard Links

To add a name to a file, use the link function. Creating a new link to a file does not copy the contents of the file; it simply makes a new name by which the file can be known, in addition to the file's existing name or names.

The function link reports an error if you try to make a hard link to the file from another file system when this cannot be done. The prototype for the link function is declared in the header file `unistd.h'. Table 6 provides the format of the function link.

Table 6: The format of the function link

<table>
<tr><td colspan="3">Function: int link (const char *oldname, const char *newname)</td></tr>
<tr><td colspan="3">The link function makes a new link to the existing file named by oldname, under the new name newname.</td></tr>
<tr><td colspan="3">This function returns a value of 0 if it is successful and -1 on failure. In addition to the usual file name syntax errors for both oldname and newname, the following errno error conditions are defined for this function:</td></tr>
<tr><td></td><td>Error condition</td><td>Meaning/description</td></tr>
<tr><td></td><td>EACCES</td><td>The directory in which the new link is to be written is not writable.</td></tr>
<tr><td></td><td>EEXIST</td><td>There is already a file named newname. If you want to replace this link with a new link, you must remove the old link explicitly first.</td></tr>
</table>

	EMLINK	There are already too many links to the file named by oldname. (The maximum number of links to a file is LINK_ MAX; Well-designed file systems never report this error, because they permit more links than your disk could possibly hold. However, you must still take account of the possibility of this error, as it could result from network access to a file system on another machine.
	ENOENT	The file named by oldname doesn't exist. You can't make a link to a file that doesn't exist
	ENOSPC	The directory or file system that would contain the new link is "full" and cannot be extended.
	EPERM	In the GNU system and some others, you cannot make links to directories. many systems allow only privileged users to do so. This error is used to report the problem.
	EROFS	The directory containing the new link can't be modified because it's on a read-only file system.
	EXDEV	The directory specified in newname is on a different file system than the existing file.

Creating Symbolic Links

The symlink and readlink functions are used to manupulate symbolic link in files. Prototypes for these functions are in `unistd.h'. . Table 7 describe the format of the function symlink.

Table 7: The description of the function symlink

Function: int symlink (const char *oldname, const char *newname) The symlink function makes a symbolic link to oldname named newname. The normal return value from symlink is 0. A return value of -1 indicates an error. In addition to the usual file name syntax errors, the following errno error conditions are defined for this function:		
	Error condition	Meaning/description
	EEXIST	There is already an existing file named newname.
	EROFS	The file newname would exist on a read-only file system.
	ENOSPC	The directory or file system cannot be extended to make the new link.
	EIO	A hardware error occurred while reading or writing data on the disk.

Function: int readlink (const char *filename, char *buffer, size_t size)

The readlink function gets the value of the symbolic link filename. The file name that the link points to is copied into buffer. This file name string is not null-terminated; readlink normally returns the number of characters copied. The size argument specifies the maximum number of characters to copy, usually the allocation size of buffer.

A value of -1 is returned in case of error. In addition to the usual file name syntax errors, the following errno error conditions are defined for this function:

	Error condition	Meaning/description
	EINVAL	The named file is not a symbolic link.
	EIO	A hardware error occurred while reading or writing data on the disk.

Conclusion

In this activity we describe the GNU C library's functions for manipulating files and directories. Specifically, functions for examining or modifying directories and working with multiple file names were presented.

Assessment

Read the provided code examples below and apply them to implement a working program

i. Here is an example showing how you could implement the behavior of GNU's getcwd (NULL, 0) using only the standard behavior of getcwd:

```
char *
gnu_getcwd ()
{
  int size = 100;
  char *buffer = (char *) xmalloc (size);

  while (1)
    {
      char *value = getcwd (buffer, size);
      if (value != 0)
        return buffer;
      size *= 2;
      free (buffer);
      buffer = (char *) xmalloc (size);
    }
}
```

Note: See section Activity 1.1 above, for information about xmalloc, which is not a library function but is a customary name used in most GNU software.

ii. Here's a simple program that prints the names of the files in the current working directory:

```
#include <stddef.h>

#include <stdio.h>

#include <sys/types.h>

#include <dirent.h>


int

main (void)

{

  DIR *dp;

  struct dirent *ep;

dp = opendir ("./");

  if (dp != NULL)

    {

      while (ep = readdir (dp))

        puts (ep->d_name);

      (void) closedir (dp);

    }

  else

    puts ("Couldn't open the directory.");


  return 0;

}
```

The order in which files appear in a directory tends to be fairly random. A more useful program would sort the entries (perhaps by alphabetizing them) before printing them.

iii. If the return value equals size, you cannot tell whether or not there was room to return the entire name. So make a bigger buffer and call readlink again. Here is an example:

```
char *
readlink_malloc (char *filename)
{
  int size = 100;

  while (1)
    {
      char *buffer = (char *) xmalloc (size);
      int nchars = readlink (filename, buffer, size);
      if (nchars < size)
        return buffer;
      free (buffer);
      size *= 2;
    }
}
```

Activity 3 - File input/output system calls

Introduction

C programmers on GNU/LINUX have two sets of input/output functions at their disposal. The standard C library provides I/O functions: printf, fopen, and so on. The Linux kernel itself provides another set of I/O operations that operate at a lower level than the C library functions.

Because this module is for people who already know the C language, we'll assume that you have encountered and know how to use the C library I/O functions. Often there are good reasons to use Linux's low-level I/O functions. Many of these are kernel system calls, and provide the most direct access to underlying system capabilities that is available to application programs. In fact, the standard C library I/O routines are implemented on top of the Linux low-level I/O system calls. Using the latter is usually the most efficient way to perform input and output operations—and is sometimes more convenient, too.

Activity Details

Reading and Writing Data

The first I/O function you likely encountered when you first learned the C language was printf. This formats a text string and then prints it to standard output. The generalized version, fprintf, can print the text to a stream other than standard output. A stream is represented by a FILE* pointer. You obtain a FILE* pointer by opening a file with fopen. When you're done, you can close it with fclose. In addition to fprintf, you can use such functions as fputc, fputs, and fwrite to write data to the stream, or fscanf, fgetc, fgets, and fread to read data.

With the Linux low-level I/O operations, you use a handle called a file descriptor instead of a FILE* pointer. A file descriptor is an integer value that refers to a particular instance of an open file in a single process. It can be open for reading, for writing, or for both reading and writing. A file descriptor doesn't have to refer to an open file; it can represent a connection with another system component that is capable of sending or receiving data. For example, a connection to a hardware device is represented by a file descriptor, as is an open socket or one end of a pipe. Include the header files <fcntl.h>, <sys/types.h>, <sys/stat.h>, and <unistd.h> if you use any of the low-level I/O functions described here.

Opening a File

To open a file and produce a file descriptor that can access that file, use the open call. It takes as arguments the path name of the file to open, as a character string, and flags specifying how to open it. You can use open to create a new file; if you do, pass a third argument that specifies the access permissions to set for the new file. If the second argument is O_RDONLY, the file is opened for reading only; an error will result if you subsequently try to write to the resulting file descriptor. Similarly, O_WRONLY causes the file descriptor to be write-only. Specifying O_RDWR produces a file descriptor that can be used both for reading and for writing. Note that not all files may be opened in all three modes. For instance, the permissions on a file might forbid a particular process from opening it for reading or for writing; a file on a read-only device such as a CD-ROM drive may not be opened for writing.

You can specify additional options by using the bitwise or of this value with one or more flags. These are the most commonly used values:

- Specify O_TRUNC to truncate the opened file, if it previously existed. Data written to the file descriptor will replace previous contents of the file.
- Specify O_APPEND to append to an existing file. Data written to the file descriptor will be added to the end of the file.
- Specify O_CREAT to create a new file. If the filename that you provide to open does not exist, a new file will be created, provided that the directory containing it exists and that the process has permission to create files in that directory. If the file already exists, it is opened instead.
- Specify O_EXCL with O_CREAT to force creation of a new file. If the file already exists, the open call will fail.

If you call open with O_CREAT, provide an additional third argument specifying the permissions for the new file. For example, the program given in question No. 1 of this activity assessment section creates a new file with the filename specified on the command line. It uses the O_EXCL flag with open, so if the file already exists, an error occurs. The new file is given read and write permissions for the owner and owning group, and read permissions only for others. (If your umask is set to a nonzero value, the actual permissions may be more restrictive.)

Closing File Descriptors

When you're done with a file descriptor, close it with close. In some cases, it's not necessary to call close explicitly because Linux closes all open file descriptors when a process terminates (that is, when the program ends). Of course, once you close a file descriptor, you should no longer use it. Closing a file descriptor may cause Linux to take a particular action, depending on the nature of the file descriptor. For example, when you close a file descriptor for a network socket, Linux closes the network connection between the two computers communicating through the socket.

Linux limits the number of open file descriptors that a process may have open at a time. Open file descriptors use kernel resources, so it's good to close file descriptors when you're done with them. A typical limit is 1,024 file descriptors per process. You can adjust this limit with the setrlimit system call.

Writing Data

Write data to a file descriptor using the write call. Provide the file descriptor, a pointer to a buffer of data, and the number of bytes to write. The file descriptor must be open for writing. The data written to the file need not be a character string; write copies arbitrary bytes from the buffer to the file descriptor. The program given in question No. 2 of this activity assessment section appends the current time to the file specified on the command line. If the file doesn't exist, it is created. This program also uses the time, localtime, and asctime functions to obtain and format the current time; check out their respective man pages for more information.

The write call returns the number of bytes that were actually written or -1 if an error occurred. For certain kinds of file descriptors, the number of bytes actually written may be less than the number of bytes requested. In this case, it's up to you to call write again to write the rest of the data. The function given in question No. 3 of this activity assessment section demonstrates how you might do this. Note that for some applications, you may have to check for special conditions in the middle of the writing operation. For example, if you're writing to a network socket, you'll have to augment this function to detect whether the network connection was closed in the middle of the write operation, and if it has, to react appropriately.

Reading Data

The corresponding call for reading data is read. Like write, it takes a file descriptor, a pointer to a buffer, and a count. The count specifies how many bytes are read from the file descriptor into the buffer. The call to read returns -1 on error or the number of bytes actually read. This may be smaller than the number of bytes requested, for example, if there aren't enough bytes left in the file.

The program given in question No. 4 of this activity assessment section provides a demonstration of read. The program prints a hexadecimal dump of the contents of the file specified on the command line. Each line displays the offset in the file and the next 16 bytes. It's shown printing out a dump of its own executable file.

Moving Around a File

A file descriptor remembers its position in a file. As you read from or write to the file descriptor, its position advances corresponding to the number of bytes you read or write. Sometimes, however, you'll need to move around a file without reading or writing data. For instance, you might want to write over the middle of a file without modifying the beginning, or you might want to jump back to the beginning of a file and re-read it without reopening it.The lseek call enables you to reposition a file descriptor in a file. Pass it the file descriptor and two additional arguments specifying the new position.

- If the third argument is SEEK_SET, lseek interprets the second argument as a position, in bytes, from the start of the file.
- If the third argument is SEEK_CUR, lseek interprets the second argument as an offset, which may be positive or negative, from the current position.
- If the third argument is SEEK_END, lseek interprets the second argument as an offset from the end of the file. A positive value indicates a position beyond the end of the file.

The call to lseek returns the new position, as an offset from the beginning of the file.

The type of the offset is off_t. If an error occurs, lseek returns -1.You can't use lseek with some types of file descriptors, such as socket file descriptors. If you want to find the position of a file descriptor in a file without changing it, specify a 0 offset from the current position—for example:

```
off_t position = lseek (file_descriptor, 0, SEEK_CUR);
```

Linux enables you to use lseek to position a file descriptor beyond the end of the file. Normally, if a file descriptor is positioned at the end of a file and you write to the file descriptor, Linux automatically expands the file to make room for the new data. If you position a file descriptor beyond the end of a file and then write to it, Linux first expands the file to accommodate the "gap" that you created with the lseek operation and then writes to the end of it. This gap, however, does not actually occupy space on the disk; instead, Linux just makes a note of how long it is.

If you later try to read from the file, it appears to your program that the gap is filled with 0 bytes. Using this behaviour of lseek, it's possible to create extremely large files that occupy almost no disk space. The program lseek-huge given in question No. 5 of this activity assessment section does this. It takes as command-line arguments a filename and a target file size, in megabytes. The program opens a new file, advances past the end of the file using lseek, and then writes a single 0 byte before closing the file.

Note that these magic gaps in files are a special feature of the ext2 file system that's typically used for GNU/Linux disks. If you try to use lseek-huge to create a file on some other type of file system, such as the fat or vfat file systems used to mount DOS and Windows partitions, you'll find that the resulting file does actually occupy the full amount of disk space. Linux does not permit you to rewind before the start of a file with lseek.

Extracting file information

Using open and read, you can extract the contents of a file. But how about other file information? For instance, invoking ls -l displays, for the files in the current directory, such information as the file size, the last modification time, permissions, and the owner. The stat call is used to obtain this information about a file.

Call stat with the path to the file you're interested in and a pointer to a variable of type struct stat. If the call to stat is successful, it returns 0 and fills in the fields of the structure with information about that file; otherwise, it returns -1.

These are the most useful fields in struct stat:

- st_mode contains the file's access permissions.
- In addition to the access permissions, the st_mode field encodes the type of the file in higher-order bits. See the text immediately following this bulleted list for instructions on decoding this information.
- st_uid and st_gid contain the IDs of the user and group, respectively, to which the file belongs.
- st_size contains the file size, in bytes.
- st_atime contains the time when this file was last accessed (read or written).
- st_mtime contains the time when this file was last modified.

These macros check the value of the st_mode field value to figure out what kind of file you've invoked stat on. A macro evaluates to true if the file is of that type.

```
S_ISBLK (mode) block device

S_ISCHR (mode) character device

S_ISDIR (mode) directory

S_ISFIFO (mode) fifo (named pipe)

S_ISLNK (mode) symbolic link

S_ISREG (mode) regular file

S_ISSOCK (mode) socket
```

The st_dev field contains the major and minor device number of the hardware device on which this file resides. The major device number is shifted left 8 bits; the minor device number occupies the least significant 8 bits. The st_ino field contains the inode number of this file. This locates the file in the file system.

If you call stat on a symbolic link, stat follows the link and you can obtain the information about the file that the link points to, not about the symbolic link itself.

This implies that S_ISLNK will never be true for the result of stat. Use the lstat function if you don't want to follow symbolic links; this function obtains information about the link itself rather than the link's target. If you call lstat on a file that isn't a symbolic link, it is equivalent to stat. Calling stat on a broken link (a link that points to a nonexistent or inaccessible target) results in an error, while calling lstat on such a link does not.

If you already have a file open for reading or writing, call fstat instead of stat. This takes a file descriptor as its first argument instead of a path. A program given in question No. 1 of this activity assessment section presents a function that allocates a buffer large enough to hold the contents of a file and then reads the file into the buffer. The function uses fstat to determine the size of the buffer that it needs to allocate and also to check that the file is indeed a regular file.

Vector Reads and Writes

The write call takes as arguments a pointer to the start of a buffer of data and the length of that buffer. It writes a contiguous region of memory to the file descriptor.

However, a program often will need to write several items of data, each residing at a different part of memory. To use write, the program either will have to copy the items into a single memory region, which obviously makes inefficient use of CPU cycles and memory, or will have to make multiple calls to write.

For some applications, multiple calls to write are inefficient or undesirable. For example, when writing to a network socket, two calls to write may cause two packets to be sent across the network, whereas the same data could be sent in a single packet if a single call to write were possible.

The writev call enables you to write multiple discontiguous regions of memory

to a file descriptor in a single operation. This is called a vector write. The cost of using writev is that you must set up a data structure specifying the start and length of each region of memory. This data structure is an array of struct iovec elements. Each element specifies one region of memory to write; the fields iov_base and iov_len specify the address of the start of the region and the length of the region, respectively.

If you know ahead of time how many regions you'll need, you can simply declare a struct iovec array variable; if the number of regions can vary, you must allocate the array dynamically. Call writev passing a file descriptor to write to, the struct iovec array, and the number of elements in the array. The return value is the total number of bytes written.

The program provided in question No. 1 of this activity assessment section writes its command-line arguments to a file using a single writev call. The first argument is the name of the file; the second and subsequent arguments are written to the file of that name, one on each line. The program allocates an array of struct iovec elements that is twice as long as the number of arguments it is writing—for each argument it writes the text of the argument itself as well as a new line character. Because we don't know the number of arguments in advance, the array is allocated using malloc.

Linux provides a corresponding function readv that reads in a single operation

into multiple discontiguous regions of memory. Similar to writev, an array of

struct iovec elements specifies the memory regions into which the data will be read from the file descriptor.

Relation to Standard C Library I/O Functions

We mentioned earlier that the standard C library I/O functions are implemented on top of these low-level I/O functions. Sometimes, though, it's handy to use standard library functions with file descriptors, or to use low-level I/O functions on a standard library FILE* stream. GNU/Linux enables you to do both.

Activity Details

If you've opened a file using fopen, you can obtain the underlying file descriptor using the fileno function. This takes a FILE* argument and returns the file descriptor. For example, to open a file with the standard library fopen call but write to it with writev, you could use this code:

```
FILE* stream = fopen (filename, "w");

int file_descriptor = fileno (stream);

writev (file_descriptor, vector, vector_length);
```

Note that stream and file_descriptor correspond to the same opened file. If you call this line, you may no longer write to file_descriptor:

```
fclose (stream);
```

Similarly, if you call this line, you may no longer write to stream:

```
close (file_descriptor);
```

To go the other way, from a file descriptor to a stream, use the fdopen function. This constructs a FILE* stream pointer corresponding to a file descriptor. The fdopen function takes a file descriptor argument and a string argument specifying the mode in which to create the stream. The syntax of the mode argument is the same as that of the second argument to fopen, and it must be compatible with the file descriptor. For example, specify a mode of r for a read file descriptor or w for a write file descriptor. As with fileno, the stream and file descriptor refer to the same open file, so if you close one, you may not subsequently use the other.

Conclusion

This activity presented how to perform the following I/O operations using low level operations (system calls):

- Open a file and create a file descriptor
- Close file descriptors
- Write data to file descriptors
- Read data from file descriptors
- Moving around a file on a file descriptor

Further, the stat system call was discussed to provide a means to extract other information associated with a file, along with the writev call used to write multiple discontiguous regions of memory to a file descriptor in a single operation. It was noted that in a similar way, the corresponding function readv can be used to read in a single operation into multiple discontiguous regions of memory.

Finally, the similarity and the conversion from the low level I/O calls to the C Library I/O functions and vice versa was demonstrated.

