FreeBSD system programming

Authors:
Nathan Boeger (nboeger at khmere.com)
Mana Tominaga (manna at dumaexplorer.com)

Welcome to the FreeBSD System Programming book. Please note that this is a work in progress and feedback is appreciated! please note a few things first:

- We have written the book in a new format. I have read many programming books that have covered many different areas. Personally, I found it hard to follow code with no comments or switching back and forth between text explaining code and the code. So in this book, after chapter 1, I have split the code and text into separate pieces. The source code for each chapter is online and fully downloaded able. That way if you only want to check out the source code examples you can view them only. And if you want to understand the concepts you can read the text or even have them side by side. Please let us know your thoughts on this
- The book was ordinally intended to be published in hard copy form. However, this has changed and the book was converted using txt2html and some quick edits by hand. Therefore you might find that the HTML formatting is not consistent or hard to follow. We would like feedback on this. Please let us know what style would be the easiest to read.
- If you would like to participate please contact the authors.You can view the work online or choose to download the chapter as txt.
- Contributors:
  Francis Gudin (updates to Chapter 1, 2, 3, 4, 5, 6, 7, and 8)
  Ceri Davies (converting to DocBook)

Thank you for reading!

Table of Contents

- All source code

I. Introduction

BSD is a family of operating systems based on what originated as a distribution of changes and enhancements to AT&T's Unix operating system. The complex history of BSD itself is recounted by key kernel developer Marshall Kirk McKusick in his essay collected in "Open Sources: Voices from the Open Source Revolution" (O'Reilly, 1999). It is a comprehensive survey of the technical and historical details surrounding the birth of one of the most popular open source projects ever. The entire chapter is available online at: http://www.oreilly.com/catalog/opensources/book/kirkmck.html

In summary, the BSD family of operating systems dates back to the late 1970s, when AT&T owned Unix. Although Unix was proprietary, source code was available, which encouraged customers to make modifications to their systems. One such customer was the University of California at Berkeley's Computer Systems Research Group. Their version of Unix was known as the Berkeley System Distribution (BSD). BSD tapes were available for a nominal fee to anyone with a Unix source license. BSD received a big boost from the United States Department of Defense, who selected BSD as the base system for implementing TCP/IP and what became the Internet. The TCP/IP code was made freely redistributable via Networking Release 1 (Net/1) in 1989.

Originally, BSD required an AT&T Unix system and source license to build and run; but in time, BSD code had almost completely supplanted every subsystem of the Unix OS, turning into a standalone OS. A nearly complete BSD system, free of AT&T code, was released as Networking Release 2. Net/2, released in 1991, was only six files short of a complete system and was freely redistributable. A distribution called 386/BSD soon emerged, which had replaced those six files. Ongoing development of 386/BSD was limited; several forks quickly emerged, including NetBSD, FreeBSD, both free software, and the commercial BSDI.

In 1992, AT&T's Unix Systems Laboratories (USL)sued BSDI and the University of California, alleging the use of proprietary AT&T code and intellectual property in both BSDI and Net/2. A 1994 settlement resulted in the release of a modified version of Net/2 called 4.4BSD-Lite. Any project based on this release would be immune from litigation by USL, which by then was owned by Novell. As a result, NetBSD, FreeBSD, and BSDI all redeveloped their systems to use the 4.4BSD-Lite code base. After one more release in 1995, 4.4BSD-Lite Release 2, the CSRG closed down for good. BSD development, post-CSRG, has splintered off into several distinct varieties. Although the development model (one core; many iterations) resembles that of Linux, the various branches of the BSD family are much more distinct than the various Linux distributions. The best-known among these are FreeBSD, OpenBSD, NetBSD, and Darwin. They are all under active development, and are freely available with source code to all. FreeBSD development was originally centered on the x86 platform and aims for maximum performance. NetBSD aims for portability (runs on virtually every platform extant) and elegance, with an eye towards embedded systems. OpenBSD, a more recent fork of NetBSD, stresses security through a code audit process and integration of cryptography. The core of Apple's OS X, Darwin, is basically a BSD system on top of a Mach microkernel, which dates back to NeXTStep.

FreeBSD is a highly sophisticated operating system designed for the x86 compatible, DEC Alpha, and PC-98 architectures. It is optimized for Internet and intranet servers, with strong networking and security features. Additional ports to platforms such as PowerPC and Sparc64 are in the works. Among the free BSDs,

FreeBSD enjoys the widest deployment, with highly visible commercial customers such as Yahoo. Software packages are also numerous - at the time of writing, 7883 applications had been ported to FreeBSD. And, if there is a program available for Linux, FreeBSD's Linux compat support lets you run many Linux programs almost natively.

FreeBSD is often praised for its technical simplicity. For example, its installation program is widely regarded as the simplest Unix installer around, and the Ports Collection is an extremely elegant systemwhich combines pristine sources with FreeBSD-specific patches to build and install additional software. At the time of writing, almost 8000 software packages are available through the Ports Collection. Both NetBSD and OpenBSD have adopted the Ports Collection to manage additional software, though neither enjoy the sheer number of applications available. FreeBSD's development model exemplifies that of BSDs. It is highly centralized, and with a clear hierarchy. There's a group of more than two hundred developers called committers, who can make any change to the official FreeBSD source code whenever. The committers are chosen by the core team, a select group of the committers, chosen by from themselves. Elections are held every other year.

While FreeBSD focused development efforts on the 386, others started porting BSD to other platforms, notably the Macintosh, at Virginia Tech. Efforts soon expanded to Atari ST, Amiga, and PC platforms. As FreeBSD increasingly focused on a deep operating system optimized for i386, NetBSD began picking up the development efforts for other platforms. As of this writing, NetBSD supports the widest range of platforms for a BSD branch. And, its also notable for the range of exotic hardware supported, such as the short-lived Sega Dreamcast game console. NetBSD also enjoys commercial backing of Wasabi Systems, a company founded by one of the NetBSD developers, for porting NetBSD to new microprocessors and systems.

OpenBSD was founded by NetBSD SPARC port lead Theo de Raadt after a disagreement between de Raadt and the core team over future development direction. OpenBSD's first release came in October 1996, with a focus on security. Because OpenBSD is based in Canada (de Raadt lives in Calgary), it isn't encumbered by U.S. export regulations. As a result, cryptographic software is integrated into the base system. OpenBSD developers also created OpenSSH, a free implementation of the popular SSH encryption protocol. OpenSSH quickly displaced the commercial free-for-noncommercial-use SSH software. When you first install OpenBSD, it feels bare compared to FreeBSD; many networking features are turned off by default to avoid potential security holes.

Darwin, unlike the other three mentioned above, is licensed under the Apple Public Souce License. It compels that you forgo patent rights on any invention that uses the code. Apple only can build commercial software using it. It is notable in providing a wider user base for BSD, but most in its gorgeous Aqua GUI, which is years ahead of X Windows in terms of usability, customizability, and overall graphic sophistication.

Although this book focuses on FreeBSD, many of the concepts discussed will apply to other BSDs. (If there iss an exception, it will be noted.) The authors choose FreeBSD for not only its technical merits, but also for its excellent documentation. In addition to the usual man pages, The FreeBSD Handbook is a detailed, extensive how-to manual written by the developers and users.

II. Why BSD?

Many developers focus on just their particular BSD branch niche but a good chunk run in multiple circles. As such, you can pick and choose areas of interest, and development work may end up getting multiple layers of review, from other distributions that not only port the code but test it as well. For example, with security issues, often bug fixes are common across the BSDs. And even with the Linux world, there is constant borrowing, particularly for driver code. From a technology perspective, BSD embraces a number of sophisticated design considerations that make it a cutting edge operating system for admins and programmers

alike, most of which are outlined in other resources, most notably the comprehensive book Absolute BSD, by Michael Lucas. The most notable non-technical fact about BSD is its licensing: BSD is certified under a truly free license, so anyone can do whatever she wants with the code, licensing it and making a profit, or using portions as part of a larger work. What restrictions exist apply to the developer's scope of responsibility - a previous developer can't be held liable for errors, for example. As such, BSD code is all around us. Take the rise of the Internet, which relies heavily on TCP/IP, networking code born of BSD. If administrators had to reinvent the wheel, the Internet's ubiquity in our lives wouldn't have happened as quickly.

III. Who this book is for

This book is intended as a resource to system programming on BSDs. The reader should be familiar with basic programming in C or C++. The reader does not need to have system administration knowledge. System administration skills would be useful, however, for creating a good development environment. For Futher Reading:

FreeBSD: http://www.freebsd.org/
OpenBSD: http://www.openbsd.org/
NetBSD: http:www.netbsd.org/
A Brief History of FreeBSD: http://www.freebsd.org/handbook/history.html
The FreeBSD Handbook: http://www.freebsd.org/handbook

IV. About the Authors

C, C++, Java developer and senior sys admin, Nathan Boeger has been using FreeBSD since 1.x . Nathan has worked at various companies as sys admin and developer, including GetRelevant, Redline Networks, and Penguin Computing. He is currently getting ready for graduate school, and lives in New York, NY.

Mana Tominaga is a technology editor and writer, covering the Internet industry since late 1998. She has written and edited product reviews at Fawcette Technical Publications, for VBPJ and DevX.com, and at CMP Media for Web Techniques magazine, later known as New Architect. Mana has a degree in English literature from the University of California at Berkeley, and lives in San Jose, CA.

Chapter 1
FreeBSD System Programming

Prev top Next

1.1 FreeBSD's make

Make: to fit, intend, or destine by or as if by creating www.m-v.com

A common and essential utility for Unix software development, make is a nifty bookkeeping utility that keeps track of all file dependencies. Managing such project details as dependencies can be very time consuming and may even stall development. Dependencies tracking can become particularly difficult when multiple developers collaborate on a project. Indeed, the proper use of make can help speed up application development and also increase productivity.

Originally designed to manage the maintenance of application builds, you can use make to perform a variety of additional tasks by creating a sequence of Unix shell commands based on a target's dependencies. These dependencies can be defined in any number of ways - including source files to compile, needed libraries, shell commands and other targets.

Make has a litany of flavors, including GNU make and System V make. Not all features discussed below are present in every version of make, and the one to use will depend on your personal taste. We'll focus on the make that ships with FreeBSD (known also as bmake or pmake), epecially how to use it to compile and upgrade a FreeBSD system, known as make world. While we focus on FreeBSD make, everything discussed here will work across the various BSD distributions.

We'll cover basic file layout and the syntax of a Makefile first. If that's too basic for you, skip ahead to the working example at the end of the chapter. (Note that the code samples inline are meant to illustrate our discussion of make targets and dependencies; they're not usually working code.)

Of course, as with any utility, first consult the man page to get a formal overview, details, and a review of make's available command line options. And, as with any utility, the best way to learn make is to use it. Create some small source files (in any language) and then try some of the examples listed. We hope that by the end of this chapter you'll understand not just make's syntactical rules, but also how it works.

1.2 Makefile Layout

Generally, you use make by specifying a targets and its dependencies by giving it a Makefile to read. Once run, make searches for a file named Makefile or makefile, in that order. This Makefile is commonly put in the root directory for a project; to specify an alternative Makefile, enter it on the command line with the -f (filename) option.

```
make -f  OtherMakefile
```

1.3 Syntax

The structure of a Makefile includes four basic lines, all of which can be extended by adding a '\' character to the end and continuing on the next line (similar to shell programming). Comments begin with a pound sign '#', and run until the ending newline.

```
#######################################
# Simple Makefile with comment example #
#######################################

# when run, it will just echo hello
all:
   echo "hello"
```

To compile a project with make, be sure that you have a proper Makefile in your present working directory, and invoke it with one of the following commands:

```
bash$  make

bash$ make all

bash$ make <target name>
```

1.4 Targets

You can specify targets in any number of ways, but the most common ways are as object files or a project name. Project names should not contain white space or punctuation marks, though this is only a convention; some white space and punctuation marks are tolerated. The name must start at the beginning of a new line and end with a single colon (:), double colon (::), or an exclamation point (!).

```
myprog:
     <some commands to the compile myprog targer>

another::
     <some commands to the compile another target>

sample!
     <some commands to the compile sample target>
```

Follow these target names with the needed dependencies including names, variables or other targets. If you have many dependencies, use a '\' character followed by a newline to separate them. All dependencies must be defined within the Makefile or exist as external files, or make won't know how to complete that dependency.

Some examples are:

```
all: driver.cpp network_class.cpp file_io_class.cpp network_libs.cpp file_io_libs.cpp

all: myprog.o

myprog.o:
```

In the above example all and myprog.o are the targets to make. Notice that myprog.o is both a target and a dependency. Make will first go to myprog.o and execute its commands, then return to 'all' and execute its

commands. This sequence is make's basic foundation.

By convention the all: target is the highest of your targets, and is meant to be the root from which make will branch out to find what it needs to complete the all: target. The all: target is not a requirement, though. If it's not there, make will simply chose the first target listed, and only act on that one unless you specify a target on the command line. We suggest you use the all: target in projects that have a central application to be maintained or built; this is a popular convention for targets and will help you avoid mistakes and unnecessary tasks.

The sequence of the dependencies shown in the above example is a simple one. Here's an example of a more complex and flexible set of sequenced dependencies, sans the commands for the specific targets:

```
all: myprog.o lib

lib: lex

lex:

myprog.o: app.h
```

Notice in this example, the all: target has 2 dependencies: myprog.o and lib. Both are targets and make will first go to compile myprog.o. When make evaluates myprog.o, there's a dependency on app.h. Although app.h is not defined in this makefile, the file app.h is a header file located in the current directory.

Once the commands for myprog.o are completed, make will return to all:, then process the next dependency, in this case lib. The dependency lib itself has a dependency, lex, so make will first complete lex: before it returns to complete lib.

NOTE: As you can see, these dependencies can be very long and deeply nested. If you have a large Makefile, be sure to keep it well organized, with your targets in order.

1.5 Evaluation Rules

Dependencies are evaluated by strict rules which depend on how the target names end. Once make decides that the rules are met make will create that specific target by execution of the associated commands (i.e. compile that target). For example, the advantage of using the single colon: is more fine grained control over which targets would need to be compiled. That is, you could specify if a specific target file needed to be re-compiled every time or only if its source files are out-dated. The rules are based on the ending characters, as below.

If the target name ends with a single colon (:) that target will be created by the following two rules:

1. If the target does not exist, as with the target named all: in our example above, make will create it.
2. If any of the source files have a more recent timestamp than the current target. In our examples above the myprog.o would be made if the apps.h or the myprog.c file had a more recent timestamp. This is easy to force by using the touch command

   ```
   touch myprog.c
   ```

If the target name ends with a double colon (::) it will be created by the following three rules.

1. If any of the source files have timestamps that are more recent than the current target.
2. The target does not exist.
3. The target has no source files associated with it.

When targets end with bang (!) the target will always be created once make has created all of its needed dependencies.

You can only use wildcard expressions for targets, ?, *, and []as part of the final component of the target or source and only to describe an existing file. For example:

```
myprog.[oc]
```

Also, expressions using curly braces, {}, may not be needed to describe an existing file. For example:

```
myprog.[oc]
```

Also, expressions using curly braces, {}, may not be needed to describe an existing file. As in this example:

```
{mypgog,test}.o

# the expression above would match myprog.o test.o only
```

One final note: variable expansions are done in directory order and not alphabetically, as they would be in shell expansions. For example, if you have some type of dependency for your targets based on alphabetical order then the following expansion might not be correct:

```
{dprog,aprog,bprog,cprog}.cpp
```

1.6 Variables

Make's ability to use variable is of singular importance. For example, if you have a source file called a.c and for some reason you need to change its name to b.c. Normally, you would then have to re-write every instance of a.c in your make file to b.c. However if you used the following

```
MYSRC = a.c
```

Then you would only need to update that single line with the new name, as in:

```
MYSRC = b.c
```

You save time, hence meeting the first role of make: Project management.

Variables are referenced by using $(<varliable name>) or a single $ but this method is not widely used and therefore not recommended.

```
$(GCC)              = /usr/local/bin/gcc
```

There are four different classes of variables for make, which we list below in the order that they are searched. (Make's search continues until it reaches the first instance of a value.)

1. Local values: These are values assigned to a specific target.
2. Command line: Command line variables are given to make on the command line.
3. Global: Global variables are assigned inside the Makefile or any included Makefiles. You'll see these most frequently in a Makefile.
4. Environment: Environment variables are set outside the Makefile in the shell that is running make.

These variable can be defined inside the Makefile with the following five operators:

1. The equals sign "=" is the most commonly used operator, and similar to those in shell. The values are assigned directly to the variable. For example:

```
VAR = < value >
```

2. The plus equals sign "+=" is the append assignment, where the variable is assigned by appending to the current value. For example:

```
VAR += < value to append >
```

3. The question mark equals sign "?=" is a conditional assignment and will only assign the value if it not yet set. Conditionals are helpful in pre-appending values to a string value. For exampe:

```
VAR ?= < value if unset >
```

4. The colon equals sign ":=" is the expansion assignment. The value will be expanded before assigning it to the value given; the expansion is normally done when the value is referenced. For example:

```
VAR := < value to expand >
```

5. The bang equals sign "!=" is a shell command assignment. The value will be assigned the result of the command, after it is expanded and sent to the shell for execution. If the result has new lines they will be replaced with spaces.For example:

```
VAR != < shell command to execute >
```

NOTE: Some variables are defined by an external system wide Makefile (found in /etc/make.conf or /etc/defaults/make.conf). If you run into problems trying to set a variable from the environment, check these system wide file settings.
A full example is:

```
#####################
# Example make file #
#####################
CURDIR != pwd
CFLAGS ?= -g
CFLAGS += -Wall -O2

all:
    echo $CFLAGS
######################

bash$ CFLAGS="-g -Wall" make
```

In the example above, the CURDIR is set to the result from the shell command pwd. (Note that the backtick command (i.e. ``) for these assignments is not needed.) The CFLAGS value first will be set to -g only if it's unset, and then it will be append -Wall -O2 regardless of its current value.

1.7 Commands

Make is nothing without commands, and it is only by passing commands to make that it can perform its tasks. Make can only execute the commands and evaluate its success if the commands are successful, based on the shell's exit status evaluation. So, if the command fails and the shell returns an error, then make will quit with an error and stop at that point. Make can be thought of as just another command - it can't actually interact with other commands other than to run them.

Commands must be associated with a target, and any target can have multiple commands. For example:

```
# example for an all-install target

all-install:
   $(CC) $(CFLAGS)  $(MYSRC)
   cp  $(MYPROG) $(INSTALL_DIR)
   echo "Finished the build and install"
```

Each command must follow a target on a new line and must have a tab before the command line starts, as shown above.

