FreeBSD 开发手册

The FreeBSD Documentation Project

FreeBSD 中文计划

FreeBSD 开发手册

由The FreeBSD Documentation Project, FreeBSD 中文计划
出版方2000 年8 月
出版方2005 年12 月
版权 © 2000, 2001, 2002, 2003, 2004, 2005 The FreeBSD Documentation Project
版权 © 2005 FreeBSD 中文计划

欢迎您阅读《FreeBSD开发手册》。这本手册还在不断由许多人继续书写。许多章节还是空白，有的章节亟待更新。如果您对这个项目感兴趣并愿意有所贡献，请发信给FreeBSD 文档计划邮件列表 (http://lists.FreeBSD.org/mailman/listinfo/freebsd-doc)。

本文档的最新英文原始版本可从FreeBSD Web 站点 (http://www.FreeBSD.org/index.html)获得，最新中文译本可从FreeBSD 中文计划Web 站点 (http://www.freebsd.org.cn/)获得。也可以各种格式和压缩形式从FreeBSD FTP 服务器 (ftp://ftp.FreeBSD.org/pub/FreeBSD/doc/) 或众多的镜像站点 (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/handbook/mirrors-ftp.html) 得到。

重要: 本文中许可证的非官方中文翻译仅供参考，不作为判定任何责任的依据。如与英文原文有出入，则以英文原文为准。

在满足下列许可条件的前提下，允许再分发或以源代码(SGML DocBook) 或“编译”(SGML, HTML, PDF, PostScript, RTF 等) 的经过修改或未修改的形式:

1. 再分发源代码(SGML DocBook) 必须不加修改的保留上述版权告示、本条件清单和下述弃权书作为该文件的最先若干行。
2. 再分发编译的形式(转换为其它DTD、PDF、PostScript、RTF 或其它形式)，必须将上述版权告示、本条件清单和下述弃权书复制到与分发品一同提供的文件，以及其它材料中。

重要: 本文档由FREEBSD DOCUMENTATION PROJECT “按现状条件”提供，并在此明示不提供任何明示或暗示的保障，包括但不限于对商业适销性、对特定目的的适用性的暗示保障。任何情况下，FREEBSD DOCUMENTATION PROJECT 均不对任何直接、间接、偶然、特殊、惩罚性的，或必然的损失(包括但不限于替代商品或服务的采购、使用、数据或利益的损失或营业中断) 负责，无论是如何导致的并以任何有责任逻辑的，无论是否是在本文档使用以外以任何方式产生的契约、严格责任或是民事侵权行为(包括疏忽或其它)中的，即使已被告知发生该损失的可能性。

Redistribution and use in source (SGML DocBook) and 'compiled' forms (SGML, HTML, PDF, PostScript, RTF and so forth) with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code (SGML DocBook) must retain the above copyright notice, this list of conditions and the following disclaimer as the first lines of this file unmodified.
2. Redistributions in compiled form (transformed to other DTDs, converted to PDF, PostScript, RTF and other formats) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

重要: THIS DOCUMENTATION IS PROVIDED BY THE FREEBSD DOCUMENTATION PROJECT "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FREEBSD DOCUMENTATION PROJECT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

重要: 从FreeBSD 和FreeBSD 中文化项目官方网站获取的本文档PostScript® 和PDF 格式文件中内嵌的中文字体，由文鼎科技开发股份有限公司按《文鼎公众授权书》授权提供。

文鼎公众授权书

版权所有(C) 1999 文鼎科技开发股份有限公司

台湾110台北市永吉路168号11楼

除以下规定外，保留所有权利。

任何人均可以复制及分发本授权书原始文件的复本，但禁止修改之。

前言

绝大多数软件的授权书是用来剥夺您共享和修改软件的自由。相反，如果您将我们所授与您的一切权利授与您的接受者，并且确保他们能得到本软件的修改物，文鼎公众授权书特别许可并鼓励您使用本软件。

法律性条款

0. 定义

本授权书中，「字型」是指「文鼎PL细上海宋」、「文鼎PL中楷」（BIG-5码）和「文鼎PL简报宋」、「文鼎PL简中楷」（GB码）等TrueType字型，以及经修改而产生的该等字型的衍生物，包括修改字型样式、更动编码顺序、转换格式、变更字型名称，或者在glyph table 中增减一些文字。

「PL」是指「Public License」（公众授权）。

「版权人」是指在字型版权声明中被称为版权所有者之人。

「您」是指被授权者或复制、再散布、修改字型的人。

「自由使用」是指您在所接受的相同条件下，有复制、修改字型及散布字型复制品的自由，而非指价格。如果您愿意，可以对此项服务收取费用。

1. 复制及分发

如果在所有复制品维持本授权书文件（ARPHICPL.TXT）不变，您可以在任何媒体无限制复制及分发本字型之复制品。

2. 修改

如果满足以下条件，您可以用任何方式修改本字型的复制品，包括修改字型样式、更换编码顺序、转换格式、变更字型名称，或者在glyph table 中增减一些文字，并且依据上述第一条的规定复制及分发该修改物。

a)您必须在每个修改过的文件中插入明显的声明，表示您如何及何时修改了该文件。

b)您必须允许所有第三方在本授权书条款下，得以「自由使用」该修改物的全部，例如通过提供指定地点的复制路径，或在通常作为软件交换的媒体上发布该修改物。

c)如果修改过的字型在执行时以交谈方式读取命令，您必须使它在开始进入一般的交谈使用方式时打印或显示声明，包括适当的版权声明、无担保声明（或者其他您提供担保的声明），以及使用者可以依据这些条件再分发本字型的声明，并且告知使用者如何看到本授权书的复本。

这些要求适用于整个修改过的作品。如果作品的可辨识部分并非衍生自本字型，且可以合理地认其为独立而个别的作品，则当您将它当作个别作品分发时，本授权书及其条款将不适用于该部分。因此，仅将非基于本字型之其他作品和本字型一起放入贮存或分发媒体的同一卷上，并不会使该其他作品纳入本授权书的适用范围。

3. 解除条件

除非依照本授权书的规定，否则您不能复制、修改、再授权或分发本字型。任何用其他方式复制、修改、再授权或分发本字型的意图，将使本授权书所授与您的权利自动失效。不过，从您那依据本授权书取得复制品和权利的人，只要他们完全遵行条款，其权利将维持有效。

4. 同意

因您未签订本授权书，所以没必要遵行它。然而，没有任何其他文件授与您复制、修改、再授权或分发本字型之权利，如果您不同意本授权书，这些行为是法律所禁止的。因此一旦复制、修改、再授权或分发本字型，就表示您接受本授权书及其一切条款和条件。

5. 自动取得授权

每当您再散布本字型时，收受者便自动从原始授权者处取得依据这些条款和条件复制、分发或修改本字型的授权。您不可对在此所授与收受者之权利实行强加其他限制，您也没有使第三方遵行本授权书的义务。

6. 抵触

若因法院判决、违反专利宣称或任何其他原因（不限于专利纠纷）的结果，使强加于您的条件（不论是法院命令、协议或其他）和授权书的条件有所冲突时，也不能让您违背授权书的条件。如果您不能同时满足本授权书所定的义务及其他相关义务来分发时，那麼您根本不可分发本字型。例如，如果专利授权不许所有直接或间接从您那里取得复制品者，在免授权费的情况下再分发本字型时，唯一能同时满足双方要求的方法是完全避免分发本字型。

如果本条款之任何部分在特定情况下无效或无法实施，条款的其他部分将继续适用，并作为整体适用于其他情况。

7. 无担保

由于本字型是免费授权，在法律允许的范围内，本字型将不附带任何担保。除非另有书面说明，版权人或其他提供本字型者「同样」不提供任何种类的担保，不论是明示还是暗示，包括但不限于销售性和适于特定目的的暗示担保。有关本字型品质和执行的一切危险均由您承担。如果本字型出现瑕疵，您应承担一切所需服务、修复和改正的费用。

8. 损害赔偿请求权的放弃

除非法律规定或以书面形式同意，在任何情况下，任何版权人或以上被允许得以复制、修改或再分发本字型者，均不对您因使用或不能使用本字型（包括但不限于替代商品或服务的采购；使用、数据或利益之损失；或者营业妨碍）所引致的任何直接、间接、必然、偶然、特殊或惩罚性损失负责，即使版权人或其他人已知该损失的可能性时亦不例外。

FreeBSD 是FreeBSD基金会的注册商标

Apple, FireWire, Mac, Macintosh, Mac OS, Quicktime, 以及TrueType 是Apple Computer, Inc. 在美国以及其他国家的注册商标。

IBM, AIX, EtherJet, Netfinity, OS/2, PowerPC, PS/2, S/390, 和ThinkPad 是国际商用机器公司在美国和其他国家的注册商标或商标。

IEEE, POSIX, 和802 是Institute of Electrical and Electronics Engineers, Inc. 在美国的注册商标。

Intel, Celeron, EtherExpress, i386, i486, Itanium, Pentium, 和Xeon 是Intel Corporation 及其分支机构在美国和其他国家的商标或注册商标。

Linux 是Linus Torvalds 的注册商标。

Microsoft, IntelliMouse, MS-DOS, Outlook, Windows, Windows Media, 和Windows NT 是Microsoft Corporation 在美国和/或其他国家的商标或注册商标。

Motif, OSF/1, 和UNIX 是The Open Group 在美国和其他国家的注册商标；IT DialTone 和The Open Group 是其商标。

Sun, Sun Microsystems, Java, Java Virtual Machine, JavaServer Pages, JDK, JSP, JVM, Netra, Solaris, StarOffice, Sun Blade, Sun Enterprise, Sun Fire, SunOS, 和Ultra 是Sun Microsystems, Inc. 在美国和其他国家的商标或注册商标。

许多制造商和经销商使用一些称为商标的图案或文字设计来彰显自己的产品。本文档中出现的，为FreeBSD Project 所知晓的商标，后面将以’™’ 或’®’ 符号来标注。

目录

I. 基础 viii

1 介绍 1
1.1 在FreeBSD平台上编程 1
1.2 BSD 观点 1
1.3 基本指导方针 1
1.4 `/usr/src`的层次结构 1

2 Programming Tools 3
2.1 Synopsis 3
2.2 Introduction 3
2.3 Introduction to Programming 3
2.4 Compiling with `cc` 6
2.5 Make 12
2.6 Debugging 16
2.7 Using Emacs as a Development Environment 20
2.8 Further Reading 28

3 安全的编程 30
3.1 提要 30
3.2 安全的设计方法 30
3.3 缓冲区溢出 30
3.4 SetUID 问题 32
3.5 限制你的程序环境 32
3.6 信任 33
3.7 竞态条件 34

4 本地化与国际化- L10N 和I18N 35
4.1 编写适应国际化的应用程序 35

5 Source Tree Guidelines and Policies 36
5.1 `MAINTAINER` on Makefiles 36
5.2 Contributed Software 36
5.3 Encumbered Files 38
5.4 Shared Libraries 39

6 Regression and Performance Testing 41
6.1. Micro Benchmark Checklist 41

II. 进程间通信 43

7 套接字 44
7.1 概述 44
7.2 联网和多样性 44
7.3 协议 44
7.4 套接字模型 46
7.5 重要的套接字函数 47
7.6 辅助函数 60
7.7 并发服务器 62

8 IPv6 Internals 65
8.1 IPv6/IPsec Implementation 65

III. 内核 82
9 DMA 83
9.1 DMA: What it is and How it Works 83
10 调试内核 94
10.1 如何将内核的崩溃转存数据保存成文件 94
10.2 Debugging a Kernel Crash Dump with `kgdb` 95
10.3 Debugging a Crash Dump with DDD 99
10.4 Post-Mortem Analysis of a Dump 100
10.5 On-Line Kernel Debugging Using DDB 100
10.6 On-Line Kernel Debugging Using Remote GDB 103
10.7 Debugging Loadable Modules Using GDB 104
10.8 Debugging a Console Driver 105
IV. 系统结构 106
11 x86 汇编语言 107
11.1 概述 107
11.2 工具 107
11.3 系统调用 107
11.4 返回值 110
11.5 建立可移植的代码 111
11.6 编写第一个程序 115
11.7 Writing UNIX® Filters 116
11.8 Buffered Input and Output 119
11.9 Command Line Arguments 125
11.10 UNIX Environment 130
11.11 Working with Files 134
11.12 One-Pointed Mind 145
11.13 Using the FPU 153
11.14 Caveats 180
11.15 Acknowledgements 182
V. 附录 183
参考书目 184
索引 184

范例清单

2-1. A sample `.emacs` file 22

I. 基础

第1章
介绍

供稿：*Murray Stokely* 和*Jeroen Ruigrok van der Werven.* 翻译：*Shi Jerome.*

1.1 在FreeBSD平台上编程

阅读本章之前,我们应该已安装好操作系统, 准备开始编程了. FreeBSD为程序员做了哪些工作呢?提供了哪些工具呢?

本章将回答大家一些问题. 当然各人编程的熟练程度不同, 对某些人而言写程序只是出于爱好,但这却是另一些人的职业. 本章内容主要针对初学者, 但对第一次接触FreeBSD平台的职业程序员也非常有用.

1.2 BSD 观点

要写出最好的在类UNIX® 操作系统上运行程序, 就必须尊重原软件工具的可用性, 可观性, 稳定性, 和原创者的思想.

1.3 基本指导方针

下面几条原则阐述了我们的观点

除非使用者在缺少这一功能的情况下不能完成所需工作, 否则不应添加任何新功能.

和决定系统应具备哪些功能一样重要的是: 决定哪些功能系统不应有. 系统不可能满足所有人的需求. 当然系统应具有良好的扩展性以跟上用户需求不断发展的潮流.

比仅有一个案例可供借鉴更糟的情况是根本无例可循.

在未充分理解问题之前,最好不要忙于解决.

可以用10%的代码完成90%的工作, 请用它.

越简单越好.

提供机制而不是策略, 尤其重要的是应由客户决定用户界面的策略.