Assessment

Read the program "write-all" provided below and give a detailed explanation of what goes on in the program.

```
//write-all.c

ssize_t write_all (int fd, const void* buffer, size_t count)

{

size_t left_to_write = count;

while (left_to_write > 0) {

size_t written = write (fd, buffer, count);

if (written == -1)

/* An error occurred; bail. */
```

```
return -1;
else
/* Keep count of how much more we need to write. */
left_to_write -= written;
}
/* We should have written no more than COUNT bytes! */
assert (left_to_write == 0);
/* The number of bytes written is exactly COUNT. */
return count;
}
```

Read the program "read-file.c" provided below and give a detailed explanation of what goes on in the program.

```
//read-file.c - Read a File into a Buffer
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
char* read_file (const char* filename, size_t* length)
{
int fd;
struct stat file_info;
char* buffer;
/* Open the file. */
fd = open (filename, O_RDONLY);
/* Get information about the file. */
fstat (fd, &file_info);
*length = file_info.st_size;
/* Make sure the file is an ordinary file. */
```

```
if (!S_ISREG (file_info.st_mode)) {

/* It's not, so give up. */

close (fd);

return NULL;

}

buffer = (char*) malloc (*length);

/* Read the file into the buffer. */

read (fd, buffer, *length);

/* Finish up. */

close (fd);

return buffer;

}
```

Unit Summary

In this unit we provided an overview of the GNU C Library, along with a number of issues pertaining the GNU C Library including the library Standards and Portability, the basics of using the library, and the case of C library functions for allocation of storage for program data.

In addition we describe the GNU C library's functions for manipulating files and directories. Specifically, the functions for examining or modifying directories and working with multiple file names were presented.

Lastly, the unit discussed file input/output system calls in GNU/LINUX platform. It explored calls for reading and writing data, calls for extracting file information, calls for writing multiple discontiguous regions of memory to a file descriptor in a single operation or reading in a single operation into multiple discontiguous regions of memory, and highlighted the kernel's I/O calls relation to Standard C Library I/O functions.

Unit Assessment

Check your understanding!

Inter-mediate Programming Exercises

Instructions

See 1 Lab 1: Question 7w

Grading Scheme

As guided by the offering Institution Grading Regulations

Answers

mailto:njulumi@gmail.com

Unit Readings and Other Resources

1. Mark Mitchell, Jeffrey Oldham, and Alex Samuel; Advanced Linux Programming; Copyright © 2001 by New Riders Publishing; FIRST EDITION: June, 2001
2. http://www.acm.uiuc.edu/webmonkeys/book/c_guide/: The C Library Reference Guide
3. http://www.delorie.com/gnu/docs/glibc/libc_toc.html: The GNU C Library

Unit 2.Shell Programming and Embedding Assembly in C

Unit Introduction

The Kernel is the heart of an Operating System (OS). It manages resource, which are the facilities available in the OS, such as the facility to store data, print data on printer, memory, file management, etc. Kernel decides who will use this resource, for how long and when. It runs your programs (or set up to execute binary files). The kernel acts as an intermediary between the computer hardware and various programs/application/shell as shown in Figure 1.

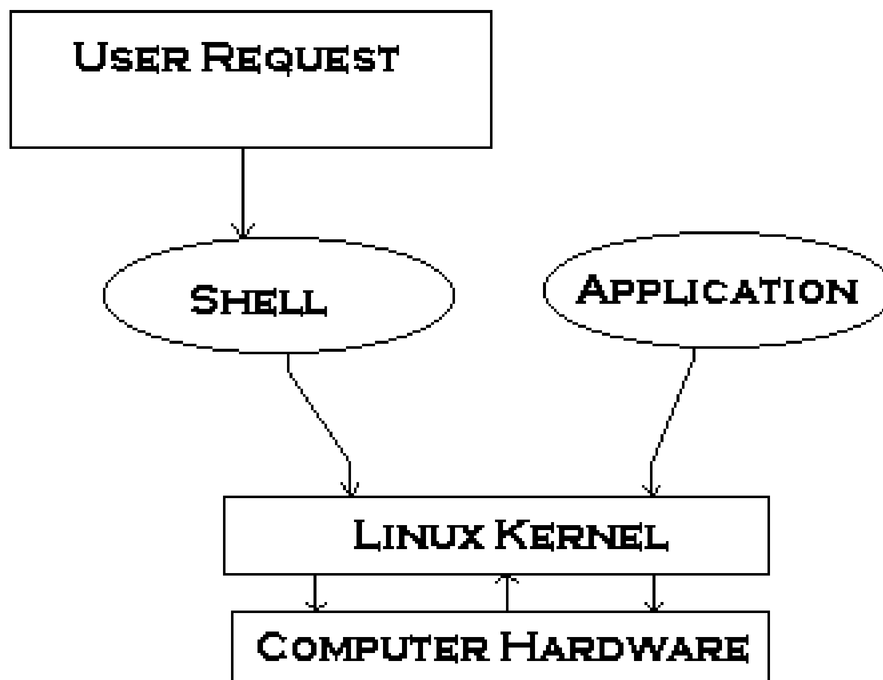

Figure 1: Kernel mediates Hardware to various users

The kernel is a Memory resident portion of the OS. It performs the following task :-

- I/O management
- Process management
- Device management
- File management
- Memory management

The computer hardware understands the language of 0's and 1's called binary language. In early days of computing, instructions to the computer were provided using binary language, which is difficult to read and write.

Operating Systems provides a special program called Shell. Shell accepts instruction or commands in English (mostly) and if its a valid command, it is pass to kernel. Shell is a user program or the OS environment provided for user interaction. Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file. Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Programming languages also provide a way to instruct the hardware to perform some user task through application development. Programs written in Higher-level languages such as C and C++ run on nearly all architectures yield higher productivity when writing and maintaining code. For occasions when programmers need to use assembly instructions in their programs, the GNU Compiler Collection permits programmers to add architecture-dependent assembly language instructions to their programs.

Assembly language instructions are architecture-dependent, so, for example, programs using x86 instructions cannot be compiled on PowerPC computers. To use them, you'll require a facility in the assembly language for your architecture. However, inline assembly statements permit you to access hardware directly and can also yield faster code. An asm instruction allows you to insert assembly instructions into C and C++ programs. Observe that unlike ordinary assembly code instructions, asm statements permit you to specify input and output operands using C syntax

Unit Objectives

Upon completion of this unit you should be able to:

- Define the shell and assembly inlining concepts
- Make use of the shell to perform different user/OS task
- Explain the process of writing shell programs and shell programming constructs
- Explain the techniques of mixing assembly instructions with C code
- Make use of shell programming/shell scripting for automating different system tasks.
- Write simple C programs embedded with assembly instructions to speed up instructions processing

Key Terms

Shell:A command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Shell program/Script: The shell interprets user commands, which are either directly entered by the user, or which can be read from a file called the shell script or shell program. Shell scripts are interpreted, not compiled. The shell reads commands from the script line per line and searches for those commands on the system, while a compiler converts a program into machine readable form, an executable file - which may then be used in a shell script.

Assembly language: Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems

High-level language:A Programming Language such as C/C++ that supports system development at a high level of abstraction, thereby freeing the developer from keeping in his head lots of details that are irrelevant to the problem at hand.

Kernel: The kernel is the central module of an operating system. It is the part of the operating system that loads first, and it remains in main memory

Operating System: Software that manages computer hardware and software resources and provides common services for computer programs

Learning Activities

Activity 1 - Shell Basics

Introduction

Several shells are available in Linux/Unix OS including BASH, CSH, KSH and TCSH as detailed in Table 1. To find all available shells in your system type following command:

$ cat /etc/shells. Note that each shell does the same job, but each understand different command syntax and provides different built-in functions.

Table 1: Some common shells in Linux/Unix OS environment

Shell Name	Developed by	Where	Remark
BASH (Bourne-Again SHell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C SHell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn SHell)	David Korn	AT & T Bell Labs	--
TCSH	See the man page. Type $ man tcsh	--	TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH).

MS-DOS provides a shell named COMMAND.COM which is also used for same purpose, but it's not as powerful as the Linux/Unix shells!

Any shell reads command from user (via Keyboard or Mouse) and tells OS what users want. If we are giving commands from keyboard it is called command line interface (Usually in-front of $ prompt for Linux/Unix OS. The prompt symbol, however, depend upon your shell and environment that is set by System Administrator, therefore you may get different prompt).

To find the current shell in Linux/Unix type following command: $ echo $SHELL

Normally shells are interactive. That means a shell accept command from user (via keyboard) and execute it. But if you need to provide a sequence of two or more commands, you can store this sequence of commands to a text file and tell the shell to execute the text file instead of entering one command at a time. This is known as shell program/script. A Shell program/script is a series of command written in plain text file. Shell scripts are just like batch files in MS-DOS but have more power than the MS-DOS batch file.

A Shell Script can take input from user, file and output them on screen. It is Useful to create own commands and save time, automate some task of day today life, also System Administration part can be also automated.

In this unit we will use the bash shell.

Activity Details

What does the shell do

In Linux/Unix, the shell is separate from the OS (change look and feel). The shell reads and executes commands

- Some handled by the shell itself (pwd, echo,...)
- some are programs stored in some directory (look in directories in PATH). Start a subshell to execute these

The shell provides support for better interaction with the computer/OS (command history, editing, configuration). The shell also supports scripting (is a programming language)

Executing a Command

After reading a command, the shell may do some processing (see wildcards, etc in the syntax description that follows), then it must find a program to execute the command. Some commands are executed directly by the shell. Other commands are executed by separate programs. These are found by looking in a list of directories for programs with the appropriate name. The shell searches directories in the PATH variable. A hash table is used to make the search fast. You can add new commands simply by adding new programs (a program can be any executable file including scripts – refer to Linux/Unix permissions) to directories in the PATH. You can modify/add directories in the PATH.

Finding out about Commands

type – tells you if a command is a built-in, an alias, or a program (external to the shell)

which – tells in which directory a utility is located

help – displays information about built-in commands (it is a builtin itself)

info bash – a good place to read about the BASH shell

For example:

```
$ which echo

/bin/echo

$ type -a echo

echo is a shell builtin

echo is /bin/echo

$ help echo # help provides information on builtin commands

echo: echo [-neE] [arg ...]
```

Output the ARGs. If -n is specified, the trailing newline is suppressed. If the -e option is given, interpretation of the following backslash-escaped characters is turned on:

```
\a alert (bell)
\b backspace
\c suppress trailing newline
\E escape character
\f form feed
\n new line
\r carriage return
\t horizontal tab
\v vertical tab
\\ backslash
```

\num the character whose ASCII code is NUM (octal).

You can explicitly turn off the interpretation of the above characters with the -E option.

Initialisation Files

Commands and variables placed in initialization files are read when shell is started. If variables are placed in system-wide init files, they are made available to every shell. Customizations must be exported to be available. Commands and aliases cannot be exported so must be placed in user-specific init files.

i. `/etc/profile - System-Wide Initialization File`

Read first by shell. The tasks it accomplishes includes:

- Sets and exports variables: PATH, TERM, PS1, etc.
- Displays contents of /etc/motd (msg of the day)
- Sets default file creation permissions (umask)

ii. User-specific initialisation

If the shell is a login shell, it looks for one of the following files (in order)

- ~/.bash_profile
- ~/.bash_login
- ~/.profile

If it is a non-login interactive shell, it reads the file

- ~/.bashrc

iii. Other customizations

- PS1 – prompt
- PATH – add to the search path
- Set shell options
  - o noclobber
  - o ignoreeof
  - o command-line editing option (vi or emacs)
  - o See the .bashrc file for examples.

Shell Variables

The shell keeps track of a set of parameter names and values. Some of these parameters determine the behavior of the shell. We can access these variables to

- set new values for some to customize the shell.
- find out the value of some to help accomplish a task.

Examples of shell variables in sh / ksh / bash:

Shell variable	Purpose
PWD	current working directory
PATH	list of places to look for commands
HOME	home directory of user
MAIL	where your email is stored
TERM	what kind of terminal you have
HISTFILE	where your command history is saved

Displaying Shell Variables

Prefix the name of a shell variable with "$" to dereference. The echo command will do:

```
echo $HOME

echo $PATH
```

You can use these variables on any command line:

```
ls -al $HOME
```

Setting Shell Variables

You can change the value of a shell variable with an assignment command (this is a shell builtin command):

```
HOME=/etc

PATH=/usr/bin:/usr/etc:/sbin

NEWVAR="avu avu avu"

Note the lack of spaces around the '='
```

export

Used to export a variable and make it available to subshells. Value passed in and changes made to a value in the subshell do not persist in the caller. Subshell is created whenever a script or a program is run and inherits parent shell's exported variables.

PATH

Each time you give the shell a command line it does the following:

- Checks to see if the command is a shell built-in.
- If not - tries to find a program whose name (the filename) is the same as the command.

The PATH variable tells the shell where to look for programs (non built-in commands).

For example:

```
$ echo $PATH

~/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games:.

$ PATH=${PATH}:/home/avu

$ echo $PATH

~/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games:.:/home/avu
```

The PATH is a list of ":" delimited directories. The PATH is a list and a search order. You can add stuff to your PATH by changing the shell startup file.

Notes about PATH:

If you do not have "." in your PATH, commands in your current directory will not be found. You may or may not want to add "." to your PATH. If you do not and want to execute a command in the current directory

./command

The PATH is searched sequentially; the first matching program is executed. Beware of naming your executables test as there is another program called test and this may be executed when you enter

test

set command

The set (shell builtin) command with no parameters will print out a list of all the shell variables.

Some common options:

- noclobber – Keeps mv, cp, redirection from deleting an existing file
- -o vi (or -o emacs) – sets command-line editing mode
- ignoreeof – ctrl-D won't exit the shell

$PS1 and $PS2

The PS1 shell variable is your command line prompt. The PS2 shell variable is used as a prompt when the shell needs more input (in the middle of processing a command). By changing PS1 and/or PS2 you can change the prompt. Bash supports some fancy stuff in the prompt string:

```
\t is replace by the current time

\w is replaced by the current directory

\h is replaced by the hostname

\u is replaced by the username

\n is replaced by a newline
```

Example bash prompt:

```
~ echo $PS1

======= [\h] - \t =======\n\w
```

You can change your prompt by changing PS1:

```
PS1="Yes Master? "
```

To make changes stick, i.e., if you want to tell the shell (bash) to always use the prompt "Yes Master ?", you need to store the change in a shell startup file. For bash - change the file ~/.bashrc.

i. SHELL

The SHELL variable holds the path of the login shell.

ii. HOME

The HOME variable contains the PATH to your home directory. When you use cd command with no arguments, the command uses the value of the HOME variable as the argument.

echo $HOME

/home/avu

A.Metacharacters

Shell metacharacters are characters that are handled specially by the shell. Below is an (incomplete) list of shell metacharacters:

- `>  >>  <  <<  |  Command redirection, pipe`
- `*  []  ?  {}  File specification`
- `;  &  ||  &&  ()  Command Sequencing`
- `",  ' Grouping Text`
  - o Double quotes and single quotes indicates that the enclosed text is to be treated as a unit, not as a group of words.

  `#  Commenting`

  - o Rest of line after these characters is ignored

- `\ (backslash) – the escape character`
  - o Indicates that the character immediately following the backslash is to be treated literally.To remove a file named "#bogus" you can use the backslash

  rm \#bogus

B.Aliases

Commands can be aliased (renamed) to alter their behavior. A common use of aliases is to add options to the default behavior of certain commands (e.g. ls, rm, mv, …). It is common practice to alias rm to prompt the user to make sure that the specified files should be removed. This is especially important since you can remove all files with (see wildcards) rm *. To see current aliases, use the alias command. This can also be used to set new aliases.

Create an alias using the builtin command alias name[=value]

- You should usually enclose value in quotes.
- no spaces around the '='
- Without a value, alias prints out the alias associated with name

For example :

```
$alias dir="ls"

$alias ls="ls -CF"

$dir

        Output same as for "ls -CF"
```

C.Variable Substitution

Shells support the use of variables. A variable is a name that is bound to a value. To set a variable (in Bash) just enter name=value (No spaces around '='). To see the settings of all variables just enter "set". To kill a variable: unset name.

When processing a command, variable substitution occurs. A variable in a command is flagged by a dollar sign prefix "$".

```
$ echo My shell is $SHELL
```

My shell is /bin/bash

echo writes out its arguments after substitution is performed.

Variable substitution will occur within double quotes

```
$ echo "My shell is $SHELL"
```

My shell is /usr/local/bin/tcsh

Substitution does not occur within single quotes.

```
'$ echo 'My shell is $SHELL'
```

My shell is $SHELL

When the usage of a variable name is not clear, enclose it within braces ${name}

```
$ prefix=cs333

$ suffix=.pdf

$ echo $prefix03$suffix

.pdf

$ echo ${prefix}03${suffix}

Cs33303.pdf
```

This occurs when constructing file names in script files.

Many programs use shell variables (also called environmental variables) to get configuration information.

For example:

- PRINTER is used by printing commands to determine the default printer.
- TERM is used by programs (e.g., vi, pine) to determine what type of terminal is being used.
- VISUAL is used by network news programs, etc., to determine what editor to use.

D.Command Substitution

Command substitution allows the output (stdout) of a command to replace the command name. There are two forms:

The original Bourne: `command`

The Bash (and Korn) extension: $(command)

The output of the command is substituted in:

```
$ echo $(ls)

    foo fee file?

$ echo "Today is $(date '+%A %d %B %Y')"

    Today is Thursday 30 September 2014
```

E.Strong quoting – Single quotes

Inhibits all substitution, and the special meaning of metacharacters:

```
$ echo '$USER is $USER'

$USER is $USER

$ echo 'today is `date`'

today is `date`

$ echo 'No background&'

No background&

$ echo 'I said, "radio!"'

I said, "radio !"
```

F.Weak quoting – double quotes

Allows command and variable substitution. Inhibits special meaning of all other metacharacters :

```
$ echo "My name is $USER &"

My name is avu &

$ echo "\$2.00 says `date`"

$2.00 says Sun Jan 15 01:43:32 EST 200
```

G.Command Execution

Sometimes we need to combine several commands. There are four formats for combining commands into one line: sequenced, grouped, chained, and conditional.

A sequence of commands can be entered on one line. Each command must be separated from its predecessor by semicolon. There is no direct relationship between the commands.

command1; command2; command3

Using grouped commands we can apply the same operation to the group. Commands are grouped by placing them into parentheses and are run in a subshell.

For example:

```
echo "Month" > file; cal 10 2000 >> file

(echo "Month" ; cal 10 2000 ) > file
```

With conditional commands we can combine two or more commands using conditional relationships AND (&&) and OR (ll). If we AND two commands, the second is executed

only if the first is successful. If we OR two commands, the second is executed only of the first fails.