For the most part, commands can be anything that is a valid shell command, and commands often include variables. For example:

```
CPP = -g++
CFLAGS = -Wall -O2

myprog.o
    $(CPP) $(CFLAGS) -c myprog.c
```

The example below tells make to compile myprog.c using the given values. These commands can be longer than one line and can be written to perform other tasks. This is important because compilers can take a number of command line options, environment settings, defines and so on, like so:

```
CLFAGS = $(LINK_FLAGS) $(LINK_LIBS) $(OTHER_LIBS) \
$(OPTIMIZER_FLAGS) $(DEFINES) $(NO_KERNEL) $(OBJS) \
$(CPU_TYPE_FLAGS) $(USE_MY_MALLOC) $(UDEFINES) \
$(SRC_FILE)  $(OTHER_SRC_FILES)
```

The example below tells make to remove all the object files, core files and the application, then to move the log file over, which is handy.

```
CPP = -g++
CFLAGS = -Wall -O2
APP = myapp
DATE != date +%m%d%y_%H_%M
LOG = debug


myprog.o:
```

```
      $(CPP) $(CFLAGS) -c myprog.c
      rm -f  *.o  *.core $(APP)
      mv  $(LOG).log  $(LOG)_$(DATE).log

clean
    rm -f *.o *.core $(APP)
    mv $(LOG).log $(LOG)_$(DATE).log
```

However if there's no log file, then make will error out and quit. To avoid this, precede the command with a "-" sign. By adding a minus sign before a command you tell make to execute that command and to ignore errors. (Make will still print the error though.) Thus, make will continue even if a command has an error after execution. For example:

```
clean
    -rm -f *.o *.core $(APP)
    -mv $(LOG).log $(LOG)_$(DATE).log
```

This would cause make to ignore the rm and mv errors if they occured.

You can also tell make to suppress the output of commands like the 'echo' shell command. Echo tells make to print out the entire command including the echo statement and then to run the command and print the string to the screen:

```
    echo $(SOME_DEFINED_STRING)
```

To avoid this place an at symbol "@" in front of the echo command, which tells make to print out only the string, like so:

```
     @echo $(SOME_DEFINED_STRING)
```

Both the "-" and "@" operators can be used with variables as well as commands as strings, but make sure you properly reference the variable command. Here's how to use the @ operator for commands:

```
ECHO = echo
MESSAGE = "Print this message"

msg::
  @$(ECHO) $(MESSAGE)
```

1.8 Conditional Statements (#if, #ifndef etc..)

If you're familiar with C and C++, you already know about conditional pre-processor directives. The versatile make has a similar feature. The conditional statements let you choose which parts of the Makefile are processed. These conditional statements can be nested up to a depth of 30 and can be placed anywhere inside the Makefile. Each statement must be prefixed with a period (.) and the conditional block must end with a .endif .

The conditional statements allow for logical operators, logical AND '&&', logical OR '||' and the entire statement can be negated with the '!' operator. The '!' operator has the highest precedence, followed by logical

AND and lastly logical OR. Parentheses can be used to specify the order of precedence. Relational operators are also available '>', '>=', '<', '<=', '==' and '!='. These operators can be used on decimal and hexadecimal values. For strings the '==', and '!=' operators can be used. If no operator is given, the value is compared with 0.

In the example below, if the VER variable has been assigned then the conditions are tested. Note that if the VER variable is not assigned then the last .else clause will evaluate to true and assign TAG the value of 2.4_stable.

```
.if $(VER) >= 2.4
  TAG = 2.4_current
.elif $(VER) == 2.3
  TAG = 2.3_release
.else
  TAG = 2.4_stable
.endif
```

Conditional statements can test variables or can be used with the following function style expressions.

Some of these have short hand notations. We list the short hand notations for compatibility issues. The long hand is much more implicit and widely understood, but takes longer to type.

When using short hand notations, you don't need the parentheses. Also, the short hand can be mixed with if / else statements along with other short hand statements:

make(< arg >) short hand [.ifmake, .ifnmake, .elifmake, .elifnmake]

```
.if make(debug)
  CFLAGS += -g
.elif make(production)
  CFLAGS += -O2
.endif
```

In the example above, make will take a target name as its argument. The value will be true if the target was given on the command line or if its the default target to make. The example below will assign the values to the CFLAGS according to the rules of the make() expression:

Here's the same code in short hand notation:

```
.ifmake debug
  CFLAGS += -g
.elifmake production
  CFLAGS += -O2
.endif
```

target(< arg >)

This will take a target name as its argument. The value will be true only if the target has been defined. There isn't a short hand notation for this expression. For example:

```
.if target(debug)
  FILES += $(DEBUG_FILES)
.endif
```

The example above will append to the FILES variable if the debug target returns true.

empty (< arg >)

This will take a variable for its argument and allows for possible modifiers. The value is true when the variable expansion would result in an empty string. There is no short hand for this expression. Also note that you don't need to reference the value when using the expression, VAR not $(VAR). For example:

```
.if empty (CFLAGS)
   CFLAGS = -Wall -g
.endif
```

defined(< arg >) short hand [.ifdef , .ifndef , .elifdef, elifndef]

The following example will take a variable for its argument. The value is true only if the variable has been defined:

```
.if defined(OS_VER)
  .if $(OS_VER) == 4.4
     DIRS += /usr/local/4.4-STABLE_src
  .endif
.else
  DIRS += /usr/src
.endif
```

Here's the short hand notation:

```
.ifdef OS_VER
. if $(OS_VER) == 4.4
  DIRS += /usr/local/4.4-STABLE_src
. endif
.else
  OS_VER = 3.2
  DIRS += /usr/src
.endif
```

As you can see, make allows for both nested conditionals and the define expression. Unlike C, indentations of if statements and variable assignments are not allowed. For visual clarity of your conditional blocks, you can have white space after the period, and before the if. Here's an example:

```
.if $(DEBUG) == 1
   $(CFLAGS) = -g
.     ifndef $(DEBUG_FLAGS)
         $(FLAGS) = $(DEBUG_FLAGS)
.     endif
.endif
```

exists(< arg >)

The example below shows how to use exists and how to add the conditional statements to a target. If the tmp directory exists then make will run the -rm tmp/*.o commands. As you can see in this example, the .if

statements will only be evaluated in the clean target; the commands run must follow the normal command syntax.

```
clean:
    -rm -f *.o *.core
.if exists(tmp)
    -rm tmp/*.o
.endif
```

1.9 System Make Files, Templates, and the .include Directive

One great feature of C is the manifest pre-processor directive, better know as the #include. This feature has been implemented in make, too. The difference is, if you want to include another Makefile then you should include it at the bottom and not at the top like in C, because variables are assigned in order. Makefiles in Makefiles can get confusing, the basic syntax to include a Makefile is simple. Note that the period (.) must precede the word include. The basic syntax is:

```
.include file name
```

To specify a Makefile located in the system make directory, use angle brackets:

```
.include <file name>
```

To specify files located in the current directory or one specified with the -I command line option, use double quotes, similar to the C #include:

```
.include "file name"
```

In the following example, if the project.mk has the variable CFLAGS defined, it will be overwritten by the newer declaration. This can cause a bit of trouble when including more than one Makefile.

```
.include "../project.mk"

CFLAGS = -g
```

FreeBSD systems have a number of system Makefiles that can be included, with routines for various tasks. On most FreeBSD systems you'll find them in the /usr/share/mk/ directory. However, there is a /etc/defaults/make.conf which can be overwritten by /etc/make.conf. (See man make.conf for more details.)

This is a brief listing of a few that are commonly used. If you are going to heavy kernel programming or applications porting, use and take advantage of Makefile. They're in the form of bsd.<type>.mk, where the <type> stands for what it is used for.

- bsd.dep.mk : This is a very helpfull include file that will handle Makefile dependencies
- bsd.port.mk : This is included and used when building ports of applications.

The good thing about using the .include directive is that you can break your project Makefile in pieces. For example, your a project could have a main make file that is included into all the other sub-make files. This main make file could contain the compiler variable along with its needed flags. That way each make file would not need to specify the compiler or needed flags, and simply reference the named compiler. These

commonly used pieces can then be used in other Makefile and modification to these routines will then be common across all Makefile.

1.10 Advanced Options

Advanced make options exist for further flexibility. We suggest a thorough read of the make man page for a deep understanding of make. The following options are the ones we use most often.

Local variables

Make has local variables that are specifically defined and only have scope within the current target. These seven are listed below with their system V compatible older notation. (The system V older notation isn't recommended; we only list it for backward compatibility.)

This variables value will be the name of the target:

.TARGET
old style notation: '@'

This variable contains the list for all the sources of this current target:

.ALLSRC
old style notation: '>'

This variable is the implied source for this target. Its value is the name and path of the source file for the target. (This will be demonstrated in the .SUFFIX section below.)

.IMPSRC
old style notation: '<'

This variable holds the list of sources that have been determined to be out of date:

.OODATE
old style notation: '?'

This variables value is the file name with out the suffix or path:

.PREFIX
old style notation: '*'

This variables value will be the name of the archive file:

.ARCHIVE
old style notation: '!'

This variables value is the name of the archive member:

.MEMBER
old style notation: '%'

When using these local variables in dependency lines, only .TARGET, .PREFIX, .ARCHIVE, and .MEMBER may have values for that target.

Another nifty directive is:

.undef <variable>

This is handy when you want to undefine a variable. Note that only global variables can be undefined. For example:

```
.ifdef DEBUG
.undef RELEASE
.endif
```

1.11 Transformation Rules (suffix rules)

Transformation rules specify how a target is to be created. You can use these rules - to save time in writing a rule for each object file. The syntax is simple: '.SUFFIXES: (suffix list)

Note that several different suffixes can share the same transformation suffix (.cpp, .cc, .c can all produce a .o transformation). If no suffixes are listed then make will delete all previous ones. This can be very confusing is you have multiple .SUFFIXES rules; we advise using only one and keeping it at the bottom of the Makefile. Consider what's listed inside the .SUFFIXES block as similar to targets to understand its structure:

```
.SUFFIXES: .cpp .c .o .sh

.c.o:
     $(CC) $(CFLAGS) -c ${.IMPSRC}

.cpp.o:
     $(CPP) $(CXXFLAGS) -c ${.IMPSRC}

.sh:
     cp ${.IMPSRC} ${.TARGET}
     chmod +x ${.TARGET}
```

These rules listed above will compile the C and C++ source. However for the .sh: rule will also tell make how to create the shell script as well. Note that listing a shell script as a target in this example will require it to be listed with out the .sh extension; however, the shell script must exist with a .sh suffix.

If we list the install_script as a target without the .sh suffix as a dependency, there should be a shell script called install_script.sh with the proper .sh suffix. That is, if a file can be listed as a target and as long as that file exists, then make will only create the targets that it deems out of date with that file, and make will not create that file. The example below illustrates this; for more information, see the example for apps.h:

```
all: install_script $(OBJS)
     $(CPP) $(CFLAGS) -o $(APP) $(OBJS) $(LINK)
```

1.12 Useful Command Line Options

Here is a list of a few command line options that are very handy to know and use. This is not a full listing, so refer to the man page for others.

-D <variable name to define>

This option will define a variable from the command line, which is handy when you have .ifdef statements in your Makefile. For example:

```
.ifdef DEBUG
  CFLAGS += -g -D__DEBUG__
.endif
```

Then when you run the command make -D DEBUG, make will set the proper CFLAGS and compile your application with your debug statements.

-E < variable name to override >

This option will override the Makefile variable assignment with the environment value instead. Before you use this option be sure to set the environment variable according to your shell. For example:

```
bash $ CFLAGS="-O2 -Wall" make -E CFLAGS
```

-e

Similar to its capitalized counterpart, -e will override all variables inside the Makefile with the environment values. If no environment variables are defined, however, then the values will be assigned normally.

-f <makefile to use>

This will allow you to specify the Makefile on the command line, helpful if you need multiple Makefiles. For example:

```
 bash$  make -f Makefile.BSD
```

-j < number of max_jobs >

This flag allows you to specify how many jobs make can spawn. Normally make spawns only one, but for a very large project and to make your hardware earn its keep, use four, as in:

```
make -j 4 world
```

If you exceed four, sometimes it'll take longer to execute, though it may be entertaining to some to watch the CPU spike with six or more jobs specified.

-n

Handy when debugging a Makefile, this option allows you to find out exactly which commands make will be running without make actually executing them. For large projects with many commands, redirect the output to an external file or all the commands will just blow by. Here's how:

```
bash $ make -n >> make_debug.log 2>&1
```

-V < variable name >

Using this option will print the variables value, based on the global context. Also make will not build any targets. You can specify multiple -V options on the command line, as in:

```
make -V CFLAGS -V LINK -V OBJS
```

1.13 A final example

The Makefile listed below, is an example of a reusable Makefile. When you include it, it will know how to compile C++ source files from the .SUFFIXES rules listed. It will also know how to install the application and clean up the development directory. By no means is this a very comprehensible Makefile, but a good example of creating a generic template style Makefile that can contain common routines for development. This will not only save time from having to retype these common rules over for every Makefile created, but will allow the developer to reuse known good routines as well.

```
#########################################################
#
# FILE: Makefile
#
# AUTHOR: Nathan Boeger
#
# NOTES:
#  This is a generic Makefile for *BSD make, you will
#  need to customize the listed variables below inside
#  the Makefile for your application.
#
# INSTALL_DIR = name of the directory that you want to install
#   this applicaion (Ex: /usr/local/bin/ )
#
# APP          = name of the application
#
# C_SRC      = C source files (Ex: pstat.c )
#
# CPP_SRC  = CPP source files (Ex: node.cpp)
```

```
#
#
# $Id: ch01.html,v 1.5 2004/08/10 14:41:39 nathan Exp $
########################################################

# Make the OBJ's from our defined C & C++ files
.ifdef CPP_SRC
OBJS            =       ${CPP_SRC:.cpp=.o}
.endif

.ifdef C_SRC
OBJS            +=      ${C_SRC:.c=.o}
.endif

# define the  Compiler. The compiler flags will be appended to
# if defined, else they are just assigned the values below
CPP             =        g++
CFLAGS          +=       -Wall -Wmissing-prototypes -O
LINK            +=       -lc

# Add a debug flag.
.ifdef DEBUG
  CFLAGS += -g
.endif

# Targets
all: ${OBJS}
    $(CPP) $(CFLAGS) -o $(APP) ${OBJS} $(LINK)

depend:
    $(CPP) -E -MM ${C_SRC} ${CPP_SRC}  > .depend

########################################################
#
#        INSTALL SECTION
#
# install will copy the defined application (APP) into the
# directory INSTALL_DIR and chmod it 0755
# for more information on install read MAN(1) install
########################################################
install: all
    install -b -s $(APP) $(INSTALL_DIR)

clean
     rm -f $(APP) *.o *.core

# SUFFIX RULES
.SUFFIXES: .cpp .c .o

.c.o:
       $(CPP) $(CFLAGS) -c ${.IMPSRC}
.cpp.o:
       $(CPP) $(CFLAGS) -c ${.IMPSRC}
```

The Makefile listed below is what you would need to create inside your project directory.

```
########################################################
#       PROJECT Makefile
#
# This is what the programs makefile would look like
```

```
# These are the only variables you will need to define
######################################################

APP             = myapp
C_SRC           = debug_logger.c
CPP_SRC         = myapp.cpp  base_classes.cpp
INSTALL_DIR     = /usr/local/bin/

# And include the template Makefile, make sure its
# path is correct.

.include "../../bsd_make.mk"
```

Prev top Next

Chapter 1
FreeBSD System Programming

Chapter 2
FreeBSD System Programming

2.1 Bootstrapping BSD

Bootstrap : to promote or develop by initiative and effort with little or no assistance <bootstrapped herself to the top> www.m-w.com

Bootstrapping a computer refers to the process of loading an operating system. The process is: initializing the hardware, reading a small amount of code into memory, and executing that code. This small bit of code then loads a larger operating system. Once the operating system is loaded, it then needs to create its entire environment. This process, called bootstrapping a computer, is a complicated, highly platform-specific process.

In this chapter we explore the i386 bootstrap process on FreeBSD in detail. The specific concepts and processes are similar to NetBSD and OpenBSD's bootstrap programs on i386 as well. Note that some Assembly code will be needed to actually accomplish the task of booting an i386 based system. However, we don't review the assembly code in detail and focus on the high level concepts mainly, so the discussion should make sense even if you aren't an expert.

Note: Although certain concepts discussed in this chapter, particularly the real and protected modes, do not exist in modern hardware architectures such as PPC and Alpha, the i386 BSD base is by far the largest and will continue to be (with the notable exception of Mac OS X), and should be relevant for many. If you're interested in the boot system details, you're likely to have custom kernel needs, custom filesystems, and device drivers. And, i386 architectures are widely used with embedded systems. Given the install base, the i386 platform and its issues will still continue to be relevant for the next few years. Even the newer 64 bit CPU's as far as we know will have the same boot process.

2.2 FreeBSD's Bootstrap Process

FreeBSD uses a three-phase bootstrap process. When you power on a computer or reboot, once the BIOS completes its system tests, it will load the first track from disk0 into memory. (Each process uses programs of 512 bytes, taking up exactly one block of a hard disk.) This first track is known as the master boot record (MBR) and this is the boot0 program, the first to be executed and loaded by the computer. The second program, boot1, is again fixed to 512 bytes and knows enough to read disk slice information and load boot2. Once boot2 is loaded it has the ability to boot the system directly or load the loader program, at a fixed size of 512 bytes, which is fairly sophisticated and designed to allow more control over how exactly the system boots.

boot0

The first program loaded from BIOS, boot0, a small program with a fixed size of 512 bytes, resides on the Master Boot Record (MBR). You can find the source for this program at /usr/src/sys/boot/i386/boot0. Of course the BIOS of most modern computers can be set to boot from different drives including CDROM, floppy, and IDE disks. For this chapter, we'll assume the computer is booting from the first hard disk, also known as disk drive 0, C: or, to the BIOS, as 0x80.

From this first disk's first sector, 512 bytes are read into the memory location of 0x7C00. After that, the BIOS will check for the number 0xAA55 at the memory location of 0x7DFE (the last two bytes of the boot block code). This location index number, 0xAA55 is so important in i386 that it's been given a suitable name - the magic number. That is, only if the magic number exists at the memory location of 0x7DFE will the BIOS transfer control to the memory location of 0x7C00 where the boot0 code lies.

This raises an important point when writing boot code on Intel i386 systems: Remember that the first memory location in your code (0x0) has to be an instruction. And, when the BIOS transfers control to the memory location of 0x7C00 that location must contain an instruction. This could be a simple jump to another location or the entry to the boot program's main routine. Otherwise you have no control over what the CPU is actually doing when the boot code is executed, and because the state of the registers is unknown as well, you can't rely on having proper segment or stack registers set. This small work must be done by the boot code because there is no operating system yet loaded; all I/O must be done using the BIOS routines. (The Intel CPU documentation includes a full list.)