摘自Scheifler & Gettys: "X Window System"

1.4 `/usr/src`的层次结构

FreeBSD的所有原代码均可从公共的CVS库里获取. 通常原代码已安装在`/usr/src` 这一目录含下列子目录:

目录	说明

目录	说明
bin/	/bin中文件的源代码
contrib/	第三方软件文件的源代码
crypto/	加密源代码
etc/	/etc中文件的源代码
games/	/usr/games中文件的源代码
gnu/	《GNU公共许可证》覆盖的工具
include/	/usr/include中文件的源代码
kerberos5/	Kerberos 版本5 源代码
lib/	/usr/lib中文件的源代码
libexec/	/usr/libexec中文件的源代码
release/	产生一个FreeBSD 发行版所需文件
rescue/	建造系统时/rescue中的工具
sbin/	/sbin中文件的源代码
secure/	FreeSec 源代码
share/	/usr/share中文件的源代码
sys/	内核源代码文件
tools/	用于维护和测试FreeBSD的工具
usr.bin/	/usr/bin中文件的源代码
usr.sbin/	/usr/sbin中文件的源代码

第2章
Programming Tools

Contributed by James Raynard 和Murray Stokely.

2.1 Synopsis

This chapter is an introduction to using some of the programming tools supplied with FreeBSD, although much of it will be applicable to many other versions of UNIX. It does *not* attempt to describe coding in any detail. Most of the chapter assumes little or no previous programming knowledge, although it is hoped that most programmers will find something of value in it.

2.2 Introduction

FreeBSD offers an excellent development environment. Compilers for C, C++, and Fortran and an assembler come with the basic system, not to mention a Perl interpreter and classic UNIX tools such as `sed` and `awk`. If that is not enough, there are many more compilers and interpreters in the Ports collection. FreeBSD is very compatible with standards such as POSIX® and ANSI C, as well with its own BSD heritage, so it is possible to write applications that will compile and run with little or no modification on a wide range of platforms.

However, all this power can be rather overwhelming at first if you have never written programs on a UNIX platform before. This document aims to help you get up and running, without getting too deeply into more advanced topics. The intention is that this document should give you enough of the basics to be able to make some sense of the documentation.

Most of the document requires little or no knowledge of programming, although it does assume a basic competence with using UNIX and a willingness to learn!

2.3 Introduction to Programming

A program is a set of instructions that tell the computer to do various things; sometimes the instruction it has to perform depends on what happened when it performed a previous instruction. This section gives an overview of the two main ways in which you can give these instructions, or "commands" as they are usually called. One way uses an *interpreter*, the other a *compiler*. As human languages are too difficult for a computer to understand in an unambiguous way, commands are usually written in one or other languages specially designed for the purpose.

2.3.1 Interpreters

With an interpreter, the language comes as an environment, where you type in commands at a prompt and the environment executes them for you. For more complicated programs, you can type the commands into a file and get the interpreter to load the file and execute the commands in it. If anything goes wrong, many interpreters will drop you into a debugger to help you track down the problem.

The advantage of this is that you can see the results of your commands immediately, and mistakes can be corrected readily. The biggest disadvantage comes when you want to share your programs with someone. They must have the same interpreter, or you must have some way of giving it to them, and they need to understand how to use it. Also users may not appreciate being thrown into a debugger if they press the wrong key! From a performance point of view, interpreters can use up a lot of memory, and generally do not generate code as efficiently as compilers.

In my opinion, interpreted languages are the best way to start if you have not done any programming before. This kind of environment is typically found with languages like Lisp, Smalltalk, Perl and Basic. It could also be argued that the UNIX shell (`sh`, `csh`) is itself an interpreter, and many people do in fact write shell "scripts" to help with various "housekeeping" tasks on their machine. Indeed, part of the original UNIX philosophy was to provide lots of small utility programs that could be linked together in shell scripts to perform useful tasks.

2.3.2 Interpreters available with FreeBSD

Here is a list of interpreters that are available from the FreeBSD Ports Collection, with a brief discussion of some of the more popular interpreted languages.

Instructions on how to get and install applications from the Ports Collection can be found in the Ports section (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/handbook/ports-using.html) of the handbook.

BASIC

Short for Beginner's All-purpose Symbolic Instruction Code. Developed in the 1950s for teaching University students to program and provided with every self-respecting personal computer in the 1980s, BASIC has been the first programming language for many programmers. It is also the foundation for Visual Basic.

The Bywater Basic Interpreter can be found in the Ports Collection as `lang/bwbasic` and the Phil Cockroft's Basic Interpreter (formerly Rabbit Basic) is available as `lang/pbasic`.

Lisp

A language that was developed in the late 1950s as an alternative to the "number-crunching" languages that were popular at the time. Instead of being based on numbers, Lisp is based on lists; in fact the name is short for "List Processing". Very popular in AI (Artificial Intelligence) circles.

Lisp is an extremely powerful and sophisticated language, but can be rather large and unwieldy.

Various implementations of Lisp that can run on UNIX systems are available in the Ports Collection for FreeBSD. GNU Common Lisp can be found as `lang/gcl`. CLISP by Bruno Haible and Michael Stoll is available as `lang/clisp`. For CMUCL, which includes a highly-optimizing compiler too, or simpler Lisp implementations like SLisp, which implements most of the Common Lisp constructs in a few hundred lines of C code, `lang/cmucl` and `lang/slisp` are available respectively.

Perl

Very popular with system administrators for writing scripts; also often used on World Wide Web servers for writing CGI scripts.

Perl is available in the Ports Collection as `lang/perl5` for all FreeBSD releases, and is installed as `/usr/bin/perl` in the base system 4.X releases.

Scheme

A dialect of Lisp that is rather more compact and cleaner than Common Lisp. Popular in Universities as it is simple enough to teach to undergraduates as a first language, while it has a high enough level of abstraction to be used in research work.

Scheme is available from the Ports Collection as `lang/elk` for the Elk Scheme Interpreter. The MIT Scheme Interpreter can be found in `lang/mit-scheme` and the SCM Scheme Interpreter in `lang/scm`.

Icon

Icon is a high-level language with extensive facilities for processing strings and structures. The version of Icon for FreeBSD can be found in the Ports Collection as `lang/icon`.

Logo

Logo is a language that is easy to learn, and has been used as an introductory programming language in various courses. It is an excellent tool to work with when teaching programming in small ages, as it makes the creation of elaborate geometric shapes an easy task even for very small children.

The lastest version of Logo for FreeBSD is available from the Ports Collection in `lang/logo`.

Python

Python is an Object-Oriented, interpreted language. Its advocates argue that it is one of the best languages to start programming with, since it is relatively easy to start with, but is not limited in comparison to other popular interpreted languages that are used for the development of large, complex applications (Perl and Tcl are two other languages that are popular for such tasks).

The latest version of Python is available from the Ports Collection in `lang/python`.

Ruby

Ruby is an interpreter, pure object-oriented programming language. It has become widely popular because of its easy to understand syntax, flexibility when writing code, and the ability to easily develop and maintain large, complex programs.

Ruby is available from the Ports Collection as `lang/ruby18`.

Tcl and Tk

Tcl is an embeddable, interpreted language, that has become widely used and became popular mostly because of its portability to many platforms. It can be used both for quickly writing small, prototype applications, or (when combined with Tk, a GUI toolkit) fully-fledged, featureful programs.

Various versions of Tcl are available as ports for FreeBSD. The latest version, Tcl 8.4, can be found in `lang/tcl84`.

2.3.3 Compilers

Compilers are rather different. First of all, you write your code in a file (or files) using an editor. You then run the compiler and see if it accepts your program. If it did not compile, grit your teeth and go back to the editor; if it did compile and gave you a program, you can run it either at a shell command prompt or in a debugger to see if it works properly. [1]

Obviously, this is not quite as direct as using an interpreter. However it allows you to do a lot of things which are very difficult or even impossible with an interpreter, such as writing code which interacts closely with the operating system——or even writing your own operating system! It is also useful if you need to write very efficient code, as the compiler can take its time and optimize the code, which would not be acceptable in an interpreter. Moreover, distributing a program written for a compiler is usually more straightforward than one written for an interpreter——you can just give them a copy of the executable, assuming they have the same operating system as you.

Compiled languages include Pascal, C and C++. C and C++ are rather unforgiving languages, and best suited to more experienced programmers; Pascal, on the other hand, was designed as an educational language, and is quite a good language to start with. FreeBSD does not include Pascal support in the base system, but both GNU Pascal Compiler (GPC) and the Free Pascal Compiler are available in the ports collection as `lang/gpc` and `lang/fpc`.

As the edit-compile-run-debug cycle is rather tedious when using separate programs, many commercial compiler makers have produced Integrated Development Environments (IDEs for short). FreeBSD does not include an IDE in the base system, but `devel/kdevelop` is available in the ports tree and many use **Emacs** for this purpose. Using **Emacs** as an IDE is discussed in 第2.7 节.

2.4 Compiling with `cc`

This section deals only with the GNU compiler for C and C++, since that comes with the base FreeBSD system. It can be invoked by either `cc` or `gcc`. The details of producing a program with an interpreter vary considerably between interpreters, and are usually well covered in the documentation and on-line help for the interpreter.

Once you have written your masterpiece, the next step is to convert it into something that will (hopefully!) run on FreeBSD. This usually involves several steps, each of which is done by a separate program.

1. Pre-process your source code to remove comments and do other tricks like expanding macros in C.
2. Check the syntax of your code to see if you have obeyed the rules of the language. If you have not, it will complain!
3. Convert the source code into assembly language——this is very close to machine code, but still understandable by humans. Allegedly. [2]
4. Convert the assembly language into machine code——yep, we are talking bits and bytes, ones and zeros here.
5. Check that you have used things like functions and global variables in a consistent way. For example, if you have called a non-existent function, it will complain.
6. If you are trying to produce an executable from several source code files, work out how to fit them all together.
7. Work out how to produce something that the system's run-time loader will be able to load into memory and run.
8. Finally, write the executable on the filesystem.

The word *compiling* is often used to refer to just steps 1 to 4——the others are referred to as *linking*. Sometimes step 1 is referred to as *pre-processing* and steps 3-4 as *assembling*.

Fortunately, almost all this detail is hidden from you, as `cc` is a front end that manages calling all these programs with the right arguments for you; simply typing

```
% cc foobar.c
```

will cause `foobar.c` to be compiled by all the steps above. If you have more than one file to compile, just do something like

```
% cc foo.c bar.c
```

Note that the syntax checking is just that——checking the syntax. It will not check for any logical mistakes you may have made, like putting the program into an infinite loop, or using a bubble sort when you meant to use a binary sort. [3]

There are lots and lots of options for `cc`, which are all in the manual page. Here are a few of the most important ones, with examples of how to use them.

`-o` *filename*

The output name of the file. If you do not use this option, `cc` will produce an executable called `a.out`. [4]

```
% cc foobar.c                  executable is a.out
% cc -o foobar foobar.c        executable is foobar
```

`-c`

Just compile the file, do not link it. Useful for toy programs where you just want to check the syntax, or if you are using a `Makefile`.

```
% cc -c foobar.c
```

This will produce an *object file* (not an executable) called `foobar.o`. This can be linked together with other object files into an executable.

`-g`

Create a debug version of the executable. This makes the compiler put information into the executable about which line of which source file corresponds to which function call. A debugger can use this information to show the source code as you step through the program, which is *very* useful; the disadvantage is that all this extra information makes the program much bigger. Normally, you compile with `-g` while you are developing a program and then compile a "release version" without `-g` when you are satisfied it works properly.

```
% cc -g foobar.c
```

This will produce a debug version of the program. [5]

`-O`

Create an optimized version of the executable. The compiler performs various clever tricks to try to produce an executable that runs faster than normal. You can add a number after the `-O` to specify a higher level of optimization, but this often exposes bugs in the compiler's optimizer. For instance, the version of `cc` that comes with the 2.1.0 release of FreeBSD is known to produce bad code with the `-O2` option in some circumstances.

Optimization is usually only turned on when compiling a release version.

```
% cc -O -o foobar foobar.c
```

This will produce an optimized version of `foobar`.

The following three flags will force `cc` to check that your code complies to the relevant international standard, often referred to as the ANSI standard, though strictly speaking it is an ISO standard.

`-Wall`

Enable all the warnings which the authors of `cc` believe are worthwhile. Despite the name, it will not enable all the warnings `cc` is capable of.

`-ansi`

Turn off most, but not all, of the non-ANSI C features provided by `cc`. Despite the name, it does not guarantee strictly that your code will comply to the standard.

`-pedantic`

Turn off *all* `cc`'s non-ANSI C features.

Without these flags, `cc` will allow you to use some of its non-standard extensions to the standard. Some of these are very useful, but will not work with other compilers——in fact, one of the main aims of the standard is to allow people to write code that will work with any compiler on any system. This is known as *portable code*.

Generally, you should try to make your code as portable as possible, as otherwise you may have to completely rewrite the program later to get it to work somewhere else——and who knows what you may be using in a few years time?

```
% cc -Wall -ansi -pedantic -o foobar foobar.c
```

This will produce an executable `foobar` after checking `foobar.c` for standard compliance.

`-l`*`library`*

Specify a function library to be used at link time.

The most common example of this is when compiling a program that uses some of the mathematical functions in C. Unlike most other platforms, these are in a separate library from the standard C one and you have to tell the compiler to add it.

The rule is that if the library is called `lib`*`something`*`.a`, you give `cc` the argument `-l`*`something`*. For example, the math library is `libm.a`, so you give `cc` the argument `-lm`. A common "gotcha" with the math library is that it has to be the last library on the command line.

```
% cc -o foobar foobar.c -lm
```

This will link the math library functions into `foobar`.

If you are compiling C++ code, you need to add `-lg++`, or `-lstdc++` if you are using FreeBSD 2.2 or later, to the command line argument to link the C++ library functions. Alternatively, you can run `c++` instead of `cc`, which does this for you. `c++` can also be invoked as `g++` on FreeBSD.

```
% cc -o foobar foobar.cc -lg++        For FreeBSD 2.1.6 and earlier
```

```
% cc -o foobar foobar.cc -lstdc++   For FreeBSD 2.2 and later
% c++ -o foobar foobar.cc
```

Each of these will both produce an executable `foobar` from the C++ source file `foobar.cc`. Note that, on UNIX systems, C++ source files traditionally end in `.C`, `.cxx` or `.cc`, rather than the MS-DOS® style `.cpp` (which was already used for something else). `gcc` used to rely on this to work out what kind of compiler to use on the source file; however, this restriction no longer applies, so you may now call your C++ files `.cpp` with impunity!