For example :

```
cp file1 file2 && echo "Copy successful"

cp file1 file2 || echo "Copy failed"
```

Shell Syntax

A.Comments -

```
# This is a comment

ls # list the files in the current directory
```

B.Line continuation - \

$echo A long \

> line

C.Command separator - ;

```
You can list more than one command per line separated by ;

ls ; who
```

D.Pathname separator - /

```
cd /home/avu
```

E.Wildcards (globbing), and pathname expansion

```
*    - match any string (including empty)

?   - match any single character

[set]   - match characters listed in set (can be range)

[!set]  - match any character not given in set
```

For example:

```
ls *.c

ls *.?

ls *.[Hh][Tt][Ll]

ls [a-z]
```

F.File redirection and pipes

```
<  - redirect input from specified source

>  - redirect output to specified source

>> - redirect output and append to specified source

|  - pipe the output from one command to the input to the next
```

For example:

```
grep word < /usr/dict/words

ls > listing

ls >> listing

ls -l| wc -l
```

G.Stderr

Note that file redirection of standard output [stdout] does not include error messages, which go to standard error [stderr] (when you are in a terminal session, both stdout and stderr go to the screen; however, when you redirect stdout to a file, stderr still goes to the screen). stdout is designated by the file descriptor 1 and stderr by 2 (standard input is 0). To redirect standard error use 2>

```
ls filenothere > listing 2> error

ls filenothere 2>&1 > listing     # both stdout and stderr redirected
to listing
```

H.Background jobs

The & operator runs command in the background.

```
grep 'we.*'< /usr/word/dict > wewords &
```

This runs the grep command in the background – you immediately get a new prompt and can continue your work while the command is run.

jobs is a builtin command. Lists active jobs (stopped, or running in the background). Also see the command ps. kill will take a PID (see ps) or jobspec (see jobs)

Conclusion

In this activity we presented the basics of Unix/Linux shell to familiarize you with shell environment.

Assessment

Practise all the examples used in this activity.

Activity 2 – Shell Programming

Introduction

A.What is A Script?

A script is a small program that is executed by the shell. The script is a text file which will contain:

- Shell commands you normally use.
- Shell flow control constructs (e.g., if-then-else, etc.)
- A heavier use of variables than you normally would use from the command line.

B.Why Write Scripts?

Any task you do (by hand) more than twice should probably be wrapped into a script. Sequences of operations which you perform often can be placed into a script file and then executed like a single command. For example, renaming all files of the form

Cs333l*.ppt to cs333l*n.ppt requires a mv command for each file.

The Unix/Linux shells are very powerful, but there are some things that they do not do well.

C.Shell scripting - Why Not?

Resource-intensive tasks, especially where speed is a factor, complex applications, where structured programming is a necessity, mission-critical applications upon which you are betting the ranch, or the future of the company, situations where security is important, where you need to protect against hacking. Project consists of subcomponents with interlocking dependencies. Extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion). Need to generate or manipulate graphics or GUIs. Need direct access to system hardware. Need port or socket I/O. Need to use libraries or interface with legacy code.

Activity Details

Getting started with Scripting

A.Creating a Script

Let's create a shell script to give us information about the system. We create the script using a text editor. Let's call the file "status".

```
#!/bin/bash

uptime

users
```

Be sure to insert an "enter" after the last line before you exit the editor.

B.Running a Script

To execute the shell we can do

```
$ bash status

10:37 up 23 days, 23:54, 14 users,

load average …

afjhj billc …
```

We can also execute the file as a command if the appropriate execute access is granted.

```
$ ./status

bash: ./status: Permission denied

$ chmod +x status

$ ./status # Works correctly.
```

C.#! – "sha-bang"

Not needed if bash kicks off the script, but…

Shells look for "#!" at the very beginning of the file. It is followed by the program

(interpreter) that will read this file:

```
#!/bin/bash

#!/usr/bin/python

# this is a Python program. Bash  doesn't read it. Gives it right

# to the Python interpreter.
```

Conditional Expressions

To perform ifs and whiles we need to be able to construct conditional expressions. A conditional expression is one that evaluates to true or false depending on its operands. A process' return value of 0 is taken to be true ; any nonzero value is false.

A.test - Conditional Expressions

Actually test is a disk utility. [ ] is its shorthand. It provides for a great many tests and is available to all shells.

test expression

or

[expression] - Separate expression from brackets spaces

test returns an exit status of zero (success) if the expression evaluates to true. test uses a variety of operators. Unary file operators can test various file properties. Here are just a few:

```
-e True if file exists

-f True if file is a regular file

-d True if file is a directory

-w True if file exists and is writable

-O True if I own the file
```

For example :

```
if  [ -e ~avu/public_html ] ; then

                 echo "Avu has a public web directory"

fi
```

B.[] – file and string operators

- Binary file operators "file1 op file2"

```
-nt True is file1 is newer than file2

-ot True is file1 is older than file2

-ef True if file1 and file2 refer to the same inode
```

- Unary string operators "op string"

```
-z True if string is of zero length

-n True if string is not of zero length

-l Returns length of string
```

For example :

```
if [ -z "$myVar" ] ; then

echo "\$myVar has null length"

fi
```

C.[] – string operators

These compare lexical order

```
== != < > <= >=

Note, < > are file redirection. Escape them

For example :

if [ "abc" != "ABC" ] ; then

echo 'See. Case matters.' ; fi

if [ 12 \< 2 ] ; then

echo "12 is less than 2?" ; fi
```

D.[] – arithmetic operators

Work only for integers.

Binary operators:

```
-lt -gt -le -ge -eq -ne

For example :

if [ 2 -le 3 ] ; then ;echo "cool!" ; fi

x=5
```

```
if [ "$x" -ne 12 ] ; then
echo "Still cool" ; fi
```

E.[] – Logical Operators

Logical expression tools

- `! expression  - Logical not (i.e., changes sense of expression)`
- `e1 -a e2 True if both expressions are true.`
- `e1 -o e2 True if e1 or e2 is true.`
- `\( expression \) Works like normal parentheses for expressions; use spaces around the expression.`

For example :

```
test -e bin -a -d /bin is true
[ -e ~/.bashrc -a ! -d ~/.bashrc ] && echo
True
```

F.[[test]]

```
Bash added [[]] for more C-like usage:
if [[ -e ~/.bashrc && ! -d ~/.bashrc ]]
then
echo "Let's parse that puppy"
fi
if [[ -z "$myFile" || ! -r $myFile ]]
…
```

It's a built-in. Why sometimes quote $myFile, sometimes not (it's usually a good idea to do so)?

Arithmetic Expressions

Bash usually treats variables as strings. You can change that by using the arithmetic expansion syntax: ((arithmeticExpr)). The notation (()) is a shorthand for the let builtin statement.

```
$ x=1
$ x=x+1  # "x+1" is just a string
echo $x
x+1
Note, $[] is deprecated
```

```
$ x=1
$ x=$x+1 # still just a string
$ echo $x
1+1
```

Closer, but still not right.

```
$ x=1
$ (( x=x+1 ))
$ echo $x
2
```

Finally!

If statement

A.Sample : Basic conditional if .. then

```
#!/bin/bash
if [ "$1" = "foo" ] ; then
echo expression \
evaluated as true
fi
```

B.Sample : Basic conditional if .. then ... else

```
#!/bin/bash
if [ "$1" = "foo" ]
then
echo 'First argument is "foo"'
else
echo 'First arg is not "foo"'
fi
```

C.Sample: Conditionals with variables

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" == "$T2" ] ; then
```

```
echo expression evaluated as true
else
echo expression evaluated as false
fi
Always quote variables in scripts!
```

D.Checking return value of a command

```
if diff "$fileA" "$fileB" > /dev/null
then
echo "Files are identical"
else
echo "Files are different"
fi
```

Case statement

```
case $opt in
a ) echo "option a";;
b ) echo "option b";;
c ) echo "option c";;
\? ) echo \
 'usage: alice [-a] [-b] [-c] args...'
exit 1;;
esac
```

Special Variables

```
$# the number of arguments
$* all arguments
$@ all arguments (quoted individually)
$? return value of last command executed
$$ process id of shell
$HOME, $IFS, $PATH, $PS1, $PS2
```

Scripts and Arguments

Scripts can be started with parameters, just like commands

aScript arg1 arg2 …

The scripts can access these arguments through shell variables:

"$n" Is the value of the nth parameter. The command is parameter zero

"$#" Is the number of parameters entered.

"$*" Expands as a list of all the parameters entered except the command.

Let's quickly write a script to see this:

(this first line is a quick and dirty way to write a file)

```
$ cat > xx # cat reads from stdin if no file specified

echo $0

echo $#

echo $1 $2

echo $*

C-d  # Control-D is the end of file character.

$ chmod +x xx  #The file xx is now an executable shell script.

$ ./xx a b c #Execute script with three parameters

./xx

3

a b

a b c

$ xx #Execute script with no parameter

./xx

0
```

Unspecified parameters expand as empty strings (i.e.., as nothing)

Loops in Scripts

The for loop is a little bit different from other programming languages. Basically, it let's you iterate over a series of 'words' within a string. The while executes a piece of code if the control expression is true, and only stops when it is false (or a explicit break is found within the executed code. The until loop is almost equivalent to the while loop, except that the code is executed while the control expression evaluates to false.

A.For Loop

i.For loop : Example 1

```
$ for x in 1 2 a; do
> echo $x
> done
1
2
a
```

ii.For loop : Example 2

```
$ for x in *; do
> echo $x
> done
bin
mail
public_html
 …
```

iii.For loop : Example 3

```
#!/bin/bash
for i in $(cat list.txt) ; do
echo item: $i
done
```

iv.For loop : Example 4

```
#!/bin/bash

for (( i=0; i<10; ++i )) ; do

echo item: $i

done
```

B.while loop

i.while loop: Example 1

```
COUNTER=0

while [ $COUNTER -lt 10 ] ; do

echo The counter is $COUNTER

let COUNTER=COUNTER+1

done
```

ii.while loop: Example 2

```
COUNTER=0

while (( COUNTER < 10 )) ; do

echo The counter is $COUNTER

(( COUNTER = COUNTER+1 ))

Done
```

C.until loop

until loop: Example 1

```
#!/bin/bash

COUNTER=20

until [ $COUNTER -lt 10 ]

do

echo COUNTER $COUNTER

let COUNTER-=1

done
```

D.Example : using loops to renaming files

Goal: Rename all the files of the form cs333l*.ppt to cs333l*.exe where * should be 02, 03 … 10, using the for loop.

Solution 1:

```
for n in 02 03 04 05 06 07 08 09 10; do

mv cs2651$n.ppt cs2651$n.exe

done
```

Solution 2:

```
for (( n=2; n<10; ++i )) ; do

mv cs33310$n.ppt cs33310$n.exe

done

mv cs333110.ppt cs333110.exe

Renames cs333102.ppt to cs333102.exe, cs333103.ppt to cs333103.exe,
etc.
```

E.Loop Control statements

break - terminates the loop

continue - causes a jump to the next iteration of the loop

F.Debugging Tip

If you want to watch the commands actually being executed in a script file, insert the line

```
"set -x" in the script.


set -x

for n in *; do

echo $n

done
```

Will display the expanded command before executing it.

```
+ echo bin

bin

+ echo mail

mail

...
```

Functions

As in almost any programming language, you can use functions to group pieces of code in a more logical way or practice the divine art of recursion. Declaring a function is just a matter of writing function:

```
my_func { my_code }.
```

Calling a function is just like calling another program, you just write its name.

A.Variables scope example

```
#!/bin/bash

HELLO=Hello            #variable HELLO with program scope

function hello {

local HELLO=World       #variable HELLO with local scope

echo $HELLO

}

$ echo $HELLO

$ hello

$ echo $HELLO
```

B.Functions with parameters : Example

```
#!/bin/bash

function quit {

echo 'Goodbye!'

exit

}

function hello {

echo "Hello $1"

}

for name in Godfrey Justo;

do

hello $name

done

quit

Output:
```

```
Hello Godfrey

Hello Justo

Goodbye
```

C.Parameter Expansion

```
${parameter:-word} - Use Default Values.

${parameter:=word} - Assign Default Values.

${parameter:?word} - Display Error if Null or Unset.

${parameter:+word} - Use Alternate Value.
```

D.More Parameter Expansion

We can remove parts of a value:

<table>
<tr><th>Parameter expansion</th><th>Purpose</th></tr>
<tr><td>${param#pattern}</td><td rowspan="2">removes shortest (#) or longest (##) leading pattern, if there's a match</td></tr>
<tr><td>${param##pattern}</td></tr>
<tr><td>${param%pattern}</td><td rowspan="2">removes shortest(%) or longest (%) trailing pattern, if match</td></tr>
<tr><td>${param%%pattern}</td></tr>
</table>

pattern is expanded just as in pathname expansion (globbing) - *, ?, []

Furthermore we can

find the length of a string: echo ${#foo}

extract substrings: echo ${foo:2:3}

perform Regex search and replace.

For more details see the Bash manpage.

E.Parameter Expansion - Example

```
$ foo=j.i.c

$ echo ${foo#*.}

i.c

$ echo ${foo##*.}

c

$ echo ${foo%.*}

j.i

$ echo ${foo%%.*}

j
```

Conclusion

This activity presented the process of writing and executing shell scripts, language constructs and highlighted the different features found in Unix/Linux shell programming/scripting.

Assessment

1. Practise the examples used in this activity

Activity 3 - Inline Assembly Code

Introduction

Higher-level languages such as C and C++ run on nearly all architectures and yield higher productivity when writing and maintaining code. For occasions when programmers need to use assembly instructions in their programs, the GNU Compiler Collection permits programmers to add architecture-dependent assembly language instructions to their programs. GCC's inline assembly statements should not be used indiscriminately. Assembly language instructions are architecture-dependent, so, for example, programs using x86 instructions cannot be compiled on PowerPC computers. To use them, you'll require a facility in the assembly language for your architecture. However, inline assembly statements permit you to access hardware directly and can also yield faster code.

An asm instruction allows you to insert assembly instructions into C and C++ programs. For example, this instruction

```
asm (“fsin” : “=t” (answer) : “0” (angle));

is an x86-specific way of coding this C statement:

answer = sin (angle);
```

The expression sin (angle) is usually implemented as a function call into the math library, but if you specify the -O1 or higher optimization flag, GCC is smart enough to replace the function call with a single fsin assembly instruction.

Observe that unlike ordinary assembly code instructions, asm statements permit you to specify input and output operands using C syntax. To read more about the x86 instruction set, which we will use in this section, see http://developer.intel.com/design/pentiumii/manuals/ and http://www.x86-64.org/documentation.

Although asm statements can be abused, they allow your programs to access the computer hardware directly, and they can produce programs that execute quickly. You can use them when writing operating system code that directly needs to interact with hardware. For example, /usr/include/asm/io.h contains assembly instructions to access input/output ports directly.

The Linux source code file /usr/src/linux/arch/i386/kernel/process.s provides another example, using hlt in idle loop code. See other Linux source code files in /usr/src/linux/arch/ and /usr/src/linux/drivers/.

Assembly instructions can also speed the innermost loop of computer programs. For example, if the majority of a program's running time is computing the sine and cosine of the same angles, this innermost loop could be recoded using the fsincos x86 instructions (Algorithmic or data structure changes may be more effective in reducing a program's running time than using assembly instructions). See, for example, /usr/include/bits/mathinline.h, which wraps up into macros some inline assembly sequences that speed transcendental function computation.

You should use inline assembly to speed up code only as a last resort. Current compilers are quite sophisticated and know a lot about the details of the processors for which they generate code. Therefore, compilers can often choose code sequences that may seem unintuitive or roundabout but that actually execute faster than other instruction sequences. Unless you understand the instruction set and scheduling attributes of your target processor very well, you're probably better off letting the compiler's optimizers generate assembly code for you for most operations.

Occasionally, one or two assembly instructions can replace several lines of higher-level language code. For example, determining the position of the most significant nonzero bit of a nonzero integer using the C programming languages requires a loop or floating-point computations. Many architectures, including the x86, have a single assembly instruction (bsr) to compute this bit position.

Activity Details

Simple Inline Assembly

Here we introduce the syntax of asm assembler instructions with an x86 example to shift a value 8 bits to the right:

```
asm ("shrl $8, %0" : "=r" (answer) : "r" (operand) : "cc");
```

The keyword asm is followed by a parenthetic expression consisting of sections separated by colons. The first section contains an assembler instruction and its operands. In this example, shrl right-shifts the bits in its first operand. Its first operand is represented by %0. Its second operand is the immediate constant $8. The second section specifies the outputs. The instruction's one output will be placed in the C variable answer, which must be an lvalue. The string "=r" contains an equals sign indicating an output operand and an r indicating that answer is stored in a register.

The third section specifies the inputs. The C variable operand specifies the value to shift. The string "r" indicates that it is stored in a register but omits an equals sign because it is an input operand, not an output operand. The fourth section indicates that the instruction changes the value in the condition code cc register.

A.Converting an asm to Assembly Instructions

GCC's treatment of asm statements is very simple. It produces assembly instructions to deal with the asm's operands, and it replaces the asm statement with the instruction that you specify. It does not analyze the instruction in any way. For example, GCC converts this program fragment

```
double foo, bar;

asm ("mycool_asm %1, %0" : "=r" (bar) : "r" (foo));

to these x86 assembly instructions:

movl -8(%ebp),%edx

movl -4(%ebp),%ecx

#APP

mycool_asm %edx, %edx

#NO_APP

movl %edx,-16(%ebp)

movl %ecx,-12(%ebp)
```

Remember that foo and bar each require two words of stack storage on a 32-bit x86 architecture. The register ebp points to data on the stack. The first two instructions copy foo into registers EDX and ECX on which mycool_asm operates. The compiler decides to use the same registers to store the answer, which is copied into bar by the final two instructions. It chooses appropriate registers, even reusing the same registers, and copies operands to and from the proper locations automatically.

Extended Assembly Syntax

In the subsections that follow, we describe the syntax rules for asm statements. Their sections are separated by colons. We will refer to this illustrative asm statement, which computes the Boolean expression x > y:

```
asm ("fucomip %%st(1), %%st; seta %%al" :
 "=a" (result) : "u" (y), "t" (x) : "cc", "st");
```

First, fucomip compares its two operands x and y, and stores values indicating the result into the condition code register. Then seta converts these values into a 0 or 1 result.

B.Assembler Instructions

The first section contains the assembler instructions, enclosed in quotation marks. The example asm contains two assembly instructions, fucomip and seta, separated by semicolons. If the assembler does not permit semicolons, use newline characters (\n) to separate instructions. The compiler ignores the contents of this first section, except that one level of percentage signs is removed, so %% changes to %. The meaning of %%st(1) and other such terms is architecture-dependent.GCC will complain if you specify the -traditional option or the -ansi option when compiling a program containing asm statements. To avoid producing these errors, such as in header files, use the alternative keyword __asm__.

C.Outputs

The second section specifies the instructions' output operands using C syntax. Each operand is specified by an operand constraint string followed by a C expression in parentheses. For output operands, which must be lvalues, the constraint string should begin with an equals sign. The compiler checks that the C expression for each output operand is in fact an lvalue. Letters specifying registers for a particular architecture can be found in the GCC source code, in the REG_CLASS_FROM_LETTER macro. For example, the gcc/config/i386/i386.h configuration file in GCC lists the register letters for the x86 architecture (You'll need to have some familiarity with GCC's internals to make sense of this file). Table 1 below summarizes these.

Table 1: Register Letters for the Intel x86 Architecture

Register Letter	Registers That GCC May Use
R	General register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
Q	General register for data (EAX, EBX, ECX, EDX)
F	Floating-point register
T	Top floating-point register
U	Second-from-top floating-point register
A	EAX register
B	EBX register
C	ECX register
D	EDX register
X	SSE register (Streaming SIMD Extension register)
Y	MMX multimedia registers
A	An 8-byte value formed from EAX and EDX
D	Destination pointer for string operations (EDI)
S	Source pointer for string operations (ESI)

Multiple operands in an asm statement, each specified by a constraint string and a C expression, are separated by commas, as illustrated in the example asm's input section. You may specify up to 10 operands, denoted %0, %1,…, %9, in the output and input sections. If there are no output operands but there are input operands or clobbered registers, leave the output section empty or mark it with a comment like