After the boot0 program is loaded and control is transferred to it, it will set up its registers and stack information. Then, boot0 relocates itself into a lower memory location and jumps to the new address offset to its main routine. At this point the boot0 program must still be able to locate and boot other bootable disks and partitions. At its end, the boot0 code has a small listing of proper known bootable types which must contain the magic number in their last 2 bytes to be bootable.

Finally, when the boot0 has finished searching for the bootable disks and partitions, it will prompt the user with a choice. If no selection is made within a small time period or if a key is pressed, boot0 will load that next boot block into memory and transfer control to it. Again, this could be any operating system's boot code - you could set it up to load the bootstrap code for Linux or even DOS. For BSD, the next stage boot program is boot1.

boot1

Similar to boot0, boot1 is a very simple program and its total sizes is 512 bytes; it must also have the magic number located at its final 2 bytes. Its purpose is to locate and read the FreeBSD disk partition, then locate and load the boot2 program.

Although in most situations, boot1 is loaded by boot0, that order isn't necessarily the only available option. With flexible FreeBSD you have the option of using what's known as a dedicated disk. (What's telling is that it's more notoriously known as a dangerously dedicated disk.) A dedicated disk is a disk where the entire disk, or every sector of the BIOS, belongs to FreeBSD. Normally, you'd find a fdisk table, or a slice table, on a PC disk, and it's used to allow multiple operating systems to boot of a single PC disk. You can choose to use a dedicated disk and then boot straight off of boot1; the boot0 block does not need to be installed on the disk or used at all. Whatever method you choose to implement though, boot1 is a very important boot block and needs to be loaded.

Boot1 is loaded into the memory location of 0x7C00 and operates in real mode; the environment isn't set, and the registers are not in known states. The boot1 program has to set up the stack, define all segment registers, and use the BIOS interface for all I/O. Once boot1 is loaded into memory and control is transferred, it must contain an instruction as its first memory location (0x0). After all this succeeds, the boot1 program will read the system disks in search of boot2.

Once boot2 is located, the program must set up boot2's environment; boot2 is a BTX client (Boot Extender) and is a little more sophisticated than the previous boot0 and boot1. The boot1 program will need to load boot2 into the memory location of 0x9000 and its entry point is located at 0x9010. However, even after boot1

loads and transfers control to boot2, there's a routine that is used by the boot2 program contained in boot1. If you read the source for the boot1 program you will notice a function called xread. This function is used to read data from the disk using the BIOS. So, the location of boot1 in memory is very important and boot2 has to be aware of its location to function properly.

boot2

So far we've loaded two boot blocks and one large program into memory, transferred control twice both times resetting up a small environment (stack, segment registers etc..), and performed some limited I/O using the BIOS. We still haven't reached the point of loading an operating system yet. If you ever watch your computer screen during a FreeBSD boot, you'll maybe see F1 and that cute spinning line of ASCII so far. You might not think it's all that impressive, but it's the exact, precise nature of Assembler code that makes the boot process seem so elegant and effortless.

Now on to the final bootstrap process, boot2. This final stage is simple and one of two things can happen: the boot2 program loads the loader (we discuss this in the following section) or, the boot2 program loads a kernel directly and boot without using the loader program at all. If you've ever interrupted boot2 program while it's loading, you may have have seen this, which boot2 prints to the screen:

```
>> FreeBSD/i386 BOOT
Default: 0:ad(0,a)/boot/loader
boot:
```

If you press enter, boot2 will simply load the default loader, as listed. However, if you just type in "boot kernel" then it will load up the kernel (/kernel) and boot. You can, of course, change these default values. If you want to find out more read the documentation for boot(8).

We mentioned earlier that boot2 is a BTX client (Boot Extender). What does that entail? The BTX provides is a basic virtual 86 addressing environment. A discussion on the history of memory addressing on Intel hardware is in order.

So far we've avoided mention of memory addressing schemes, which can be confusing because Intel CPUs suffer from legacy issues, and boot code design is usually left to those developers who absolutely need to write it. Unless you are porting a system to a new architecture so your code is completely platform dependent, usually, a programmer will never be tasked with writing a boot loader.. However the boot process is very important to developers who need to write device drivers or other kernel related programming. This is where some developers will encounter the BTX loader.

Starting from around the 8088 until the 80186, Intel processors had only one way to address memory, called real mode. These early CPUs had whopping 16 bit registers and 20 bit memory addresses. The question then arises, how do you make a 20 bit address in a 16 bit register? The answer was, to use two 16 bit registers, with one register serving as a base the other as an offset to this base. The base register is shifted left 4 bits and thus when the two are combined an astounding 20 bit address can be calculated. With all these nifty segment registers and bit shifting the early Intel processors could address a total of 1 megabyte. Today this would not even be large enough to hold a Word document, as a bloated example.

Once the 80386 rolled around, addressing 1 megabyte was not enough; users demanded more memory and programs started to use more memory. A new addressing mode called protected mode was devised. The new protected mode allowed for addressing of up to 4Gigs of memory.

Another advantage of this new scheme was that it was easier to implement for assembly programmers. The main difference is that your extended registers (these are the same 16 bit registers that since the 386 are now 32 bit) can contain a full 32 bit address, even while your previous segment registers are now protected. The program cannot write to them nor read them. These segment registers are now used to locate your real address in memory and this process includes checking bits for permission (read, write etc..) and involves the MMU (memory management unit).

Now back to the BTX client issues. What advantage does this give us to use this BTX program? Simple: flexibility. BTX provides enough services so that a small program with a nice interface could be written and allow for greater flexibility in loading the kernel. On FreeBSD systems that would be the loader. From the next section you will see just how nice and flexible the loader program really is. So for the rest of this section we'll cover basic BTX services.

The BTX services can be categorized into two basic groups. The first group is system services provided by direct function calling (similar to system calls). The other group is services, which are environment services and are not directly called by the client. These services are similar to an operating system; however the BTX program operates as a single task environment.

The BTX services provided by direct calls consist of two system calls, exit and exec. When exit is called the BTX loader terminates and the system is rebooted. The final system call exec will transfer control to the provided address. This transfer of control is done in Supervisor mode and the new program can leave the protected CPU mode.

The environment services the BTX loader provides are very basic. The BTX loader handles all hardware interrupts. These interrupts are then sent to the proper BIOS handlers. BTX also emulates the BIOS extended memory transfer call. And finally several Assembler instructions are emulated. These are pushf, popf, cli, sti, iret, and hlt.

A final note of caution: All programs written to run in the BTX environment will need to be compiled and then linked with the btxld. For more information please read man pages for the BTX Loader.

2.3 Loader

The final boot stage consists of the loader. The loader program is a combination of standard commands (referred to as "built in commands") and a Forth interpreter (which is based on ficl). The loader will allow the user to interact with how the system boots or allow for system recovery and maintenance. From the loader the end user can choose to load or unload kernel or kernel modules. The user can also set and un-set specific variables, such as rootdevice and module_path. These can also be changed in the /boot/loader.conf. The default file the loader reads is located in /boot/defaults/loader.conf. This default file also contains many of the available options. Both of these files are constructed similarly to the /etc/rc.conf file.

The loader program is very useful for kernel and device driver debugging. From the loader you can tell the kernel to boot with the debugger (ddb) enabled. Or, you can load a specific kernel module for device driver testing. If you're going to be writing any kernel modules or device drivers you should read all the documentation on the loader. First start with the man page and then review all the options contained in the /boot/loader.conf. The loader program could be very useful down the road when you need to extend your BSD system or diagnose a kernel crash.

2.4 Beginning Kernel Services

We're finally at the stage when the kernel is loaded into memory and control of the CPU is transferred to it. Once the kernel is loaded it needs to run through its initialization and prepare the system for multitasking. This initialization includes three main components. The first two are machine specific and written in a combination of assembly and C. These first two stages prepare the system and initialize the CPU Memory Management Unit (MMU, if it exists) as well as handle initialization of hardware. The final stage entails setting up the basic kernel environment and getting the system ready to run process 1 (/sbin/init). These first two stages are highly platform dependant. Because every architecture has specific needs, we'll provide a high level overview for these two first stages and later in the book when we cover device drivers we'll go into these concepts in detail.

Stage 1 & 2 kernel assembly and C start-up routines

Although once loaded the kernel will assume nothing about the system, the loader program does pass some information to the kernel, such as the boot device and the boot flags. In addition, the kernel must create its environment and prepare the system for process 0 (explained below). These tasks include CPU detection, creation of a run-time task, and detection of memory amount.

The CPU identification is an important step. Because each platform can have multiple different types of CPUs (i386 being one of them), not all CPUs will support the same features. For example, take the MMX instruction set. Although it's not that important of a feature for the kernel, the floating point unit is, so if this feature is not supported on that type of CPU then it will have to be emulated in software. This is true for all other unsupported features as well as know bugs or idiosyncracies with the CPU.

The second stage will initialize the system's hardware and memory. This includes probing for hardware, initializing I/O devices, allocating memory for kernel structures, and creating the system message buffer. This stage is what you see during the boot screen, with lists of hardware flashing by. Traditionally in BSD, this stage is initialized by calling the function cpu_startup().

Stage 3 and process 0

Once the function cpu_startup() returns, the kernel will need to create process 0. This process 0 is commonly known as the swapper. If you run the ps command you will see it in action. However it really does not exist in the sense that there's no such binary named swapper associated with this process. This is true for these four other important processes found on a modern FreeBSD system: pagedaemon, vmdaemon, bufdaemon and syncer. To avoid complications, we'll just say that these processes are part of the VM subsystem; we discuss them in the process chapter. What's important to understand is that they are created during boot by the kernel and that they are not binary programs in the filesystem, and are fairly platform independent. They are written in C and are started after the beginning platform environment is initialized.

init and system shell scripts

Finally after all of that assembly and platform-dependant code gets executed, the kernel finally executes the first real program /sbin/init. This program is quite simple and is fairly small (on FreeBSD, it totals about 1,700 lines). As we discuss in the chapter on BSD processes, this is the one process that all processes are descendant from. The strength of the design is, because the /sbin/init program is just a binary in the file system, you can write a custom version. The main goal on start-up for /sbin/init is to run the system start up scripts and prepare the system for multiuser mode. Be aware of signals: the /sbin/init process should be able to handle signals with some grace, or your system could end up in a strange state of the /sbin/init program and crash on boot. Also during runtime /sbin/init can be sent signals to perform certain tasks. For examplee, if you want to tell the system to not spawn a process for a specific terminal, as listed in /etc/ttys, you can mark the desired

terminal off and run the following which will in effect have init read the /etc/ttys and only spawn processes for the terminals listed as on:

```
bash$ kill -HUP 1
```

Note that unless you're careful, you could end up with a system that you can't log into! (Look at the chapter on signals for more details.)

The init program will on boot set up the system for multiuser mode. It's quite a feat, involving tedious tasks such as starting every daemon and setting network information. On Unix systems, a few ways to perform these tasks are available, mainly involving shell scripts. On some versions of Linux and System V systems /etc/rc<n>.d/ directories that correspond to the certain runlevels that the scripts should be started on are available. However the BSDs have a much simpler method. These are the rc scripts found in /etc/.

These scripts normally should not be edited; instead, set the values from /etc/rc.conf. With custom installations such as PicoBSD you might have to create your own scripts; PicoBSD is highly diskspace conscious, and has specific filesystem needs. One important note, the /usr/local/etc/rc.d/ directory is special. This directory is special in the sense that every executable found in this directory with a .sh will be executed after the /etc/rc scripts. For good system admins this directory replaces the older /etc/rc.local file. (The /etc/rc.local was the older method of running custom scripts or programs at the very end of the system start up.

The BSD rc scripts include the following notable ones listed below:

```
/etc/rc - main script and first to be called. Mounts the file system and then runs all the need
/etc/rc.atm - used to configure ATM networking
/etc/rc.devfs - set up /dev/ permissions and links
/etc/rc.diskless1 - first diskless client script
/etc/rc.diskless2 - second diskless client script
/etc/rc.firewall - used for firewall rules
/etc/rc.firewall6 - used for ipv6 firewall rules
/etc/rc.i386 - intel system specific needs
/etc/rc.isdn - isdn network settings
/etc/rc.network - ipv4 network settings
/etc/rc.network6 - ipv6 network settings
/etc/rc.pccard - for laptop pc card controlers
/etc/rc.serial - set up serial devices
/etc/rc.sysctl - used to set sysctl options at boot time
/usr/local/etc/rc.d/ - general directory with custom start up scripts
```

Here is an example:

If you want to start rsync as a daemon then this script (although very simple) will do it:

```
#!/bin/sh

if [ -x  /usr/local/bin/rsync ] ; then
         /usr/local/bin/rsync --daemon
fi
```

This will first check to see if the rsync program exists then run it in daemon mode.

Chapter 2
FreeBSD System Programming

Chapter 3
FreeBSD System Programming

Code samples for this chapter: here

3.1 Process and Kernel Services

process: a series of actions or operations conducing to an end; especially : a continuous operation or treatment especially in manufacture www.m-w.com

The preceding chapter covered the boot process for BSD systems. Now, let's look at what happens once the kernel has booted and is running. Like most other modern operating systems, BSD is a multiuser, multitasking operating system, meaning it supports multiple users using system resources, each running different multiple processes. The concept of a 'process' offers a useful abstraction that comprises all the activities managed by an operating system. The term was first introduced by Multics designers during the 1960s, as a general encompassing term - as opposed to a 'job' - in a multiprogramming environment.

3.2 Scheduling

A process is a single instance of a running program. For example, when you run Netscape on a BSD system it creates a process while it's being executed. If you have three users logged into a BSD system all running the same Netscape program all at the same time, each user will have his or her own instance of Netscape, independent of the others. BSD systems can support many such processes at once. Every process will have associated with it a Process Identifier (PID). These processes will need resources and might have to access devices such as external storage.

When multiple processes are running on a system, the illusion that they are all executing at the same time is handled by the operating system. Assigning priorities to processes is managed by scheduling algorithms unique to the operating system; this area of computer science is extensive, and highly specialized; see the Resources section for more information.

The operating system actually moves executing processes in and out of the CPU(s). This way, each process gets a specific amount of time in execution on the CPU(s). This amount of time is called a 'slice'. The slice length is almost entirely determined by the kernel's scheduling algorithm. An adjustable value of this algorithm is the 'nice' value, which offers programmability for processes to specify the execution priority. These priority values are as follows:

Nice values	Priority
-20 - 0	Higher priority for execution
0 - 20	Lower priority for execution

Note: A higher nice value yields a lower priority, which may seem counterintuitive. However, consider this calculation:

(scheduling algo calculation) + (nice value)

Simple math will show you that adding a negative value to a positive one will result in a lower number, and as such, when these numbers are sorted, the lower ones will all come at the front of the execution queue.

The default nice value for all processes is 0. The executing process can raise its nice value (that is, lower its priority), but only processes running as root can lower them (that is, raise its priority). BSDs provide two basic interfaces to manipulate and retrieve these values. These are:

```
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

and

```
int nice(int incr);
```

The nice function will set the calling process' nice value to the incr passed as its parameter. Note that the nice() function is rather obsolete because it's not very flexible and the nice function is really just wrapped by the newer functions (get and setpriority). The preferred method is to use setpriority().

Because a valid priority can be -1, the return value from getpriority should not be regarded as success or failure for the program execution - instead, check if errno (error number) has been set. If so, clear it prior to calling getpriority. (For more information on errno routines, see the man(2) pages on errno and read the example program, nice_code.c for an example.) Also setpriority and getpriority can set or get priority value for external processes, by setting the 'which' and 'who' parameters, discussed later.

The example program, nice_code.c demonstrates retrieving and setting the nice value for the current process. If it is executed by root and given a command line value of -20, the system will seem to stop responding. This is because this program has the highest priority that can be set, and will dominate the system. Use caution and have much patience if setting the priority below 0 is attempted; depending on the CPU, full execution can take upwards of twenty minutes. It's recommended that you run the program using the time command, as below:

```
bash$ time nice_code 10
```

Then, adjust the command line value. That way how long the process took to execute is evident. For example, this adjusts the value below 0:

```
$ time nice_code -1
```

This makes the value large:

```
bash$ time nice_code 20
```

Also, try running the program as any other non-root user, and attempt to set the priority below 0. The system should deny this; only processes running as root can lower their nice value. (Because Unix systems are multi-user and each process should get a fair amount of time on the CPU, only root can change the process priority to values below 0 to avoid users from monopolizing the CPU resources by lowering their priority so that only their process would be executed.)

3.3 System Processes

The concept of 'protected mode' was introduced in the previous chapter. In short, this is a special mode in modern CPUs that allows, amongst other things, for the operating system to protect memory. Given this, there are two such modes of operation. The first is kernel-land, meaning that the process will be executing inside the kernel's memory space and hence operates within the kernel's privileged protected mode in the CPU. The second is userland, referring to any proves that's executed and doesn't operate in the kernel's protected mode.

This distinction is very important, because any given process uses resources in both kernel and userland. These resources are in various forms, such as kernel structures that account for the process, allocated memory within the process, open files, and other execution context.

On a FreeBSD system, a few critical processes aid the kernel in performing its tasks. Some of these processes are completely implemented and run within kernel space, while others run in userland.These processes are as follows:

PID	Name
0	swapper
1	init
2	pagedaemon
3	vmdaemon
4	bufdaemon
5	syncer

All the processes listed above, with the exception of init, are all implemented within the kernel. That means there is no regular binary program for these processes. These processes only resemble userland processes and because they are implemented within the kernel, they operate within the kernel's privileged mode. Such architecture results from various design considerations. For example, the pagedaemon process, which prevents thrashing, is only awakened when the system is low on memory. So if the system has lots of free memory then it never needs to be awakened. Thus, the advantage for running the pagedaemon in userland is that it can avoid using the CPU unless its really necessary. However it does add another process to the system and thus will need to be scheduled. But the scheduler calculations are very small and thus almost insignificant.

The init process is the mother of all processes. Every process, other than processes that operate in the kernelland privileged mode, are descendant from init. Also, zombies, or all processes that are abandoned, are then inherited by init. It also performs some administrative tasks also, including the spawning of gettys for the system's ttys, and executing an orderly shutdown of the system.

3.4 Process Creation and Process IDs

As mentioned above, when a program is executed, the kernel assigns it a unique PID. The PID is a positive integer and its value will range from 0 - PID_MAX, depending on the system. (For FreeBSD, the /usr/include/sys/proc.h has PID_MAX set at 99999.) The kernel will assign the PID value using the next sequential available PID. So, when the PID_MAX is reached, the values will loop around. The looping is important when using the PID for anything other than current process accounting.

Every process that runs on the system is created by another process. This is done by a few system calls that are discussed in the next chapter. When a process creates a new process, the original process is referred to as

the parent and the new process is referred to as the child. This parent/child relationship provides an excellent analogy - every parent can have multiple children and parent processes are descendent from another process. Processes can retrieve their own or their parent's PID with the getpid function.

Processes can also be grouped into process groups. These process groups are identified by a unique grpID. Process groups were introduced to BSD as a mechanism for shells to control jobs. Take this example:

```
bash$ ps auwx  | grep root  | awk {'print $2' }
```

The programs, ps, grep, awk, and print, all belong to the same process group. This allows all the commands to be controlled by referencing a single job.

A process can get its group ID by calling getpgrp or getpgid:

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

     pid_t   getpid(void);
     pid_t   getppid(void);
     pid_t   getpgrp(void);
```

All of the functions listed above are guaranteed to be successful. However, as the FreeBSD man pages strongly suggest, the PID should not be used to create a unique file. This is due to the fact that the PID is guaranteed to be unique only at the time of creation. Upon exit, the PID value is returned to the pool of unused PIDs and, will be reused at some point (of course, provided that the system stays running).