2.4.1 Common `cc` Queries and Problems

1. I am trying to write a program which uses the `sin()` function and I get an error like this. What does it mean?

```
/var/tmp/cc0143941.o: Undefined symbol `_sin' referenced from text segment
```

When using mathematical functions like `sin()`, you have to tell `cc` to link in the math library, like so:

```
% cc -o foobar foobar.c -lm
```

2. All right, I wrote this simple program to practice using `-lm`. All it does is raise 2.1 to the power of 6.

```
#include <stdio.h>

int main() {
float f;

f = pow(2.1, 6);
printf("2.1 ^ 6 = %f\n", f);
return 0;
}
```

and I compiled it as:

```
% cc temp.c -lm
```

like you said I should, but I get this when I run it:

```
% ./a.out
2.1 ^ 6 = 1023.000000
```

This is *not* the right answer! What is going on?

When the compiler sees you call a function, it checks if it has already seen a prototype for it. If it has not, it assumes the function returns an int, which is definitely not what you want here.

3. So how do I fix this?

The prototypes for the mathematical functions are in `math.h`. If you include this file, the compiler will be able to find the prototype and it will stop doing strange things to your calculation!

```
#include <math.h>
#include <stdio.h>

int main() {
...
```

After recompiling it as you did before, run it:

```
% ./a.out
2.1 ^ 6 = 85.766121
```

If you are using any of the mathematical functions, *always* include `math.h` and remember to link in the math library.

4. I compiled a file called `foobar.c` and I cannot find an executable called `foobar`. Where has it gone?

Remember, `cc` will call the executable `a.out` unless you tell it differently. Use the `-o` *`filename`* option:

```
% cc -o foobar foobar.c
```

5. OK, I have an executable called `foobar`, I can see it when I run `ls`, but when I type in `foobar` at the command prompt it tells me there is no such file. Why can it not find it?

Unlike MS-DOS, UNIX does not look in the current directory when it is trying to find out which executable you want it to run, unless you tell it to. Either type `./foobar`, which means "run the file called `foobar` in the current directory", or change your `PATH` environment variable so that it looks something like

```
bin:/usr/bin:/usr/local/bin:.
```

The dot at the end means "look in the current directory if it is not in any of the others".

6. I called my executable `test`, but nothing happens when I run it. What is going on?

Most UNIX systems have a program called `test` in `/usr/bin` and the shell is picking that one up before it gets to checking the current directory. Either type:

```
% ./test
```

or choose a better name for your program!

7. I compiled my program and it seemed to run all right at first, then there was an error and it said something about `core dumped`. What does that mean?

The name *core dump* dates back to the very early days of UNIX, when the machines used core memory for storing data. Basically, if the program failed under certain conditions, the system would write the contents of core memory to disk in a file called `core`, which the programmer could then pore over to find out what went wrong.

8. Fascinating stuff, but what I am supposed to do now?

Use `gdb` to analyze the core (see 第2.6 节).

9. When my program dumped core, it said something about a `segmentation fault`. What is that?

This basically means that your program tried to perform some sort of illegal operation on memory; UNIX is designed to protect the operating system and other programs from rogue programs.

Common causes for this are:

Trying to write to a NULL pointer, eg

```
char *foo = NULL;
strcpy(foo, "bang!");
```

Using a pointer that has not been initialized, eg

```
char *foo;
strcpy(foo, "bang!");
```

The pointer will have some random value that, with luck, will point into an area of memory that is not available to your program and the kernel will kill your program before it can do any damage. If you are unlucky, it will point somewhere inside your own program and corrupt one of your data structures, causing the program to fail mysteriously.

Trying to access past the end of an array, eg

```
int bar[20];
bar[27] = 6;
```

Trying to store something in read-only memory, eg

```
char *foo = "My string";
strcpy(foo, "bang!");
```

UNIX compilers often put string literals like `"My string"` into read-only areas of memory.

Doing naughty things with `malloc()` and `free()`, eg

```
char bar[80];
```

```
free(bar);
```

or

```
char *foo = malloc(27);
free(foo);
free(foo);
```

Making one of these mistakes will not always lead to an error, but they are always bad practice. Some systems and compilers are more tolerant than others, which is why programs that ran well on one system can crash when you try them on an another.

10. Sometimes when I get a core dump it says `bus error`. It says in my UNIX book that this means a hardware problem, but the computer still seems to be working. Is this true?

No, fortunately not (unless of course you really do have a hardware problem. . .). This is usually another way of saying that you accessed memory in a way you should not have.

11. This dumping core business sounds as though it could be quite useful, if I can make it happen when I want to. Can I do this, or do I have to wait until there is an error?

Yes, just go to another console or xterm, do

```
% ps
```

to find out the process ID of your program, and do

```
% kill -ABRT pid
```

where *pid* is the process ID you looked up.

This is useful if your program has got stuck in an infinite loop, for instance. If your program happens to trap SIGABRT, there are several other signals which have a similar effect.

Alternatively, you can create a core dump from inside your program, by calling the `abort()` function. See the manual page of abort(3) to learn more.

If you want to create a core dump from outside your program, but do not want the process to terminate, you can use the `gcore` program. See the manual page of gcore(1) for more information.

2.5 Make

2.5.1 What is `make`?

When you are working on a simple program with only one or two source files, typing in

```
% cc file1.c file2.c
```

is not too bad, but it quickly becomes very tedious when there are several files——and it can take a while to compile, too.

One way to get around this is to use object files and only recompile the source file if the source code has changed. So we could have something like:

```
% cc file1.o file2.o ... file37.c ...
```

if we had changed `file37.c`, but not any of the others, since the last time we compiled. This may speed up the compilation quite a bit, but does not solve the typing problem.

Or we could write a shell script to solve the typing problem, but it would have to re-compile everything, making it very inefficient on a large project.

What happens if we have hundreds of source files lying about? What if we are working in a team with other people who forget to tell us when they have changed one of their source files that we use?

Perhaps we could put the two solutions together and write something like a shell script that would contain some kind of magic rule saying when a source file needs compiling. Now all we need now is a program that can understand these rules, as it is a bit too complicated for the shell.

This program is called `make`. It reads in a file, called a *makefile*, that tells it how different files depend on each other, and works out which files need to be re-compiled and which ones do not. For example, a rule could say something like "if `fromboz.o` is older than `fromboz.c`, that means someone must have changed `fromboz.c`, so it needs to be re-compiled." The makefile also has rules telling make *how* to re-compile the source file, making it a much more powerful tool.

Makefiles are typically kept in the same directory as the source they apply to, and can be called `makefile`, `Makefile` or `MAKEFILE`. Most programmers use the name `Makefile`, as this puts it near the top of a directory listing, where it can easily be seen. [6]

2.5.2 Example of using `make`

Here is a very simple make file:

```
foo: foo.c
cc -o foo foo.c
```

It consists of two lines, a dependency line and a creation line.

The dependency line here consists of the name of the program (known as the *target*), followed by a colon, then whitespace, then the name of the source file. When `make` reads this line, it looks to see if `foo` exists; if it exists, it compares the time `foo` was last modified to the time `foo.c` was last modified. If `foo` does not exist, or is older than `foo.c`, it then looks at the creation line to find out what to do. In other words, this is the rule for working out when `foo.c` needs to be re-compiled.

The creation line starts with a tab (press the **tab** key) and then the command you would type to create `foo` if you were doing it at a command prompt. If `foo` is out of date, or does not exist, `make` then executes this command to create it. In other words, this is the rule which tells make how to re-compile `foo.c`.

So, when you type **`make`**, it will make sure that `foo` is up to date with respect to your latest changes to `foo.c`. This principle can be extended to `Makefile`s with hundreds of targets——in fact, on FreeBSD, it is possible to compile the entire operating system just by typing **`make world`** in the appropriate directory!

Another useful property of makefiles is that the targets do not have to be programs. For instance, we could have a make file that looks like this:

```
foo: foo.c
cc -o foo foo.c

install:
cp foo /home/me
```

We can tell make which target we want to make by typing:

```
% make target
```

`make` will then only look at that target and ignore any others. For example, if we type **`make foo`** with the makefile above, make will ignore the `install` target.

If we just type **`make`** on its own, make will always look at the first target and then stop without looking at any others. So if we typed **`make`** here, it will just go to the `foo` target, re-compile `foo` if necessary, and then stop without going on to the `install` target.

Notice that the `install` target does not actually depend on anything! This means that the command on the following line is always executed when we try to make that target by typing **`make install`**. In this case, it will copy `foo` into the user's home directory. This is often used by application makefiles, so that the application can be installed in the correct directory when it has been correctly compiled.

This is a slightly confusing subject to try to explain. If you do not quite understand how `make` works, the best thing to do is to write a simple program like "hello world" and a make file like the one above and experiment. Then progress to using more than one source file, or having the source file include a header file. The `touch` command is very useful here——it changes the date on a file without you having to edit it.

2.5.3 Make and include-files

C code often starts with a list of files to include, for example stdio.h. Some of these files are system-include files, some of them are from the project you are now working on:

```
#include <stdio.h>
#include "foo.h"

int main(....
```

To make sure that this file is recompiled the moment `foo.h` is changed, you have to add it in your `Makefile`:

```
foo: foo.c foo.h
```

The moment your project is getting bigger and you have more and more own include-files to maintain, it will be a pain to keep track of all include files and the files which are depending on it. If you change an include-file but forget to recompile all the files which are depending on it, the results will be devastating. `gcc` has an option to analyze your files and to produce a list of include-files and their dependencies: `-MM`.

If you add this to your Makefile:

```
depend:
gcc -E -MM *.c > .depend
```

and run **`make depend`**, the file `.depend` will appear with a list of object-files, C-files and the include-files:

```
foo.o: foo.c foo.h
```

If you change `foo.h`, next time you run `make` all files depending on `foo.h` will be recompiled.

Do not forget to run `make depend` each time you add an include-file to one of your files.

2.5.4 FreeBSD Makefiles

Makefiles can be rather complicated to write. Fortunately, BSD-based systems like FreeBSD come with some very powerful ones as part of the system. One very good example of this is the FreeBSD ports system. Here is the essential part of a typical ports `Makefile`:

```
MASTER_SITES=   ftp://freefall.cdrom.com/pub/FreeBSD/LOCAL_PORTS/
DISTFILES=      scheme-microcode+dist-7.3-freebsd.tgz

.include <bsd.port.mk>
```

Now, if we go to the directory for this port and type **`make`**, the following happens:

1. A check is made to see if the source code for this port is already on the system.
2. If it is not, an FTP connection to the URL in MASTER_SITES is set up to download the source.
3. The checksum for the source is calculated and compared it with one for a known, good, copy of the source. This is to make sure that the source was not corrupted while in transit.
4. Any changes required to make the source work on FreeBSD are applied——this is known as *patching*.
5. Any special configuration needed for the source is done. (Many UNIX program distributions try to work out which version of UNIX they are being compiled on and which optional UNIX features are present——this is where they are given the information in the FreeBSD ports scenario).
6. The source code for the program is compiled. In effect, we change to the directory where the source was unpacked and do `make`——the program's own make file has the necessary information to build the program.
7. We now have a compiled version of the program. If we wish, we can test it now; when we feel confident about the program, we can type **`make install`**. This will cause the program and any supporting files it needs to be copied into the correct location; an entry is also made into a package database, so that the port can easily be uninstalled later if we change our mind about it.

Now I think you will agree that is rather impressive for a four line script!

The secret lies in the last line, which tells `make` to look in the system makefile called `bsd.port.mk`. It is easy to overlook this line, but this is where all the clever stuff comes from——someone has written a makefile that tells `make` to do all the things above (plus a couple of other things I did not mention, including handling any errors that may occur) and anyone can get access to that just by putting a single line in their own make file!

If you want to have a look at these system makefiles, they are in `/usr/share/mk`, but it is probably best to wait until you have had a bit of practice with makefiles, as they are very complicated (and if you do look at them, make sure you have a flask of strong coffee handy!)

2.5.5 More advanced uses of `make`

`Make` is a very powerful tool, and can do much more than the simple example above shows. Unfortunately, there are several different versions of `make`, and they all differ considerably. The best way to learn what they can do is probably to read the documentation——hopefully this introduction will have given you a base from which you can do this.

The version of make that comes with FreeBSD is the **Berkeley make**; there is a tutorial for it in `/usr/share/doc/psd/12.make`. To view it, do

```
% zmore paper.ascii.gz
```

in that directory.

Many applications in the ports use **GNU make**, which has a very good set of "info" pages. If you have installed any of these ports, **GNU make** will automatically have been installed as `gmake`. It is also available as a port and package in its own right.

To view the info pages for **GNU make**, you will have to edit the `dir` file in the `/usr/local/info` directory to add an entry for it. This involves adding a line like

```
* Make: (make).                The GNU Make utility.
```

to the file. Once you have done this, you can type **`info`** and then select make from the menu (or in **Emacs**, do **`C-h i`**).

2.6 Debugging

2.6.1 The Debugger

The debugger that comes with FreeBSD is called `gdb` (**GNU debugger**). You start it up by typing

```
% gdb progname
```

although most people prefer to run it inside **Emacs**. You can do this by:

```
M-x gdb RET progname RET
```

Using a debugger allows you to run the program under more controlled circumstances. Typically, you can step through the program a line at a time, inspect the value of variables, change them, tell the debugger to run up to a certain point and then stop, and so on. You can even attach to a program that is already running, or load a core file to investigate why the program crashed. It is even possible to debug the kernel, though that is a little trickier than the user applications we will be discussing in this section.

`gdb` has quite good on-line help, as well as a set of info pages, so this section will concentrate on a few of the basic commands.

Finally, if you find its text-based command-prompt style off-putting, there is a graphical front-end for it (xxgdb (http://www.FreeBSD.org/ports/devel.html)) in the ports collection.