```
/* no outputs */.
```

A.Inputs

The third section specifies the input operands for the assembler instructions. The constraint string for an input operand should not have an equals sign, which indicates an lvalue. Otherwise, an input operand's syntax is the same as for output operands. To indicate that a register is both read from and written to in the same asm, use an input constraint string of the output operand's number. For example, to indicate that an input register is the same as the first output register number, use 0. Output operands are numbered left to right, starting with 0. Merely specifying the same C expression for an output operand and an input operand does not guarantee that the two values will be placed in the same register. This input section can be omitted if there are no input operands and the subsequent clobber section is empty.

B. Clobbers

If an instruction modifies the values of one or more registers as a side effect, specify the clobbered registers in the asm's fourth section. For example, the fucomip instruction modifies the condition code register, which is denoted cc. Separate strings representing clobbered registers with commas. If the instruction can modify an arbitrary memory location, specify memory.

Using the clobber information, the compiler determines which values must be reloaded after the asm executes. If you don't specify this information correctly, GCC may assume incorrectly that registers still contain values that have, in fact, been overwritten, which will affect your program's correctness.

Example

The x86 architecture includes instructions that determine the positions of the least significant set bit and the most significant set bit in a word. The processor can execute these instructions quite efficiently. In contrast, implementing the same operation in C requires a loop and a bit shift. For example, the bsrl assembly instruction computes the position of the most significant bit set in its first operand, and places the bit position (counting from 0, the least significant bit) into its second operand. To place the bit position for number into position, we could use this asm statement:

asm ("bsrl %1, %0" : "=r" (position) : "r" (number));

One way you could implement the same operation in C is using this loop:

```
long i;

for (i = (number >> 1), position = 0; i != 0; ++position)

i >>= 1;
```

To test the relative speeds of these two versions, we'll place them in a loop that computes the bit positions for a large number of values. The program "bit-pos-loop.c" given below does this using the C loop implementation. The program loops over integers, from 1 up to the value specified on the command line. For each value of number, it computes the most significant bit that is set.

The subsequent program "bit-pos-asm.c" does the same thing using the inline assembly instruction. Note that in both versions, we assign the computed bit position to a volatile variable result. This is to coerce the compiler's optimizer so that it does not eliminate the entire bit position computation; if the result is not used or stored in memory, the optimizer eliminates the computation as "dead code."

```
//Program "bit-pos-loop.c" - Find Bit Position Using a Loop

#include <stdio.h>

#include <stdlib.h>

int main (int argc, char* argv[])
```

```
{
long max = atoi (argv[1]);
long number;
long i;
unsigned position;
volatile unsigned result;
/* Repeat the operation for a large number of values. */
for (number = 1; number <= max; ++number) {
/* Repeatedly shift the number to the right, until the result is
zero. Keep count of the number of shifts this requires. */
for (i = (number >> 1), position = 0; i != 0; ++position)
i >>= 1;
/* The position of the most significant set bit is the number of
shifts we needed after the first one. */
result = position;
}
return 0;
}
//Program "bit-pos-asm.c" - Find Bit Position Using bsrl
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char* argv[])
{
long max = atoi (argv[1]);
long number;
unsigned position;
volatile unsigned result;
/* Repeat the operation for a large number of values. */
for (number = 1; number <= max; ++number) {
/* Compute the position of the most significant set bit using the
```

```
bsrl assembly instruction. */
asm ("bsrl %1, %0" : "=r" (position) : "r" (number));
result = position;
}
return 0;
}
```

We'll compile both versions with full optimization:

```
$ cc -O2 -o bit-pos-loop bit-pos-loop.c
$ cc -O2 -o bit-pos-asm bit-pos-asm.c
```

Now let's run each using the time command to measure execution time. We'll specify a large value as the command-line argument, to make sure that each version takes at least a few seconds to run.

```
$ time ./bit-pos-loop 250000000
19.51user 0.00system 0:20.40elapsed 95%CPU (0avgtext+0avgdata
0maxresident)k0inputs+0outputs (73major+11minor)pagefaults 0swaps
$ time ./bit-pos-asm 250000000
3.19user 0.00system 0:03.32elapsed 95%CPU (0avgtext+0avgdata
0maxresident)k0inputs+0outputs (73major+11minor)pagefaults 0swaps
```

Notice that the version that uses inline assembly executes a great deal faster (your results for this example may vary).

Optimization Issues

GCC's optimizer attempts to rearrange and rewrite programs' code to minimize execution time even in the presence of asm expressions. If the optimizer determines that an asm's output values are not used, the instruction will be omitted unless the keyword volatile occurs between asm and its arguments. (As a special case, GCC will not move an asm without any output operands outside a loop.) Any asm can be moved in ways that are difficult to predict, even across jumps. The only way to guarantee a particular assembly instruction ordering is to include all the instructions in the same asm. Using asms can restrict the optimizer's effectiveness because the compiler does not know the asms' semantics. GCC is forced to make conservative guesses that may prevent some optimizations. Caveat emptor!

Maintenance and Portability Issues

If you decide to use nonportable, architecture-dependent asm statements, encapsulating these statements within macros or functions can aid in maintenance and porting. Placing all these macros in one file and documenting them will ease porting to a different architecture, something that occurs with surprising frequency even for "throwaway" programs. Thus, the programmer will need to rewrite only one file for the different architecture. For example, most asm statements in the Linux source code are grouped into /usr/src/linux/include/asm and usr/src/linux/include/asm-i386 header files, and /usr/src/linux/arch/i386/ and /usr/src/linux/drivers/ source files.

Conclusion

In this activity we presented the basics of inling assembly code into a C/C++ program.

Assessment

Practise the examples given in this activity, to help you get more insights of assembly inlining.

Unit Summary

This unit presented three aspects of programming a computer system, namely, using the shell, high-level languages such as C/C++, and assembly language. Specifically, the Unix/Linux shell was introduced alongside the process of writing and the constructs of a shell program. We presented the commonly used shells are available in Linux/Unix OS, and explained the process of writing a shell script and working with variables in shells Finally, the techniques of inlining assembly code into a C/C++ program were presented.

Unit Assessment

Check your understanding!

Miscellaneous Exercises

Instructions

1. Write the following shell script, save it, execute it and note down it's output.

```
# Script to print user information who currently login
, current date & time

#

clear

echo "Hello $USER"

echo "Today is \c ";date

echo "Number of user login : \c" ; who | wc -l

echo "Calendar"

cal

exit 0
```

```
# Script to test MY knowledge about
variables!

#

myname=Vivek

myos = TroubleOS

myno=5

echo "My name is $myname"

echo "My os is $myos"

echo "My number is myno, can you see
this number"
```

6. Write a script, pad, that will pad on the left. Count directly from 2 to 10.

7. Write the following inline assembly program, save it, execute it and note down it's output (The examplemakes use of extended inline assembly statements. It performs simple arithmetic operations on integer operands and displays the result

```
#include <stdio.h>

int main() {

    int arg1, arg2, add, sub, mul, quo, rem ;

    printf( "Enter two integer numbers : " );
    scanf( "%d%d", &arg1, &arg2 );

    /* Perform Addition, Subtraction, Multiplication & Division */
    __asm__ ( "addl %%ebx, %%eax;" : "=a" (add) : "a" (arg1) , "b"
(arg2) );
    __asm__ ( "subl %%ebx, %%eax;" : "=a" (sub) : "a" (arg1) , "b"
(arg2) );
    __asm__ ( "imull %%ebx, %%eax;" : "=a" (mul) : "a" (arg1) , "b"
(arg2) );
```

```
    __asm__ ( "movl $0x0, %%edx;"
              "movl %2, %%eax;"
              "movl %3, %%ebx;"
               "idivl %%ebx;" : "=a" (quo), "=d" (rem) : "g" (arg1),
"g" (arg2) );

    printf( "%d + %d = %d\n", arg1, arg2, add );
    printf( "%d - %d = %d\n", arg1, arg2, sub );
    printf( "%d * %d = %d\n", arg1, arg2, mul );
    printf( "%d / %d = %d\n", arg1, arg2, quo );
    printf( "%d %% %d = %d\n", arg1, arg2, rem );

    return 0 ;
}
```

8. Write the following inline assembly program, save it, execute it and note down it's output (The example aim to compute the Greatest Common Divisor using well known Euclid's Algorithm).

```
#include <stdio.h>
int gcd( int a, int b ) {
    int result ;
    /* Compute Greatest Common Divisor using Euclid's Algorithm */
    __asm__ __volatile__ ( "movl %1, %%eax;"
                          "movl %2, %%ebx;"
                          "CONTD: cmpl $0, %%ebx;"
                          "je DONE;"
                          "xorl %%edx, %%edx;"
                          "idivl %%ebx;"
                          "movl %%ebx, %%eax;"
                          "movl %%edx, %%ebx;"
                          "jmp CONTD;"
```

```
                    "DONE: movl %%eax, %0;" : "=g" (result) : "g"
(a), "g" (b)
    );

    return result ;
}
int main() {
    int first, second ;
    printf( "Enter two integers : " ) ;
    scanf( "%d%d", &first, &second );

   printf( "GCD of %d & %d is %d\n", first, second, gcd(first, second)
) ;

    return 0 ;
}
```

Grading Scheme

As guided by the offering Institution Grading Regulations

Answers mailto:njulumi@gmail.com

Unit Readings and Other Resources

1. Learning the bash Shell: Unix Shell Programming (In a Nutshell (O'Reilly)), Kindle Edition, Cameron Newham (Author)
2. bash Cookbook: Solutions and Examples for bash Users (Cookbooks (O'Reilly)), Kindle Edition, Carl Albing (Author), JP Vossen (Author), Cameron Newham (Author)
3. Mark Mitchell, Jeffrey Oldham, and Alex Samuel; Advanced Linux Programming; Copyright © 2001 by New Riders Publishing; FIRST EDITION: June, 2001
4. Professional Assembly Language, John Wiley & Sons, 22 Feb 2005, By Richard Blumhttp://www.codeproject.com/Articles/15971/

   Using-Inline-Assembly-in-C-C

Unit 3. Processes,Threads and Memory management

Unit Introduction

A running instance of a program is called a process. For example, if you have two terminal windows showing on your screen, then you are probably running the same terminal program twice, i.e. you have two terminal processes. Each terminal window is probably running a shell; each running shell is another process. When you invoke a command from a shell, the corresponding program is executed in a new process; the shell process resumes when that process completes. Advanced programmers often use multiple cooperating processes in a single application to enable the application to do more than one thing at once, to increase application robustness, and to make use of already-existing programs.

Threads, like processes, are a mechanism to allow a program to do more than one thing at a time. As with processes, threads appear to run concurrently; the Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute. Conceptually, a thread exists within a process. Threads are a finer-grained unit of execution than processes. When you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially. That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time.

Program must be brought into memory and placed within a process for it to be run. Memory is the primary data storage area for computers. We call the basic memory unit a bit. A bit may contain two different values: either 0 or 1. Memory consists of a number of cells which can store some number of bits. The memory is just a byte array. Each cell has a number to identify it, called its address. Programs refer addresses to reach memory. A management system called virtual memory organizes memory into "pages", which are memory units typically a few Kbytes in size. CPU has a unit called Memory Management Unit which is responsible for operating virtual memory.

In this unit we describe process, thread and memory manipulation functions in Linux systems most of whose are similar to those on other UNIX systems. Most of the described functions are declared in the header file <unistd.h>; check the man page for each function to be sure.

Because a process and all its threads can be executing only one program at a time, if any thread inside a process calls one of the exec functions, all the other threads are ended (the new program may, of course, create new threads). GNU/Linux implements the POSIX standard thread API (known as pthreads). All thread functions and data types are declared in the header file <pthread.h>.The pthread functions are not included in the standard C library. Instead, they are in libpthread, so you should add -lpthread to the command line when you link your program.

Unit Objectives

Upon completion of this unit you should be able to:

- explain the process, thread and memory management concepts
- create processes using the system, fork and exec functions
- manage and manipulate processes using various system signals
- create and manage threads
- analyze processes from threads
- demonstrate understanding of memory management in C programs

Key Terms

Process:A process is an instance of a program running in a computer. It is close in meaning to task, a term used in some operating systems. In UNIX/Linux and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program).

Thread:A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its instructions (executable code) and its context (the values of its variables at any given moment).

Memory management: Memory management is the act of managing computer memory at the system level. The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. This is critical to any advanced computer system where more than a single process might be underway at any time.

Learning Activities

Activity 1 - Processes

Introduction

Even as you sit down at your computer, there are processes running. Every executing program uses one or more processes. Let's start by taking a look at the processes already on your computer.

Activity Details

Working with Processes

A.Process Ids

Each process in a Linux system is identified by its unique process ID, sometimes referred to as pid. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created. Every process also has a parent process (except the special init process-the first process in Linux Systems. Thus, you can think of the processes on a Linux system as arranged in a tree, with the init process at its root. The parent process ID, or ppid, is simply the process ID of the process's parent. When referring to process IDs in a C or C++ program, always use the pid_t typedef, which is defined in <sys/types.h>. A program can obtain the process ID of the process it's running in with the getpid() system call, and it can obtain the process ID of its parent process with the getppid() system call. For instance, the program "print-pid.c" provided below prints its process ID and its parent's process ID.

```
// Program print-pid.c - Printing the Process ID

#include <stdio.h>

#include <unistd.h>

int main ()

{

printf ("The process ID is %d\n", (int) getpid ());

printf ("The parent process ID is %d\n", (int) getppid ());

return 0;

}
```

Observe that if you invoke this program several times, a different process ID is reported because each invocation is in a new process. However, if you invoke it every time from the same shell, the parent process ID (that is, the process ID of the shell process) is the same.

B.Viewing Active Processes

The ps command displays the processes that are running on your system. The GNU/Linux version of ps has lots of options because it tries to be compatible with versions of ps on several other UNIX variants. These options control which processes are listed and what information about each is shown. By default, invoking ps displays the processes controlled by the terminal or terminal window in which ps is invoked. For example:

```
$ ps

 PID   TTY   TIME   CMD

21693 pts/8 00:00:00 bash

21694 pts/8 00:00:00 ps
```

This invocation of ps shows two processes. The first, bash, is the shell running on this terminal. The second is the running instance of the ps program itself. The first column, labeled PID, displays the process ID of each. For a more detailed look at what's running on your GNU/Linux system, invoke this:

$ ps -e -o pid,ppid,command

The -e option instructs ps to display all processes running on the system. The -o pid, ppid, command option tells ps what information to show about each process - in this case, the process ID, the parent process ID, and the command running in this process. Below are the first few lines and last few lines of output from this command on typical system. You may see different output, depending on what's running on your system.

```
$ ps -e -o pid,ppid,command
PID  PPID   COMMAND
1    0      init [5]
2    1      [kflushd]
3    1      [kupdate]
     ...
21725       21693 Xterm
21727       21725 Bash
21728       21727 ps -e -o pid,ppid,command
```

Note that the parent process ID of the ps command, 21727, is the process ID of bash, the shell from which I invoked ps. The parent process ID of bash is in turn 21725, the process ID of the xterm program in which the shell is running.

C.Killing a Process

You can kill a running process with the kill command. Simply specify on the command line the process ID of the process to be killed. By default the kill command works by sending the process a SIGTERM, or termination signal. This causes the process to terminate, unless the executing program explicitly handles or masks the SIGTERM signal.

Creating Processes

Two common techniques are used for creating a new process, the system function and the fork function along with exec functions. The first is relatively simple but should be used sparingly because it is inefficient and considerably has security risks. The second technique is more complex but provides greater flexibility, speed, and security.

A.Using system

The system function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell. In fact, system creates a subprocess by running the standard Bourne shell (/bin/sh) and then hands the command to

that shell for execution. For example, the program "system.c" provided below invokes the ls command to display the contents of the root directory, as if you typed ls -l / into a shell.

```
//Program "system.c" - Using the system Call
#include <stdlib.h>
int main ()
{
int return_value;
return_value = system ("ls -l /");
return return_value;
}
```

The system function returns the exit status of the shell command. If the shell itself cannot be run, system returns 127; if another error occurs, system returns –1. Because the system function uses a shell to invoke your command, it's subject to the features, limitations, and security flaws of the system's shell. You can't rely on the availability of any particular version of the Bourne shell. On many UNIX systems, /bin/sh is a symbolic link to another shell. For instance, on most GNU/Linux systems, /bin/sh points to bash (the Bourne-Again SHell), and different GNU/Linux distributions use different versions of bash. Invoking a program with root privilege with the system function, for instance, can have different results on different GNU/Linux systems. Therefore, it's preferable to use the fork and exec method for creating processes.

B.Using fork and exec

The DOS and Windows API contain the spawn family of functions. These functions take as an argument the name of a program to run and create a new process instance of that program. Linux doesn't contain a single function that does all this in one step. Instead, Linux provides one function, fork, which makes a child process that is an exact copy of its parent process. Linux provides another set of functions, the exec family, which causes a particular process to cease being an instance of one program and to instead become an instance of another program. To spawn a new process, you first use fork to make a copy of the current process. Then you use exec to transform one of these processes into an instance of the program you want to spawn.

i.Calling fork

When a program calls fork, a duplicate process, called the child process, is created. The parent process continues executing the program from the point that fork was called. The child process, too, executes the same program from the same place. So how do the two processes differ? First, the child process is a new process and therefore has a new process ID, distinct from its parent's process ID. One way for a program to distinguish whether it's in the parent process or the child process is to call getpid. However, the fork function provides different return values to the parent and child processes—one process "goes in" to the fork call, and two processes "come out," with different return values.

The return value in the parent process is the process ID of the child. The return value in the child process is zero. Because no process ever has a process ID of zero, this makes it easy for the program whether it is now running as the parent or the child process.

Program "fork.c" provided below is an example of using fork to duplicate a program's process. Note that the first block of the "if" statement is executed only in the parent process, while the "else" clause is executed in the child process.