A simple source for getting these values is listed in proc_ids.c.

If you run the program like so:

```
bash$ sleep 10 |  ps auwx |  awk {'print $2'}  | ./proc_ids
```

And in another terminal run:

```
bash$ ps -j
```

Only one shell should run for these commands and each will have the same PPID and PGID when executed.

3.5 Processes from Processes

Processes can be created by other processes, and there are three ways to accomplish this in BSD. They are: fork, vfork and rfork, discussed in order. There are other calls (like system) that are really just wrappers for these three.

fork

When a process calls fork, a new process is created that is a duplicate of the parent:

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

   pid_t   fork(void);
```

Unlike other function calls, when successful, fork will return twice - once in the parent process, where the return value will be the new child's PID and two, in the child process, where the return value will be 0. This way, you can distinguish between processes. Once a new process is created with fork, it will be almost an exact duplicate of its parent. These common traits are, in random order:

- Controlling terminal
- Working directory
- Root directory
- Nice value
- Resource limits
- Real, effective and saved user ID
- Real, effective and saved group ID
- Process group ID
- Supplementary group ID
- The set-user-id bits
- The set-group-id bits
- All saved environment variables
- All open file descriptors and current offsets
- Close-on-exec flags
- File mode creation (umask)
- Signals handling settings (SIG_DFL, SIG_IGN, addresses)
- Signals mask

What is actually unique in the child is its new PID, a PPID which is set to the parent's PID, process resource utilization values set to 0, and a copy of its parent's file descriptors. The child can close the file descriptors without disturbing the parent. If the child wishes to read or write from them, however, it will retain the parent's offsets. Potentially, this may cause strange output, or cause both processes to crash if they both try to read or write from the same file descriptors.

After a new process is created, the order of execution is not known. The best way to control this is through the use of semiphores, pipes, or signals, discussed later. That way, reads and writes are controlled, making it impossible for one process to clobber the other and cause both to crash.

wait

The parent process should collect the child's exit status using one of the following wait system calls:

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

 pid_t     wait(int *status);


 #include <sys/time.h>
 #include <sys/resource.h>
```

```
pid_t     waitpid(pid_t wpid, int *status, int options);
pid_t     wait3(int *status, int options, struct rusage *rusage);
pid_t     wait4(pid_t  wpid, int *status, int options, struct rusage *rusage);
```

With wait calls, the options parameter is a bitwise, or is one of the following:

WNOHANG - do not block on wait. This will cause wait to return even if no child process have terminated.

WUNTRACED - set this if you want to wait for status of stopped and untraced children (due to SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signals).

WLINUXCLONE - set this if you want to wait for kthread spawned from linux_clone.

When using the wait4 and waitpid calls, note that the wpid parameter is the actual PID waited on. Specifying a -1 value will cause the call to wait for any available child process to terminate. When using calls with a rusage structure, if the structure is non-NULL then a summary of the resources used by the terminated process is returned. Of course, if the process has stopped then the rusage information is not available.

The wait function calls provide a means for the parent to gather information about its child processes upon exit. Once called, the actual calling process is blocked until either a child process terminates, or a signal is received. This can be avoided; a common tactic is to set the WNOHANG option in the call to wait. Upon a successful return, the status parameter will contain information about the exited process. Also, if the calling process is not interested in the exit status, a NULL parameter can be passed as status. More details on using the WNOHANG will be covered in the Signals section.

If the exit status information is of interest, macros definitions are available in /usr/include/sys/wait.h. The preferred method is to use these, for greater cross-platform portability. The three are listed below, with explanations on usage:

WIFEXITED(status) - If this evaluates to true (that is, it has a non-zero value) then the process has terminated normally by either a call to exit() or _exit().

WIFSIGNALED(status) - If this evaluates to true then the process terminated due to a signal.

WIFSTOPPED(status) - If this evaluates to true then the process has stopped and can be restarted. This macro should be used with the WUNTRACED option or when the child is being traced (such as with ptrace).

If needed, the following macros will extract the remaining status information:

WEXITSTATUS(status) - This is used when the WIFEXITED(status) macro evaluates to true. This will evaluate the low-order 8 bits of the argument passed to exit() or _exit().

WTERMSIG(status) - This is used when the WIFSIGNALED(status) evaluates to true. This will produce the signal number that caused the process to terminate.

WCOREDUMP(status) - This is used when the WIFSIGNALED(status) evaluates to true. This macro will also evaluate as true if the terminated process has created a core dump at the point at which the signal was received.

WSTOPSIG(status) - This is used when the WIFSTOPPED(status) evaluates to true. This macro will produce the signal number that caused the process to stop.

If the parent process never collects the exit status of its child processes, or if the parent process dies before the child exits, in either case, init will by default automatically inherit the children and collect their exit status.

vfork and rfork

The function vfork is similar to fork and was introduced in 2.9BSD. The difference between the two is that vfork will suppress the parent's execution and use the parent's current thread of execution. This is designed to accommodate the execv function calls (discussed later), in order to prevent fully copying the parent's address space, which would be rather inefficient. Actually, the use of vfork is not recommended because it is not guaranteed across platforms. For example, Irix as of 5.x did not have vfork. Here's a sample vfork call:

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

  int  vfork(void);
```

The function call rfork is also quite similar to fork and vfork. It was introduced in Plan9. Its main goal was to create a more sophisticated method for controlling process creation and creating kernel threads. In FreeBSD/OpenBSD its support extends to simulate threads and the Linux clone call. In other words, rfork allows a faster and smaller process creation routine than fork. The caller can specifiy the resources that they want the child(ren) to share by logical OR'ing them. Here's a sample rfork call:

```
#include <unistd.h>

  int    rfork(int flags);
```

The resources that can be selected with rfork are as follows:

RFPROC - Set this flag when you want to create a new process; otherwise the other flags will only affect the current process. By default, this flag must always be set when used.

RFNOWAIT - Set this flag if you want the child process to be dissociated from the parent. Once the child process exits, it will not leave status information for the parent to collect.

RFFDG - Set this flag when you want the parents file descriptor table to be copied. If not, the parent and child will share a common file descriptor table.

RFCFDG - This flag is mutually exclusive with the RFDG flag. Set this flag if you want the child process to have a new clean file descriptor table.

RFMEM - Set this flag if you want to force the kernel to share the entire address space. This is typically done by directly sharing the hardware page table. This cannot be called directly from C because the child process will return on the same stack as the parent. If that's what you want, then the best method is to use the rfork_thread function, as listed below:

```
#include <unistd.h>

   int     rfork_thread(int flags, void *stack, int (*func)(void*arg), void *arg);
```

This will create a new process that will run on the specified stack, and call the specified function with its arguments. Unlike fork, the return value when successful will be the new process PID to the parent only, because the child will execute the supplied function directly. If it fails, the return value will be -1 and errno

will be set.

RFSIGSHARE - This is a FreeBSD specific flag and was recently used in an well-known buffer overflow exploit in FreeBSD 4.3. This flag will allow the child process to share signals with parent (done by sharing the sigacts structure).

RFLINUXTHPN - This is another FreeBSD specific flag. This will cause the kernel to return SIGUSR1 instead of SIGCHILD upon child exit. We will discuss signals in the next chapter but for now, consider it as a way for the rfork to mimic the Linux clone call.

The return values from rfork are similar to fork. The child will get a value of 0 and the parent will get the PID of the new process. One subtle distinction - rfork will sleep, if needed, until the necessary system resources become available. Also a fork call may be simulated by a rfork call as in RFFDG | RFPROC. However it's not designed for backwards compatibility. If the call to rfork results in a failure, the value -1 is returned and the errno is set. (See the man page or header file for more information on the error codes.)

As evident, the rfork call offers a slimmed down version of fork. One drawback is that there have been some widely exposed security flaws with this function call. Also it's not very compatible even across the BSDs, let alone with other Unix variants. For example, currently NetBSD-1.5 does not support rfork, and not all of the flags that are available on FreeBSD are supported on OpenBSD. As such, the recommended interface for threads is to use pthreads.

3.6 Executing Binary Programs

A process would be useless if it was limited to being an exact copy of its parent process. Hence, the exec functions are used. These are designed to replace the current process image with the new process image. Take, for example, the shell running the ls command. First, the shell will fork, or vfork, depending on its implementation, and then, it will call one of the exec functions. Once the exec function is successfully called the newly created process is replaced with the ls executable and the exec will itself never return.If scripts, such as shell or Perl, are the target programs, the process is similar to that of binary program execution. There is an additional step - calling the actual interpreter. That is, the first two characters will contain #!. For example, the following shows a form for an interpreter invocation, along with the remaining arguments:

```
#!  interpreter [arg]
```

This command below invokes the Perl interpreter, with -w as the argument:

```
#!/usr/bin/perl  -w
```

All of the exec calls are listed below. For a source code example see exec.c. The basic structure for calling them is:
(target program), (args), (environment if needed).

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

    extern char **environ;

    int     execl(const char *path, const char *arg0, ...  const char *argn,   NULL);
    int     execle(const char *path, const char *arg0, ... const char  *argn, NULL, char *const
    int     execlp(const char *file, const char *arg0, ... const char *argn , NULL );
    int     exect(const char *path, char *const argv[], char *const envp[]);
```

```
int     execv(const char *path, char *const argv[]);
int     execvp(const char *file, char *const argv[]);
```

NOTE: When using the exec calls that have the arg, ... parameters, arguments to the exec function must be a sequence of NULL-terminated strings, as in arg0, arg1, arg2... argn, with the final terminating character of NULL, or 0. If the ending NULL argument is not specified, the call will fail. The sequence will be the arguments to the target program to be executed.

The exec calls that have the *argv[] parameter are comprised of NULL terminated array of null-terminated strings. This array is the argument list that will be given to the target program to be executed. Again, this list must be terminated with the final pointer as NULL.

Another distinction between binary or scripts execution is in how the target program is specified. The functions execlp and execvp will search your PATH environment variable for the target program, as long as the target does not contain a solidus (/) character. The other function calls will require a path specified for the executable.

Quick guide:

Array arguments and Sequence list

array: exect, execv, execvp

sequence: execl, execle, eseclp

Path search and direct file

path: execl, execle, exect, execv

file: execlp, exevp

Specify environment and Inherit environment

specify: execle, exect

inherit: execl, execlp, execv, execvp

system

```
#include <stdlib.h>

  int     system(const char *string);
```

Another key process execution function call is the system call. This function is fairly intuitive. The supplied argument is passed directly to the shell. If a NULL argument is specified, the function will return 1 if the shell is available and 0 if not. Once called, the process will wait until the shell exits and will return the shell's exit status. If a value of -1 is returned then fork or waitpid failed; a value of 127 specifies that the shell's execution failed.

Once called, some signals such as SIGINT and SIGQUIT will be ignored, and also may block SIGCHILD. Also, it's possible that the system function call can crash the calling process, such as if the child process writes

to stderr.

As evident, BSD design has an easy but robust interface for process creation. This sophisticated design is comparable to any other modern operating system. The next chapter will cover signals and process management including resource usage, threads, and process limits.

Prev top Next

Chapter 3
FreeBSD System Programming

Chapter 4
FreeBSD System Programming

Code samples for this chapter: here

4.1 Advanced Process Controls and Signals

Signal: 4 a : an object used to transmit or convey information beyond the range of human voice. www.m-w.com

So far we have covered process creation and other system calls. There will be times that you need to communicate between multiple processes, gain more fine grain process control, or even allow programs or operators to notify your program of an event. For example, you might want your program to re-read its configuration file. Or, your database program may need to terminate gracefully by writing out transactions prior to exiting. These are just a few possibilities for using signals. Although there are many ways to do similar tasks, including using sockets, fifos, pipes, and semaphores, we'll focus on signals and other process control mechanisms, which offer most of the features and power you'd need for use in the real world.

4.2 Signals

Signals are similar to hardware interrupts. Just as when a device wants service it tells the CPU so by generating a hardware interrupt, a process that wants to notify another process can use a signal.

Most Unix system administrators will be familiar with the SIGHUP signal. Most server daemons will re-read their configuration files or restart when you send them a SIGHUP signal via the kill command. Some of these signals have direct correlation to the hardware like SIGFPE, "floating point exception" or SIGILL, "illegal instruction"; others are software related like SIGSYS, "non-existent system call invoked".

What the process does once a signal is received depends on the signal and what the process wants to do with it. Some signals can be blocked, ignored, or caught, while others cannot. If a process wants to catch a signal and perform some actions based on it, you can set the process with a signal handler for that specific signal. A signal handler is just a function that is called once that signal has been received. Or rather, a signal handler is just a function call that you can specify.

There are defaults performed by the operating system if no signal handler has been specified. These default actions vary from termination to full core dumps. Note that there are two signals that cannot be caught or ignored: SIGSTOP and SIGKILL, explained below.

There are many signals defined on a BSD system; we discuss the standard signals defined in the /usr/include/sys/signals.h. Note that NetBSD has a few more signals that are not included here so if you need a specific that is not found below, check your header files.

```
#define SIGHUP          1       /* hangup */
```

The SIGHUP is a very common signal for Unix system administrators. Many server daemons will re-read their configuration files once they receive this signal. Its original function, however, was to notify a process

that its controlling terminal has disconnected. The default action is to terminate the process.

```
#define SIGINT          2       /* interrupt */
```

The SIGINT is another common signal for Unix users, and is better known as CTRL-C for most shells. The formal name is the Interrupt signal. The default action is to terminate the process.

```
#define SIGQUIT         3       /* quit */
```

The SIGQUIT signal is used by shells that accept the CTRL-/ keys. Otherwise, it acts as a signal to tell the process to quit. This is a commonly used signal for an application to be given notice to shutdown gracefully. The default action is to terminate the process and create a core dump.

```
#define SIGILL          4       /* illegal instr. (not reset when caught) */
```

The SIGILL signal will be sent to a process if it tries to execute an illegal instruction. If your program makes use of use of threads, or pointer functions, try to catch this signal if possible for aid in debugging. The default action is to terminate the process and create a core dump.

```
#define SIGTRAP         5       /* trace trap (not reset when caught) */
```

The SIGTRAP is a POSIX signal, and is used for debugging. It notifies a process being debugged that it has reached a break point. Once delivered, the process being debugged would stop and the parent process would be notified. The default action is to terminate the process and create a core dump.

```
#define SIGABRT         6       /* abort() */
```

The SIGABRT signal provides a way to abort a process and create a core dump. If this is caught and the signal handler does not return, however, the program will not terminate. The default action is to terminate the process and create a core dump.

```
#define SIGFPE          8       /* floating point exception */
```

The SIGFPE is sent to a process when a floating-point error has occurred. For programs that handle complex mathematics, it's recommended that you catch this signal. The default action is to terminate the process and create a core dump.

```
#define SIGKILL         9       /* kill (cannot be caught or ignored) */
```

The SIGKILL is the most evil signal of them all. As you can see from its comment, this signal cannot be caught or ignored. Once this signal is delivered to the process, it is terminated. There are rare cases where this is not the case and even a SIGKILL will not terminate the process, however. These rare conditions occur when the process is doing an "un-interruptible operation" such as disk I/O. Although these conditions are rare, once it happens and the process is deadlocked, the only way to terminate the process is to reboot. The default action is to terminate the process.

```
#define SIGBUS          10      /* bus error */
```

As the name implies, the SIGBUS signal is a result of the CPU detecting an error on its data bus. This error could be the result of a program trying to access an improperly aligned memory address. The default action is to terminate the process and create a core dump.

```
#define SIGSEGV         11      /* segmentation violation */
```

The SIGSEGV is another common signal for many C/C++ programmers, this is a result of the program trying to access a protected memory location that it does not have rights to or an invalid virtual memory address (dirty pointers). The default action is to terminate the process and create a core dump.

```
#define SIGSYS          12      /* non-existent system call invoked */
```

The SIGSYS signal will be delivered to a process after the program tries to execute a system call that does not exist. The operating system will deliver this signal and the process will be terminated. The default action is to terminate the process and create a core dump.

```
#define SIGPIPE         13      /* write on a pipe with no one to read it */
```

Pipes allow process to communicate with each other, as in a phone call. If a process tries to write to the pipe and there isn't a responder, however, the operating system will deliver the SIGPIPE signal to the offending process (the one who attempted the write). The default action is to terminate the process.

```
#define SIGALRM         14      /* alarm clock */
```

The SIGALRM is delivered to a process when its alarm has expired. These alarms are set with the setitimer and alarm calls, covered later in this chapter. The default action is to terminate the process.

```
#define SIGTERM         15      /* software termination signal from kill */
```

The SIGTERM signal is sent to a process to let it know it needs to clean up after its self and terminate. The SIGTERM is also the default signal sent by the Unix kill command, as well as by the operating system when shutting down. The default action is to terminate the process.

```
#define SIGURG          16      /* urgent condition on IO channel */
```

The SIGURG signal is sent to a process when certain conditions exist on an open socket, and will be discarded if not caught. The default action is to discard the signal.

```
#define SIGSTOP         17      /* sendable stop signal not from tty */
```

This signal cannot be caught or ignored. Once a process receives the SIGSTOP signal it will stop until it receives another SIGCONT signal. The default action is to stop the process until a SIGCONT signal has been received.

```
#define SIGTSTP         18      /* stop signal from tty */
```

The SIGSTP signal is similar to the SIGSTOP; however this signal can be caught or ignored. Shells will deliver this signal to a process when it receives the CTRL-Z from the keyboard. The default action is to stop the process until a SIGCONT signal has been received.

```
#define SIGCONT         19      /* continue a stopped process */
```

The SIGCONT signal is also an interesting signal. As mentioned earlier, the SIGCONT is sent to a process once it has stopped to notify it to resume. This signal is interesting because it cannot be ignored or blocked but can be caught. This makes sense, because a process probably would not want to ignore or block the SIGCONT signal. Otherwise what would it do once it receives a SIGSTOP or SIGSTP? The default action is

to discard the signal.

```
#define SIGCHLD         20      /* to parent on child stop or exit */
```

The SIGCHLD was introduced by Berkeley Unix and has a better interface than SRV 4 Unix's implementation. (The BSD implementations is better in that the signal is not a retroactive process. In system V Unix if a process requests to catch this signal, the operating system will check to see if any outstanding children exist (these are children that have exited and are waiting for the parent to call wait to collect their status). If child processes exist with terminating information, the signal handler will be called. So, merely requesting to catch this signal can result in the signal handler being called, a rather messy situation.)