This section is intended to be an introduction to using `gdb` and does not cover specialized topics such as debugging the kernel.

2.6.2 Running a program in the debugger

You will need to have compiled the program with the `-g` option to get the most out of using `gdb`. It will work without, but you will only see the name of the function you are in, instead of the source code. If you see a line like:

```
... (no debugging symbols found) ...
```

when `gdb` starts up, you will know that the program was not compiled with the `-g` option.

At the `gdb` prompt, type **`break main`**. This will tell the debugger to skip over the preliminary set-up code in the program and start at the beginning of your code. Now type **`run`** to start the program——it will start at the beginning of the set-up code and then get stopped by the debugger when it calls `main()`. (If you have ever wondered where `main()` gets called from, now you know!).

You can now step through the program, a line at a time, by pressing `n`. If you get to a function call, you can step into it by pressing `s`. Once you are in a function call, you can return from stepping into a function call by pressing `f`. You can also use `up` and `down` to take a quick look at the caller.

Here is a simple example of how to spot a mistake in a program with `gdb`. This is our program (with a deliberate mistake):

```
#include <stdio.h>

int bazz(int anint);

main() {
int i;

printf("This is my program\n");
bazz(i);
return 0;
}

int bazz(int anint) {
printf("You gave me %d\n", anint);
return anint;
}
```

This program sets i to be `5` and passes it to a function `bazz()` which prints out the number we gave it.

When we compile and run the program we get

```
% cc -g -o temp temp.c
% ./temp
This is my program
anint = 4231
```

That was not what we expected! Time to see what is going on!

```
% gdb temp
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i386-unknown-freebsd), Copyright 1994 Free Software Foundation, Inc.
(gdb) break main Skip the set-up code
```

```
Breakpoint 1 at 0x160f: file temp.c, line 9. gdb puts breakpoint at main()
(gdb) run Run as far as main()
Starting program: /home/james/tmp/temp Program starts running

Breakpoint 1, main () at temp.c:9 gdb stops at main()
(gdb) n Go to next line
This is my program Program prints out
(gdb) s step into bazz()
bazz (anint=4231) at temp.c:17 gdb displays stack frame
(gdb)
```

Hang on a minute! How did anint get to be `4231`? Did we not we set it to be `5` in `main()`? Let's move up to `main()` and have a look.

```
(gdb) up Move up call stack
#1  0x1625 in main () at temp.c:11 gdb displays stack frame
(gdb) p i Show us the value of i
$1 = 4231 gdb displays 4231
```

Oh dear! Looking at the code, we forgot to initialize i. We meant to put

```
...
main() {
int i;

i = 5;
printf("This is my program\n");
...
```

but we left the `i=5;` line out. As we did not initialize i, it had whatever number happened to be in that area of memory when the program ran, which in this case happened to be `4231`.

> **注意:** `gdb` displays the stack frame every time we go into or out of a function, even if we are using `up` and `down` to move around the call stack. This shows the name of the function and the values of its arguments, which helps us keep track of where we are and what is going on. (The stack is a storage area where the program stores information about the arguments passed to functions and where to go when it returns from a function call).

2.6.3 Examining a core file

A core file is basically a file which contains the complete state of the process when it crashed. In "the good old days", programmers had to print out hex listings of core files and sweat over machine code manuals, but now life is a bit easier. Incidentally, under FreeBSD and other 4.4BSD systems, a core file is called *`progname`*`.core` instead of just `core`, to make it clearer which program a core file belongs to.

To examine a core file, start up `gdb` in the usual way. Instead of typing `break` or `run`, type

```
(gdb) core progname.core
```

If you are not in the same directory as the core file, you will have to do **`dir /path/to/core/file`** first.

You should see something like this:

```
% gdb a.out
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i386-unknown-freebsd), Copyright 1994 Free Software Foundation, Inc.
(gdb) core a.out.core
Core was generated by `a.out'.
Program terminated with signal 11, Segmentation fault.
Cannot access memory at address 0x7020796d.
#0  0x164a in bazz (anint=0x5) at temp.c:17
(gdb)
```

In this case, the program was called `a.out`, so the core file is called `a.out.core`. We can see that the program crashed due to trying to access an area in memory that was not available to it in a function called `bazz`.

Sometimes it is useful to be able to see how a function was called, as the problem could have occurred a long way up the call stack in a complex program. The `bt` command causes `gdb` to print out a back-trace of the call stack:

```
(gdb) bt
#0  0x164a in bazz (anint=0x5) at temp.c:17
#1  0xefbfd888 in end ()
#2  0x162c in main () at temp.c:11
(gdb)
```

The `end()` function is called when a program crashes; in this case, the `bazz()` function was called from `main()`.

2.6.4 Attaching to a running program

One of the neatest features about `gdb` is that it can attach to a program that is already running. Of course, that assumes you have sufficient permissions to do so. A common problem is when you are stepping through a program that forks, and you want to trace the child, but the debugger will only let you trace the parent.

What you do is start up another `gdb`, use `ps` to find the process ID for the child, and do

```
(gdb) attach pid
```

in `gdb`, and then debug as usual.

"That is all very well," you are probably thinking, "but by the time I have done that, the child process will be over the hill and far away". Fear not, gentle reader, here is how to do it (courtesy of the `gdb` info pages):

```
...
if ((pid = fork()) < 0) /* _Always_ check this */
error();
else if (pid == 0) { /* child */
int PauseMode = 1;

while (PauseMode)
sleep(10); /* Wait until someone attaches to us */
...
} else { /* parent */
...
```

Now all you have to do is attach to the child, set PauseMode to `0`, and wait for the `sleep()` call to return!

2.7 Using Emacs as a Development Environment

2.7.1 Emacs

Unfortunately, UNIX systems do not come with the kind of everything-you-ever-wanted-and-lots-more-you-did-not-in-one-gigantic-package integrated development environments that other systems have. [7] However, it is possible to set up your own environment. It may not be as pretty, and it may not be quite as integrated, but you can set it up the way you want it. And it is free. And you have the source to it.

The key to it all is Emacs. Now there are some people who loathe it, but many who love it. If you are one of the former, I am afraid this section will hold little of interest to you. Also, you will need a fair amount of memory to run it——I would recommend 8MB in text mode and 16MB in X as the bare minimum to get reasonable performance.

Emacs is basically a highly customizable editor——indeed, it has been customized to the point where it is more like an operating system than an editor! Many developers and sysadmins do in fact spend practically all their time working inside Emacs, leaving it only to log out.

It is impossible even to summarize everything Emacs can do here, but here are some of the features of interest to developers:

Very powerful editor, allowing search-and-replace on both strings and regular expressions (patterns), jumping to start/end of block expression, etc, etc.

Pull-down menus and online help.

Language-dependent syntax highlighting and indentation.

Completely customizable.

You can compile and debug programs within Emacs.

On a compilation error, you can jump to the offending line of source code.

Friendly-ish front-end to the `info` program used for reading GNU hypertext documentation, including the documentation on Emacs itself.

Friendly front-end to `gdb`, allowing you to look at the source code as you step through your program.

You can read Usenet news and mail while your program is compiling.

And doubtless many more that I have overlooked.

Emacs can be installed on FreeBSD using the Emacs port (http://www.FreeBSD.org/ports/editors.html).

Once it is installed, start it up and do **`C-h t`** to read an Emacs tutorial——that means hold down the **control** key, press **h**, let go of the **control** key, and then press **t**. (Alternatively, you can you use the mouse to select Emacs Tutorial from the Help menu).

Although Emacs does have menus, it is well worth learning the key bindings, as it is much quicker when you are editing something to press a couple of keys than to try to find the mouse and then click on the right place. And, when you are talking to seasoned Emacs users, you will find they often casually throw around expressions like

"`M-x replace-s RET foo RET bar RET`" so it is useful to know what they mean. And in any case, Emacs has far too many useful functions for them to all fit on the menu bars.

Fortunately, it is quite easy to pick up the key-bindings, as they are displayed next to the menu item. My advice is to use the menu item for, say, opening a file until you understand how it works and feel confident with it, then try doing C-x C-f. When you are happy with that, move on to another menu command.

If you can not remember what a particular combination of keys does, select Describe Key from the Help menu and type it in——Emacs will tell you what it does. You can also use the Command Apropos menu item to find out all the commands which contain a particular word in them, with the key binding next to it.

By the way, the expression above means hold down the Meta key, press x, release the Meta key, type **`replace-s`** (short for `replace-string`——another feature of Emacs is that you can abbreviate commands), press the return key, type **`foo`** (the string you want replaced), press the return key, type bar (the string you want to replace `foo` with) and press return again. Emacs will then do the search-and-replace operation you have just requested.

If you are wondering what on earth the Meta key is, it is a special key that many UNIX workstations have. Unfortunately, PC's do not have one, so it is usually the **alt** key (or if you are unlucky, the escape key).

Oh, and to get out of Emacs, do `C-x C-c` (that means hold down the control key, press x, press c and release the control key). If you have any unsaved files open, Emacs will ask you if you want to save them. (Ignore the bit in the documentation where it says `C-z` is the usual way to leave Emacs——that leaves Emacs hanging around in the background, and is only really useful if you are on a system which does not have virtual terminals).

2.7.2 Configuring Emacs

Emacs does many wonderful things; some of them are built in, some of them need to be configured.

Instead of using a proprietary macro language for configuration, Emacs uses a version of Lisp specially adapted for editors, known as Emacs Lisp. Working with Emacs Lisp can be quite helpful if you want to go on and learn something like Common Lisp. Emacs Lisp has many features of Common Lisp, although it is considerably smaller (and thus easier to master).

The best way to learn Emacs Lisp is to download the Emacs Tutorial (ftp://ftp.gnu.org/old-gnu/emacs/elisp-manual-19-2.4.tar.gz)

However, there is no need to actually know any Lisp to get started with configuring Emacs, as I have included a sample `.emacs` file, which should be enough to get you started. Just copy it into your home directory and restart Emacs if it is already running; it will read the commands from the file and (hopefully) give you a useful basic setup.

2.7.3 A sample `.emacs` file

Unfortunately, there is far too much here to explain it in detail; however there are one or two points worth mentioning.

Everything beginning with a `;` is a comment and is ignored by Emacs.

In the first line, the `-*- Emacs-Lisp -*-` is so that we can edit the `.emacs` file itself within Emacs and get all the fancy features for editing Emacs Lisp. Emacs usually tries to guess this based on the filename, and may not get it right for `.emacs`.

The tab key is bound to an indentation function in some modes, so when you press the tab key, it will indent the current line of code. If you want to put a tab character in whatever you are writing, hold the control key down while you are pressing the tab key.

This file supports syntax highlighting for C, C++, Perl, Lisp and Scheme, by guessing the language from the filename.

Emacs already has a pre-defined function called `next-error`. In a compilation output window, this allows you to move from one compilation error to the next by doing `M-n`; we define a complementary function, `previous-error`, that allows you to go to a previous error by doing `M-p`. The nicest feature of all is that `C-c C-c` will open up the source file in which the error occurred and jump to the appropriate line.

We enable Emacs's ability to act as a server, so that if you are doing something outside Emacs and you want to edit a file, you can just type in

```
% emacsclient filename
```

and then you can edit the file in your Emacs! [8]