```
//Program  fork.c - Using fork to Duplicate a Program's Process

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main ()

{

    pid_t child_pid;

    printf ("the main program process ID is %d\n", (int) getpid ());

    child_pid = fork ();

    if (child_pid != 0) {

printf ("this is the parent process, with id %d\n", (int) getpid ());

printf ("the child's process ID is %d\n", (int) child_pid);

   }

   else

printf ("this is the child process, with id %d\n", (int) getpid ());

              return 0;

}
```

ii.Using the exec Family

The exec functions replace the program running in a process with another program. When a program calls an exec function, that process immediately ceases executing that program and begins executing a new program from the beginning, assuming that the exec call doesn't encounter an error. Within the exec family, there are functions that vary slightly in their capabilities and how they are called. Functions that contain the letter p in their names (execvp and execlp) accept a program name and search for a program by that name in the current execution path; functions that don't contain the p must be given the full path of the program to be executed.

Functions that contain the letter v in their names (execv, execvp, and execve) accept the argument list for the new program as a NULL-terminated array of pointers to strings. Functions that contain the letter l (execl, execlp, and execle) accept the argument list using the C language's varargs mechanism. Functions that contain the letter e in their names (execve and execle) accept an additional argument, an array of environment variables. The argument should be a NULL-terminated array of pointers to character strings. Each character string should be of the form "VARIABLE=value".

Because exec replaces the calling program with another one, it never returns unless an error occurs. The argument list passed to the program is analogous to the command-line arguments that you specify to a program when you run it from the shell. They are available through the argc and argv parameters to main. Remember, when a program is invoked from the shell, the shell sets the first element of the argument list argv[0] to the name of the program, the second element of the argument list argv[1] to the first command-line argument, and so on. When you use an exec function in your programs, you, too, should pass the name of the function as the first element of the argument list.

iii.Using fork and exec Together

A common pattern to run a subprogram within a program is first to fork the process and then exec the subprogram. This allows the calling program to continue execution in the parent process while the calling program is replaced by the subprogram in the child process. The program "fork-exec.c" provided below, like program "system.c" given above lists the contents of the root directory using the ls command. Unlike the program "system.c", though, it invokes the ls command directly, passing it the command-line arguments -l and / rather than invoking it through a shell.

```
//Program "fork-exec.c" - Using fork and exec Together

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

    /* Spawn a child process running a new program. PROGRAM is the name

     of the program to run; the path will be searched for this program.

      ARG_LIST is a NULL-terminated list of character strings to be

      passed as the program's argument list. Returns the process ID of

      the spawned process. */

 int spawn (char* program, char** arg_list)

{
```

```
pid_t child_pid;
/* Duplicate this process. */
child_pid = fork ();
if (child_pid != 0)
      /* This is the parent process. */
  return child_pid;
else {
    /* Now execute PROGRAM, searching for it in the path. */
     execvp (program, arg_list);
    /* The execvp function returns only if an error occurs. */
     fprintf (stderr, "an error occurred in execvp\n");
     abort ();
         }
}
int main ()
{
     /* The argument list to pass to the "ls" command. */
     char* arg_list[] = {
     "ls", /* argv[0], the name of the program. */
      "-l",
      "/",
      NULL /* The argument list must end with a NULL. */
      };
      /* Spawn a child process running the "ls" command. Ignore the
       returned child process ID. */
       spawn ("ls", arg_list);
       printf ("done with main program\n");
return 0;
}
```

A.Process Scheduling

Linux schedules the parent and child processes independently; there's no guarantee of which one will run first, or how long it will run before Linux interrupts it and lets the other process (or some other process on the system) run. In particular, none, part, or all of the ls command may run in the child process before the parent completes. Linux promises that each process will run eventually—no process will be completely starved of execution resources.

You may specify that a process is less important—and should be given a lower priority —by assigning it a higher niceness value. By default, every process has a niceness of zero. A higher niceness value means that the process is given a lesser execution priority; conversely, a process with a lower (that is, negative) niceness gets more execution time. To run a program with a nonzero niceness, use the nice command, specifying the niceness value with the -n option. For example, this is how you might invoke the command "sort input.txt > output.txt", a long sorting operation, with a reduced priority so that it doesn't slow down the system too much:

$nice -n 10 sort input.txt > output.txt

You can use the renice command to change the niceness of a running process from the command line. To change the niceness of a running process programmatically, use the nice function. Its argument is an increment value, which is added to the niceness value of the process that calls it. Remember that a positive value raises the niceness value and thus reduces the process's execution priority.

Note that only a process with root privilege can run a process with a negative niceness value or reduce the niceness value of a running process. This means that you may specify negative values to the nice and renice commands only when logged in as root, and only a process running as root can pass a negative value to the nice function. This prevents ordinary users from grabbing execution priority away from others using the system.

Signals

Signals are mechanisms for communicating with and manipulating processes in Linux. In this section we present some of the most important signals and techniques that are used for controlling processes.

A signal is a special message sent to a process. Signals are asynchronous; when a process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code. There are several dozen different signals, each with a different meaning. Each signal type is specified by its signal number, but in programs, you usually refer to a signal by its name. In Linux, these are defined in /usr/include/bits/signum.h. You shouldn't include this header file directly in your programs, though, instead, use <signal.h>.

When a process receives a signal, it may do one of several things, depending on the signal's disposition. For each signal, there is a default disposition, which determines what happens to theprocess if the program does not specify some other behavior. For most signal types, a program may specify some other behavior—either to ignore the signal or to call a special signal-handler function to respond to the signal.

If a signal handler is used, the currently executing program is paused, the signal handler is executed, and, when the signal handler returns, the program resumes.

The Linux system sends signals to processes in response to specific conditions. For instance, SIGBUS (bus error), SIGSEGV (segmentation violation), and SIGFPE (floating point exception) may be sent to a process that attempts to perform an illegal operation. The default disposition for these signals it to terminate the process and produce a core file. A process may also send a signal to another process. One common use of this mechanism is to end another process by sending it a SIGTERM or SIGKILL signal. Another common use is to send a command to a running program. Two "userdefined" signals are reserved for this purpose: SIGUSR1 and SIGUSR2.The SIGHUP signal is sometimes used for this purpose as well, commonly to wake up an idling program or cause a program to re-read its configuration files.

The sigaction function can be used to set a signal disposition. The first parameter is the signal number. The next two parameters are pointers to sigaction structures; the first of these contains the desired disposition for that signal number, while the second receives the previous disposition. The most important field in the first or second sigaction structure is sa_handler. It can take one of three values:

- SIG_DFL, which specifies the default disposition for the signal.
- SIG_IGN, which specifies that the signal should be ignored
- A pointer to a signal-handler function. The function should take one parameter, the signal number, and return void.

Because signals are asynchronous, the main program may be in a very fragile state when a signal is processed and thus while a signal handler function executes. Therefore, you should avoid performing any I/O operations or calling most library and system functions from signal handlers.

A signal handler should perform the minimum work necessary to respond to the signal, and then return control to the main program (or terminate the program). In most cases, this consists simply of recording the fact that a signal occurred. The main program then checks periodically whether a signal has occurred and reacts accordingly.

It is possible for a signal handler to be interrupted by the delivery of another signal. While this may sound like a rare occurrence, if it does occur, it will be very difficult to diagnose and debug the problem. Therefore, you should be very careful about what your program does in a signal handler.

Even assigning a value to a global variable can be dangerous because the assignment may actually be carried out in two or more machine instructions, and a second signal may occur between them, leaving the variable in a corrupted state.

If you use a global variable to flag a signal from a signal-handler function, it should be of the special type sig_atomic_t. Linux guarantees that assignments to variables of this type are performed in a single instruction and therefore cannot be interrupted midway. In Linux, sig_atomic_t is an ordinary int; in fact, assignment to integer types the size of int or smaller, or to pointers, are atomic. If you want to write a program that's portable to any standard UNIX system, though, use sig_atomic_t for these global variables. The program skeleton "sigusr1.c" below, for instance, uses a signal-handler function tocount the number of times that the program receives SIGUSR1, one of the signals reserved for application use.

```
//Program sigusr1.c - Using a Signal Handler
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
sig_atomic_t sigusr1_count = 0;
void handler (int signal_number)
{
++sigusr1_count;
}
int main ()
{
struct sigaction sa;
memset (&sa, 0, sizeof (sa));
sa.sa_handler = &handler;
sigaction (SIGUSR1, &sa, NULL);
/* Do some lengthy stuff here. */
/* ... */
printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
return 0;
}
```

Process Termination

Normally, a process terminates in one of two ways. Either the executing program calls the exit function, or the program's main function returns. Each process has an exit code: a number that the process returns to its parent. The exit code is the argument passed to the exit function, or the value returned from main.

A process may also terminate abnormally, in response to a signal. For instance, the SIGBUS, SIGSEGV, and SIGFPE signals mentioned previously cause the process to terminate. Other signals are used to terminate a process explicitly. The SIGINT signal is sent to a process when the user attempts to end it by typing Ctrl+C in its terminal. The SIGTERM signal is sent by the kill command. The default disposition for both of these is to terminate the process. By calling the abort function, a process sends itself the SIGABRT signal, which terminates the process and produces a core file. The most powerful termination signal is SIGKILL, which ends a process immediately and cannot be blocked or handled by a program.

Any of these signals can be sent using the kill command by specifying an extra command-line flag; for instance, to end a troublesome process by sending it a SIGKILL, invoke the following, where pid is its process ID:

```
$kill -KILL pid
```

To send a signal from a program, use the kill function. The first parameter is the target process ID. The second parameter is the signal number; use SIGTERM to simulate the default behavior of the kill command. For instance, where child pid contains the process ID of the child process, you can use the kill function to terminate a child process from the parent by calling it like this:

```
kill (child_pid, SIGTERM);
```

nclude the <sys/types.h> and <signal.h> headers if you use the kill function. By convention, the exit code is used to indicate whether the program executed correctly. An exit code of zero indicates correct execution, while a nonzero exit code indicates that an error occurred. In the latter case, the particular value returned may give some indication of the nature of the error. It's a good idea to stick with this convention in your programs because other components of the GNU/Linux system assume this behavior. For instance, shells assume this convention when you connect multiple programs with the && (logical and) and ll (logical or) operators. Therefore, you should explicitly return zero from your main function, unless an error occurs.

With most shells, it's possible to obtain the exit code of the most recently executed program using the special $? variable. Here's an example in which the ls command is invoked twice and its exit code is displayed after each invocation. In the first case, ls execute correctly and return the exit code zero. In the second case, ls encounters an error (because the filename specified on the command line does not exist) and thus returns a nonzero exit code.

```
$ ls /

bin coda etc lib misc nfs proc sbin usr

boot dev home lost+found mnt opt root tmp var

$ echo $?

0

$ ls bogusfile

ls: bogusfile: No such file or directory

$ echo $?

1
```

Note that even though the parameter type of the exit function is int and the main function returns an int, Linux does not preserve the full 32 bits of the return code. In fact, you should use exit codes only between zero and 127. Exit codes above 128 have a special meaning—when a process is terminated by a signal; its exit code is 128 plus the signal number.

A.Waiting for Process Termination

If you typed in and ran the fork and exec example program "fork-exec.c" in Section 1.1.1.2, you may have noticed that the output from the ls program often appears after the "main program" has already completed. That's because the child process, in which ls is run, is scheduled independently of the parent process.

Because Linux is a multitasking operating system, both processes appear to execute simultaneously, and you can't predict whether the ls program will have a chance to run before or after the parent process runs. In some situations, though, it is desirable for the parent process to wait until one or more child processes have completed. This can be done with the wait family of system calls. These functions allow you to wait for a process to finish executing, and enable the parent process to retrieve information about its child's termination. There are four different system calls in the wait family; you can choose to get a little or a lot of information about the process that exited, and you can choose whether you care about which child process terminated.

The simplest such function is called simply wait. It blocks the calling process until one of its child processes exits (or an error occurs). It returns a status code via an integer pointer argument, from which you can extract information about how the child process exited. For instance, the WEXITSTATUS macro extracts the child process's exit code.

You can use the WIFEXITED macro to determine from a child process's exit status whether that process exited normally (via the exit function or returning from main) or died from an unhandled signal. In the latter case, use the WTERMSIG macro to extract from its exit status the signal number by which it died. Below is the main function from the fork and exec example again. This time, the parent process calls wait to wait until the child process, in which the ls command executes, is finished.

```
int main ()
```

```
{
    int child_status;
        /* The argument list to pass to the "ls" command. */
    char* arg_list[] = {
    "ls", /* argv[0], the name of the program. */
    "-l",
    "/",
     NULL /* The argument list must end with a NULL. */
     };
     /* Spawn a child process running the "ls" command. Ignore the
        returned child process ID. */
    spawn ("ls", arg_list);
     /* Wait for the child process to complete. */
     wait (&child_status);
     if (WIFEXITED (child_status))
     printf ("the child process exited normally, with exit code %d\n",
             WEXITSTATUS (child_status));
     else
  printf ("the child process exited abnormally\n");
    return 0;
}
```

Several similar system calls are available in Linux, which are more flexible or provide more information about the exiting child process. The waitpid function can be used to wait for a specific child process to exit instead of any child process. The wait3 function returns CPU usage statistics about the exiting child process, and the wait4 function allows you to specify additional options about which processes to wait for.

B.Zombie Processes

If a child process terminates while its parent is calling a wait function, the child process vanishes and its termination status is passed to its parent via the wait call. But what happens when a child process terminates and the parent is not calling wait? Does it simply vanish? No, because then information about its termination—such as whether it exited normally and, if so, what its exit status is—would be lost. Instead, when a child process terminates, it becomes a zombie process.

A zombie process is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children. The wait functions do this, too, so it's not necessary to track whether your child process is still executing before waiting for it. Suppose, for instance, that a program forks a child process, performs some other computations, and then calls wait. If the child process has not terminated at that point, the parent process will block in the wait call until the child process finishes. If the child process finishes before the parent process calls wait, the child process becomes a zombie. When the parent process calls wait, the zombie child's termination status is extracted, the child process is deleted, and the wait call returns immediately.

What happens if the parent does not clean up its children? They stay around in the system, as zombie processes. The program "zombie.c" provided below forks a child process, which terminates immediately and then goes to sleep for a minute, without ever cleaning up the child process.

```
//Program zombie.c - Making a Zombie Process
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
    pid_t child_pid;
   /* Create a child process. */
    child_pid = fork ();
     if (child_pid > 0) {
     /* This is the parent process. Sleep for a minute. */
     sleep (60);
    }
    else {
          /* This is the child process. Exit immediately. */
       exit (0);
    }
     return 0;
}
```

Try compiling this file to an executable named make-zombie. Run it, and while it's still running, list the processes on the system by invoking the following command in another window:

```
$ ps -e -o pid,ppid,stat,cmd
```

This lists the process ID, parent process ID, process status, and process command line. Observe that, in addition to the parent make-zombie process, there is another make-zombie process listed. It's the child process; note that its parent process ID is the process ID of the main make-zombie process. The child process is marked as <defunct>, and its status code is Z, for zombie.

What happens when the main make-zombie program ends when the parent process exits, without ever calling wait? Does the zombie process stay around? No—try running ps again, and note that both of the make-zombie processes are gone. When a program exits, its children are inherited by a special process, the init program, which always runs with process ID of 1 (it's the first process started when Linux boots).The init process automatically cleans up any zombie child processes that it inherits.

C.Cleaning Up Children Asynchronously

If you're using a child process simply to exec another program, it's fine to call wait immediately in the parent process, which will block until the child process completes. But often, you'll want the parent process to continue running, as one or more children execute synchronously. How can you be sure that you clean up child processes that have completed so that you don't leave zombie processes, which consume system resources, lying around?

One approach would be for the parent process to call wait3 or wait4 periodically, to clean up zombie children. Calling wait for this purpose doesn't work well because, if no children have terminated, the call will block until one does. However, wait3 and wait4 take an additional flag parameter, to which you can pass the flag value WNOHANG. With this flag, the function runs in nonblocking mode—it will clean up a terminated child process if there is one, or simply return if there isn't. The return value of the call is the process ID of the terminated child in the former case, or zero in the latter case.

A more elegant solution is to notify the parent process when a child terminates. There are several ways to do this using the methods of "Interprocess Communication," but fortunately Linux does this for you, using signals. When a child process terminates, Linux sends the parent process the SIGCHLD signal. The default disposition of this signal is to do nothing, which is why you might not have noticed it before. Thus, an easy way to clean up child processes is by handling SIGCHLD. Of course, when cleaning up the child process, it's important to store its termination status if this information is needed, because once the process is cleaned up using wait, that information is no longer available. The program "sigchld.c" provided below is what it looks like for a program to use a SIGCHLD handler to clean up its child processes.

```
//Program "sigchld.c" - Cleaning Up Children by Handling SIGCHLD
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
sig_atomic_t child_exit_status;
void clean_up_child_process (int signal_number)
{
    /* Clean up the child process. */
    int status;
    wait (&status);
     /* Store its exit status in a global variable. */
     child_exit_status = status;
}
int main ()
{
     /* Handle SIGCHLD by calling clean_up_child_process. */
     struct sigaction sigchld_action;
     memset (&sigchld_action, 0, sizeof (sigchld_action));
     sigchld_action.sa_handler = &clean_up_child_process;
     sigaction (SIGCHLD, &sigchld_action, NULL);
     /* Now do things, including forking a child process. */
      /* ... */
   return 0;
}
```

Note how the signal handler stores the child process's exit status in a global variable, from which the main program can access it. Because the variable is assigned in a signal handler, its type is sig_atomic_t.

Conclusion

In this activity we presented the process concepts and explained various techniques and tools for creating and working with processes. In particular, process creation using the system, fork and exec functions was presented. Further, the management and manipulation of processes using various system signals was highlighted.

Assessment

1. Practice the example code provided in this activity

Activity 2 - Threads

Introduction

We've seen how a program can fork a child process. The child process is initially running its parent's program, with its parent's virtual memory, file descriptors, and so on copied. The child process can modify its memory, close file descriptors, and the like without affecting its parent, and vice versa. When a program creates another thread, though, nothing is copied. The creating and the created thread share the same memory space, file descriptors, and other system resources as the original. If one thread changes the value of a variable, for instance, the other thread subsequently will see the modified value. Similarly, if one thread closes a file descriptor, other threads may not read from or write to that file descriptor.

Each thread in a process is identified by a thread ID. When referring to thread IDs in C or C++ programs, use the type pthread_t. Upon creation, each thread executes a thread function. This is just an ordinary function and contains the code that the thread should run. When the function returns, the thread exits. On GNU/Linux, thread functions take a single parameter, of type void*, and have a void* return type. The parameter is the thread argument: GNU/Linux passes the value along to the thread without looking at it. Your program can use this parameter to pass data to a new thread. Similarly, your program can use the return value to pass data from an exiting thread back to its creator.

The pthread_create function creates a new thread. You provide it with the following:

i. A pointer to a pthread_t variable, in which the thread ID of the new thread is stored.

ii. A pointer to a thread attribute object. This object controls details of how the thread interacts with the rest of the program. If you pass NULL as the thread attribute, a thread will be created with the default thread attributes. Thread attributes are presented in later section,"Thread Attributes."

iii. A pointer to the thread function. This is an ordinary function pointer, of this type:

iv. void* (*) (void*)

v. A thread argument value of type void*. Whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

A call to pthread_create returns immediately, and the original thread continues executing the instructions following the call. Meanwhile, the new thread begins executing the thread function. Linux schedules both threads asynchronously, and your program must not rely on the relative order in which instructions are executed in the two threads.The program "thread-create.c" provided below creates a thread that prints x's continuously to standard error. After calling pthread_create, the main thread prints o's continuously to standard error.