The SIGCHLD signal will be sent to a process once a child has changed status. As I mentioned in the previous chapter, a parent can fork but does not have to wait for a child to exit. Normally this is bad since the process could turn into a zombie once it exits. However if the parent catches the SIGCHLD signal then it can use one of the wait system calls to collect the status or determine what happened. The SIGCHLD is also sent to the parent once a SIGSTOP, SIGSTP or SIGCONT has been sent to any of the child processes as well. The default action is to discard the signal.

```
#define SIGTTIN         21      /* to readers pgrp upon background tty read */
```

The SIGTTIN signal is sent when a background process attempts a read. The process will then be stopped until a SIGCONT signal is received. The default action is to stop the process until a SIGCONT signal has been received.

```
#define SIGTTOU         22      /* like TTIN if (tp->t_local&LTOSTOP) */
```

The SIGTTOU signal is similar to the SIGTTIN, except the SIGTTOU signal is sent when a background process attempts to write to a tty that has set the TOSTOP attribute. If this attribute is not set on the tty, however, then the SIGTTOU is not sent. The default action is to stop the process until a SIGCONT signal has been received.

```
#define SIGIO           23      /* input/output possible signal */
```

The SIGIO signal is sent to a process that has possible I/O on a file descriptor. The process should set this using the fcntl call. The default action is to discard the signal.

```
#define SIGXCPU         24      /* exceeded CPU time limit */
```

The SIGXCPU signal is sent to a process once it exceeds the CPU limit imposed on it. The limits can be set by the setrlimit, discussed later. The default action is to terminate the process.

```
#define SIGXFSZ         25      /* exceeded file size limit */
```

The SIGXFSZ signal is sent to a process when it has exceeded its imposed file size limit, discussed later. The default action is to terminate the process.

```
#define SIGVTALRM       26      /* virtual time alarm */
```

The SIGVTALRM is sent to a process once its virtual alarm time has expired. The default action is to terminate the process.

```
#define SIGPROF         27      /* profiling time alarm */
```

The SIGPROF is another signal sent to a process that has set an alarm. The default action is to terminate the process.

```
#define SIGWINCH        28      /* window size changes */
```

The SIGWINCH signal is sent to a process that has adjusted the columns or rows of the terminal, such as to increase the size of your xterm. The default action is to discard the signal.

```
#define SIGUSR1         29      /* user defined signal 1 */
#define SIGUSR2         30      /* user defined signal 2 */
```

The SIGUSR1 and SIGUSR2 are designed to be defined by the user. They can be set to do whatever is needed. In other words, the operating system does not have any actions associated with these signals. The defaults are to terminate the process.

4.3 System calls

So how do you use signals? Sometimes it's not even clear whether to use them. Or, after a signal is delivered, you may need to analyze the situation, and find out why was the signal sent or from where did it originate before taking action. Other times you'll simply want to exit the program and create a core file after cleanup. Look at the code samples at the end for detailed examples on each function.

The kill function

The kill function will be familiar to those who have ever killed processes from the command line. The basic syntax is:

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

The kill function will send the specified signal to the process with the process id PID. The signal will only be delivered if the process matches the following criteria:

- The sending and receiving process have the same effective user id (UID);
- The sending process has the appropriate permissions (ie: setuid programs);
- The sending process has the UID of the super user (root).

Note: SIGCONT is an exception in that it can be sent to any descendent of the current process.

The action of kill function varies widely depending on its arguments. These actions are as follows:

- If PID is greater than 0 and the sending process has appropriate permissions, then the signal sig will be delivered.
- If PID is equal to 0 then the signal sig is delivered to all processes whose group ID matches the sending processes PID. (The sending process must still meet the permission requirements.)
- If PID is -1 then the signal is sent to all processes whose effective user id (UID) matches the senders. Naturally this excludes the sending process. If the sending processes effective user id (UID) matches the super user (root), however, then the signal is delivered to all processes except for the system processes (defined by having P_SYSTEM set in their proc->p_flag). With this specific instance, the kill function will not return an error if some of the process could not be sent the signal sig.
- If sig is 0 then the kill function will only check for errors (ie: invalid permission, nonexistent process, etc.) Sometimes this is used to determine if a specific process exists.

- The return value from kill will be 0 if successful and -1 otherwise. On failure the kill function will set the errno with the respective error.

The other version of kill is the raise function, as in:

```
int    raise(int sig);
```

The raise function will send the current process the signal sig. This function is not very useful because it can only send signals to the current process. The raise function will return 0 if its successful and -1 on failure. On failure the raise function will set the errno with the respective error, as in:

```
void (*signal(int sig, void (*func)(int)))(int);
```

4.4 Handling Signals

Now that we know how to raise and send signals, how do you handle them?

The Signal Function

The signal function call offers the easiest paradigm. It does look a bit intimidating, however, because C prototypes can look more complex than they really are. The signal function associates a given function with a specific signal. Here's the FreeBSD definition with added typedefs:

```
typedef void (*sig_t) (int);

sig_t   signal(int sig, sig_t func);
```

The first parameter is the target signal, and can be any one of the ones outlined above. The func parameter is a pointer to a function that will handle the signal. This function should return nothing (void) and take a single integer as its argument. The func parameter can also be set to the following values:

SIG_IGN: If the func parameter is set to SIG_IGN then the signal will be ignored.

SIG_DFL: If the func parameter is set to SIG_DFL then the signal will be set to the default action.

Sigaction

The sigaction is a more flexible conterpart to the signal function. The first parameter is the target signal. The next parameter act is the sigaction structure that will describe what to do with this signal. The final parameter oact is a pointer to a location to store the previous settings. The sigaction structure is as follows:

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

The struct sigaction has the following members:

```
 void     (*sa_handler)(int);
```

This member is a pointer to a function that returns nothing (void) and takes a single integer as an argument. This is the same as the function argument to signal and can also be set to SIG_IGN and SIG_DFL, achieving

the same results as a call to signal.

```
void     (*sa_sigaction)(int, siginfo_t *, void *);
```

This member is a pointer to a function that again returns nothing (void) and takes three arguments. These arguments are an integer that will specify the signal sent; a pointer to a siginfo_t structure that will contain information regarding the signal; a pointer to the specific context as to where the signal was delivered.

```
sigset_t sa_mask;
```

This member is a bitwise mask of signals to block while the signal is being delivered. Attempts to block SIGKILL or SIGSTOP will be ignored. The signals will be then postponed until they are unblocked. See sigprocmask for more on global masks.

```
int      sa_flags;
```

This member is a bitwise mask of the following flags:

SA_NOCLDSTOP: If the SA_NOCLDSTOP bit is set and the target signal is SIGCHLD, then a parent process will not be notified when the child stops, but only if the child exits.

SA_NOCLDWAIT: The SA_NOCLDWAIT flag will prevent children from becoming zombie processes. This is used when the target signal is SIGCHLD. If the process should set this and then call one of the wait system calls, the process will block until all of the children have terminated and return -1 with errno set to ECHILD. To use this feature you must set this bit, and the target signal should be SIGCHLD.

SA_ONSTACK: Sometimes there's a need for a signal to be handled on a specified stack. The sigaction system call provides these means. If the this bit is set, then the signal will be delivered on the stack specified.

SA_NODEFER: When the SA_NODEFER bit is set the system will not mask further deliveries of the current signal while the handler is executing.

SA_RESETHAND: If the SA_RESETHAND bit is set, then the signal handler is set to SIG_DFL once the signal is delivered.

SA_SIGINFO: When set, the function pointed to by sa_sigaction member of the sigaction structure is used. Note: this bit should not be set when using the SIG_IGN or SIG_DFL. Upon a successful call to sigaction, the return value is 0 and -1 otherwise, with errno set to the respective error.

4.5 Signal Masks (blocking and unblocking signals)

A process can decide to block a signal or a set of signals. Once a signal is blocked its delivery is postponed until the process unblocks it. This is useful when a program enters a certain part of code that cannot be interrupted and still wishes to receive and process sent signals that could have been missed. The ability to reliably deliver signals that weren't swallowed up by the operating system was not always present but was introduced in 4.2BSD and later adopted by SVR3.

With the advent of reliable signals, the life and delivery of signals changed. Signals before could be generated and delivered. Now, once a signal is pending, a process can then decide what to do with the signal prior to accepting it. The process may choose to handle it, set it back to the default or ignore and discard the signal.

NOTE: If multiple signals are pending the system will deliver the signals that could change the process state, such as SIGBUS first.

Sigprocmask

Any given process can block signals by using sigprocmask. The syntax is:

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

The sigprocmask function will modify or examine the current signal mask. The action sigprocmask takes when the set parameter is not null depends on the first parameter how. The actions along with their associated meanings are listed below:

SIG_BLOCK: The signals specified to be blocked in the set parameter are added to the list of blocked signals.

SIG_UNBLOCK: The signals that are specified in the set parameter are to be removed from the signal mask.

SIG_SETMASK: The set parameter will replace the current signal mask entirely. If the oset parameter is not null, then it will be set to the previous signal mask. If the set value is null, then the how parameter is ignored and the signal mask is left unchanged. So, to examine the signal mask call the sigprocmask with set null and oset non-null. Once the mask is retrieved, you probably want to examine or manipulate it. The following routines are available. Note that these are currently implemented as macros.

```
int    sigemptyset(sigset_t *set)
```

When called the set parameter will be initialized to an empty set of signals.

```
int    sigfillset(sigset_t *set)
```

When called the set parameter will be initalize to contain all signals.

```
int   sigaddset(sigset_t *set, int signo)
```

When called the signal specified by signo will be added to the set of signals pointed to by the set parameter.

```
int    sigdelset(sigset_t *set, int signo)
```

When called the signal specified by signo will be removed from the set of signals prointed to by the set parameter.

```
int   sigismember(const sigset_t *set, int signo)
```

When called this will return 1 of the signal specified by signo is in the set pointed to by the set parameter. If the signal is not set then the return value will be 0.

```
int     sigpending(sigset_t *set);
```

A process can use the sigpending function to discover what signals are currently pending. The sigpending function will return a mask containing all pending signals. This mask can then be examined by the above routines. The sigpending function will return 0 on success and -1 otherwise, with errno set accordingly.

4.6 Customizing Behavior

Sometimes a program may require that the signal handler to run on a specified stack. In order for this to happen an alternate stack must be specified with the signaltstack function. The structure used for this function is the signaltstack:

```
int    sigaltstack(const struct sigaltstack *ss, struct sigaltstack *oss);
```

Its members are described below.

```
 char    *ss_sp;
```

This member points to an area of memory to be used as the stack. There is a minimum amount of memory needed for the signal handler to operate, defined as MINSIGSTKSZ. There is also another predefined amount that should cover the usual case SIGSTKSZ. This memory should be allocated prior to calling the signaltstack function.

```
   size_t  ss_size;
```

The ss_size member specifies the size of the new stack. If this value is inaccurate, the behavior of the signal handler when executing will be unpredictable - you won't have a say in how the system handles this signal.

```
    int     ss_flags;
```

The ss_flags member can take on a few values depending on the calling circumstances. The first is when the process wishes to disable the alternate stack; the ss_flags will be set to SS_DISABLE. In that case, the ss_sp and ss_size values are ignored and the alternate stack is disabled. Note that the alternate stack can only be disabled if currently the handler is not operating on it.

If calling the signaltstack with a non-null value for oss, then the ss_flags will contain information specifying the current state. These are:

SS_DISABLE: The alternate stack is disabled.

SS_ONSTACK: The alternate stack is currently in use and attempts to disable it will fail.

If calling signaltstack and the oss parameter is not null, it will return the current state. The return value for signaltstack is 0 for success and -1 otherwise. On failure, the errno value will be set accordingly. Because signals can be delivered at any point, they are hard to anticipate. For this reason 4.2 BSD's default behavior was to restart interrupted system calls, provided data has not been transmitted yet. This is for the most part good behavior and still the default action across BSD. There are rare cases, though, when you may want to turn this feature off. You can accomplish this using the siginterrupt function call. The schema is simple:

```
int   siginterrupt(int sig, int flag);
```

Set the sig parameter value to the target signal, and set the flag to true (in this case 1). If the flag parameter is set to false (in this case 0) then the default behavior is that of restarting the system calls.

4.7 Waiting for signals

The sigsuspend function call will temporarily change the current set of blocked signals to the new set specified by the sigmask parameter. After that, the sigsuspend will wait until a signal is delivered. Once a signal is delivered the original signal mask is restored. Because the sigsuspend always terminates due to a signal, the return value from sigsuspend will always be -1 with errno set to EINTR. Here's the syntax:

```
int     sigsuspend(const sigset_t *sigmask);
```

The sigwait function will take a set of signals specified by the set parameter as a signal mask. It will then check for any pending signals contained in this specified set . If so the sigwait function will clear the pending signal, and return with the sig parameter set to the numerical value of the cleared signal. If no signals are pending then the sigwait function will wait until one of the specified signals is generated. The syntax:

```
int    sigwait(const sigset_t *set, int *sig);
```

When a signal is delivered to a process and the process has a signal handler set to catch it, process execution will switch to the signal handler. This can pose a problem once the handler returns. For example, say your program listens on a port specified by a configuration file. Your program sets a signal handler to catch the SIGHUP signal to re-read the configuration file. Once the SIGHUP signal is delivered your program will execute the signal handler and re-read the configuration file. One problem is, you will have no way to know where in your execution will a signal be delivered. You could use some of the function calls listed above to narrow it down but what if you have open sockets, open connections and other things that you need to clean up before you listen on the new port? How do you then determine where to start and when to do so? If your program is currently waiting for input, the system call will be restarted if no data has been transferred, and so the return from a SIGHUP will just continue to wait.

This is one of the uses for the setjmp and longjmp functions, which allow for non-local branching. To use them you need to call one of the setjmp functions with the env parameter, as below:

```
jmp_buff  env;

int    sigsetjmp(sigjmp_buf env, int savemask);
void   siglongjmp(sigjmp_buf env, int val);
int    setjmp(jmp_buf env);
void   longjmp(jmp_buf env, int val);
int     _setjmp(jmp_buf env);
void    _longjmp(jmp_buf env, int val);
void    longjmperror(void);
```

The return from setjmp will be 0 and the current environment will be saved into env. You can then call the corresponding longjmp from inside a signal handler. Once called, longjmp will restore the execution environment to env and the program will return to the original location in which setjmp was called. The original call to setjmp will then return the value of val that was sent as the parameter to the longjmp function.

A few notes about the setjmp and longjmp functions: first, the two are not intermixable. That is, a call to setjmp cannot pass a env variable to a call to _longjmp. Also, when a function calls setjmp returns a subsequent call to longjmp will fail.

The different calls have specific actions that they take. These actions are listed below:

setjmp and longjmp: These will save and restore the signal masks, register set, and the stack.

_setjmp and _longjmp: These will save and restore only the resister set and the stack.

sigsetjmp and siglongjmp: These will save and restore the register set, stack, and the signal masks as long as the save mask parameter value is non-zero.

If, for some reason, the env parameter is corrupted or the function that called setjmp has returned, the longjmp functions will call the function longjmperror. If the longjmperror function then returns, the program is aborted. You can customize the longjmperror function with a function that has the same prototype. The default longjmperror will write "longjmp botch" to standard error and return.

4.8 Alarms

```
unsigned int   alarm(unsigned int seconds);
```

The alarm function is basically a simple alarm clock, and is a useful function that allows a process to be notified once a specified amount of seconds has expired. Upon expiration, the calling process will be sent a SIGALRM signal. All subsequent calls to alarm will supersede any previous calls. Unlike the sleep function, alarm will not block.

It has a few return values that you should note. First, if the calling process has no alarms set then the return value is 0. Second, if an alarm has been set but has not expired then the remaining amount of time since the previous call is returned.

The current maximum amount of seconds that can be specified is 100,000,000 - a pretty long time.

```
int getitimer(int which, struct itimerval *value);
```

The getitimer function will retrieve the proper struct itimerval as described by the first argument (the which argument). The options for the first argument are described below:

```
int  setitimer(int which, const struct itimerval *value,  struct itimerval *ovalue);
```

The setitmer provides a more robust interface than the previous alarm call. On a BSD system each process is provided with three interval timers. These timers are described below:

```
#define ITIMER_REAL      0
```

The real timer is decremented in real time regardless of how long the process spends actually executing on the CPU (in other words, it tracks natural time). This will allow a process to set a timer based on natural real time. When real timer expires, the process is sent a SIGALRM signal.

```
#define ITIMER_VIRTUAL   1
```

The virtual timer is decremented only when the process is executing on the CPU, and allows a process to set an alarm based on CPU usage. When the virtual timer has expired the process is sent a SIGVTALRM signal.

```
#define ITIMER_PROF      2
```

The profile timer is decremented when the process is both executing in the CPU and when the system is executing system calls on behalf of the process. This is helpful for interpreters that want to perform statistical profiling. When the profile timer has expired the process is sent a SIGPROF signal. Unlike the real and virtual timers, however, the SIGPROF can be sent during a system call; the process should be prepared to restart the interrupted system call(s).

This chapter has focused on the signal libraries. These functions and the usage of signals are very important to system programming. Signals can make programs more robust by allowing a system administrator to notify the application to re-read a configuration file. Other important signals deal with pending I/O on an open file descriptor. The next chapter will show how to make use of these I/O related signals.

Prev top Next

Chapter 4
FreeBSD System Programming

Chapter 5
FreeBSD System Programming

Code samples for this chapter: here

5.1 Basic I/O

For the most part, Unix espouses the simple design philosophy, "everything is a file." This is quite a powerful feature - it means a text file you edit has the same programmatic interface as a modem, printer, or network card. And just as you can edit a text file, you should be able to perform the basic operations - read and write - on them as well. Although the actual implementation of this ideal isn't perfect, BSD Unix did actually adhere rather closely. It's another one of BSD's strengths - it's simple and elegant. Things that are not truly files are devices, which have entries in the /dev/ directory. Some devices will require specialized operations, such as block reading and writing. An extreme example is an Ethernet device, which doesn't even have an entry in /dev/ before FreeBSD 5.