例2-1. A sample `.emacs` file

```
;; -*-Emacs-Lisp-*-

;; This file is designed to be re-evaled; use the variable first-time
;; to avoid any problems with this.
(defvar first-time t
  "Flag signifying this is the first time that .emacs has been evaled")

;; Meta
(global-set-key "\M- " 'set-mark-command)
(global-set-key "\M-\C-h" 'backward-kill-word)
(global-set-key "\M-\C-r" 'query-replace)
(global-set-key "\M-r" 'replace-string)
(global-set-key "\M-g" 'goto-line)
(global-set-key "\M-h" 'help-command)

;; Function keys
(global-set-key [f1] 'manual-entry)
(global-set-key [f2] 'info)
(global-set-key [f3] 'repeat-complex-command)
(global-set-key [f4] 'advertised-undo)
(global-set-key [f5] 'eval-current-buffer)
(global-set-key [f6] 'buffer-menu)
(global-set-key [f7] 'other-window)
(global-set-key [f8] 'find-file)
(global-set-key [f9] 'save-buffer)
(global-set-key [f10] 'next-error)
(global-set-key [f11] 'compile)
(global-set-key [f12] 'grep)
(global-set-key [C-f1] 'compile)
(global-set-key [C-f2] 'grep)
(global-set-key [C-f3] 'next-error)
(global-set-key [C-f4] 'previous-error)
```

```
(global-set-key [C-f5] 'display-faces)
(global-set-key [C-f8] 'dired)
(global-set-key [C-f10] 'kill-compilation)

;; Keypad bindings
(global-set-key [up] "\C-p")
(global-set-key [down] "\C-n")
(global-set-key [left] "\C-b")
(global-set-key [right] "\C-f")
(global-set-key [home] "\C-a")
(global-set-key [end] "\C-e")
(global-set-key [prior] "\M-v")
(global-set-key [next] "\C-v")
(global-set-key [C-up] "\M-\C-b")
(global-set-key [C-down] "\M-\C-f")
(global-set-key [C-left] "\M-b")
(global-set-key [C-right] "\M-f")
(global-set-key [C-home] "\M-<")
(global-set-key [C-end] "\M->")
(global-set-key [C-prior] "\M-<")
(global-set-key [C-next] "\M->")

;; Mouse
(global-set-key [mouse-3] 'imenu)

;; Misc
(global-set-key [C-tab] "\C-q\t") ; Control tab quotes a tab.
(setq backup-by-copying-when-mismatch t)

;; Treat 'y' or <CR> as yes, 'n' as no.
(fset 'yes-or-no-p 'y-or-n-p)
(define-key query-replace-map [return] 'act)
(define-key query-replace-map [?\C-m] 'act)

;; Load packages
(require 'desktop)
(require 'tar-mode)

;; Pretty diff mode
(autoload 'ediff-buffers "ediff" "Intelligent Emacs interface to diff" t)
(autoload 'ediff-files "ediff" "Intelligent Emacs interface to diff" t)
(autoload 'ediff-files-remote "ediff"
  "Intelligent Emacs interface to diff")

(if first-time
    (setq auto-mode-alist
  (append '(("\\.cpp$" . c++-mode)
    ("\\.hpp$" . c++-mode)
    ("\\.lsp$" . lisp-mode)
    ("\\.scm$" . scheme-mode)
    ("\\.pl$" . perl-mode)
    ) auto-mode-alist)))
```

```
;; Auto font lock mode
(defvar font-lock-auto-mode-list
  (list 'c-mode 'c++-mode 'c++-c-mode 'emacs-lisp-mode 'lisp-mode 'perl-mode 'scheme-mode)
  "List of modes to always start in font-lock-mode")

(defvar font-lock-mode-keyword-alist
  '((c++-c-mode . c-font-lock-keywords)
    (perl-mode . perl-font-lock-keywords))
  "Associations between modes and keywords")

(defun font-lock-auto-mode-select ()
  "Automatically select font-lock-mode if the current major mode is in font-lock-auto-mode-list"
  (if (memq major-mode font-lock-auto-mode-list)
      (progn
(font-lock-mode t))
    )
  )

(global-set-key [M-f1] 'font-lock-fontify-buffer)

;; New dabbrev stuff
;(require 'new-dabbrev)
(setq dabbrev-always-check-other-buffers t)
(setq dabbrev-abbrev-char-regexp "\\sw\\|\\s_")
(add-hook 'emacs-lisp-mode-hook
  '(lambda ()
     (set (make-local-variable 'dabbrev-case-fold-search) nil)
     (set (make-local-variable 'dabbrev-case-replace) nil)))
(add-hook 'c-mode-hook
  '(lambda ()
     (set (make-local-variable 'dabbrev-case-fold-search) nil)
     (set (make-local-variable 'dabbrev-case-replace) nil)))
(add-hook 'text-mode-hook
  '(lambda ()
     (set (make-local-variable 'dabbrev-case-fold-search) t)
     (set (make-local-variable 'dabbrev-case-replace) t)))

;; C++ and C mode...
(defun my-c++-mode-hook ()
  (setq tab-width 4)
  (define-key c++-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (define-key c++-mode-map "\C-ce" 'c-comment-edit)
  (setq c++-auto-hungry-initial-state 'none)
  (setq c++-delete-function 'backward-delete-char)
  (setq c++-tab-always-indent t)
  (setq c-indent-level 4)
  (setq c-continued-statement-offset 4)
  (setq c++-empty-arglist-indent 4))

(defun my-c-mode-hook ()
  (setq tab-width 4)
  (define-key c-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (define-key c-mode-map "\C-ce" 'c-comment-edit)
```

```
  (setq c-auto-hungry-initial-state 'none)
  (setq c-delete-function 'backward-delete-char)
  (setq c-tab-always-indent t)
;; BSD-ish indentation style
  (setq c-indent-level 4)
  (setq c-continued-statement-offset 4)
  (setq c-brace-offset -4)
  (setq c-argdecl-indent 0)
  (setq c-label-offset -4))

;; Perl mode
(defun my-perl-mode-hook ()
  (setq tab-width 4)
  (define-key c++-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (setq perl-indent-level 4)
  (setq perl-continued-statement-offset 4))

;; Scheme mode...
(defun my-scheme-mode-hook ()
  (define-key scheme-mode-map "\C-m" 'reindent-then-newline-and-indent))

;; Emacs-Lisp mode...
(defun my-lisp-mode-hook ()
  (define-key lisp-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (define-key lisp-mode-map "\C-i" 'lisp-indent-line)
  (define-key lisp-mode-map "\C-j" 'eval-print-last-sexp))

;; Add all of the hooks...
(add-hook 'c++-mode-hook 'my-c++-mode-hook)
(add-hook 'c-mode-hook 'my-c-mode-hook)
(add-hook 'scheme-mode-hook 'my-scheme-mode-hook)
(add-hook 'emacs-lisp-mode-hook 'my-lisp-mode-hook)
(add-hook 'lisp-mode-hook 'my-lisp-mode-hook)
(add-hook 'perl-mode-hook 'my-perl-mode-hook)

;; Complement to next-error
(defun previous-error (n)
  "Visit previous compilation error message and corresponding source code."
  (interactive "p")
  (next-error (- n)))

;; Misc...
(transient-mark-mode 1)
(setq mark-even-if-inactive t)
(setq visible-bell nil)
(setq next-line-add-newlines nil)
(setq compile-command "make")
(setq suggest-key-bindings nil)
(put 'eval-expression 'disabled nil)
(put 'narrow-to-region 'disabled nil)
(put 'set-goal-column 'disabled nil)
(if (>= emacs-major-version 21)
(setq show-trailing-whitespace t))
```

```
;; Elisp archive searching
(autoload 'format-lisp-code-directory "lispdir" nil t)
(autoload 'lisp-dir-apropos "lispdir" nil t)
(autoload 'lisp-dir-retrieve "lispdir" nil t)
(autoload 'lisp-dir-verify "lispdir" nil t)

;; Font lock mode
(defun my-make-face (face color &optional bold)
  "Create a face from a color and optionally make it bold"
  (make-face face)
  (copy-face 'default face)
  (set-face-foreground face color)
  (if bold (make-face-bold face))
  )

(if (eq window-system 'x)
    (progn
      (my-make-face 'blue "blue")
      (my-make-face 'red "red")
      (my-make-face 'green "dark green")
      (setq font-lock-comment-face 'blue)
      (setq font-lock-string-face 'bold)
      (setq font-lock-type-face 'bold)
      (setq font-lock-keyword-face 'bold)
      (setq font-lock-function-name-face 'red)
      (setq font-lock-doc-string-face 'green)
      (add-hook 'find-file-hooks 'font-lock-auto-mode-select)

      (setq baud-rate 1000000)
      (global-set-key "\C-cmm" 'menu-bar-mode)
      (global-set-key "\C-cms" 'scroll-bar-mode)
      (global-set-key [backspace] 'backward-delete-char)
;      (global-set-key [delete] 'delete-char)
      (standard-display-european t)
      (load-library "iso-transl")))

;; X11 or PC using direct screen writes
(if window-system
    (progn
      ;;      (global-set-key [M-f1] 'hilit-repaint-command)
      ;;      (global-set-key [M-f2] [?\C-u M-f1])
      (setq hilit-mode-enable-list
    '(not text-mode c-mode c++-mode emacs-lisp-mode lisp-mode
  scheme-mode)
    hilit-auto-highlight nil
    hilit-auto-rehighlight 'visible
    hilit-inhibit-hooks nil
    hilit-inhibit-rebinding t)
      (require 'hilit19)
      (require 'paren))
  (setq baud-rate 2400) ; For slow serial connections
  )
```

```
;; TTY type terminal
(if (and (not window-system)
 (not (equal system-type 'ms-dos)))
    (progn
      (if first-time
  (progn
    (keyboard-translate ?\C-h ?\C-?)
    (keyboard-translate ?\C-? ?\C-h)))))

;; Under UNIX
(if (not (equal system-type 'ms-dos))
    (progn
      (if first-time
  (server-start))))

;; Add any face changes here
(add-hook 'term-setup-hook 'my-term-setup-hook)
(defun my-term-setup-hook ()
  (if (eq window-system 'pc)
      (progn
;; (set-face-background 'default "red")
)))

;; Restore the "desktop" - do this as late as possible
(if first-time
    (progn
      (desktop-load-default)
      (desktop-read)))

;; Indicate that this file has been read at least once
(setq first-time nil)

;; No need to debug anything now

(setq debug-on-error nil)

;; All done
(message "All done, %s%s" (user-login-name) ".")
```

2.7.4 Extending the Range of Languages Emacs Understands

Now, this is all very well if you only want to program in the languages already catered for in the `.emacs` file (C, C++, Perl, Lisp and Scheme), but what happens if a new language called "whizbang" comes out, full of exciting features?

The first thing to do is find out if whizbang comes with any files that tell Emacs about the language. These usually end in `.el`, short for "Emacs Lisp". For example, if whizbang is a FreeBSD port, we can locate these files by doing

```
% find /usr/ports/lang/whizbang -name "*.el" -print
```

and install them by copying them into the Emacs site Lisp directory. On FreeBSD 2.1.0-RELEASE, this is `/usr/local/share/emacs/site-lisp`.

So for example, if the output from the find command was

```
/usr/ports/lang/whizbang/work/misc/whizbang.el
```

we would do

```
# cp /usr/ports/lang/whizbang/work/misc/whizbang.el /usr/local/share/emacs/site-lisp
```

Next, we need to decide what extension whizbang source files have. Let's say for the sake of argument that they all end in `.wiz`. We need to add an entry to our `.emacs` file to make sure Emacs will be able to use the information in `whizbang.el`.

Find the auto-mode-alist entry in `.emacs` and add a line for whizbang, such as:

```
...
("\\.lsp$" . lisp-mode)
("\\.wiz$" . whizbang-mode)
("\\.scm$" . scheme-mode)
...
```

This means that Emacs will automatically go into `whizbang-mode` when you edit a file ending in `.wiz`.

Just below this, you will find the font-lock-auto-mode-list entry. Add `whizbang-mode` to it like so:

```
;; Auto font lock mode
(defvar font-lock-auto-mode-list
  (list 'c-mode 'c++-mode 'c++-c-mode 'emacs-lisp-mode 'whizbang-mode 'lisp-mode 'perl-mode 'scheme-
  "List of modes to always start in font-lock-mode")
```

This means that Emacs will always enable `font-lock-mode` (ie syntax highlighting) when editing a `.wiz` file.

And that is all that is needed. If there is anything else you want done automatically when you open up a `.wiz` file, you can add a `whizbang-mode hook` (see `my-scheme-mode-hook` for a simple example that adds `auto-indent`).

2.8 Further Reading

For information about setting up a development environment for contributing fixes to FreeBSD itself, please see development(7).

Brian Harvey and Matthew Wright *Simply Scheme* MIT 1994. ISBN 0-262-08226-8

Randall Schwartz *Learning Perl* O'Reilly 1993 ISBN 1-56592-042-2

Patrick Henry Winston and Berthold Klaus Paul Horn *Lisp (3rd Edition)* Addison-Wesley 1989 ISBN 0-201-08319-1

Brian W. Kernighan and Rob Pike *The Unix Programming Environment* Prentice-Hall 1984 ISBN 0-13-937681-X

Brian W. Kernighan and Dennis M. Ritchie *The C Programming Language (2nd Edition)* Prentice-Hall 1988 ISBN 0-13-110362-8

Bjarne Stroustrup *The C++ Programming Language* Addison-Wesley 1991 ISBN 0-201-53992-6

W. Richard Stevens *Advanced Programming in the Unix Environment* Addison-Wesley 1992 ISBN 0-201-56317-7

W. Richard Stevens *Unix Network Programming* Prentice-Hall 1990 ISBN 0-13-949876-1

备注

1. If you run it in the shell, you may get a core dump.
2. To be strictly accurate, `cc` converts the source code into its own, machine-independent *p-code* instead of assembly language at this stage.
3. In case you did not know, a binary sort is an efficient way of sorting things into order and a bubble sort is not.
4. The reasons for this are buried in the mists of history.
5. Note, we did not use the `-o` flag to specify the executable name, so we will get an executable called `a.out`. Producing a debug version called `foobar` is left as an exercise for the reader!
6. They do not use the `MAKEFILE` form as block capitals are often used for documentation files like `README`.
7. Some powerful, free IDEs now exist, such as KDevelop in the ports collection.
8. Many Emacs users set their `EDITOR` environment to `emacsclient` so this happens every time they need to edit a file.

第3章
安全的编程

供稿：*Murray Stokely*. 翻译：*susn @NewSMTH.*

3.1 提要

本章描述了十年间一些令UNIX程序员感到困惑的安全问题，并提供了一些新的工具来帮助程序员避免生成可被利用的代码。

3.2 安全的设计方法

编写安全的应用程序要带着谨慎和略有悲观的生活观点。程序应该本着“最小特权”的原则运行，这样就不会有带着大于足够能完成其功能的权限的进程在运行。预先测试的代码应该随时可以重用以避免遇到一些本已经修复的通常错误。

UNIX环境的陷阱之一就是很容易的制造一个稳健环境的假象。程序应该永远不要相信用户的输入(以各种形式)，系统资源，进程间通讯，或者触发事件的时钟。UNIX进程不是同步运行，所以逻辑操作很少是原子类型。

3.3 缓冲区溢出

缓冲区溢出的漏洞随着冯诺依曼1 构架的出现就已经开始出现了。 在1988年随着莫里斯互联网蠕虫的广泛传播他们开始声名狼藉。不幸的是，同样的这种攻击一直持续到今天。 1999年的17个CERT(Computer Emergency Response Team, 卡内基梅隆大学计算机紧急响应小组)的安全通告中， 他们中的10个直接是由软件的缓冲区溢出而导致的。到目前为止，大部分的缓冲区溢出的攻击都是基于摧毁栈的方式。

大部分现代计算机系统使用栈来给进程传递参数并且存储局部变量。栈是一种在进程映象内存的高地址内的后进先出(LIFO)的缓冲区。 当程序调用一个函数时一个新的“栈帧”会被创建。这个栈帧包含着传递给函数的各种参数和一些动态的局部变量空间。 “栈指针”记录着当前栈顶的位置。 由于栈指针的值会因为新变量的压入栈顶而经常的变化，许多实现也提供了一种"帧指针"来定位在栈帧的起始位置，以便局部变量可以更容易的被访问。1调用函数的返回地址也同样存储在栈中， 由于在函数中的局部变量覆盖了函数的返回地址成为了栈溢出的一个原因，这就潜在的准许了一个恶意用户可以执行他(她)所想运行的任何代码。

虽然基于栈的攻击是目前最广泛的，这也可以使基于堆的攻击(malloc/ free)变成可能。

C程序语言并不像其他一些编程语言一样自动的做数组或者指针的边界检查。另外，C标准库还具有相当一些非常危险的操作函数。

`strcpy`(char *dest, const char *src)	可导致dest缓冲区溢出
`strcat`(char *dest, const char *src)	可导致dest缓冲区溢出
`getwd`(char *buf)	可导致buf缓冲区溢出
`gets`(char *s)	可导致s缓冲区溢出
`[vf]scanf`(const char *format, ...)	可导致参数溢出

realpath(char *path, char resolved_path[])	可导致path缓冲区溢出
[v]sprintf(char *str, const char *format, ...)	可导致str缓冲区溢出

3.3.1 缓冲区溢出示例

下面的示例代码包含了一个缓冲区溢出的情况，它会覆盖函数的返回地址并且立即跳过了紧随此函数之后调用。(授权于5)

```
#include <stdio.h>

void manipulate(char *buffer) {
  char newbuffer[80];
  strcpy(newbuffer,buffer);
}

int main() {
  char ch,buffer[4096];
  int i=0;

  while ((buffer[i++] = getchar()) != '\n') {};

  i=1;
  manipulate(buffer);
  i=2;
  printf("The value of i is : %d\n",i);
  return 0;
}
```

让我们来查看一下如果在输入回车之前输入160个空格后这个小程序的内存映象是个什么样子。

[XXX figure here!]