```
//Program "thread-create.c" - Create a Thread

#include <pthread.h>

#include <stdio.h>

/* Prints x's to stderr. The parameter is unused. Does not return. */

void* print_xs (void* unused)

{

while (1)

fputc ('x', stderr);

return NULL;

}

/* The main program. */

int main ()

{

pthread_t thread_id;

/* Create a new thread. The new thread will run the print_xs

function. */

pthread_create (&thread_id, NULL, &print_xs, NULL);

/* Print o's continuously to stderr. */

while (1)

fputc ('o', stderr);

return 0;

}
```

Compile and link this program using the following code:

$ cc -o thread-create thread-create.c -lpthread

Try running it to see what happens. Notice the unpredictable pattern of x's and o's as Linux alternately schedules the two threads. Under normal circumstances, a thread exits in one of two ways. One way, as illustrated previously, is by returning from the thread function. The return value from the thread function is taken to be the return value of the thread. Alternately, a thread can exit explicitly by calling pthread_exit. This function may be called from within the thread function or from some other function called directly or indirectly by the thread function. The argument to pthread_exit is the thread's return value.

Activity Details

Passing Data to Threads

The thread argument provides a convenient method of passing data to threads. Because the type of the argument is void*, though, you can't pass a lot of data directly via the argument. Instead, use the thread argument to pass a pointer to some structure or array of data. One commonly used technique is to define a structure for each thread function, which contains the "parameters" that the thread function expects.

Using the thread argument, it's easy to reuse the same thread function for many threads. All these threads execute the same code, but on different data. The program "thread-create2" given below is similar to the previous example. This one creates two new threads, one to print x's and the other to print o's. Instead of printing infinitely, though, each thread prints a fixed number of characters and then exits by returning from the thread function. The same thread function, char_print, is used by both threads, but each is configured differently using struct char_print_parms.

```
//Program "thread-create2" - Create Two Threads

#include <pthread.h>

#include <stdio.h>

/* Parameters to print_function. */

struct char_print_parms

{

/* The character to print. */

char character;

/* The number of times to print it. */

int count;

};

/* Prints a number of characters to stderr, as given by PARAMETERS,
```

```
which is a pointer to a struct char_print_parms. */
void* char_print (void* parameters)
{
/* Cast the cookie pointer to the right type. */
struct char_print_parms* p = (struct char_print_parms*)  parameters;
int i;
for (i = 0; i < p->count; ++i)
fputc (p->character, stderr);
return NULL;
}
/* The main program. */
int main ()
{
pthread_t thread1_id;
pthread_t thread2_id;
struct char_print_parms thread1_args;
struct char_print_parms thread2_args;
/* Create a new thread to print 30,000 'x's. */
thread1_args.character = 'x';
thread1_args.count = 30000;
pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
/* Create a new thread to print 20,000 o's. */
thread2_args.character = 'o';
thread2_args.count = 20000;
pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
return 0;
}
```

But wait! The program “thread-create2” has a serious bug in it. The main thread (which runs the main function) creates the thread parameter structures (thread1_args and thread2_args) as local variables, and then passes pointers to these structures to the threads it creates. What can prevent Linux from scheduling the three threads in such a way that main finishes executing before either of the other two threads are done? Nothing! But if this happens, the memory containing the thread parameter structures will be deallocated while the other two threads are still accessing it!!

Joining Threads

One solution is to force main to wait until the other two threads are done. What we need is a function similar to wait that waits for a thread to finish instead of a process. That function is pthread_join, which takes two arguments: the thread ID of the thread to wait for, and a pointer to a void* variable that will receive the finished thread’s return value. If you don’t care about the thread return value, pass NULL as the second argument.

The revised main program provided below shows the corrected main function for the buggy example program “thread-create2.c”. In this version, main does not exit until both of the threads printing x’s and o’s have completed, so they are no longer using the argument structures.

```
// Revised main function for program “thread-create2.c”

int main ()

{

pthread_t thread1_id;

pthread_t thread2_id;

struct char_print_parms thread1_args;

struct char_print_parms thread2_args;

/* Create a new thread to print 30,000 x’s. */

thread1_args.character = ’x’;

thread1_args.count = 30000;

pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

/* Create a new thread to print 20,000 o’s. */

thread2_args.character = ’o’;

thread2_args.count = 20000;

pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

/* Make sure the first thread has finished. */

pthread_join (thread1_id, NULL);
```

```
/* Make sure the second thread has finished. */

pthread_join (thread2_id, NULL);

/* Now we can safely return. */

return 0;

}
```

The moral of the story: Make sure that any data you pass to a thread by reference is not deallocated, even by a different thread, until you're sure that the thread is done with it. This is true both for local variables, which are deallocated when they go out of scope, and for heap-allocated variables, which you deallocate by calling free (or using delete in C++).

Thread Return Values

If the second argument you pass to pthread_join is non-null, the thread's return value will be placed in the location pointed to by that argument. The thread return value, like the thread argument, is of type void*. If you want to pass back a single int or other small number, you can do this easily by casting the value to void* and then casting back to the appropriate type after calling pthread_join.

The program "primes.c" given below computes the nth prime number in a separate thread. That thread returns the desired prime number as its thread return value. The main thread, meanwhile, is free to execute other code. Note that the successive division algorithm used in compute_prime function is quite inefficient; consult a book on numerical algorithms if you need to compute many prime numbers in your programs.

```
//Program "primes.c" - Compute Prime Numbers in a Thread

#include <pthread.h>

#include <stdio.h>

/* Compute successive prime numbers (very inefficiently). Return the

Nth prime number, where N is the value pointed to by *ARG. */

void* compute_prime (void* arg)

{

int candidate = 2;

int n = *((int*) arg);


while (1) {

int factor;

int is_prime = 1;

/* Test primality by successive division. */
```

```
for (factor = 2; factor < candidate; ++factor)
if (candidate % factor == 0) {
is_prime = 0;
break;
}
/* Is this the prime number we're looking for? */
if (is_prime) {
if (--n == 0)
/* Return the desired prime number as the thread return value. */
return (void*) candidate;
}
++candidate;
}
return NULL;
}
int main ()
{
pthread_t thread;
int which_prime = 5000;
int prime;
/* Start the computing thread, up to the 5,000th prime number. */
pthread_create (&thread, NULL, &compute_prime, &which_prime);
/* Do some other work here... */
/* Wait for the prime number thread to complete, and get the result. */
pthread_join (thread, (void*) &prime);

/* Print the largest prime it computed. */
printf("The %dth prime number is %d.\n", which_prime, prime);
return 0;
    }
```

More on Thread Ids

Occasionally, it is useful for a sequence of code to determine which thread is executing it. The pthread_self function returns the thread ID of the thread in which it is called. This Thread ID may be compared with another thread ID using the pthread_equal function. These functions can be useful for determining whether a particular thread ID corresponds to the current thread. For instance, it is an error for a thread to call pthread_join to join itself. (In this case, pthread_join would return the error code EDEADLK.) To check for this beforehand, you might use code like this:

```
if (!pthread_equal (pthread_self (), other_thread))

pthread_join (other_thread, NULL);
```

Thread Attributes

Thread attributes provide a mechanism for fine-tuning the behavior of individual threads. Recall that pthread_create accepts an argument that is a pointer to a thread attribute object. If you pass a null pointer, the default thread attributes are used to configure the new thread. However, you may create and customize a thread attribute object to specify other values for the attributes.

To specify customized thread attributes, you must follow these steps:

i. Create a pthread_attr_t object. The easiest way is simply to declare an automatic variable of this type.

ii. Call pthread_attr_init, passing a pointer to this object. This initializes the attributes to their default values.

iii. Modify the attribute object to contain the desired attribute values.

iv. Pass a pointer to the attribute object when calling pthread_create.

v. Call pthread_attr_destroy to release the attribute object. The pthread_attr_t variable itself is not deallocated; it may be reinitialized with pthread_attr_init.

A single thread attribute object may be used to start several threads. It is not necessary to keep the thread attribute object around after the threads have been created. For most GNU/Linux application programming tasks, only one thread attribute is typically of interest (the other available attributes are primarily for specialty real-time programming). This attribute is the thread's detach state. A thread may be created as a joinable thread (the default) or as a detached thread. A joinable thread, like a process, is not automatically cleaned up by GNU/Linux when it terminates. Instead, the thread's exit state hangs around in the system (kind of like a zombie process) until another thread calls pthread_join to obtain its return value. Only then are its resources released. A detached thread, in contrast, is cleaned up automatically when it terminates. Because a detached thread is immediately cleaned up, another thread may not synchronize on its completion by using pthread_join or obtain its return value.

To set the detach state in a thread attribute object, use pthread_attr_setdetachstate. The first argument is a pointer to the thread attribute object, and the second is the desired detach state. Because the joinable state is the default, it is necessary to call this only to create detached threads; pass PTHREAD_CREATE_DETACHED as the second argument. The program "detached.c" given below creates a detached thread by setting the detach state thread attribute for the thread.

```
//Program "detached.c" - Skeleton Program That Creates a Detached
Thread

#include <pthread.h>

void* thread_function (void* thread_arg)

{

  /* Do work here... */

  }

  int main ()

  {

  pthread_attr_t attr;

  pthread_t thread;

  pthread_attr_init (&attr);

  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);

  pthread_create (&thread, &attr, &thread_function, NULL);

  pthread_attr_destroy (&attr);

  /* Do work here... */

  /* No need to join the second thread. */

  return 0;

  }
```

Even if a thread is created in a joinable state, it may later be turned into a detached thread. To do this, call pthread_detach. Once a thread is detached, it cannot be made joinable again.

Conclusion

In this activity we presented the thread concepts and explained various techniques and tools for creating and managing thread execution. In particular, thread creation using pthread_create function was presented. Further, various threads management techniques were highlighted.

Assessment

1. Practice the example code provided in this activity

Activity 3 - Memory Management

Introduction

Memory is the primary data storage area for computers. We call the basic memory unit a bit. A bit may contain two different values: either 0 or 1. Why do computers use binary arithmetic? It is the most reliable and efficient way to express data. Because, the digital information can be stored as voltage. We have to distinguish each different value. If we have more values (such as decimal system), it causes less separation between adjacent values. But with two values (binary system: 0 and 1) different values will be distinguished with maximum distance. Like a rule: To get the most distance we must use two points. The distance will be distance of the rule.

Memory consists of a number of cells which can store some number of bits. The memory is just a byte array. Each cell has a number to identify it, called its address. Programs refer addresses to reach memory. Adjacent cells have consecutive addresses. If memory has m cells, the cells will have addresses 0 to m-1. If CPU supports n bit, it can refer addresses from 0 to 2^n – 1. For example, Intel Pentium II is a 32-bit CPU and can address 4 GBytes of memory. Each cell stores an integer. An integer is n-bit number. It is 32 bit (4 bytes) if you have Intel Pentium II. Therefore, the maximum addressable memory size = n * 2^n. In recent years, nearly all manufactures have standardized on an 8-bit cell which is called byte. Bytes are grouped into words. A 32-bit CPU has 4-bytes/word. A CPU with 32-bit registers can holds 32-bits at a time. For that reason, registers that access to memory are also 32 bits. So it can point maximum to 11111111111111111111111111111111 in binary format (0xFFFFFFFF in hexal form). That is same with the 2^n – 1.

In the early years, computer memories were small and more expensive. Programmers were using a total memory size of only 4096 18-bit words for both user programs and operating system in PDP-1. So, the programmer had to fit his program in this small memory. Nowadays, computers have some gigabytes of memory but the modern programs need much more memory. To solve this problem, operating systems use secondary memories such as disk as main memory. In the first technique, the programmer divided the program up into a number of pieces called overlays. At the start of the program, first overlay was loaded into memory. When it finished, loads next overlay. Programmers must manage overlays between memory and disk. Programmer was responsible to find it from disk and load it to memory. This was a difficult undertaking for programmers.

In 1961, a group of researchers from Manchester established automatic overlay management system called virtual memory. Virtual memory is organized into "pages". A page is a memory unit typically a few Kbytes in size. It is mostly 4-Kbytes. You can learn page size by typing pagesize command in Linux/Unix. When a program references to an address on a page not present in main memory, a page fault occurs. After a page fault, the operating system seeks for the corresponding page on the disk and loads it onto main memory by using a page replacement algorithm such as LRU (Least Recently Used (LRU) - works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too).

That way programmer can start a program when none of the program is in main memory. When the CPU tries to fetch the first instruction of the program, it gets a page fault, because the memory doesn't contain any piece of the program in the main memory. This method is called demand paging. If a process in main memory has low priority or is sleeping, that means it won't run soon. In this case, the process can be backed up on disk by the operating system. This process is swapped out. The swap space is used for holding memory data. Processes use virtual addresses for transparency. They don't know about physical memory. CPU has a unit called Memory Management Unit (MMU) which is responsible for operating virtual memory. When a process makes a reference to a page that isn't in main memory, the MMU generates a page fault. The kernel catches it and decides whether the reference is valid or not. If invalid, the kernel sends signal "segmentation violation" to the process. If valid, the kernel retrieves the page process referenced from the disk.

Activity details

Memory Layout for a Process

Memory is an array of words. But it is not functional with this simple structure. Operating systems divide it into some pieces each one has its custom behaviour. For example, the kernel may protect a part of process memory against write and execute. In the process memory, each memory section which has different behaviours is called segment.

When a program is loaded into memory, it resides in memory as shown by the following memory layout:

```
+---------------------------+--------------------------+--------------
+---------------+---------------+

 |   Code segment(r+x)   |   Data segment(r+w)   |  BSS(r+w)   |
Heap(r+w)   |  Stack(r+w)   |

+---------------------------+--------------------------+--------------
+---------------+---------------+
```

A. Code(Text) Segment

This is the area in which the executable instructions reside. In the Linux/UNIX world, it is called as "text segment". It has an execute permission. Some old architecture allowed the code change itself. For that reason, the code segment had also the 'write' permission.

Suppose we have a function named func(), and addr points somewhere in the code segment. Because functions reside in code segment, therefore, addr = &func;

B.Data Segment

It contains initialized global variables declared by programmer. Global variables have a fixed area in memory where they will be defined at startup.

C.BSS

It contains uninitialized global variables.

D.Heap

It contains variables generated dynamically at runtime. Data segment holds variables which we create at compile time. So it is fixed in size. Often, the programmer needs to create variables at runtime. This is called dynamic memory allocation. Modern C libraries provide some functions to allocate area from heap like malloc().The free() function destroys variables which have been dynamically allocated from heap space.

Everything is unnamed in the heap. You cannot reach any variable directly by using its name. But you can reference indirectly using a pointer. The end of the heap is marked by a pointer called "break". When a program reference past the break, it will break. When the heap manager needs more memory, it calls brk() and sbrk() system calls. The brk() and sbrk() functions are used to change the amount of memory allocated in a process's data segment. They do this by moving the location of the `break'. The break is the first address after the end of the process's uninitialized data segment (also known as the `BSS').

```
NAME

              brk, sbrk

 -- change data segment size

LIBRARY

            Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>

#include <unistd.h>

    int brk(const void *addr);

void * sbrk(intptr_t incr);
```

The brk() function sets the break to addr. The sbrk() function raises the break by incr bytes, thus allocating at least incr bytes of new memory in the data segment. If incr is negative, the break is lowered by incr bytes. The current value of the break may be determined by calling sbrk(0).

E.Stack

Stack is a data structure which is accessed in last-in first-out order. There are two operations for stacks: To insert a new item, it must be PUSHed and to retrieve item, it must be POPped. SP(stack pointer) is a CPU register which points to the top of the stack.

Data resides from higher addresses to lower addresses in stack. A function call is the typical usage of stack. What happens after a function call? Consider the program example below:

```
10] i = 4;

11] func(i);

12] k = 2;

13] ...
```

Note that CPU always executes instruction which IP(instruction pointer) shows. In above code, IP is 10 before calling func(). It will be the address of func() when func() is called. And then IP will be 12 after func() exited. Before func() call, we must save address of the next code line. Because we'll return back after function exited and continue to execute program from next line. To store address of the next line, the program will use stack. After this step the program PUSHes function parameters (in above code, 'i' is the parameter) to stack, and then local variables of the function. When the function ends, it POPs local variables and function parameters orderly. Thus, only address of the next line (in above code, it is 12) remains in the stack. The return is a CPU specific instruction. After return, the address will be POPped from stack and will be assigned to IP. Hence the program will continue execution from line 12.

Memory Leak

A memory leak is where allocated memory is not freed although it is never used again. There are two common types of heap problems:

i. Freeing or overwriting data that is still in use will cause "memory corruption".

ii. Not freeing data which is no longer in use will cause "memory leak".

If the memory leak is in a loop, after a while the program will consume all of the memory. You will see that, your operating system is getting slower. A good programmer always frees allocated memory explicitly. Whenever he uses malloc(), puts a corresponding free() statement. Garbage collection (GC), also known as automatic memory management, is the automatic recycling of dynamically allocated memory. Garbage collection is performed by a garbage collector which frees memory that will never be used again. There are many ways for automatic memory managers to determine what memory is no longer required. In the main, garbage collection relies on determining which blocks are not pointed to by any program variables.

Garbage collection was first invented by John McCarthy in 1958 as part of the implementation of Lisp. Systems and languages which use garbage collection can be described as garbage-collected. Java, Prolog, Smalltalk etc. are garbage collected languages. C provides more control over program to programmer. For that reason it doesn't worry about freeing unused memory. All local variables in the stack will be freed and available for reuse after exit from its scope. But dynamic allocated variables will not be freed without a garbage collector. Since C doesn't usually perform garbage collection, programmers must be careful if they use malloc().

Why we need dynamic memory allocation? All variables declared statically at the compile time at stack, will be destroyed while functions are exiting (when the main function in the program will be exiting). But sometimes the programmer cannot know how much space the program needs. For example, the program reads spam words from a file and put them onto the memory. There may be 10 lines or 1,000 lines. If we assume a line can be maximum 32-bytes, then in the first case we need 320-bytes of memory and in the second case we need 32,000-bytes of memory. As result, the programmer doesn't know about memory requirements beforehand.

As a solution he can allocate 1000 word lines statically. But if we have 100 words, this will waste our memory. Or if the spam database is so big (for example 10,000 lines) the program won't work correctly.

```
char wordtable[1000][32];
```

The best solution is using dynamic memory. The program allocates 32-bytes for each line in a loop.