A good example of an operating system that treats everything like a file is Plan 9. Plan 9 has files for everything, even Ethernets and network protocols. Please see the Plan 9 Web site http://www.cs.bell-labs.com/plan9dist/ for more information about Plan 9.

In general, files represent the most basic, elemental form of data on a computer, essentially as a linear sequence of bits. When a compiled program is executed with the exec command, the system will read the binary file into memory, which is placed into memory then the code is executed. It's irrelevant to the exec command where the program is located - it can reside on a floppy, hard disk, CD-ROM, or even a network mounted file system half -way around the world. What matters is that it can read the bytes sequentially. The same is true of sending data over a network link. When a program sends data, the data itself is simply a linear sequence of bits, sometimes called streams. The program does not care that it's sending data over a network link, just how to write that data. These two fundamental operations, read and write, are found in most aspects of computing. \ This chapter will cover the basic I/O subsystem and process resources.

5.2 I/O

Processes on Unix keep references called file descripters to open files, which are integers. Whenever a processes is created on Unix, it is given three file descriptors. These are:

- 0 standard in
- 1 standard out
- 2 standard error

These can be descriptors to read and write from the terminal, a file, or even another processes set of descriptors. Take the shell redirect, cat /etc/hosts >> hosts.out. The shell will open the file hosts.out and then execute the cat command with /etc/hosts as its argument. However, when the cat process writes to the standard out (1), the result doesn't get to the tty, but to the file hosts.out. (The cat program does not even know that it's writing to a file on the file system, and writes to the standard out file descriptor. We will see how to do this later in the chapter.)

The open and close functions are basic ways to manipulate descriptors.

The Open Function

```
int     open(const char *path, int flags, /*  mode */ );
```

Upon a successful call, the open function will return a file descriptor associated with the file given in its arguments. This descriptor is just an integer index into the processes file descriptor table. This file descriptor has associated with it a structure that will describe to the kernel how to operate on this file. On BSD this structure is called filedesc, and can be found in the /usr/include/sys/filedesc.h header file. When the process wants to perform any operations on this descriptor, it will use this integer value for arguments to read, write, and execute other IO calls that require a file descriptor.

The kernel keeps a reference count on all file descriptors. This reference count is incremented when the processes opens it, duplicates it, or when the descriptor is kept open across forks or exec calls. When this reference reaches 0, the file is closed. That means that if you have a program and it calls fork or exec and if close-on-exec bit is not set, the reference count will increase and then, if a new program calls fork or exec, it will increase yet again. So, the file will stay open until its reference count reaches 0 or until all of the processes close it or exit.

The Close Function

```
int close(int fd);
```

When a process wants to remove or close an open file descriptor, it calls the close function. This closes the given file descriptor, and decrements the file descriptor reference count. It's similar to exit - when a process calls exit, all open file descriptors associated with that process is automatically decremented. Once the file reference reaches 0, the kernel will fully release the associated file entry.

The getdtablesize Function

```
int getdtablesize();
```

The getdtablesize function will return the size of the file descriptor table. This can be used to examine a system imposed limit. You can also use the sysctl program to examine this like so:

```
bash$ sysctl kern.maxfilesperproc
kern.maxfilesperproc: 3722
```

Depending on your system, you can tune this at run time or when you compile your kernel. This function will not tell you how many files your current process has open, (as getdtablesize) but only the maximum amount your process could potentially have open.

The fcntl Function

```
int    fcntl(int fd, int cmd, ...);
```

The fcntl function will allow a process to manipulate file descriptors. The fcntl function takes at a minimum two arguments, a valid file descriptor, and a command. Then depending on the command the fcntl could require a third argument. The following values are defined for the command argument. On FreeBSD, you can find them inside the /usr/include/fcntl.h header.

```
#define  F_DUPFD     0
```

The F_DUPFD is used to create a new file descriptor that is similar to the original. (You can accomplish the same thing with the dup calls that are covered later.) Upon a successful call to fcntl with the F_DUPFD flag, fcntl will return a new file descriptor with the following attributes:

- If a third argument is given, the descriptor returned is the lowest available descriptor greater than or equal to the value of the third argument. The descriptor returned will reference the file descriptor given as the first argument to fcntl.
- If the file descriptor argument to fcntl is a file, the new file descriptor will have the same file offsets, and the new file descriptor will have the same access mode (ie: O_RDONLY, O_RDWR, O_WRONLY).
- The new file descriptor will share the same file status flags.
- The new file will have the close on exec flag turned off. That means the new file descriptor will remain open across exec calls.

The F_GETFD Command

```
#define  F_GETFD     1
```

The F_GETFD command is used to retrieve the close-on-exec flag status. The return value when ANDed with FD_CLOEXEC will either result in 0 or 1. If 0, the close-on-exec flag is not set, so the file descriptor will remain open across exec calls. If 1, the close-on-exec flag is set, and the file descriptor will be closed on a successful call to one of the exec functions.

The F_SETFD Command

```
#define  F_SETFD     2
```

The F_SETFD command is used to set the file descriptors close-on-exec flag. The third argument is either FD_CLOEXEC to set the close-on-exec flag, or 0 to unset the close-on-exec flag.

The F_GETFL and F_SETFL Commands

```
#define  F_GETFL     3
#define  F_SETFL     4
```

The F_GETFL command will tell fcntl to return the current file descriptor status flags. The opened mode can be retrieved by anding the O_ACCMODE (#define O_ACCMODE 0x0003) along with the returned value. The F_SETFL command will set the file status flags according to the third argument.

Common Flags

Some of these flags are also used in calls to open, and can only be set by calling open with the desired flags. Here are the most common found; check your system header file for other values.

```
#define  O_RDONLY 0x0000
```

If the O_RDONLY flag is set, then the file is opened for reading only. Note that this O_RDONLY flag can only be set by a call to open; it cannot be set by calling fcntl with the F_SETFL command.

```
#define  O_WRONLY 0x0001
```

If the O_WRONLY flag is set, the file is opened for writing only. This flag is set by open and cannot be set by calling fcntl with the F_SETFL command.

```
#define  O_RDWR      0x0002
```

If the O_RDWR flag is set, the file is opened for reading and writing. Again, this flag can only be set by a call to open.

```
#define  O_NONBLOCK  0x0004
```

If the O_NONBLOCK flag is set, the file descriptor will not block but instead return immediately. An example would be an open tty. If the user does not type anything into the terminal the call to read will block until the user types. When the O_NONBLOCK flag is set, the call to read will return immediately with the return value set to EAGAIN.

```
#define  O_APPEND 0x0008
```

If the O_APPEND flag is set, the file is opened for append mode, and writes to the file will begin at the end.

```
#define  O_SHLOCK 0x0010
```

If the O_SHLOCK flag is set, the file descriptor has a shared lock. A shared lock can be set on a file so that multiple processes can perform operations on the file. See the F_GETLK and F_SETLK commands to fcntl for more on shared file locks.

```
#define  O_EXLOCK 0x0020
```

If the O_EXLOCK flag is set, the file descriptor has an exclusive lock on the file. Again, refer to the F_GETLK and F_SETLK commands to fcntl for details.

```
#define  O_ASYNC     0x0040
```

If the O_ASYNC flag is set, the process group will be sent the SIGIO signal to notify them that IO is possible on the file descriptor. See the Signals chapter for details.

```
#define  O_FSYNC     0x0080
```

If the O_FSYNC flag is set, all writes to the file descriptor will not be cached by the kernel. Instead, it will be written to media and all calls to write will block until the kernel finishes.

```
#define  O_NOFOLLOW  0x0100
```

When the O_NOFOLLOW flag is set then the call to open would have failed if the file was a symbolic link. If this flag is set on a valid file descriptor, then the current file is not a symbolic link.

```
#define  O_CREAT     0x0200
```

If the O_CREAT flag is set then the file would have been created if it did not exist upon a call to open. (The misspelling is interesting; when one of the original creators of C was asked "What one thing would you change about C?" he replied, "I would change O_CREAT to O_CREATE!", or at least how the rumor goes.)

```
#define  O_TRUNC     0x0400
```

If the O_TRUNC flag is set, the file would have been truncated upon a successful call to open.

```
#define  O_EXCL      0x0800
```

If the O_EXCL flag is set, the call to open would have resulted in an error if the file had already existed.

```
#define F_GETOWN          5
```

The F_GETOWN command is used to retrieve the current process or process group receiving the SIGIO signal for this descriptor. If the value is positive then it will represent a process; negative values represent process groups.

```
#define F_SETOWN  6
```

The F_SETOWN command is used to set the process or process group to receive the SIGIO signal when IO is ready. To specify a process use a positive value (a PID) as the third argument to fcntl. Otherwise use a negative value for the 3rd argument to fcntl to specify a process group.

5.3 File Locking

So what happens when multiple processes attempt writes to a file? They can trample each other, in something known as file locking. This occurs because each process has its own file descriptor with its own offsets. When each process writes their file, offset gets advanced independently so no process knows that others are writing. The resulting file will contain garbage because the multiple independent writes to the file can get intermixed. One way to solve this problem was to have file level locking, so only one process can write to a file at any time. Another was to allow locks of regions inside the file in a scheme called advisory file locking. The fcntl function can provide this functionality. For the most part there are two types of locks. The first is read and the second is write. The difference is that the read locks will not interfere with other processes reading the file, but only one write lock can exist on a specified region.

The following structure is used as the third argument to fcntl when using advisory locks.

```
struct flock {
          off_t   l_start;    /* starting offset */
          off_t   l_len;      /* len = 0 means until end of file */
          pid_t   l_pid;      /* lock owner */
          short   l_type;     /* lock type:   */
          short   l_whence;   /* type of l_start */
  };
```

Now let's go over each element in detail.

l_start

This is an offset in bytes relative to the l_whence. In other words, the desired location is actually measured from l_whence + l_start.

l_len

This needs to be set to the length of the desired region in bytes. The lock will begin from l_whence + l_start for l_len bytes. If you want a lock for the entire file, then set this value to 0. If the value of l_len is negative,

however, the behavior is unpredictable.

l_pid

This needs to be set to the process ID (PID) of the process that is working on the lock.

l_type

This needs to be set to the desired type of lock. The values are as follows:

- F_RDLCK - a read lock
- F_WRLCK - a write lock
- F_UNLCK - used to clear the lock

l_whence

This is the most confusing part of this system call. This field will determine the offset of the l_start position. This will need to be set to:

- SEEK_CUR - the current location in the file
- SEEK_SET - the beginning of the file
- SEEK_END - the end of the file

Commands to fcntl

The following commands to fcntl are as follows.

```
#define  F_GETLK      7
```

The F_GETLK will try to check to see if a lock can be granted. When using this command fcntl will check to see if there is a conflicting lock. If a conflicting lock exists, fcntl will overwrite the flock structure passed to it with the conflicting locks information. If there are no conflicting locks, then the original information in the flock structure will be preserved, except the l_type field will be set to F_UNLCK.

```
#define  F_SETLK       8
```

The F_SETLK command will try to obtain the lock as described by the flock structure. This call will not block if the lock cannot be granted, however; fctnl will return immediately with EAGAIN and errno will be set accordingly. You can use this to clear a lock when the l_type value in the flock structure is set to F_UNLCK.

```
#define  F_SETLKW   9
```

The F_GETLK command will try to obtain the lock as described by the flock structure. This command to fcntl will block until the lock can be granted.

5.4 Why flock?

For the most part, the advisory file locking scheme is a good thing. However the POSIX.1 interface has a few drawbacks. For one, all locks associated with a file must be removed when any file descriptor for that file is closed. In other words, if you have a process that has a file open, and it calls a function that opens that same

file, it reads and then closes it, so that all of your previous locks will be removed. This can cause serious problems if you are not sure what a library routine might do. Also, locks are not passed to children processes, so a child process must create its own locks independently. In addition, all locks obtained prior to an exec call will not be released until the process releases, close the file or exits. So, if you need to lock a region of a file, then you'll exec without releasing the locks or closing the file descriptor. That region will be locked until the process terminates, which might not be the expected behavior that you wanted. The designers of BSD, however, created a much simpler interface to file locking that is preferred by many - flock.

Flock is used to lock entire files, the BSD preferred method, as opposed to the fcntl advisory locks. The flock scheme allows the locks to be passed down to children processes. Another advantage to using the flock call is that locks are done at the file level and not at the descriptor level, which may be preferred in some cases. It means that multiple file descriptors that reference the same file, such as calls to dup(), or multiple calls to open(), would each refer to the same file lock. File locks with flock are similar to fcntl locks in that only one writer can exist while multiple readers are allowed. However, lock can be upgraded. When calling flock the following operations are defined:

```
#define   LOCK_SH        0x01      /* shared file lock */
```

The LOCK_SH operation is used to create a shared lock on a file (similar to a read lock with fcntl). Multiple processes can have a shared lock on a file.

```
#define   LOCK_EX        0x02      /* exclusive file lock */
```

The LOCK_EX operation is used to create an exclusive lock on a file. When an exclusive lock is granted, no other shared locks can exist on the file, including shared locks.

```
#define   LOCK_NB        0x04      /* don't block when locking */
```

With this, calls to flock will block until the lock is granted. However if the LOCK_NB is ORed with the desired operation, the call to flock will return with either success (0) or (-1) with errno set to EWOULDBLOCK.

```
#define   LOCK_UN        0x08      /* unlock file */
```

The LOCK_UN is used to remove a lock on a file.

Locks with flock can be upgraded or downgraded, by calling flock with the desired operation. New successful calls will replace the previous lock with the newly granted one.

The dup Function

```
int     dup(int old);
```

Just like the fcntl call can be used to create duplicate descriptors for existing file descriptors, the dup function also creates duplicate file descriptors. The dup call will return a new file descriptor that is indistinguishable from the old parameter. This means all calls to read(), write() and lseek() will move both descriptors. Also, all options set with fcntl will remain, with the exception of the close on exec bit. The close on exec bit will be turned off, so you can dup a file descriptor then allow a child process to call one of the exec functions, a very common use of the dup function. The old parameter is the desired target descriptor to be duped and must be a valid descriptor. The new file descriptor returned by a successful call to dup() will be the lowest unused file descriptor. That means if you close STDIN_FILENO (its value is 0) then immediately call dup() the value of your new file descriptor will be STDIN_FILENO. If the dup function fails for any reason, the return value

will be -1 and errno will be set accordingly.

The dup2 Function

```
int     dup2(int old, int new);
```

The dup2 function is similar to the dup function, except the new parameter is the desired target value. If the new parameter already references a valid open descriptor and its value is not the same as the old parameter, the new file descriptor will be closed first. If the new parameter equals the old parameter, then nothing happens. The returned value from a successful call to dup2 will be equal to the new parameter. If a call to dup2 fails, the return value will be -1 and the errno be set accordingly.

5.5 Interprocess Communication

Basic interprocess communication, or better known as IPC, are mainly functions from System V. They are still very commonly used amongst the BSDs. The IPC functions allow programs to share data with each other. This is similar to the redirect that we just covered, but whereas the redirect is a one way process and is not bidirectional, in the example program redirect could share data with the cat command by setting its STDIN_FILENO. One problem: the cat command cannot share data with the redirect program. We could modify both to allow sharing in both directions using a fancy algorithm in which they read from each others file descriptor but using open files is clumsy. BSD offers much better methods for interprocess communication.

The Pipe Function

```
int pipe(int *array)
```

The pipe function will allocate two file descriptors, given by calling pipe with a valid two dimensional array (ex: int array[2];). If successful, the array will contain two distinct file descriptors that will allow for unidirectional communication. The first descriptor (array[0]) is opened for reading, and the other (array[1]) is opened for writing. So once you have successfully called pipe, you get a unidirectional communication channel between these two descriptors. When you write to one of them, you will be able to read the output from the other. The benefit from the pipe function over a redirect is that you don't have to use any files.

The file descriptors will behave exactly the same way however they will not have any file associated with them. Unix shells make use of the pipe function for commands that are piped to each other like so:

```
bash$ find /  -user frankie  | grep -i jpg  | more
```

Here the find command will have its output piped into the grep command that will in turn have its output piped into the more command. When creating this sequence, the shell will handle the actual setting of the pipes. These programs have no idea that they are writing to another program, because they don't really need to know. From the example you can see that once pipe is called, it is normal for the process to fork. Once this is done the processes can communicate. If bidirectional communication is the goal, you can create two sets of pipes, one for the parent to communicate with the child, and the other for the child to communicate with the parent.

Communication with pipes have the following two rules:

1. If the read end of the pipe is closed, attempts to write to that pipe will result in a SIGPIPE being delivered to the writing process.

2. If the write end of the pipe is closed, attempts to read from that pipe will result in read returning 0 or end of file. Closing the write end of the pipe is the only way to deliver the end of file to the reading end of the pipe.

A successful call to the pipe function will result in 0 being returned. If the call fails then -1 is returned and the errno is set accordingly.

NOTE: On more modern BSDs, the pipe function supports bidirectional communication from a single set of descriptors. However this behavior is not portable, and thus is not recommended.

The mkfifo Function

```
int mkfifo(const char *path, mode_t mode);
```

Pipes are useful when communicating between related processes, but for communicating between unrelated processes, use the mkfifo function. The mkfifo actually creates a file in the file system. This file is only a marker for other processes to use when communicating. These files are called fifos (first in first out). When a process creates a fifo, writes to this fifo will not be written to the file, but will be read by another processes instead. This behavior is very similar to a pipe; the fifos are also known as named pipes.

The mkfifo function takes two arguments. The first argument is a null-terminated string that specifies the path and file name. The second argument is the mode for that file. The mode argument is the standard Unix permissions for owner read and write (see S_IRUSR, S_IRGRP etc. found in /usr/include/sys/stat.h).

Once the mkfifo has been successfully called, the fifo will need to be opened for reading and writing using the open function. If the call fails, then -1 is returned and the errno is set accordingly.

Creating fifos is similar to file creation in that the process must have sufficient permissions to create the fifo, because the user id of the fifo will be set to the effective user id of the process and the group id of will be set to the effective group id of the process.

An important note about fifos is that they are blocking by default. Thus, reads to a fifo will block until the other end writes and vice versa. To avoid this, use the O_NOBLOCK option to open. You'll get the following behaviors; calls to read will return immediately with the read value of 0 or, calls to write will result in a SIGPIPE signal.