很明显更多的恶意输入能被设计出执行实际的编译指令(例如exec(/bin/sh))。

3.3.2 避免缓冲区溢出

对于栈溢出的最直接的解决方法就是总是使用长度有限的内存和字符串复制函数。strncpy和strncat是C标准库的一部分。 这些函数接收一个不大于目标缓冲区长度的值作为参数。这些函数会从源地址复制此值长的字节数到目标地址。然而这些函数还是有一些问题。如果输入缓冲区的长度和目标缓冲区的一样长则函数不保证两者都以NUL 作为结束符。 长度参数在strncpy和strncat函数中同样的不一致很容易导致程序员在正常使用时感到困惑。同时当复制一个较短的字符串到一个很大的缓冲区中时相对于strcpy也有很重大的性能损失，因为strncpy会用NUL填充所指定的长度。

在OpenBSD中，另一个内存复制的实现已经规避了这些问题。 函数strlcpy和strlcat 保证了当指定了非零的长度参数时目标字符串总是以NUL作为结束符。关于这些函数的更多信息请参考7。OpenBSD的strlcpy和strlcat 自从FreeBSD3.3的版本已经被引入了。

3.3.2.1 基于编译器运行时边界检查

不幸的是扔然有相当数量的代码在广泛使用盲目的内存复制功能而不是我们所提及到的任何有限制的复制例程。幸运的是还有另一个解决方案。有一些编译器插件和库在C/C++中一直在做运行时的边界检查。

作为gcc代码生成器的一个小补丁StackGuard就是这样一款插件。源自StackGuard 站点 (http://immunix.org/stackguard.html):

> "StackGuard 检测并靠着保护在栈中的返回地址不受到更改来防御针对栈的剧烈攻击。当函数被调用时StackGuard在栈中紧邻返回地址放置了一个‘canary’(哨兵或探针)。如果函数返回时哨兵已经被改变了，就是有针对栈的攻击实施了，那么程序会在syslog中发出一个入侵警报并且停止运行。

> "StackGuard作为gcc代码生成器的一个小补丁来实现，特别是function_prolog()和function_epilog()程序。增强的function_prolog()在函数开始时在栈中安装了哨兵，而function_epilog()在函数退出时检查哨兵的完整性。任何其他破坏返回地址的行为在函数返回时就这样被检测到了。

使用StackGuard重新编译你的程序可以有效的防止大部分的缓冲区溢出的攻击，但是这仍然是个折衷的办法。

3.3.2.2 基于库运行时边界检查

基于编译器的机制对于不能重新编译的只有二进制的软件完全无用。对于这些情况仍还是有很多库可以对C库中的不安全的函数(`strcpy`, `fscanf`, `getwd`等)重新实现并确保这些函数决不回写栈指针。

libsafe

libverify

libparanoia

不幸的是这些基于库的防护有一些缺点。这些库仅仅保护和安全相关的一小部分集合，他们忽略了实际的问题。如果程序使用参数-fomit-frame-pointer进行编译的话这些防护也许会失败。同样，环境变量LD_PRELOAD和LD_LIBRARY_PATH也可以被用户取消或者重置。

3.4 SetUID 问题

对于给定的进程至少有6个不同的ID与之关联。因此你不得不非常关注你的程序在任何特定时刻的权限问题。特别的，所有seteuid的程序在不需要的时候会立刻放弃他们的特权。

实际用户ID只能被超级用户进程改变。当用户初始登陆时 **login**程序设置它并且极少进行更改。

如果程序准许seteuid位设置的话有效用户ID会被`exec()` 函数设置。应用程序可以调用`seteuid()` 在任何时候设置有效的用户ID为任意的实际用户ID或者保存设置-用户-ID。当有效用户ID被`exec()`函数设置后，前一个ID的值会被保存在设置-用户-ID中。

3.5 限制你的程序环境

传统的限制进程的方法是使用系统调用`chroot()` 。这个系统调用使得从进程及其任何子进程所引用的其他的路径变为根路径。对于要使程序运行成功这个调用必须在引用的目录上拥有执行(搜索)的权限。直到你使用了`chdir()` 在你的新环境中它才会实际的生效。同时应该注意到如果程序具有超级用户的权限它

很容易的摆脱chroot所设置的环境。它可能靠创建设备节点来读取内核的内容，对程序在jail外绑定一个调试器，或者靠其他创造性的方法来完成操作。

系统调用chroot()的行为可以被sysctl变量kern.chroot_allow_open_directories 的值在一定程度上控制。当此值为0时，如果有任何目录被打开 chroot()将会返回EPERM并失败。当被置为默认值1，如果任何目录被打开并且进程已经准备调用chroot() 那么chroot()将会返回EPERM并失败。对于其他的值，对打开目录的检查会被完全的忽视。

3.5.1 FreeBSD的jail功能

Jail的概念在chroot()之上作了延伸，它靠限制超级用户的权力来创建了一个真正的"虚拟服务器"。一旦一个监狱被设置好后整个网络必须通过特别的IP地址才能到达，在这里"超级用户权限"的力量完全的受到限制。

当在jail中时，所有在内核中使用suser() 调用的超级用户权限的尝试都会失败。然而，一些对suser() 的调用已经被更改为新的接口suser_xxx() 。这个函数对认可或者拒绝被限制的进程去取得超级用户的权限的行为负责。

一个在Jail环境中的超级用户进程有以下权力:

使用可信任的操作：setuid, seteuid, setgid, setegid, setgroups, setreuid, setregid, setlogin

使用setrlimit设置资源限制

编辑一些sysctl节点值(kern.hostname)

chroot()

在vnode(V-节点)上设置标志: chflags, fchflags

设置V-节点属性例如文件权限，所有者，所有组，大小，存取时间，更新时间

在互联网域中帮定有特权端口(端口号小于1024)

Jail是一个对于在一个安全环境中运行一个仍有一些缺点的程序非常有用的工具。目前，IPC机制还没有被更改到suser_xxx以至于像MySQL之类的程序还不能运行在jail中。在jail中超级用户的存取可能还有非常有限的含义，但是没有途径能正确的指出"非常有限"意味着什么。

3.5.2 POSIX®.1e 处理能力

POSIX已经发布了一个工作草案，增加了事件审计，访问控制列表，精细特权控制，信息标签和强制访问控制。

这是一个正在进展中的工作并且是 TrustedBSD (http://www.trustedbsd.org/)项目的重点。一些初始化的工作已经被提交到FreeBSD-CURRENT(cap_set_proc(3))。

3.6 信任

一个程序应该永远不要假设用户环境是健全的。这包括(但是决不限于此): 用户输入，信号，环境变量，资源，IPC，mmap(内存映射)，工作目录的文件系统，文件描述符，打开文件的数量，等等

你永远不要假设你可以捕捉到所有的用户可能产生的非法输入样式。换言之，你的程序应该过滤只准许一些你认为安全的特别的输入子集。不正确的确认数据会导致各种错误，特别是在互联网上的CGI脚本。对于文件名你应该额外小心比如路径("../", "/")，符号连接和shell的退出符。

Perl有一个非常棒的特性叫做“Taint”模式能避免脚本从外部程序在不安全的途径得到使用的数据。这个方式会检查命令行参数，环境变量，位置信息确定系统调用(`readdir()`,`readlink()` ,`getpwxxx()`)的结果和所有文件的输入。

3.7 竞态条件

竞态条件是由和事件时间相关的意料之外的依赖所导致的反常行为。换句话说，一个程序员不正确的假设一个特殊的事件总是在另一个事件之前发生。

一些通常的导致竞态条件的原因是信号，存取检查和打开文件操作。由于信号生来就是异步事件所以在处理他们时要特别当心。存取检查中使用`access(2)`然后使用`open(2)` 是很明显的非原子操作。用户可以在两次调用中移走文件。换言之，有特权的程序应该使用`seteuid()`然后直接调用`open()`。沿着同一思路，一个程序应该总是在`open()`之前设置正确的掩码来排除不合逻辑的`chmod()`调用。

第4章
本地化与国际化- *L10N* 和*I18N*

翻译：*susn @NewSMTH.*

4.1 编写适应国际化的应用程序

为了使你的程序对于使用其他语言的用户更加有用，我们希望你的程序应该国际化。GNU的gcc编译器和GUI库比如QT或者GTK等通过对字符串的特殊处理来支持国际化。生成一个支持国际化的程序非常容易。它使得发布者很快的把你的程序移植成其他语言。请参看详细的I18N文档来获得更多的信息。

对比与通常的理解，兼容国际化的代码非常容易编写。通常，它只需要使用一些特殊的功能函数来包装一下你的字符串。另外，请确认支持了宽字符或者多字节字符。

4.1.1 整合I18N成果的号召

值得我们注意的是各个国家自己的I18N/L10N 的工作已经变成了重复彼此的工作。我们中的许多人一直在做着重复的无效率的重复发明工作。我们希望各个主要I18N的组织能聚集成一个大的组织担负起像核心团队一样的责任。

当前，我们希望，当你在编写或者移植I18N程序时，能够发布给各个国家相关的FreeBSD邮件列表作为测试。将来，我们希望能生成可以直接使用各种语言的应用程序而不是还要做一些混乱的破解。

FreeBSD 国际化邮件列表 (http://lists.FreeBSD.org/mailman/listinfo/freebsd-i18n) 已经建立。如果你是一个I18N/L10N的开发者，请发送给我们任何你能想到的与之相关的说明，想法，问题等等。

Michael C. Wu 将会维护I18N的工作，主页在http://www.FreeBSD.org/~keichii/i18n/index.html。请同时阅读BSDCon2000 I18N的由Clive Lin，Chia-Liang Kao和Michael C.Wu
在http://www.FreeBSD.org/~keichii/papers/ 提出的文章。

4.1.2 Perl 和Python

Perl和Python拥有I18N和处理宽字符的库。请使用他们来兼容I18N。

在旧的FreeBSD版本中，Perl会给出一些没有在你的系统中安装了本地宽字符的警告。你可以在你的shell中设置环境变量`LD_PRELOAD`为`/usr/lib/libxpg4.so`。

在基于`sh`的shell中:

```
LD_PRELOAD=/usr/lib/libxpg4.so
```

在基于`C-sh`的shell中:

```
setenv LD_PRELOAD /usr/lib/libxpg4.so
```

第5章
Source Tree Guidelines and Policies

Contributed by Poul-Henning Kamp.

This chapter documents various guidelines and policies in force for the FreeBSD source tree.

5.1 MAINTAINER on Makefiles

If a particular portion of the FreeBSD distribution is being maintained by a person or group of persons, they can communicate this fact to the world by adding a

```
MAINTAINER= email-addresses
```

line to the Makefiles covering this portion of the source tree.

The semantics of this are as follows:

The maintainer owns and is responsible for that code. This means that he is responsible for fixing bugs and answering problem reports pertaining to that piece of the code, and in the case of contributed software, for tracking new versions, as appropriate.

Changes to directories which have a maintainer defined shall be sent to the maintainer for review before being committed. Only if the maintainer does not respond for an unacceptable period of time, to several emails, will it be acceptable to commit changes without review by the maintainer. However, it is suggested that you try to have the changes reviewed by someone else if at all possible.

It is of course not acceptable to add a person or group as maintainer unless they agree to assume this duty. On the other hand it does not have to be a committer and it can easily be a group of people.

5.2 Contributed Software

Contributed by Poul-Henning Kamp 和David O'Brien.

Some parts of the FreeBSD distribution consist of software that is actively being maintained outside the FreeBSD project. For historical reasons, we call this *contributed* software. Some examples are **sendmail**, **gcc** and **patch**.

Over the last couple of years, various methods have been used in dealing with this type of software and all have some number of advantages and drawbacks. No clear winner has emerged.

Since this is the case, after some debate one of these methods has been selected as the "official" method and will be required for future imports of software of this kind. Furthermore, it is strongly suggested that existing contributed software converge on this model over time, as it has significant advantages over the old method, including the ability to easily obtain diffs relative to the "official" versions of the source by everyone (even without cvs access). This will make it significantly easier to return changes to the primary developers of the contributed software.

Ultimately, however, it comes down to the people actually doing the work. If using this model is particularly unsuited to the package being dealt with, exceptions to these rules may be granted only with the approval of the core team and with the general consensus of the other developers. The ability to maintain the package in the future will be a key issue in the decisions.

> **注意:** Because of some unfortunate design limitations with the RCS file format and CVS's use of vendor branches, minor, trivial and/or cosmetic changes are *strongly discouraged* on files that are still tracking the vendor branch. "Spelling fixes" are explicitly included here under the "cosmetic" category and are to be avoided for files with revision 1.1.x.x. The repository bloat impact from a single character change can be rather dramatic.

The **Tcl** embedded programming language will be used as example of how this model works:

`src/contrib/tcl` contains the source as distributed by the maintainers of this package. Parts that are entirely not applicable for FreeBSD can be removed. In the case of Tcl, the `mac`, `win` and `compat` subdirectories were eliminated before the import.