```
NAME

malloc, calloc, realloc, free, reallocf

-- general purpose memory allocation functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

void * malloc(size_t size);

void free(void *ptr);
```

The malloc() function allocates size bytes of memory. The free() function causes the allocated memory referenced by ptr to be made available for future allocations.

The following code illustrates a basic memory leak programming mistake:

```
01]
02] int main()
03] {
04] char* str;
05] char* tmp;
06]
07] str = malloc(sizeof(char)*32);
08] tmp = malloc(sizeof(char)*32);
09] fscanf(stdin, "%s", str);
10] tmp = str;
11]
12] do_something_with_tmp();
13] do_something_with_str();
14] free(str);
15] free(tmp);
16]
17] return 0;
18] }
19]
```

In line 10, the programmer is using tmp pointer to preserve address of str pointer. Then performing some processes with tmp. And then using str. In line 14 and 15, he is freeing pointers like a good programmer. But he forgets his assignment (tmp = str). Line 14 will succeed, but line 15 will fail if tmp still points at str. Because str cannot be de-allocated again. Another point is that, the programmer lost address of tmp allocated at line 8. So he never frees tmp.

Pointers

A pointer is a group of cells (often two or four) that can hold an address. If ch is a char and p is a pointer to this char:

```
                p                    &ch
 +-----+------+------+-----+-----+-------+-----+
 |  ...  |  ...  | &ch |........ |.......|  char  |  ...  |
 +------+-----+------+-----+-----+-------+-----+
```

The unary operator & gives the address of an object. The indirection or dereference operator * gives the "contents of an object pointed by a pointer". To declare a pointer we use dereference operator.

Consider the program "ptr.c" below which demonstrates use of pointers:

```
//Program ptr.c – Use of pointers
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6 char ch1 = 'g';
7 char ch2 = 's';
8 char *ptr;
9
10 printf("Char 1 is %c\n", ch1);
11 printf("Char 2 is %c\n", ch2);
12
13 printf("Address of Char 1 is %p\n", &ch1);
14 printf("Address of Char 2 is %p\n", &ch2);
15
16 ptr = &ch1;
17 ch2 = *ptr;
18
19 printf("Char 1 is %c\n", ch1);
20 printf("Char 2 is %c\n", ch2);
21
```

```
22 printf("Address of ptr is %p\n", ptr);

23

24 return 0;

25 }
```

We declared two char variables in line 6 and 7. Then we declared a char pointer in line 8. ptr is a variable which contains address of ch1 after line 16. ch2 is assigned to value "resides in address ptr points" in line 17. The address ptr points to address of ch1 because of line 16. So these two lines are equal to the following line:

```
ch2 = ch1;
```

We can make arithmetic operations on pointers.

```
ptr++;  /* Points next address. */

(*ptr)++;  /* Increments what ptr points to. */
```

ptr++ increments ptr to point next object. It doesn't increments ptr by 1 byte. Added value depends on the size of object. Management of dynamically created variables requires use of pointers. When the programmer need memory at runtime to store a data structure, he demands it by using malloc() function.

```
void * malloc(size_t size);
```

The malloc() function allocates size bytes of memory and returns a pointer to allocated space.

```
typedef struct rulelist rulelist;

struct rulelist {

int attr;

char ruleline[256];

rulelist *next;

};

rulelist *ll;

if((ll = (rulelist *) malloc(sizeof(rulelist))) == NULL) return -1;
```

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function if it is not a global variable. All local variables are accessible locally. If you pass it to function and then alter its value, it doesn't effect on variable of calling function.

```
void swap(int x, int y) /* WRONG */
{
int temp;
temp = x;
x = y;
y = temp;
}
swap(a, b);
But addresses are accessible from anywhere.
void swap(int *px, int *py) /* interchange *px and *py */
{
int temp;
temp = *px;
*px = *py;
*py = temp;
}
swap(&a, &b);
```

Pointer arguments enable a function to access and change objects in the function that called it. Pointer subtraction is also valid: if p and q point to elements of the same array, and p<q, then q-p+ 1 is the number of elements from p to q inclusive. This fact can be used to write yet another version of strlen:

```
int strlen(char *s)
{
char *p = s;
while (*p != '\0')
p++;
return p - s;
}
```

In its declaration, p is initialized to s, that is, to point to the first character of the string. In the while loop, each character in turn is examined until the '\0' at the end is seen. Because p points to characters, p++ advances p to the next character each time, and p-s gives the number of characters advanced over, that is, the string length.

IMPORTANT!!!

When a pointer is declared it does not point anywhere. Used this way program will crash. This is one of the famous security bug. You must set pointer variable to point somewhere before you use it. This can be by two ways:

i. Allocating memory for this pointer

ii. Assigning it to address of an existing variable.

```
int *Num;
*Num = 11;
```

The above code will crash. Correct code should be something like below:

```
int i = 5;
int *Num;
Num = &i;
*Num = 11;
```

Another way is allocating memory for the pointer from heap:

```
int *Num;
Num = (int *) malloc(sizeof(int));
*Num = 11;
```

Conclusion

In this activity we review the memory layout for processes and presented the memory management techniques in C programs. In particular, memory creation using the malloc function, the memory leak condition and use of pointers in C were highlighted.

Assessment

1. Write the following code which illustrates segments.

```
//mem.c
#include <stdio.h>
#include <stdlib.h>
int uig;
int ig = 5;
int func()
```

```
{
 return 0;
 }
 int main()
 {
 int local;
 int *ptr;
 ptr = (int *) malloc(sizeof(int));
 printf("An address from BSS: %p\n", &uig);
 printf("An address from Data segment: %p\n", &ig);
 printf("An address from Code segment: %p\n", &func);
 printf("An address from Stack segment: %p\n", &local);
 printf("An address from Heap: %p\n", ptr);
 printf("Another address from Stack: %p\n", &ptr);
 free(ptr);
 return 0;
 }
```

Execute the code and examine last line of output, which shows another address from stack:

a. Why &ptr is in the stack segment?

b. Why distance is 4 byte between local and &ptr?

c. Why &ptr is less than &local?

2. Consider the program "ptr.c" given in section 1.3.2.3. Execute this program and examine the output. Observe that after line 7, Char1 and Char2 both are equal to 'g' (value of Char1). Notice also that, the address of ptr is equal to address of ch1.

Unit Summary

In this unit we presented the process, thread and memory management concepts and highlighted the techniques and functions for creation and management. In particular, process creation functions were presented and the management and manipulation of processes using various system signals was highlighted. The thread creation function and various associated threads management techniques were described. Finally, memory creation function, the memory leak condition and use of pointers in C were explained.

Unit Assessment

Check your understanding!

Miscellaneous Exercises

Instructions

1. Write the following C Program which Forks a Separate Process, save it, execute it and note down it's output.

```
int main()
{
Pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-1);
    }
    else if (pid == 0) { /* child process */
          execlp("/bin/ls", "ls", NULL);
    }
    wait (NULL);
          printf ("Child Complete");
          exit(0);
    }
}
```

2.Consider the code skeleton given below for the bounded-Buffer problem with Shared-Memory Solution, which models the Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process. The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in = out; the buffer is full when in + 1 mod n = out. Write a complete program that uses threads to implement the insert and remove functions. Save it, execute it and note down it's output. Assume that the items are randomly generated integer numbers.

```
//Shared data
#define BUFFER_SIZE 10
Typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
// code for Insert function
while (true) {
   /* Produce an item */
      while (((in + 1) % BUFFER SIZE)  == out) ;  /*do nothing -- no
free buffers */
        buffer[in] = item;
        in = (in + 1) % BUFFER SIZE;
     {
//code for remove function
while (true) {
          while (in == out)  ;  // do nothing -- nothing to consume
         // remove an item from the buffer
         item = buffer[out];
         out = (out + 1) % BUFFER SIZE;
    return item;
```

3. Consider the program below. Describe the state of the memory after line 10, 12 and 14. What is the output of the program?

```
1   #include <stdio.h>
2   int main() {
3     int i = 0;
4     int j = 2;
5     int** p1;
6     int* p2;
7     int* p3;
8     p2 = &i;
9     p3 = &j;
10    p1 = &p2;
11    j = *p3 + 1;
12    *p2 = **p1 - 1;
13    p1 -= 2;
14    *(p1[2]) = 3 * *p2;
15    printf("i = %d, j= %d\n", i, j);
16  return 0;
17  }
```

Grading Scheme

As guided by the offering Institution Grading Regulation

Answers

mailto:njulumi@gmail.com

Unit Readings and Other Resources

1. Mark Mitchell, Jeffrey Oldham, and Alex Samuel; Advanced Linux Programming; Copyright © 2001 by New Riders Publishing; FIRST EDITION: June, 2001

2. Linux System Programming: Talking Directly to the Kernel and C Library, Robert Love "O'Reilly Media, Inc.", 14 May 2013

3. Linux Kernel Development, Robert Love, Pearson Education, 22 Jun 2010

Unit 4. Inter Process Communication

Unit Introduction

The previous unit on "processes," discussed the creation of processes and showed how one process can obtain the exit status of a child process. That's the simplest form of communication between two processes, but it's by no means the most powerful. Since the mechanisms don't provide any way for the parent to communicate with the child except via command-line arguments and environment variables, nor any way for the child to communicate with the parent except via the child's exit status. None of these mechanisms provides any means for communicating with the child process while it is not actually running, nor do these mechanisms allow communication with a process outside the parent-child relationship.

In this unit we describe the means for inter-process communication that circumvent these limitations. We present various ways for communicating between parents and children, between "unrelated" processes, and even between processes on different machines. Inter-process communication (IPC) is the transfer of data among processes. For example, a Web browser may request a Web page from a Web server, which then sends HTML data. This transfer of data usually uses sockets in a telephone-like connection. In another example, you may want to print the filenames in a directory using a command such as ls I lpr. The shell creates an ls process and a separate lpr process, connecting the two with a pipe, represented by the "I" symbol. A pipe permits one-way communication between two related processes. The ls process writes data into the pipe, and the lpr process reads data from the pipe.

In this unit, we will discuss three types of inter-process communication:

i. Pipes permit sequential communication from one process to a related process.

ii. FIFOs are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem.

iii. Sockets support communication between unrelated processes even on different computers.

These types of IPC differ by the following criteria:

i. Whether they restrict communication to related processes (processes with a common ancestor), to unrelated processes sharing the same filesystem, or to any computer connected to a network

ii. Whether a communicating process is limited to only write data or only read data

iii. The number of processes permitted to communicate

iv. Whether the communicating processes are synchronized by the IPC—for example, a reading process halts until data is available to read

Unit Objectives

Upon completion of this unit you should be able to:

- explain the inter-process communication concepts
- contrast the three types of IPC
- create each of three types of IPC
- apply IPC mechanisms to manage collaboration among processes

Key Terms

Inter-process communication (IPC): Is the transfer of data among processes

Pipes:A pipe permits one-way communication between two related processes

FIFOs: Similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem.

Sockets: Support communication between unrelated processes even on different computers.

Learning Activities

Activity 1 - Pipes

Introduction

A pipe is a communication device that permits unidirectional communication. Data written to the "write end" of the pipe is read back from the "read end." Pipes are serial devices; the data is always read from the pipe in the same order it was written. Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes.

In a shell, the symbol | creates a pipe. For example, this shell command causes the shell to produce two child processes, one for ls and one for less:

$ ls | less

The shell also creates a pipe connecting the standard output of the ls sub-process with the standard input of the less process. The filenames listed by ls are sent to less in exactly the same order as if they were sent directly to the terminal. A pipe's data capacity is limited. If the writer process writes faster than the reader process consumes the data, and if the pipe cannot store more data, the writer process blocks until more capacity becomes available. If the reader tries to read but no data is available, it blocks until data becomes available. Thus, the pipe automatically synchronizes the two processes.

Activity Details

Creating Pipes

To create a pipe, invoke the pipe command. Supply an integer array of size 2.The call to pipe stores the reading file descriptor in array position 0 and the writing file descriptor in position 1. For example, consider this code:

```
int pipe_fds[2];

int read_fd;

int write_fd;

pipe (pipe_fds);

read_fd = pipe_fds[0];

write_fd = pipe_fds[1];
```

Data written to the file descriptor read_fd can be read back from write_fd.

Communication between Parent and Child Processes

A call to pipe creates file descriptors, which are valid only within that process and its children. A process's file descriptors cannot be passed to unrelated processes; however, when the process calls fork, file descriptors are copied to the new child process. Thus, pipes can connect only related processes.

In the program "pipe.c" provided below, a fork spawns a child process. The child inherits the pipe file descriptors. The parent writes a string to the pipe, and the child reads it out. The sample program converts these file descriptors into FILE* streams using fdopen. Because we use streams rather than file descriptors, we can use the higher-level standard C library I/O functions such as printf and fgets.

```
//Program "pipe.c" - Using a Pipe to Communicate with a Child Process

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

/* Write COUNT copies of MESSAGE to STREAM, pausing for a second

between each. */


void writer (const char* message, int count, FILE* stream)

{
```

```
for (; count > 0; --count) {

/* Write the message to the stream, and send it off immediately. */

fprintf (stream, "%s\n", message);

fflush (stream);

/* Snooze a while. */

sleep (1);

}

}

/* Read random strings from the stream as long as possible. */

void reader (FILE* stream)

{

char buffer[1024];

/* Read until we hit the end of the stream. fgets reads until

either a newline or the end-of-file. */

while (!feof (stream) && !ferror (stream) && fgets (buffer, sizeof
(buffer),

stream) != NULL) fputs (buffer, stdout);

}

int main ()

{

int fds[2];

pid_t pid;


/* Create a pipe. File descriptors for the two ends of the pipe are

placed in fds. */

pipe (fds);

/* Fork a child process. */

pid = fork ();

if (pid == (pid_t) 0) {

FILE* stream;

/* This is the child process. Close our copy of the write end of
```

```
the file descriptor. */
close (fds[1]);
/* Convert the read file descriptor to a FILE object, and read
from it. */
stream = fdopen (fds[0], "r");
reader (stream);
close (fds[0]);
}
else {
/* This is the parent process. */
FILE* stream;
/* Close our copy of the read end of the file descriptor. */
close (fds[0]);
/* Convert the write file descriptor to a FILE object, and write
to it. */
stream = fdopen (fds[1], "w");
writer ("Hello, world.", 5, stream);
close (fds[1]);
}
return 0;
}
```

At the beginning of main, fds is declared to be an integer array with size 2.The pipe call creates a pipe and places the read and write file descriptors in that array. The program then forks a child process. After closing the read end of the pipe, the parent process starts writing strings to the pipe. After closing the write end of the pipe, the child reads strings from the pipe. Note that after writing in the writer function, the parent flushes the pipe by calling fflush. Otherwise, the string may not be sent through the pipe immediately.

When you invoke the command ls l less, two forks occur: one for the ls child process and one for the less child process. Both of these processes inherit the pipe file descriptors so they can communicate using a pipe. To have unrelated processes communicate, use a FIFO instead, as will be discussed in later sections on "FIFOs."

Redirecting the Standard Input, Output, and Error Streams

Frequently, you'll want to create a child process and set up one end of a pipe as its standard input or standard output. Using the dup2 call, you can equate one file descriptor with another. For example, to redirect a process's standard input to a file descriptor fd, use this line:

dup2 (fd, STDIN_FILENO);

The symbolic constant STDIN_FILENO represents the file descriptor for the standard input, which has the value 0.The call closes standard input and then reopens it as a duplicate of fd so that the two may be used interchangeably. Equated file descriptors share the same file position and the same set of file status flags. Thus, characters read from fd are not reread from standard input.

The program "dup2.c" given below uses dup2 to send the output from a pipe to the sort command (sort reads lines of text from standard input, sorts them into alphabetical order, and prints them to standard output). After creating a pipe, the program forks. The parent process prints some strings to the pipe. The child process attaches the read file descriptor of the pipe to its standard input using dup2. It then executes the sort program.

```
//Program "dup2.c" - Redirect Output from a Pipe with dup2

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main ()

{

int fds[2];

pid_t pid;

/* Create a pipe. File descriptors for the two ends of the pipe are

placed in fds. */

pipe (fds);

/* Fork a child process. */

pid = fork ();

   if (pid == (pid_t) 0) {

/* This is the child process. Close our copy of the write end of

the file descriptor. */

close (fds[1]);
```

```
/* Connect the read end of the pipe to standard input. */
dup2 (fds[0], STDIN_FILENO);
/* Replace the child process with the "sort" program. */
execlp ("sort", "sort", 0);
    }
    else {
/* This is the parent process. */
FILE* stream;
/* Close our copy of the read end of the file descriptor. */
close (fds[0]);
/* Convert the write file descriptor to a FILE object, and write
to it. */
stream = fdopen (fds[1], "w");
fprintf (stream, "This is a test.\n");
fprintf (stream, "Hello, world.\n");
fprintf (stream, "My dog has fleas.\n");
fprintf (stream, "This program is great.\n");
fprintf (stream, "One fish, two fish.\n");
fflush (stream);
close (fds[1]);
/* Wait for the child process to finish. */
waitpid (pid, NULL, 0);
    }
return 0;
}
```

popen and pclose

A common use of pipes is to send data to or receive data from a program being run in a subprocess. The popen and pclose functions ease this paradigm by eliminating the need to invoke pipe, fork, dup2, exec, and fdopen. Compare the program "popen.c" provided below, which uses popen and pclose, to the previous example program "dup2.c".

```
//Program "popen.c" - Example using popen

#include <stdio.h>

#include <unistd.h>

int main ()

{

FILE* stream = popen ("sort", "w");

fprintf (stream, "This is a test.\n");

fprintf (stream, "Hello, world.\n");

fprintf (stream, "My dog has fleas.\n");

fprintf (stream, "This program is great.\n");

fprintf (stream, "One fish, two fish.\n");

return pclose (stream);

}
```

The call to popen creates a child process executing the sort command, replacing calls to pipe, fork, dup2, and execlp. The second argument, "w", indicates that this process wants to write to the child process. The return value from popen is one end of a pipe; the other end is connected to the child process's standard input. After the writing finishes, pclose closes the child process's stream, waits for the process to terminate, and returns its status value.

The first argument to popen is executed as a shell command in a sub-process running /bin/sh. The shell searches the PATH environment variable in the usual way to find programs to execute. If the second argument is "r", the function returns the child process's standard output stream so that the parent can read the output. If the second argument is "w", the function returns the child process's standard input stream so that the parent can send data. If an error occurs, popen returns a null pointer. Call pclose to close a stream returned by popen. After closing the specified stream, pclose waits for the child process to terminate.

Conclusion

This activity presented a pipe as an IPC device that permits unidirectional communication. We described the constructs for creating a pipe, and how the pipe can be used for communication between Parent and Child Processes. Further, we showed how to redirect a process's Standard Input, Output, and Error Streams, and the use of the popen and pclose functions to eliminate the need to invoke pipe, fork, dup2, exec, and fdopen functions.

Assessment

1. Practice the example code provided in this activity

Activity 2 - FIFOs

Introduction

A first-in, first-out (FIFO) file is a pipe that has a name in the filesystem. Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other. FIFOs are also called named pipes.

You can make a FIFO using the mkfifo command. Specify the path to the FIFO on the command line. For example, create a FIFO in /tmp/fifo by invoking this:

```
$mkfifo /tmp/fifo

$ls -l /tmp/fifo

prw-rw-rw- 1 samuel users 0 Jan 16 14:04 /tmp/fifo
```

The first character of the output from ls is p, indicating that this file is actually a FIFO (named pipe). In one window, read from the FIFO by invoking the following:

```
$ cat < /tmp/fifo
```

In a second window, write to the FIFO by invoking this:

```
$ cat > /tmp/fifo
```

Then type in some lines of text. Each time you press Enter, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window. Remove the FIFO with this line:

```
$ rm /tmp/fifo
```

Activity Details

Creating a FIFO

Create a FIFO programmatically using the mkfifo function. The first argument is the path at which to create the FIFO; the second parameter specifies the pipe's owner, group, and world permissions (File System Permissions). Because a pipe must have a reader and a writer, the permissions must include both read and write permissions. If the pipe cannot be created (for instance, if a file with that name already exists), mkfifo returns –1. Include <sys/types.h> and <sys/stat.h> if you call mkfifo.

Accessing a FIFO

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it for reading. Either low-level I/O functions (open, write, read, close, and so on) or C library I/O functions (fopen, fprintf, fscanf, fclose, and so on) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
int fd = open (fifo_path, O_WRONLY);

write (fd, data, data_length);

close (fd);
```

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, "r");

fscanf (fifo, "%s", buffer);

fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of PIPE_BUF (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

Differences from Windows Named Pipes

Pipes in the Win32 operating systems are very similar to Linux pipes (Refer to the Win32 library documentation for technical details about these). The main difference for Win32 is that named pipes functions are more like sockets. Win32 named pipes can connect processes on separate computers connected via a network. On Linux, sockets are used for this purpose. Also, Win32 allows multiple reader-writer connections on a named pipe without interleaving data, and Win32 pipes can be used for two-way communication (Note that only Windows NT can create a named pipe; Windows 9x programs can form only client connections).