5.6 Message Queues

Another IPC mechanism, message queues, provides another way for processes to communicate. However unlike the others that I have mentioned, you should try to avoid using these if possible. If your program uses message queues, try to re-implement them as fifos or even Unix domain sockets. Before I discuss the reasons, here's a quick overview.

Message queues are similar to fifos but instead of using a file for a reference, they use a key. This key is an unsigned integer. Once a message queue is created, data sent to a message queue is buffered by the kernel. The kernel has a finite amount of memory allocated for message queues. Once this buffer is filled then no more data can be sent until a process reads from the message queue. In that sense queues are reliable and for the most part non-blocking if both processes are reading and writing at different speeds. This is different from a fifo, where a slow read process can actually slow down a faster writing process (unless the O_NONBLOCK option is set). Another benefit is that data written to the message queue will be saved until another process

reads it even if the writing process exits, unlike a fifo where if the writing process exists the fifo is closed and the reading process will receive and end of file marker.

All of this good behavior of a message queue seems nice but let's take a closer look. Say a process opens a message queue, writes a large amount of data and fills up the kernel buffer then exits, the kernel will have to store this data until another process reads it, and any other process that wish to create a message queue and write will be denied. This will stay in effect until a process reads the data or until the system is rebooted. In a sense it's possible to create a simple denial of service against a message queue.

Another problem is that keys are not guaranteed to be unique. In other words, a process has no way to determine a method to determine weither it's the only one using a specific message queue. With fifos, when a process creates a fifo, it has a better chance to know it's unique because a pre-agreed upon file location can be specified (ie: /usr/local/myapp/fifo_dir). Your application will be able to create a unique directory upon installation and thus almost guarantee a unique fifo location. Valid message queue keys can be generated by a function called ftok to help produce unique keys, but it's not a sure bet. The side effects of this problem can be hard to determine - your application could be reading data that it was not intended to read, or your program could be writing data that is being read by some other unintended process. In short, strange and errors that are tough to debug will result when you use message queues.

If you still insisit in using message queues, see the following man pages: ftok(3), msgget(3), msgctl(3), msgrcv(3), and msgsnd(3).

5.7 Conclusion

This chapter has covered various system calls for manipulating open file descriptors, including scenarios where the descriptor must be closed before we fork and exec. We've also discussed file locking, and setting and removing file locks, and special file descriptors such as fifos and queues, which in a sense don't store data on a file system at all. These system calls add a great deal of programmability and flexibility to BSD. But what happens once a process has multiple file descriptors open? The next chapter discusses handling multiple descriptors efficiently.

Prev top Next
Chapter 5
FreeBSD System Programming

Chapter 6
FreeBSD System Programming

Code samples for this chapter: here

6.1 Advanced I/O and Process Resources

As we have seen from the previous chapter, programs can have multiple file descriptors open at the same time. These file descriptors aren't necessarily files, but can be fifos, pipes, or sockets. As such, the ability to multiplex these open descriptors becomes important. For example, consider a simple mail reader program, like pine. It should of course allow the user to not only read and write email, but also simultaneously check for new email. That means the program receives input from at least two sources at any given point: one source is the user, the other is the descriptor checking for new mail. Handling multiplexing descriptors is a complex issue. One method is to mark all open descriptors non-blocking (O_NONBLOCK), and then loop through them until one is found that will allow an I/O operation. The problem with this approach is that the program constantly loops, and if no I/O is available for a long time, the process will tie up the CPU. Your CPU load only worsens when multiple processes are looping on a small set of descriptors.

Another approach is to set signal handlers to catch when I/O is available, and then put the process to sleep. This sounds good in theory, if you only have a few open descriptors and infrequent I/O requests. Because the process is sleeping, it will not tie up the CPU, and it will then only execute when I/O is available. However, the problem with this approach is that signal handling is somewhat expensive. For example a web server receiving 100 requests per minute, would need to catch signals almost constantly. The overhead from catching hundreds of signals per minute would be significant, not only for the process but for the kernel to send these signals as well.

So far, both options are limited and ineffective, with the common problem being that a process needs to know when I/O is available. However, this information is actually only known in advance by the kernel, because the kernel ultimately handles all the open descriptors on the system. For example, when a process sends data over a fifo to another, the sending process calls write, which is a system call and thus involves the kernel. The receiver will not know until the write system call is executed by the sender. So, a better way to multiplex file descriptors suggests itself: have the kernel manage it for the process. In other words, send the kernel a list of open descriptors and then wait until the kernel has one or more ready, or until a time-out has expired.

This is the approach taken by the select(), poll() and kqueue() interfaces. Through these, the kernel will manage the descriptors and awake the process when I/O is available. These interfaces elegantly handle the problems mentioned above. The process doesn't need to loop through the open descriptors, nor does it need to handle signals. The process will still incur a slight overhead, however, when using these functions. This is because the I/O operations are executed after the return from these interfaces. Thus it takes at least two system calls to perform any operation. For example, say your program has two descriptors used for reading. You use select and wait for them to have data to read. This requires the process to first call select, and then when select returns, to call read on the descriptor. More ideally, you could do a large blocking read against all of the open descriptors. Once one is ready to read, the read will return with the data inside the buffer and an indication of which descriptor the data was read from.

6.2 select

The first interface I will cover is select(). The format is:

```
int  select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,  struct timeval *
```

The first argument to select has caused some confusion over the years. The proper usage of the nfds argument is to set this to the maximum file descriptor value plus one. In other words, if you have a set of descriptors {0, 1, 8}, the ndfs parameter should be set to 9, because the highest numbered descriptor in your set is 8. Some mistake this parameter to mean the number of total descriptors to be n+1, which in our example would result in 4 incorrectly. Remember that a descriptor is simply an integer value, so your program will need to figure out which is the largest valued descriptor you want to select on.

Select will then examine the next three arguments, readfds, writefds and exceptfds for any pending reading, writing or exceptional conditions, in that order. (For more information, see man(2) select). Note that if no descriptors in readfds, writefds or execptfds are set, the values to select should be set to NULL.

The readfds, writefds and exceptfds arguments are to be set with the four macros listed below.

FD_ZERO(&fdset);

The FD_ZERO macro is used to clear the set bits in the desired descriptor set. One very important note: this macro should always be called when using select; otherwise, select will behave unpredictably.

FD_SET(fd, &fdset);

The FD_SET macro is used when you want to add an individual descriptor to a set of active descriptors.

FD_CLR(fd, &fdset);

The FD_CLR macro is used when you want to remove an individual descriptor from a set of active descriptors.

FD_ISSET(fd, &fdset);

The FD_ISSET macro is used once select returns to test if a particular descriptor is ready for an I/O operation. The final parameter is a time-out value. If the time-out value is set to NULL, then the call to select will block indefinitely until an operation is ready. However, if a specified time-out is desired then the time-out value should be a non-null timeval structure. The timeval structure is as follows:

```
struct timeval {
    long    tv_sec;         /* seconds */
    long    tv_usec;        /* microseconds */
};
```

Upon a successful call to select, the number of ready descriptors is returned. If select returns due to the expiration of a time limit 0 is returned. If an error occurs then -1 is returned and the errno is set accordingly.

6.3 Poll

For the most part, the I/O discussed is BSD specific. System V includes support for a special type of I/O known as streams. Streams, along with sockets, have a priority attribute sometimes referred to as the

data-band. This data-band can be set to specify a high priority for certain data on the stream. BSD originally did not have support for this feature, but some have added System V emulation, and can support certain types. Because we don't focus on System V, we'll make reference to the data-band or data priority band only. For more information, see System V STREAMS.

The poll function is similar to select:

```
int  poll(struct pollfd *fds, unsigned int nfds, int timeout);
```

Unlike select, which is native to BSD, poll was created by System V Unix and was not supported on earlier versions of BSD. Currently poll is supported on all major BSD systems.

Similar to select, poll will multiplex a set of given file descriptors. When specifying them, you have to use an array of structures, with each structure representing a single file descriptor. One advantage of using poll over select is that you can detect a few rare conditions that select will not. These conditions are POLLERR, POLLHUP, and POLLNVAL, discussed later. Although there is much discussion about choosing select or poll, for the most part, it'll depend on your personal taste. The structure used by poll is the pollfd structure, as below:

```
struct pollfd {
    int     fd;             /* which file descriptor to poll */
    short   events;         /* events we are interested in */
    short   revents;        /* events found on return */
};
```

fd

The fd member is used to specify the file descriptor you wish to use poll on. If you want to remove a particular descriptor, set the fd member for that descriptor to -1. That way you can avoid having to shuffle the array around, and will also clear all events listed in the revents member.

events, revents

The events member is a bitmask specifying the events in which you are interested in for that specific descriptor. The revents member is also a bitmask, but its value is set by poll with the event(s) that have occurred on that specific descriptor. These events are defined bellow.

```
#define POLLIN          0x0001
```

The POLLIN event lets you specify that your program will poll for readable data events for the descriptor. Note that this data will not include high priority data, such as out-of-bound data on sockets.

```
#define POLLPRI         0x0002
```

The POLLPRI event is used to specify that your program is interested in polling for any high priority events for the descriptor.

```
#define POLLOUT         0x0004
#define POLLWRNORM      POLLOUT
```

The POLLOUT and POLLWRNORM events are used to specify that your program is interested in when a write on a descriptor can be performed. On FreeBSD and OpenBSD they are same; you can also check this in

your system header file (/usr/include/poll.h). Technically speaking, the difference between them is that POLLWRNORM will only detect when it's possible to perform a write when the data priority band is equal to 0.

```
#define POLLRDNORM      0x0040
```

The POLLRDNORM event is used to specify that your program is interested in polling for normal data on the descriptor. Note that on some systems, this is specified to have the exact same behavior as POLLIN. However on NetBSD and FreeBSD, this is not the same as the POLLIN event. Again, check your system header file (/usr/include/poll.h). Strictly speaking, POLLRDNORM will only detect when it is possible to perform a read when the data-band is equal to 0.

```
#define POLLRDBAND      0x0080
```

The POLLRDBAND event is used to specify that your program is interested in knowing when it can read data with a non zero data-band value.

```
#define POLLWRBAND      0x0100
```

The POLLWRBAND event is used to specify that your program is interested in knowing when it can write data to the descriptor with a non-zero data-band value.

FreeBSD Specific Options

The next options are FreeBSD specific and are not well known or widely used. They're worth mentioning, however, because of the flexibility afforded. These are new options and poll is not guaranteed to detect these conditions; plus, they only work with the UFS file system. If you need your program to detect these types of events, it's best to use the kqueue interface, covered later.

```
#define POLLEXTEND      0x0200
```

The POLLEXTEND event will be set if the file has been executed.

```
#define POLLATTRIB      0x0400
```

The POLLATTRIB event will be set if any file attributes have changed.

```
#define POLLNLINK       0x0800
```

The POLLNLINK event will be set if the file has been renamed, deleted, or unlinked.

```
#define POLLWRITE       0x1000
```

The POLLWRITE event will be set if the file contents have been modified.

The next events are not valid flags for the pollfd events member and poll will ignore them. They are returned in the pollfd revents instead, to specify certain events that have happened.

```
#define POLLERR         0x0008
```

The POLLERR event will specify that an error has occurred.

```
#define POLLHUP         0x0010
```

The POLLHUP will specify that a hang-up has occurred on the stream. The POLLHUP and POLLOUT are mutually exclusive events, because a stream is no longer writable once a hang-up has occurred.

```
#define POLLNVAL        0x0020
```

The POLLNVAL event will specify that the request to poll was invalid.

The final argument to poll is timeout. This argument can specify to poll a desired timeout in milliseconds. When timeout is set to -1, poll will block until a requested event has occurred. When the timeout is set to 0, poll will return immediately.

A positive integer is returned upon a successful call to poll. The value of the positive integer will specify the number of descriptors in which events have occurred. If the time out has expired, then poll will return 0. If an error has occurred, the poll will return -1.

6.4 kqueue

So far, poll and select seem like elegant ways to multiplex file descriptors. To use either of those two functions, however, you need to create the list of descriptors, then send them to kernel, and then upon return, look through the list again. That seems a bit inefficient. A better model would be to give the descriptor list to the kernel and then wait. Once one or more events happen, the kernel can notify the process with a list of only the file descriptors which had events, avoiding loops through the entire list every time a function returns. Although this small gain is not noticeable if the process only has a few descriptors open, for programs with thousands of open file descriptors, the performance gains are significant. This was one of the main goals behind the creation of kqueue. Also, the designers wanted a process to be able to detect a wider variety of events, such as file modification, file deletion, signals delivered, or a child process exit, with a flexible function call that subsumed other tasks. Handling signals, multiplexing file descriptors, and waiting for child processes can all be wrapped into this single kqueue interface because they are all waiting for an event to occur.

Another design consideration was for a process to have multiple instances of kqueues without interference. As you have seen, a process can set a signal handler; however, what happens if another part of the code also wants to catch that specific signal? Or worse, say a library function sets a signal handler for the same signal your code wants to catch? Debugging this to figure out why your program is not executing the signal handler you set would take hours. For the most part, these situations don't happen often. Good programmers will avoid setting signal handlers inside library functions. With large, complex programs these situations become hard to avoid, so ideally, we should be able to detect these events, as kqueue allows.

The way kqueue works is with filters. These filters are identified by a unique identifier and filter tuple called a kevent (ident, filter). Only one unique kevent for each kqueue is allowed. These filters are processed during the initial registration with kqueue and during any retrieval of events. At registration if preexisting events exist, they will be placed onto the kqueue for retrieval. If multiple events exist for a given filter, they will be combined into a single kevent.

The kqueue API consists of two function calls and a macro to aid in setting events. These functions are outlined below.

```
int kqueue(void);
```

The kqueue function will start a new kqueue. Upon a successful call the return value will be a descriptor that is to be used when interacting with the newly created kqueue. Each kqueue will have a unique descriptor associated with it. So, a program can have more than one unique kqueue open at once. The kqueue descriptors behave similarly to a regular file descriptor: they can be multiplexed.

One final note, the descriptors are not inherited by child processes created by fork. If the child process was created by the rfork call, however, the RFFDG flag will need to be set to avoid them from being shared with child processes. If the kqueue function fails, -1 is returned and errno is set accordingly.

```
int kevent(int kq, const struct kevent *changelist, int nchanges, struct kevent *eventlist, i
```

The kevent function is used to interact with the kqueue. The first argument is a descriptor returned by kqueue. The changelist argument is an array of kevent structures of size n changes. The changelist argument is used to register or modify events and will be processed before pending events are read from kqueue.

The eventlist argument is an array of kevent structures of size nevents. Kevent will return events to the calling process by placing events inside the eventlist argument. The eventlist and changelist arguments can point to the same array if desired. The final argument is the time-out desired for kevent. When the time-out is specified as NULL, kevent will block until an event has occurred. When the time-out argument is non-NULL, then kevent will block until the time out has expired. When the time-out is specified with a zero value structure, kevent will return immediately with any pending events.

The return value from kevent will specify the number of events placed inside the eventlist array. If the number of events exceeds the size of the eventlist argument then they can be retrieved with subsequent calls to kevent. Errors that occur from processing events will be placed in the eventlist argument providing space is available. The events with errors will have the EV_ERROR bit set along with the system error placed inside the data member. For all other errors -1 is returned and errno is set accordingly.

The kevent structure is used to communicate with kqueue. The header file on FreeBSD can be found in the /usr/include/sys/event.h file. This will have the declaration of the struct kevent as well as other options and flags. Because kqueue is still fairly new compared with select or poll, it's constantly evolving and adding new features. Check your system header for any new or system specific options.

The kevent structure declaration in its native form:

```
struct kevent {
      uintptr_t       ident;
      short           filter;
      u_short         flags;
      u_int           fflags;
      intptr_t        data;
      void            *udata;
};
```

Now, let's look at the individual members:

ident

The ident member is used to store the unique identifier for the kqueue. In other words, if you want to add a file descriptor to an event, the ident member would be set to the target descriptor's value.

filter

The filter member is used to specify the filter you want the kernel to apply to the ident member.

flags

The flags member will specify to the kernel what actions along with any needed flags should be processed with the event. Upon return, the flags member can contain error conditions.

fflags

The fflags member is used to specify filter specific flags you want the kernel to use. Upon return, the fflags member can contain filter-specific return values.

data

The data member is used to hold any filter-specific data.

udata

The udata member is not used by kqueue, but its value passes through kqueue unmodified. This member can be used by the process to send information or even a function to itself for use or considered upon an event detection.

kqueue filters

The filters used by kqueue are listed below. Some filters will have filter-specific flags. Set these flags in the fflags member of the kevent structure.

```
#define EVFILT_READ     (-1)
```

The EVFILT_READ filter will detect when data is available to read. The ident member of the kevent should be set to a valid descriptor. Although this filter behaves in the same manner as select or poll would, the filter will return events specific to the type of descriptor used.