`src/lib/libtcl` contains only a **bmake** style `Makefile` that uses the standard `bsd.lib.mk` makefile rules to produce the library and install the documentation.

`src/usr.bin/tclsh` contains only a bmake style `Makefile` which will produce and install the `tclsh` program and its associated man-pages using the standard `bsd.prog.mk` rules.

`src/tools/tools/tcl_bmake` contains a couple of shell-scripts that can be of help when the tcl software needs updating. These are not part of the built or installed software.

The important thing here is that the `src/contrib/tcl` directory is created according to the rules: it is supposed to contain the sources as distributed (on a proper CVS vendor-branch and without RCS keyword expansion) with as few FreeBSD-specific changes as possible. The 'easy-import' tool on `freefall` will assist in doing the import, but if there are any doubts on how to go about it, it is imperative that you ask first and not blunder ahead and hope it "works out". CVS is not forgiving of import accidents and a fair amount of effort is required to back out major mistakes.

Because of the previously mentioned design limitations with CVS's vendor branches, it is required that "official" patches from the vendor be applied to the original distributed sources and the result re-imported onto the vendor branch again. Official patches should never be patched into the FreeBSD checked out version and "committed", as this destroys the vendor branch coherency and makes importing future versions rather difficult as there will be conflicts.

Since many packages contain files that are meant for compatibility with other architectures and environments that FreeBSD, it is permissible to remove parts of the distribution tree that are of no interest to FreeBSD in order to save space. Files containing copyright notices and release-note kind of information applicable to the remaining files shall *not* be removed.

If it seems easier, the `bmake` `Makefile`s can be produced from the dist tree automatically by some utility, something which would hopefully make it even easier to upgrade to a new version. If this is done, be sure to check in such utilities (as necessary) in the `src/tools` directory along with the port itself so that it is available to future maintainers.

In the `src/contrib/tcl` level directory, a file called `FREEBSD-upgrade` should be added and it should state things like:

Which files have been left out.

Where the original distribution was obtained from and/or the official master site.

Where to send patches back to the original authors.

Perhaps an overview of the FreeBSD-specific changes that have been made.

However, please do not import `FREEBSD-upgrade` with the contributed source. Rather you should `cvs add FREEBSD-upgrade ; cvs ci` after the initial import. Example wording from `src/contrib/cpio` is below:

```
This directory contains virgin sources of the original distribution files
on a "vendor" branch.  Do not, under any circumstances, attempt to upgrade
the files in this directory via patches and a cvs commit.  New versions or
official-patch versions must be imported.  Please remember to import with
"-ko" to prevent CVS from corrupting any vendor RCS Ids.

For the import of GNU cpio 2.4.2, the following files were removed:

        INSTALL         cpio.info       mkdir.c
        Makefile.in     cpio.texi       mkinstalldirs

To upgrade to a newer version of cpio, when it is available:
        1. Unpack the new version into an empty directory.
           [Do not make ANY changes to the files.]

        2. Remove the files listed above and any others that don't apply to
           FreeBSD.

        3. Use the command:
                cvs import -ko -m 'Virgin import of GNU cpio v<version>' \
                        src/contrib/cpio GNU cpio_<version>

           For example, to do the import of version 2.4.2, I typed:
                cvs import -ko -m 'Virgin import of GNU v2.4.2' \
                        src/contrib/cpio GNU cpio_2_4_2

        4. Follow the instructions printed out in step 3 to resolve any
           conflicts between local FreeBSD changes and the newer version.

Do not, under any circumstances, deviate from this procedure.

To make local changes to cpio, simply patch and commit to the main
branch (aka HEAD).  Never make local changes on the GNU branch.

All local changes should be submitted to "cpio@gnu.ai.mit.edu" for
inclusion in the next vendor release.

obrien@FreeBSD.org - 30 March 1997
```

5.3 Encumbered Files

It might occasionally be necessary to include an encumbered file in the FreeBSD source tree. For example, if a device requires a small piece of binary code to be loaded to it before the device will operate, and we do not have the source to that code, then the binary file is said to be encumbered. The following policies apply to including encumbered files in the FreeBSD source tree.

1. Any file which is interpreted or executed by the system CPU(s) and not in source format is encumbered.
2. Any file with a license more restrictive than BSD or GNU is encumbered.
3. A file which contains downloadable binary data for use by the hardware is not encumbered, unless (1) or (2) apply to it. It must be stored in an architecture neutral ASCII format (file2c or uuencoding is recommended).
4. Any encumbered file requires specific approval from the Core team (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/contributors/staff-core.html) before it is added to the CVS repository.
5. Encumbered files go in `src/contrib` or `src/sys/contrib`.
6. The entire module should be kept together. There is no point in splitting it, unless there is code-sharing with non-encumbered code.
7. Object files are named *`arch/filename`*`.o.uu>`.
8. Kernel files:
   a. Should always be referenced in `conf/files.*` (for build simplicity).
   b. Should always be in `LINT`, but the Core team (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/contributors/staff-core.html) decides per case if it should be commented out or not. The Core team (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/contributors/staff-core.html) can, of course, change their minds later on.
   c. The *Release Engineer* decides whether or not it goes into the release.
9. User-land files:
   a.

      The Core team (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/contributors/staff-core.html) decides if the code should be part of `make world`.

   b.

      The Release Engineer (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/contributors/staff-who.html) decides if it goes into the release.

5.4 Shared Libraries

Contributed by Satoshi Asami、Peter Wemm 和David O'Brien.

If you are adding shared library support to a port or other piece of software that does not have one, the version numbers should follow these rules. Generally, the resulting numbers will have nothing to do with the release version of the software.

The three principles of shared library building are:

Start from `1.0`

If there is a change that is backwards compatible, bump minor number (note that ELF systems ignore the minor number)

If there is an incompatible change, bump major number

For instance, added functions and bugfixes result in the minor version number being bumped, while deleted functions, changed function call syntax, etc. will force the major version number to change.

Stick to version numbers of the form major.minor (*x.y*). Our a.out dynamic linker does not handle version numbers of the form *x.y.z* well. Any version number after the *y* (i.e. the third digit) is totally ignored when comparing shared lib version numbers to decide which library to link with. Given two shared libraries that differ only in the "micro" revision, `ld.so` will link with the higher one. That is, if you link with `libfoo.so.3.3.3`, the linker only records `3.3` in the headers, and will link with anything starting with *libfoo.so.3.(anything >= 3).(highest available)*.

注意: `ld.so` will always use the highest "minor" revision. For instance, it will use `libc.so.2.2` in preference to `libc.so.2.0`, even if the program was initially linked with `libc.so.2.0`.

In addition, our ELF dynamic linker does not handle minor version numbers at all. However, one should still specify a major and minor version number as our `Makefile`s "do the right thing" based on the type of system.

For non-port libraries, it is also our policy to change the shared library version number only once between releases. In addition, it is our policy to change the major shared library version number only once between major OS releases (i.e. from 3.0 to 4.0). When you make a change to a system library that requires the version number to be bumped, check the `Makefile`'s commit logs. It is the responsibility of the committer to ensure that the first such change since the release will result in the shared library version number in the `Makefile` to be updated, and any subsequent changes will not.

第6章
Regression and Performance Testing

Regression tests are used to exercise a particular bit of the system to check that it works as expected, and to make sure that old bugs are not reintroduced.

The FreeBSD regression testing tools can be found in the FreeBSD source tree in the directory `src/tools/regression`.

6.1. Micro Benchmark Checklist

This section contains hints for doing proper micro-benchmarking on FreeBSD or of FreeBSD itself.

It is not possible to use all of the suggestions below every single time, but the more used, the better the benchmark's ability to test small differences will be.

Disable APM and any other kind of clock fiddling (ACPI ?).

Run in single user mode. E.g. cron(8), and and other daemons only add noise. The sshd(8) daemon can also cause problems. If ssh access is required during test either disable the SSHv1 key regeneration, or kill the parent `sshd` daemon during the tests.

Do not run ntpd(8).

If syslog(3) events are generated, run syslogd(8) with an empty `/etc/syslogd.conf`, otherwise, do not run it.

Minimize disk-I/O, avoid it entirely if possible.

Do not mount file systems that are not needed.

Mount `/`, `/usr`, and any other file system as read-only if possible. This removes atime updates to disk (etc.) from the I/O picture.

Reinitialize the read/write test file system with newfs(8) and populate it from a tar(1) or dump(8) file before every run. Unmount and mount it before starting the test. This results in a consistent file system layout. For a worldstone test this would apply to `/usr/obj` (just reinitialize with `newfs` and mount). To get 100% reproducibility, populate the file system from a dd(1) file (i.e.: `dd if=myimage of=/dev/ad0s1h bs=1m`)

Use malloc backed or preloaded md(4) partitions.

Reboot between individual iterations of the test, this gives a more consistent state.

Remove all non-essential device drivers from the kernel. For instance if USB is not needed for the test, do not put USB in the kernel. Drivers which attach often have timeouts ticking away.

Unconfigure hardware that are not in use. Detach disks with atacontrol(8) and camcontrol(8) if the disks are not used for the test.

Do not configure the network unless it is being tested, or wait until after the test has been performed to ship the results off to another computer.

If the system must be connected to a public network, watch out for spikes of broadcast traffic. Even though it is hardly noticeable, it will take up CPU cycles. Multicast has similar caveats.

Put each file system on its own disk. This minimizes jitter from head-seek optimizations.

Minimize output to serial or VGA consoles. Running output into files gives less jitter. (Serial consoles easily become a bottleneck.) Do not touch keyboard while the test is running, even **space** or **back-space** shows up in the numbers.

Make sure the test is long enough, but not too long. If the test is too short, timestamping is a problem. If it is too long temperature changes and drift will affect the frequency of the quartz crystals in the computer. Rule of thumb: more than a minute, less than an hour.

Try to keep the temperature as stable as possible around the machine. This affects both quartz crystals and disk drive algorithms. To get real stable clock, consider stabilized clock injection. E.g. get a OCXO + PLL, inject output into clock circuits instead of motherboard xtal. Contact Poul-Henning Kamp <`phk@FreeBSD.org`> for more information about this.

Run the test at least 3 times but it is better to run more than 20 times both for "before" and "after" code. Try to interleave if possible (i.e.: do not run 20 times before then 20 times after), this makes it possible to spot environmental effects. Do not interleave 1:1, but 3:3, this makes it possible to spot interaction effects.

A good pattern is: `bababa{bbbaaa}*`. This gives hint after the first 1+1 runs (so it is possible to stop the test if it goes entirely the wrong way), a standard deviation after the first 3+3 (gives a good indication if it is going to be worth a long run) and trending and interaction numbers later on.

Use `usr/src/tools/tools/ministat` to see if the numbers are significant. Consider buying "Cartoon guide to statistics" ISBN: 0062731025, highly recommended, if you have forgotten or never learned about standard deviation and Student's T.

Do not use background fsck(8) unless the test is a benchmark of background `fsck`. Also, disable `background_fsck` in `/etc/rc.conf` unless the benchmark is not started at least 60+ "`fsck` runtime" seconds after the boot, as rc(8) wakes up and checks if `fsck` needs to run on any file systems when background `fsck` is enabled. Likewise, make sure there are no snapshots lying around unless the benchmark is a test with snapshots.

If the benchmark show unexpected bad performance, check for things like high interrupt volume from an unexpected source. Some versions of ACPI have been reported to "misbehave" and generate excess interrupts. To help diagnose odd test results, take a few snapshots of `vmstat -i` and look for anything unusual.

Make sure to be careful about optimization parameters for kernel and userspace, likewise debugging. It is easy to let something slip through and realize later the test was not comparing the same thing.

Do not ever benchmark with the `WITNESS` and `INVARIANTS` kernel options enabled unless the test is interested to benchmarking those features. `WITNESS` can cause 400%+ drops in performance. Likewise, userspace malloc(3) parameters default differently in -CURRENT from the way they ship in production releases.

II. 进程间通信

第7章
套接字

供稿：*G. Adam Stanislav.* 翻译：*intron @NewSMTH.*

7.1 概述

BSD 套接字(socket)将进程间通信推到一个新的水平。彼此通信的进程可不再必须运行在同一计算机上。它们仍然还能够运行在同一计算机上，但不再必须那样。

不仅这些进程不必运行在同一计算机上，它们也不必运行在同一种操作系统上。有了BSD 套接字，你的FreeBSD 软件能够与运行在Macintosh®中的程序顺利的协同工作，也可以与另一个在Sun™ 工作站上的，或是另一个运行在Windows® 2000中的，只要这些系统用以太网型的局域网相连。

你的软件还可以很好的与运行在另一幢大楼，或是在另一个大陆、在一艘潜艇中的，或是一架航天飞机中的进程协同工作。

它也能够与并非属于计算机一部分(至少从术语的严格意义上说不是) 的组件协同工作，这种设备像打印机、数码相机、医疗设备，大致只要是任何能够进行数字通信的东西。

7.2 联网和多样性

我们已经暗示了联网的多样性问题。许多不同的系统要彼此对话。它们必须说同一种语言。与此同时，它们也必须理解同一种语言。

人们常常认为肢体语言是通用的。事实并非如此。回想在我刚刚十几岁时，我的父亲带我去保加利亚。一次我们正坐在索非亚一座公园里的桌子旁，一个小贩上来向我们推销烤杏仁。

那时我还没有学习多少保加利亚语，我没有说“不”，而是摇了摇头，那是“通用的”说不的肢体语言。小贩很快开始装给我们一些杏仁。

然后我想起我曾被告知在保加利亚摇头表示是。很快，我又开始上下点头。小贩注意到了，就拿起他的杏仁走开了。对于一个统一的观察者，我没有改变肢体语言：我继续使用摇头和点头的语言。被改变的是肢体语言的意义。最初，小贩和我将同一种语言理解为完全不同的意义。我必须校正我自己对那种语言的了解，这样小贩才会明白。

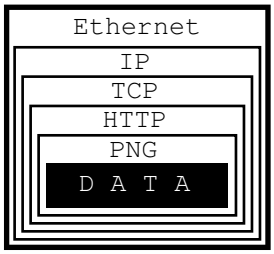
对于计算机也是同样的：同样的符号可能会有不同的，乃至截然相反的意义。所以，为了让两台计算机明白彼此，它们不仅要对于相同的语言有默契，还必须对这种语言的理解有默契。

7.3 协议

当各种各样的编程语言动辄有着复杂的语法，并且使用了许多多字母保留字(这使用它们易于被人类程序员明白)；数据通信的语言则倾向于简洁。它们经常使用一个个二进制位，而不是多字节单词。这有一个很令人信服的理由：数据在你的计算机内部 可以以光速高速行进，它却通常只能在两台计算机之间以慢的多速度行进。

因为数据通信中使用的语言很简洁，我们通常把它们称为协议，而不是语言。

当数据从一台计算机行进到另一台时，它一般使用超过一种协议。这样协议是分层次的。数据可以与一头洋葱的芯类比：只有你剥开几层“表皮” 后才可取得数据。这最好用一张图说明：



在这个例子中，我们尝试从用以太网连着的网页上获取一幅图像。

图像由原始数据组成，那是一个我们的软件能够处理的(转换为一幅图片并显示在我们的显示器上)红绿蓝值序列。

唉，我们的软件无法知道原始数据是如何组织的：那是一个红绿蓝值序列，还是一个灰度序列，或者可能是CMYK编码的色彩？数据是表现为8位离散值，还是16位大小，或是4位？图像由多少行和列组织？有的像素应当是透明的吗?