Conclusion

This activity presented a FIFO (also referred as "named pipes") as an IPC device that permits unrelated process communication. We described the constructs for creating and accessing a FIFO and highlighted the differences of Linux from Windows named pipes.

Assessment

1.Practice by making the example code provided in this activity into complete programs

Activity 3 - Sockets

Introduction

A socket is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines. Sockets permit communication between processes on different computers. Internet programs such as Telnet, rlogin, FTP, talk, and the World Wide Web use sockets. For example, you can obtain the WWW page from a Web server using the Telnet program because they both use sockets for network communications (Usually, you'd use telnet to connect a Telnet server for remote logins. But you can also use telnet to connect to a server of a different kind and then type comments directly at it).

To open a connection to a WWW server at www.codesourcery.com, use telnet www.codesourcery.com 80. The magic constant 80 specifies a connection to the Web server program running www.codesourcery.com instead of some other process. Try typing GET / after the connection is established. This sends a message through the socket to the Web server, which replies by sending the home page's HTML source and then closing the connection—for example:

```
$ telnet www.codesourcery.com 80

Trying 206.168.99.1...

Connected to merlin.codesourcery.com (206.168.99.1).

Escape character is '^]'.

GET /

<html>

<head>

<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
```

Activity Details

Socket Concepts

When you create a socket, you must specify three parameters: communication style, namespace, and protocol. A communication style controls how the socket treats transmitted data and specifies the number of communication partners. When data is sent through a socket, it is packaged into chunks called packets. The communication style determines how these packets are handled and how they are addressed from the sender to the receiver.

- Connection styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender.

A connection-style socket is like a telephone call: The addresses of the sender and receiver are fixed at the beginning of the communication when the connection is established.

Datagram styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions. Each packet must be labelled with its destination and is not guaranteed to be delivered. The system guarantees only "best effort," so packets may disappear or arrive in a different order than shipping. A datagram-style socket behaves more like postal mail. The sender specifies the receiver's address for each individual message.

A socket namespace specifies how socket addresses are written. A socket address identifies one end of a socket connection. For example, socket addresses in the "local namespace" are ordinary filenames. In "Internet namespace," a socket address is composed of the Internet address (also known as an Internet Protocol address or IP address) of a host attached to the network and a port number. The port number distinguishes among multiple sockets on the same host.

A protocol specifies how data is transmitted. Some protocols are TCP/IP, the primary networking protocols used by the Internet; the AppleTalk network protocol; and the UNIX local communication protocol. Not all combinations of styles, namespaces, and protocols are supported.

System Calls

Sockets are more flexible than previously discussed communication techniques. These are the system calls involving sockets:

socket—Creates a socket

closes—Destroys a socket

connect—Creates a connection between two sockets

bind—Labels a server socket with an address

listen—Configures a socket to accept conditions

accept—Accepts a connection and creates a new socket for the connection

Sockets are represented by file descriptors.

A.Creating and Destroying Sockets

The socket and close functions create and destroy sockets, respectively. When you create a socket, specify the three socket choices: namespace, communication style, and protocol. For the namespace parameter, use constants beginning with PF_ (abbreviating "protocol families"). For example, PF_LOCAL or PF_UNIX specifies the local namespace, and PF_INET specifies Internet namespaces. For the communication style parameter, use constants beginning with SOCK_. Use SOCK_STREAM for a connection-style socket, or use SOCK_DGRAM for a datagram-style socket.The third parameter, the protocol, specifies the low-level mechanism to transmit and receive data. Each protocol is valid for a particular namespace-style combination. Because there is usually one best protocol for each such pair, specifying 0 is usually the correct protocol. If socket succeeds, it returns a file descriptor for the socket. You can read from or write to the socket using read, write, and so on, as with other file descriptors. When you are finished with a socket, call close to remove it.

B.Calling connect

To create a connection between two sockets, the client calls connect, specifying the address of a server socket to connect to. A client is the process initiating the connection, and a server is the process waiting to accept connections. The client calls connect to initiate a connection from a local socket to the server socket specified by the second argument. The third argument is the length, in bytes, of the address structure pointed to by the second argument. Socket address formats differ according to the socket namespace.

C.Sending Information

Any technique to write to a file descriptor can be used to write to a socket. For further details, revisit the unit that presents Linux's low-level I/O functions and some of the issues surrounding their use. The send function, which is specific to the socket file descriptors, provides an alternative to write with a few additional choices; see the man page for information.

Servers

A server's life cycle consists of the creation of a connection-style socket, binding an address to its socket, placing a call to listen that enables connections to the socket, placing calls to accept incoming connections, and then closing the socket. Data isn't read and written directly via the server socket; instead, each time a program accepts a new connection, Linux creates a separate socket to use in transferring data over that connection. In this section, we introduce bind, listen, and accept.An address must be bound to the server's socket using bind if a client is to find it. Its first argument is the socket file descriptor. The second argument is a pointer to a socket address structure; the format of this depends on the socket's address family. The third argument is the length of the address structure, in bytes. When an address is bound to a connection-style socket, it must invoke listen to indicate that it is a server. Its first argument is the socket file descriptor. The second argument specifies how many pending connections are queued. If the queue is full, additional connections will be rejected. This does not limit the total number of connections that a server can handle; it limits just the number of clients attempting to connect that have not yet been accepted.

A server accepts connection requests from a client by invoking accept. The first argument is the socket file descriptor. The second argument points to a socket address structure, which is filled with the client socket's address. The third argument is the length, in bytes, of the socket address structure. The server can use the client address to determine whether it really wants to communicate with the client. The call to accept creates a new socket for communicating with the client and returns the corresponding file descriptor. The original server socket continues to accept new client connections. To read data from a socket without removing it from the input queue, use recv. It takes the same arguments as read, plus an additional FLAGS argument. A flag of MSG_PEEK causes data to be read but not removed from the input queue.

Local Sockets

Sockets connecting processes on the same computer can use the local namespace represented by the synonyms PF_LOCAL and PF_UNIX. These are called local sockets or UNIX-domain sockets. Their socket addresses, specified by filenames, are used only when creating connections. The socket's name is specified in struct sockaddr_un. You must set the sun_family field to AF_LOCAL, indicating that this is a local namespace. The sun_path field specifies the filename to use and may be, at most, 108 bytes long.

The actual length of struct sockaddr_un should be computed using the SUN_LEN macro. Any filename can be used, but the process must have directory write permissions, which permit adding files to the directory. To connect to a socket, a process must have read permission for the file. Even though different computers may share the same filesystem, only processes running on the same computer can communicate with local namespace sockets.

The only permissible protocol for the local namespace is 0. Because it resides in a file system, a local socket is listed as a file. For example, notice the initial s:

```
$ ls -l /tmp/socket

srwxrwx--x 1 user group 0 Nov 13 19:18 /tmp/socket
```

Call unlink to remove a local socket when you're done with it.

An Example Using Local Namespace Sockets

We illustrate sockets with two programs. The server program,"socket-server.c", creates a local namespace socket and listens for connections on it. When it receives a connection, it reads text messages from the connection and prints them until the connection closes. If one of these messages is "quit," the server program removes the socket and ends. The socket-server program takes the path to the socket as its command-line argument.

```
//Program "socket-server.c" -  Local Namespace Socket Server

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <unistd.h>

/* Read text from the socket and print it out. Continue until the

socket closes. Return nonzero if the client sent a "quit"

message, zero otherwise. */

int server (int client_socket)

{
```

```
   while (1) {
int length;
char* text;
/* First, read the length of the text message from the socket. If
read returns zero, the client closed the connection. */
if (read (client_socket, &length, sizeof (length)) == 0)
return 0;
/* Allocate a buffer to hold the text. */
text = (char*) malloc (length);
/* Read the text itself, and print it. */
read (client_socket, text, length);
printf ("%s\n", text);
/* Free the buffer. */
free (text);
/* If the client sent the message "quit," we're all done. */
if (!strcmp (text, "quit"))
return 1;
    }
}
int main (int argc, char* const argv[])
{
   const char* const socket_name = argv[1];
   int socket_fd;
struct sockaddr_un name;
   int client_sent_quit_message;
/* Create the socket. */
   socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
/* Indicate that this is a server. */
   name.sun_family = AF_LOCAL;
   strcpy (name.sun_path, socket_name);
```

```
    bind (socket_fd, &name, SUN_LEN (&name));
  /* Listen for connections. */
    listen (socket_fd, 5);
  /* Repeatedly accept connections, spinning off one server() to deal
  with each client. Continue until a client sends a "quit" message. */
    do {
  struct sockaddr_un client_name;
  socklen_t client_name_len;
  int client_socket_fd;
  /* Accept a connection. */
  client_socket_fd = accept (socket_fd, &client_name, &client_name_len);
  /* Handle the connection. */
  client_sent_quit_message = server (client_socket_fd);
  /* Close our end of the connection. */
  close (client_socket_fd);
     }
  while (!client_sent_quit_message);
  /* Remove the socket file. */
  close (socket_fd);
  unlink (socket_name);
  return 0;
  }
```

The client program, "socket-client.c", connects to a local namespace socket and sends a message. The name path to the socket and the message are specified on the command line.

```
//Program "socket-client.c" - Local Namespace Socket Client
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
```

```
/* Write TEXT to the socket given by file descriptor SOCKET_FD. */
void write_text (int socket_fd, const char* text)
{
/* Write the number of bytes in the string, including
NUL-termination. */
int length = strlen (text) + 1;
write (socket_fd, &length, sizeof (length));
/* Write the string. */
write (socket_fd, text, length);
}
int main (int argc, char* const argv[])
{
const char* const socket_name = argv[1];
const char* const message = argv[2];
int socket_fd;
struct sockaddr_un name;
/* Create the socket. */
socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
/* Store the server's name in the socket address. */
name.sun_family = AF_LOCAL;
strcpy (name.sun_path, socket_name);
/* Connect the socket. */
connect (socket_fd, &name, SUN_LEN (&name));
/* Write the text on the command line to the socket. */
write_text (socket_fd, message);
close (socket_fd);
return 0;
}
```

Before the client sends the message text, it sends the length of that text by sending the bytes of the integer variable length. Likewise, the server reads the length of the text by reading from the socket into an integer variable. This allows the server to allocate an appropriately sized buffer to hold the message text before reading it from the socket.

To try this example, start the server program in one window. Specify a path to a socket—for example, /tmp/socket.

```
$./socket-server /tmp/socket
```

In another window, run the client a few times, specifying the same sockets path plus messages to send to the client:

```
$./socket-client /tmp/socket "Hello, world."
$ ./socket-client /tmp/socket "This is a test."
```

The server program receives and prints these messages. To close the server, send the message "quit" from a client:

```
$ ./socket-client /tmp/socket "quit"
```

The server program terminates.

Internet-Domain Sockets

UNIX-domain sockets can be used only for communication between two processes on the same computer. Internet-domain sockets, on the other hand, may be used to connect processes on different machines connected by a network.

Sockets connecting processes through the Internet use the Internet namespace represented by PF_INET. The most common protocols are TCP/IP. The Internet Protocol (IP), a low-level protocol, moves packets through the Internet, splitting and rejoining the packets, if necessary. It guarantees only "best-effort" delivery, so packets may vanish or be reordered during transport. Every participating computer is specified using a unique IP number. The Transmission Control Protocol (TCP), layered on top of IP, provides reliable connection-ordered transport. It permits telephone-like connections to be established between computers and ensures that data is delivered reliably and in order.

Internet socket addresses contain two parts: a machine and a port number. This information is stored in a struct sockaddr_in variable. Set the sin_family field to AF_INET to indicate that this is an Internet namespace address. The sin_addr field stores the Internet address of the desired machine as a 32-bit integer IP number. A port number distinguishes a given machine's different sockets. Because different machines store multibyte values in different byte orders, use htons to convert the port number to network byte order. See the man page for ip for more information.

To convert human-readable hostnames, either numbers in standard dot notation (such as 10.0.0.1) or DNS names (such as www.codesourcery.com) into 32-bit IP numbers, you can use gethostbyname. This returns a pointer to the struct hostent structure; the h_addr field contains the host's IP number.

Note that DNS names are useful because it is easier to remember names than numbers. The Domain Name Service (DNS) associates names such as www.codesourcery.com with computers' unique IP numbers. DNS is implemented by a worldwide hierarchy of name servers, but you don't need to understand DNS protocols to use Internet host names in your programs.

The sample program "socket-inet.c" provided below illustrates the use of Internet-domain sockets. The program obtains the home page from the Web server whose hostname is specified on the command line.

```
//Program socket-inet.c - Read from a WWW Server
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
/* Print the contents of the home page for the server's socket.
Return an indication of success. */
void get_home_page (int socket_fd)
{
   char buffer[10000];
   ssize_t number_characters_read;
/* Send the HTTP GET command for the home page. */
   sprintf (buffer, "GET /\n");
   write (socket_fd, buffer, strlen (buffer));
/* Read from the socket. The call to read may not
    return all the data at one time, so keep
   trying until we run out. */
    while (1) {
number_characters_read = read (socket_fd, buffer, 10000);
if (number_characters_read == 0)
return;
/* Write the data to standard output. */
```

```
fwrite (buffer, sizeof (char), number_characters_read, stdout);
     }
}
int main (int argc, char* const argv[])
{
 int socket_fd;
   struct sockaddr_in name;
   struct hostent* hostinfo;
/* Create the socket. */
   socket_fd = socket (PF_INET, SOCK_STREAM, 0);
/* Store the server's name in the socket address. */
    name.sin_family = AF_INET;
/* Convert from strings to numbers. */
    hostinfo = gethostbyname (argv[1]);
    if (hostinfo == NULL)
return 1;
   else
name.sin_addr = *((struct in_addr *) hostinfo->h_addr);
/* Web servers use port 80. */
name.sin_port = htons (80);
/* Connect to the Web server */
if (connect (socket_fd, &name, sizeof (struct sockaddr_in)) == -1) {
          perror ("connect");
return 1;
      }
/* Retrieve the server's home page. */
    get_home_page (socket_fd);
     return 0;
}
```

This program takes the hostname of the Web server on the command line (not a URL—that is, without the "http://"). It calls gethostbyname to translate the hostname into a numerical IP address and then connects a stream (TCP) socket to port 80 on that host. Web-Servers speak the Hypertext Transport Protocol (HTTP), so the program issues the HTTP GET command and the server responds by sending the text of the home page.

For example, to retrieve the home page from the Web site www.codesourcery.com, invoke this:

```
$ ./socket-inet www.codesourcery.com

<html>

<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">

...
```

Note on Standard Port Numbers: By convention, Web servers listen for connections on port 80. Most Internet network services are associated with a standard port number. For example, secure Web servers that use SSL listen for connections on port 443, and mail servers (which speak SMTP) use port 25. On GNU/Linux systems, the associations between protocol/service names and standard port numbers are listed in the file /etc/services. The first column is the protocol or service name. The second column lists the port number and the connection type: tcp for connection-oriented, or udp for datagram. If you implement custom network services using Internet-domain sockets, use port numbers greater than 1024.

Socket Pairs

As we saw previously, the pipe function creates two file descriptors for the beginning and end of a pipe. Pipes are limited because the file descriptors must be used by related processes and because communication is unidirectional. The socketpair function creates two file descriptors for two connected sockets on the same computer.These file descriptors permit two-way communication between related processes. Its first three parameters are the same as those of the socket call: They specify the domain, connection style, and protocol. The last parameter is a two-integer array, which is filled with the file descriptions of the two sockets, similar to pipe. When you call socketpair, you must specify PF_LOCAL as the domain.

Conclusion

This activity presented socket as a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines. Various concepts related to sockets were described and demonstrated, including System calls involving sockets, servers and server's life cycle, internet-domain sockets, and socket Pairs.

Assessment

1. Practice the example code provided in this activity

Unit Summary

This unit presented three type of IPC communication devices, the pipes, FIFOs and Sockets. A pipe permits one-way communication between two related processes. FIFOs are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem. Sockets support communication between unrelated.

processes even on different computers. The concepts behind each communication device were described, and the constructs for creation and manipulation of the device highlighted.

Unit Assessment

Check your understanding!

Miscellaneous Exercises

Instructions

Consider the code skeleton given below that demonstrates the bounded-Buffer problem, which models the Paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. Write a suite of working programs, that uses Pipes, FIFOs, and Sockets IPC mechanisms respectively as solution method for the bounded buffer problem.

```
//Shared data

#define BUFFER_SIZE 10

Typedef struct {

    . . .

} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;


// code for Insert function

while (true) {

   /* Produce an item */

      while (((in + 1) % BUFFER SIZE)  == out) ;  /*do nothing -- no
free buffers */

        buffer[in] = item;
```

```
  in = (in + 1) % BUFFER SIZE;
      {
//code for remove function
while (true) {
           while (in == out)   ;  // do nothing -- nothing to consume
          // remove an item from the buffer
          item = buffer[out];
          out = (out + 1) % BUFFER SIZE;
     return item;
      {
```

The African Virtual University Headquarters

Cape Office Park

Ring Road Kilimani

PO Box 25405-00603

Nairobi, Kenya

Tel: +254 20 25283333

contact@avu.org

oer@avu.org

The African Virtual University Regional Office in Dakar

Université Virtuelle Africaine

Bureau Régional de l'Afrique de l'Ouest

Sicap Liberté VI Extension

Villa No.8 VDN

B.P. 50609 Dakar, Sénégal

Tel: +221 338670324

bureauregional@avu.org

2017 AVU