If the descriptor references an open file refereed to as a vnode, the event will specify that the read offset has not reached the end of the file. The data member will contain the current offset relative to the end of the file and can be negative.If the descriptor references a pipe or a fifo, then the filter will return when there is actual data to be read. The data member will contain the number of bytes available to read. The EV_EOF bit will specify that one of the writers has closed their connection. (See the kqueue man page for details on how the EVFILT_READ will behave when a socket is used.)

```
#define EVFILT_WRITE    (-2)
```

The EVFILT_WRITE filter will detect whether it is possible to perform a write on the descriptor. If the descriptor references a pipe, fifo, or socket then the data member will contain the amount of bytes available in the write buffer. The EV_EOF bit will specify that the reader has closed its connection. This flag is not valid for open file descriptors.

```
#define EVFILT_AIO      (-3)
```

The EVFILT_AIO is used with asynchronous I/O operations and will detect similar conditions as the aio_error system call.

```
#define EVFILT_VNODE    (-4)
```

The EVFILT_VNODE filter detects certain changes to a file on a file system. Set the ident member to a valid open file descriptor, and specify which events are desired with the fflags member. Upon return, the fflags member will contain the bitwise mask of events that have happened. These events are as follows:

```
#define NOTE_DELETE     0x0001
```

The NOTE_DELETE fflag will specify that the process wants to know when the file has been deleted.

```
#define NOTE_WRITE      0x0002
```

The NOTE_WRITE fflag will specify that the process wants to know when the file contents have changed.

```
#define NOTE_EXTEND     0x0004
```

The NOTE_EXTEND fflag will specify that the process wants to know when the file has been extended.

```
#define NOTE_ATTRIB     0x0008
```

The NOTE_ATTRIB fflag will specify that the process wants to know when the file attributes have changed.

```
#define NOTE_LINK       0x0010
```

The NOTE_LINK fflag will specify that the process wants to know when the files link count has changed. The link count can change when a file is hard linked by the link function call. (For more information see man(2) link.)

```
#define NOTE_RENAME     0x0020
```

The NOTE_RENAME fflag will specify that the process wants to know if the file gets renamed.

```
#define NOTE_REVOKE     0x0040
```

The NOTE_REVOKE fflag will specify that access to the file was revoked. For more information, see man(2) revoke.

```
#define EVFILT_PROC     (-5)  /* attached to struct proc */
```

The EVFILT_PROC filter is used by a process to detect events that occur in another process. Include the PID of the desired process in the ident member, and set the fflag member with the desired events to monitor. Upon return the events will be placed inside the fflags member. These events are set by bitwise OR'ing the following events:

```
#define NOTE_EXIT       0x80000000
```

The NOTE_EXIT fflag will detect when the process has exited.

```
#define NOTE_FORK       0x40000000
```

The NOTE_FORK fflag will detect when the process has called fork.

```
#define NOTE_EXEC       0x20000000
```

The NOTE_EXEC fflag will detect when the process has called an exec function.

```
#define NOTE_TRACK      0x00000001
```

The NOTE_TRACK fflag will cause kqueue to track a process across a fork call. The child process will be returned with the NOTE_CHILD flag set in fflags and the parent processes PID will be placed into the data member.

```
#define NOTE_TRACKERR   0x00000002
```

The NOTE_TRACKERR fflag will be set if an error occurred when trying to track a child process. This is a return only fflag.

```
#define NOTE_CHILD      0x00000004
```

The NOTE_CHILD fflag is set on a child process. This is a return only fflag.

```
#define EVFILT_SIGNAL      (-6)
```

The EVFILT_SIGNAL filter is used to detect if a signal has been delivered to the process. This filter will detect every time a signal has been delivered by placing the count inside the data member. This includes signal that have the SIG_IGN flag set. The event will be placed onto the kqueue after the execution of the normal signal handling process. Note that this filter will set the EV_CLEAR flag internally.

```
#define EVFILT_TIMER    (-7)
```

The EVFILT_TIMER filter will create a timer for kqueue to count the number of timer expirations. If a one time timer is desired then set the EV_ONESHOT flag. Specify this timer with the ident member and use the data member to specify the timeout in milliseconds. The return value will be specified in the data member. Note that this filter will set the EV_CLEAR flag internally.

kqueue actions

The kqueue actions are set by bitwise OR'ing the desired actions along with any desired flags.

```
#define EV_ADD          0x0001
```

The EV_ADD action will add the event to the kqueue. Because duplicates are not allowed in kqueue, if you try to add an event when one already exists, the existing event will be overwritten by the newer addition. Note that when events are added, they are enabled by default unless you've set the EV_DISABLE flag.

```
#define EV_DELETE       0x0002
```

The EV_DELETE action will remove the event from the kqueue.

```
#define EV_ENABLE       0x0004
```

The EV_ENABLE will enable the event in the kqueue. Note that newly added events will be enabled by default.

```
#define EV_DISABLE      0x0008
```

The EV_DISABLE will disable kqueue from returning information on the event. Note that kqueue will not remove the filter, however.

kqueue action flags

The kqueue action flags are defined below. These are to be used in conjunction with the actions listed above. They are set by bitwise OR'ing the desired actions.

```
#define EV_ONESHOT      0x0010
```

The EV_ONESHOT flag will notify kqueue to return only the first

```
#define EV_CLEAR        0x0020
```

The EV_CLEAR flag will notify kqueue to reset the state for the event once the process retrieves the event from the kqueue.

kqueue returned values

Return only values are placed in the flag member of the kevent structure. These values are defined below.

```
#define EV_EOF          0x8000
```

The EV_EOF will indicate an end of file condition.

```
#define EV_ERROR        0x4000
```

The EV_ERROR will indicate that an error has occurred. The system error will be placed inside the data member.

6.5 Conclusion

This chapter explored multiplexing descriptors in BSD. As a programmer, you can choose three interfaces: select, poll or kqueue. The three are similar in performance for a small set of descriptors, but with a large amount, kqueue is an elegant choice. In addition, kqueue can detect much more than I/O events. It can also detect signals, file modifications and child processes related events. In the next chapter, we explore other ways to get information regarding child processes and determine the statistics of our current process, with a focus on new features in FreeBSD 5.x.

Prev top Next
Chapter 6
FreeBSD System Programming

Chapter 7
FreeBSD System Programming

Code samples for this chapter: here

7.1 Process Resources and System Limits

The BSD Unix system provides ways for system administrators to control system resources, in order to support multiple simultaneous users and application connections. Such resources include CPU time, memory usage, and disk usage. The resource controls let you tune the system to best accommodate its usage. In earlier versions of Unix, some system limits where set at compile time, and as such, changing a limit would require a recompilation of the entire system. However, modern BSD systems are able to adjust most if not all of the system resources at runtime without recompiling the entire system.

This chapter discusses limits associated with processes, both for system wide and user imposed limits. We will look at ways we can find these limits as well as possibly modify them, and also discuss ways a process can actually query its resource usage.

7.2 Determining System Limits

getrlimit, setrlimit

The getrlimit will allow a process to query the imposed system limits. These system limits are specified by a hard limit and a soft limit pair. When a soft limit is exceeded, the process will be allowed to continue, but depending on the type of limit, a signal may be sent to the process. On the other hand, a process may not exceed its hard limit. The soft limit value may be set by the process to any value between 0 and the hard limit maximum. Hard limits can be irreversibly lowered by any process; only the super user can increase them, however.

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>

  int  getrlimit(int resource, struct rlimit *rlp);
  int  setrlimit(int resource, const struct rlimit *rlp);
```

The getrlimit and setrlimit both make use of the following structure:

```
struct rlimit {
         rlim_t  rlim_cur;
         rlim_t  rlim_max;
};
```

Now let's look at each member. The rlim_cur member will specify the current system soft limit for the specified resource. The rlim_max member will specify the current system hard limit for the specified resource.

The first argument to the getrlimit and setrlimit functions is the resource parameter. This will specify the resource that the process is interested in receiving information about. The possible resource values are listed below. You can also find them in /usr/include/sys/resource.h.

```
#define  RLIMIT_CPU  0     /* cpu time in milliseconds */
```

The RLIMIT_CPU resource specifies how many milliseconds a process can take for execution on the CPU. Normally, a process has only a soft limit, and no hard limit. If the soft limit is exceeded, the process will receive a SIGXCPU signal.

```
#define  RLIMIT_FSIZE   1     /* maximum file size */
```

The RLIMIT_FSIZE limit specifies how large a file in bytes a process can create. For example, if the RLIMIT_FSIZE is set to 0, then the process will not be able to create a file at all. If the process exceeded this limit, it will be sent a SIGFSZ signal.

```
#define  RLIMIT_DATA 2     /* data size */
```

The RLIMIT_DATA limit specifies the maximum amount of bytes the process data segment can occupy. The data segment for a process is the area in which dynamic memory is located (that is, memory allocated by malloc() in C, or in C++, with new()). If this limit is exceeded, calls to allocate new memory will fail.

```
#define  RLIMIT_STACK   3     /* stack size */
```

The RLIMIT_STACK limit specifies the maximum amount of bytes the process's stack can occupy. Once the hard limit is exceeded, the process will receive a SIGSEV signal.

```
#define  RLIMIT_CORE 4     /* core file size */
```

The RLIMIT_CORE will specify the maximum size of a core file a process can create. If this limit is set to 0, then a core file will not be created. On the other hand, all processes writing to a core file will be terminated once this limit has been reached.

```
#define  RLIMIT_RSS  5     /* resident set size */
```

The RLIMIT_RSS limit specifies the maximum amount of bytes that a process's resident set size can occupy. The resident set size of a process refers to the amount of physical memory a process is using.

```
#define  RLIMIT_MEMLOCK 6     /* locked-in-memory address space */
```

The RLIMIT_MEMLOCK limit specifies the maximum amount of bytes a process can lock using a call to mlock.

```
#define  RLIMIT_NPROC   7     /* number of processes */
```

The RLIMIT_NPROC limit specifies the maximum amount of concurrent processes allowed for a given user. The user is determined by the effective user id of the process.

```
#define  RLIMIT_NOFILE  8     /* number of open files */
```

The RLIMIT_NOFILE limit specifies the maximum number of files the process is allowed to have open.

```
#define  RLIMIT_SBSIZE  9     /* maximum size of all socket buffers */
```

The RLIMIT_SBSIZE limit specifies the amount of mbufs a user can use at any moment. See the socket man page for a definition of mbufs.

```
#define  RLIMIT_VMEM 10    /* virtual process size (inclusive of mmap) */
```

The RLIMIT_VMEM limit indicates the amount of bytes a process' mapped address space can occupy. Once this limit is exceeded, calls to allocate dynamic memory or calls to mmap will fail.

```
#define  RLIM_INFINITY
```

The RLIM_INFINITY macro is used to remove a limit on a resource. In other words, setting a resource hard limit to RLIM_INFINITY will allow usage without any system imposed limit. Setting a soft limit to RLIM_INFINITY will prevent the process from receiving any soft limit notices. This might be of use if your process does not want to set a signal handler for a resource that would cause the process to be sent a signal once the soft limit is exceeded.

When calling getrlimit the second parameter needs to be a valid pointer to a rlimit structure. The getrlimit will then place the proper limit values inside this structure. On the other hand, setrlimit will use the values supplied inside the second parameter when altering limits. Setting a limit value to 0 will prevent usage of that resource. Setting a resource limit to RLIM_INFINITY will remove any limits on that resource. These functions both return 0 on success and -1 otherwise. If an error occurs, these functions will set errno accordingly.

The getpagesize Function

```
#include <unistd.h>

    int  getpagesize(void);
```

Before we cover the getrusage function, we should discuss the getpagesize function. Some process statistics are reported in terms of pages used. A page of memory is just a chunk of memory usually around 4096 bytes. The size of a page can vary, however, and it should not be hard coded into your programs. Instead, to determine the local system pagesize you should use the getpagesize function. The return value from getpagesize will be the amount of bytes used per page.

7.3 Determining Process Resource Usage

The getrusage Function

Now that we can determine the system limits, we should then be able to determine current processes resource usage. That is where the getrusage function is used. The getrusage function is quite comprehensive. A process can determine its memory usage, amount of time running inside the CPU and even find out information regarding child processes. As such, one use for the getrusage function is to help avoid runaway processes. A runaway process is one that is out of control, either using up too much CPU (being caught in an infinite loop) or maybe exceeding some memory usage limit (resulting from a memory leak).

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>

#define   RUSAGE_SELF     0
#define   RUSAGE_CHILDREN     -1
```

```
int   getrusage(int who, struct rusage *rusage);
```

The getrusage function takes two parameters. The first parameter who is set to either RUSAGE_SELF or RUSAGE_CHILDREN. Setting the parameter RUSAGE_SELF will fill the rusage structure with information regarding the current process. Setting the parameter RUSAGE_CHILDREN will fill the rusage structure with aggregate information regarding child processes.

The rusage structure definition can be found inside /usr/include/sys/resource.h. It includes the following members:

```
struct   rusage {
   struct timeval ru_utime;   /* user time used */
   struct timeval ru_stime;   /* system time used */
   long  ru_maxrss;           /* max resident set size */
   long  ru_ixrss;            /* integral shared memory size */
   long  ru_idrss;            /* integral unshared data */
   long  ru_isrss;            /* integral unshared stack */
   long  ru_minflt;           /* page reclaims */
   long  ru_majflt;           /* page faults */
   long  ru_nswap;            /* swaps */
   long  ru_inblock;          /* block input operations */
   long  ru_oublock;          /* block output operations */
   long  ru_msgsnd;           /* messages sent */
   long  ru_msgrcv;           /* messages received */
   long  ru_nsignals;         /* signals received */
   long  ru_nvcsw;            /* voluntary context switches */
   long  ru_nivcsw;           /* involuntary " */
};
```

Now let's look at each member in detail.

ru_utime,ru_stime

The ru_utime and ru_stime members contain the total amount of time spent executing in user mode and system mode, respectively. They both make use of the timeval structure. (Please see the previous chapter for a brief description of this structure.)

ru_maxrss

The ru_maxrss member contains the total amount of resident set memory used. The value will be in terms of memory pages used.

ru_ixrss

The ru_ixrss value is expressed as the total amount of memory used by the text segment in kilobytes multiplied by the execution-ticks. The text segment of a program refers to the part of the binary that, amongst other things, contains read-only instructions and data.

ru_idrss

The ru_idrss value is expressed as the total amount of private memory used by a process in kilobytes

multiplied by execution-ticks.

ru_isrss

The ru_isrss value is expressed as the total amount of memory used by the stack in kilobytes multiplied by execution-ticks.

ru_minflt

The ru_minflt value is the number of page faults that required no I/O activity. A page fault occurs when the kernel needs to retrieve a page of memory for the process to access.

ru_majflt

The ru_majflt value is the number of page faults that required I/O. A page fault occurs when the kernel needs to retrieve a page of memory for a process to access.

ru_nswap

Occasionally, a process will be swapped out of memory to make room for another to execute. The ru_nswap value indicates the number of times a process was swapped out of memory.

ru_inblock

The ru_inblock member contains the total number of inputs the file system had to perform for a read request.

ru_oublock

The ru_oublock member contains the total number of times the file system had to perform output for a write request.

ru_msgsnd

The ru_msgsnd member indicates the total number of IPC messages sent.

ru_msgrcv

The ru_msgrcv member contains the total number of IPC messages received.

ru_nsignals

The ru_nsignals member contains the total number of signals the process received.

ru_nvcsw

The ru_nvsw member indicates the total number of voluntary context switches for a process. A voluntary context switch occurs when the process "gives up" its time slice on the CPU. Usually this happens when the process is waiting for some type of resource to become available.

ru_nivcsw

The ru_nivcsw member contains the total number of context switches due to a higher priority process becoming runnable.

Upon a successful call to getrusage, 0 will be returned. Otherwise, -1 will be returned with the errno set accordingly.

7.4 Conclusion

In this chapter we covered ways a program can obtain the system limits. These limits should never be hardwired inside your code. Your program should make use of these interfaces instead. This is because these limits are platform-dependent and even differ from a system to another. For example, a system administrator can adjust the amount of allowed open file descriptors, so hard coding a value of 4096 might not be the case if they are lowered or increased. Another example is page sizes. Some 64 bits systems use 8096 as the default page size whereas most 32 bits systems use 4096 bytes. Again this is a tunable parameter and cannot be guaranteed to stay at a fixed value.

We also looked at ways a program can obtain its current usage or query for usage statistics of its child processes. Using these interfaces may help to detect and avoid a process from going out of control. They are also potentially useful in debugging errors that occur when a program tries to exceed its established hard and soft limits.

Prev top Next

Chapter 7
FreeBSD System Programming

Chapter 8
FreeBSD System Programming

Prev top

8.1 FreeBSD 5.X

The FreeBSD project arrived at a significant milestone on January, 2003 with the release of the FreeBSD-5.x branch. Under development for nearly three years, the FreeBSD-5.x branch includes numerous changes to both the core kernel and the base system. For the most part these changes affect BSD administrators, not programmers, and thus will not affect any of the interfaces discussed in this book. Notable exceptions are discussed below.

8.2 Boot Layout

The first change is the organization of the boot files. The FreeBSD-5.x system has moved all modules and kernels under the /boot directory. This directory still holds all boot related configuration files, as in previous versions of FreeBSD. This seemingly minor change actually allows for more flexibility, because / and /boot (including all kernels and kernel modules) are now much easier to separate between devices.

8.3 Devfs

Our favorite new feature in FreeBSD-5.x is devfs. Prior to this branch, the /dev directory was filled with more than a thousand files. These files where device nodes and almost every supported device had an entry as a file in the /dev directory. As one can surmise, this directory became quite big and contained a lot of unnecessary files. For example, a system with IDE drives will contain device entries for scsi devices in /dev, even though the system does not have a scsi device. Such annoyances are gone with devfs. The only entries in /dev now are for devices that actually exist on the system. In fact, unlike prior releases /dev is only a mount point to a file system called devfs.

The devfs file system is similar to the proc file system. They both are mount points that contain files that don't exist on the hard disk. These files are created by the kernel and only appear as files. In fact, they are created after the system boots. Devfs allows much more flexibility because you can support multiple devfs mount points. For example if you wanted to chroot or jail a process, you don't have to create the /dev directory by hand, but instead, you can simply create a new mount point for your process and mount devfs.

Another benefit of devfs is that you can easily tell what devices a system has. All you need to do is just cd into the /dev directory and list the files. This is a very convenient way for a user to tell what devices are on a box, or more importantly, what devices have been detected.

8.4 a.out

FreeBSD-5.x series removes support for a.out binaries in the base system. You can still add support for a.out binaries, though. The reasoning behind this is that the a.out format is fairly old and the newer ELF format is more preferred. The ELF format is more flexible and more commonly used today.

8.5 gcc-3.2 tool chain

FreeBSD-5.x now uses the gcc-3.2 tool chain for the base system. This is a significant change: gcc-3.x is more ISO complaint, and its C++ ABI is much more stable. However, this might cause some growing pains for a few people. We have had issues with some code that required updating prior to compiling with gcc-3.x. If you use flex or yacc, make sure you have the latest versions, or patch your current version because they have known issues.

8.6 SMPng

FreeBSD-5.x has improved support for Shared Memory Multiprocessor (SMP) systems, under a project commonly referred to as SMPng (SMP next generation). Although previous versions of FreeBSD did have support for SMP, performance needed improvements.

8.7 Kernel Scheduler Entities (KSE)

Another new feature is the Kernel Scheduler Entities (KSE). KSE is a kernel-supported threading system conceptually similar to Scheduler Activations. On the kernel side, specifically, KSE consists of modifications to the FreeBSD scheduler; and on the user half is a POSIX threads implementation which utilizes the extra facilities provided by the kernel. However, as the changes are inextricable, you don't need special kernel configuration options to build a kernel with the KSE-related changes.

In order to use KSE in an application, you link it against libpthreads. These are not built by default, so first, build and install lipbthreads on your system. Then, in its makefiles, change the -pthread option to -lpthread and relink.

8.8 Conclusion

FreeBSD has matured over the years and now is one of the most stable operating system available. With the changes to improve SMP support and kernel threads, FreeBSD will continue the rich tradition of offering rock solid stability and high performance. Support for new platforms have been added, such as sparc64 and ia64. These new platforms will help BSD distributions to continue to offer a high quality open source alternative for many years to come.

Prev top

Chapter 8
FreeBSD System Programming

Source code
FreeBSD System Programming
top

Source for each chapter

- Chapter 3
- Chapter 4
- Chapter 5
- Chapter 6
- Chapter 7

Other files

- Top level Makefile and the src
- Top level make include file _bsd_make.mk and the src