我想你得到了图片……

为了统一我们的软件处理原始数据的方式，数据被编码为PNG文件。那也可以是GIF，或JPEG文件，不一定只是PNG文件。

于是PNG就是一种协议。

对于这一点，我可以听见你的喊声：“不，那不是！那是一种文件格式！”

好，那当然是一种文件格式。但从数据通信的方面说，一种文件格式也是一种协议：文件结构是一种语言，而且还是一种简洁的语言，与我们的进程通信，确定数据如何被组织。因此，那是一种协议。

唉，假如我们接收到的只有PNG文件，我们的软件将要面对一个严峻的问题：将如何知道数据代表一幅图像，而不是一些文本、或可能是一段声音，或者这些都不是？其次，将如何知道图像是PNG格式的，而不是GIF，或是JPEG，或是其它图像格式?

要取得那些信息，我们使用另一种协议：HTTP。这种协议能告诉我们数据确实代表一幅图像，并且图像使用PNG协议。它也能告诉我们其它一些东西，不过还是让我们把注意力停留在协议层次这里吧。

至此，我们有一些数据被包裹在PNG 协议中，而后又被包裹在HTTP协议中。我们如何从服务器上取得它?

通过在以太网上使用TCP/IP，这就是方法。实际上，有比三种更多的协议。我不再继续深入了，我现在开始说说以太网，只因为这样更容易解释其余的问题。

以太网是一种有趣的系统，它将计算机连接在一个局域网 (*local area network*，LAN)中。每台计算机有一个网络接口卡(中文简称“网卡”) (*network interface card*，NIC)。每个网卡有一个唯一的48位标识，称为它的地址。世界上没有两块网卡会有相同的地址。

这些网卡彼此相连。一旦一台计算机要与在同一以太网局域网中的另一台计算机通信时，就在网络上发送一条消息。每个网卡都会看见这条消息。但是作为以太网协议的一部分，数据包含目的网卡的地址(还有其它内容)。所以，在所有网卡中只有一个会注意那条消息，其余的则会忽略。

但并非所有的计算机都被连接在同一网络上。因为我们在我们的以太网上所接收到的数据并不意味着那一定源自于我们的局域网。可能有来自其它通过Internet 与我们自己的网络相连的网络的数据来我们面

前。

在Internet上传送的所有数据都使用IP。IP表示网间协议 (*Internet Protocol*)。它的基本功能是让我们知道世界上的数据从哪里到来，应该会到哪里去。它并不保证我们一定会接收到数据，只保证假如我们接收到数据时会知道它从哪里来。

甚至即使我们接收到数据，IP 也不保证我们会按照其它计算机发送数据段的顺序接收到这些数据段。举个例子，我们接收到图像的中心部分可能在接收到左上角之前，又可能在接收到右下角之后。

是TCP (*Transmission Control Protocol*，传输控制协议) 要求发送方重发丢失的数据，并且把数据都排成正确的顺序。

总结起来，一台计算机与另一台计算机通信一幅图像的样子需要五个不同的协议。我们接收到的数据被包裹进PNG协议，这又被包裹进HTTP协议，而后又被包裹进TCP协议，再后来又被包裹进IP协议，最后被包裹进Ethernet协议。

欧，顺便说一下，可能有几个其它的协议包含在那其中的某个位置。例如，如果我们的局域网通过电话呼叫接入Internet，就会在调制解调器上使用PPP协议，而调制解调器还可能使用一个(或多个)调制解调器协议，等等，等等，等等……

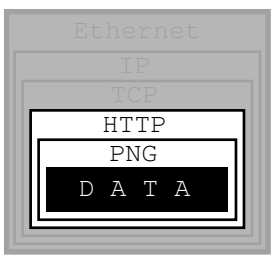
到现在为止作为一个开发者你应该问：“我应该如何掌握它们全部？”

你是幸运的，你不必掌握它们全部。你只要掌握其中的一部分，而不是全部。尤其你不需要担心物理连接(在我们的情形中是以太网和可能的PPP等)。你也不需要掌握网间协议，或是传输控制协议。

换句话说，你不必为从其它计算机接收数据做所有的事情。好，你又要问要做什么，事实上就像打开一个文件一样简单。

一旦你收到数据，就需要你指出如何处理。在我们的情形中，你需要明白HTTP协议和PNG文件结构。

以此类推，所有联网协议变成一个灰色区域：并非因为我们不明白它们如何工作，而是因为我们不必关心它们。套接字接口为我们照管这些灰色区域：



我们只需要明白告诉我们如何理解数据的协议，而不是如何从其它进程接收数据，也不是如何向其它进程发送数据。

7.4 套接字模型

BSD套接字构建在基本的UNIX模型上：一切都是文件。那么，在我们的例子中，套接字将使我们接收一个*HTTP*文件，就这么说。然后我们要负责将*PNG*文件从中提取出来。

由于联网的复杂性，我们不能只使用`open`系统调用，或`open()` C 函数。而是我们需要分几步“打开”一个套接字。

一旦我们做了这些，我们就能以处理任何文件描述符 的方式处理套接字。我们从它读取(read)，向它写入(write)，建立管道(pipe)，必定还要关闭(close)它。

7.5 重要的套接字函数

FreeBSD提供了与套接字相关的不同函数，“打开”一个套接字我们只需要四个函数。有时我们只需要两个。

7.5.1 客户端-服务器差异

典型情况中，以套接字为基础的数据通信一端是一个服务器，另一端是一个客户端。

7.5.1.1 通用元素

7.5.1.1.1 socket

这一个函数在客户端和服务器都要使用：socket(2)。它是这样被声明的：

```
int socket(int domain, int type, int protocol);
```

返回值的类型与open的相同，一个整数。FreeBSD从和文件句柄相同的池中分配它的值。这就是允许套接字被以对文件相同的方式处理的原因。

参数domain告诉系统你需要使用什么协议族。有许多种协议族存在，有些是某些厂商专有的，其它的都非常通用。协议族的声明在sys/socket.h中

使用PF_INET是对于UDP, TCP 和其它网间协议(IPv4)的情况。

对于参数type有五个定义好的值，也在sys/socket.h中。这些值都以“SOCK_”开头。其中最通用的是SOCK_STREAM，它告诉系统你正需要一个可靠的流传送服务 (和PF_INET一起使用时是指TCP)。

如果指定SOCK_DGRAM，你是在请求无连接报文传送服务 (在我们的情形中是UDP)。

如何你需要处理基层协议(例如IP)，或者甚至是网络接口(例如，以太网)，你就需要指定SOCK_RAW。

最后，参数protocol取决于前两个参数，并非总是有意义。在以上情形中，使用取值0。

未连接的套接字: 对于函数socket 我们还没有指定我们要连往什么其它(主机)系统。我们新建的套接字还是未连接的。

这是有意的：拿电话类比，我们刚把调制解调器接在电话线上。我们既没有告诉调制解调器发起一个呼叫，也不会应答电话振铃。

7.5.1.1.2 sockaddr

各种各样的套接字函数需要指定地址，那是一小块内存空间(用C语言术语是指向一小块内存空间的指针)。在sys/socket.h中有各种各样如struct sockaddr的声明。这个结构是这样被声明的：

```
/*
 * 内核用来存储大多数种类地址的结构
```

```
 */
struct sockaddr {
unsigned char sa_len; /* 总长度 */
sa_family_t sa_family; /* 地址族 */
char sa_data[14]; /* 地址值，实际可能更长 */
};
#define SOCK_MAXADDRLEN 255 /* 可能的最长的地址长度 */
```

注意对于sa_data域的定义有些不确定性。那只是被定义为14字节的数组，注释暗示内容可能超过14字节

这种不确定性是经过深思熟虑的。套接字是个非常强大的接口。多数人可能认为比Internet接口强不到哪里——大多数应用现在很可能都用它——套接字可被用于几乎任何种类的进程间通信，Internet(更精确的说是IP)只是其中的一种。

sys/socket.h提到的各种类型的协议将被按照地址族对待，并把它们就列在sockaddr定义的前面：

```
/*
 * 地址族
 */
#define AF_UNSPEC       0               /* 未指定 */
#define AF_LOCAL        1               /* 本机 (管道，portal) */
#define AF_UNIX         AF_LOCAL        /* 为了向前兼容 */
#define AF_INET         2               /* 网间协议: UDP, TCP, 等等 */
#define AF_IMPLINK      3               /* arpanet imp 地址 */
#define AF_PUP          4               /* pup 协议: 例如BSP */
#define AF_CHAOS        5               /* MIT CHAOS 协议 */
#define AF_NS           6               /* 施乐(XEROX) NS 协议 */
#define AF_ISO          7               /* ISO 协议 */
#define AF_OSI          AF_ISO
#define AF_ECMA         8               /* 欧洲计算机制造商协会 */
#define AF_DATAKIT      9               /* datakit 协议 */
#define AF_CCITT        10              /* CCITT 协议, X.25 等 */
#define AF_SNA          11              /* IBM SNA */
#define AF_DECnet       12              /* DECnet */
#define AF_DLI          13              /* DEC 直接数据链路接口 */
#define AF_LAT          14              /* LAT */
#define AF_HYLINK       15              /* NSC Hyperchannel */
#define AF_APPLETALK    16              /* Apple Talk */
#define AF_ROUTE        17              /* 内部路由协议 */
#define AF_LINK         18              /* 协路层接口 */
#define pseudo_AF_XTP   19              /* eXpress Transfer Protocol (no AF) */
#define AF_COIP         20              /* 面向连接的IP, 又名 ST II */
#define AF_CNT          21              /* Computer Network Technology */
#define pseudo_AF_RTIP  22              /* 用于识别RTIP包 */
#define AF_IPX          23              /* Novell 网间协议 */
#define AF_SIP          24              /* Simple 网间协议 */
#define pseudo_AF_PIP   25              /* 用于识别PIP包 */
#define AF_ISDN         26              /* 综合业务数字网(Integrated Services Digital Network) */
#define AF_E164         AF_ISDN         /* CCITT E.164 推荐 */
#define pseudo_AF_KEY   27              /* 内部密钥管理功能 */
#define AF_INET6        28              /* IPv6 */
#define AF_NATM         29              /* 本征ATM访问 */
#define AF_ATM          30              /* ATM */
```

```
#define pseudo_AF_HDRCMPLT 31           /* 由BPF使用，就不必在接口输出例程
                                         * 中重写头文件了
                                         */
#define AF_NETGRAPH     32              /* Netgraph 套接字 */
#define AF_SLOW         33              /* 802.3ad 慢速协议 */
#define AF_SCLUSTER     34              /* Sitara 集群协议 */
#define AF_ARP          35
#define AF_BLUETOOTH    36              /* 蓝牙套接字 */
#define AF_MAX          37
```

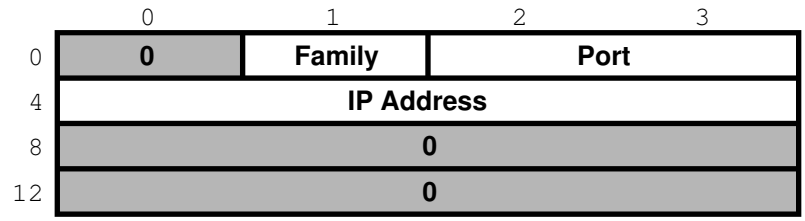
用于指定IP的是AF_INET。这个符号对应着恒量2。

在sockaddr中的域sa_family指定地址族，从而决定预先只确定下大致字节数的sa_data的实际大小。

特别是当地址族 是AF_INET时，我们可以使用struct sockaddr_in，这可在netinet/in.h中找到，任何需要sockaddr的地方都以此作为实际替代。

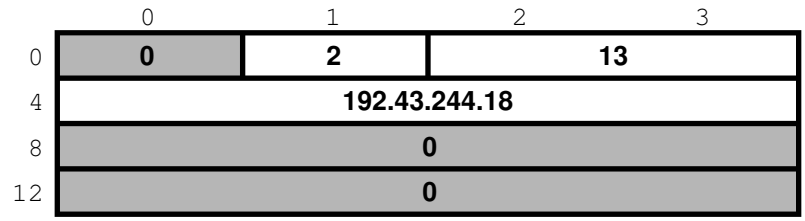
```
/*
 * 套接字地址，Internet风格
 */
struct sockaddr_in {
uint8_t sin_len;
sa_family_t sin_family;
in_port_t sin_port;
struct in_addr sin_addr;
char sin_zero[8];
};
```

我们可这样描绘它的结构：



三个重要的域是：sin_family，结构体的字节1；sin_port，16位值，在字节2和3；sin_addr，一个32位整数，表示IP地址，存储在字节4-7。

现在，让我们尝试填满它。让我们假设我们正在写一个*daytime*协议的客户端，这个协议只是简单的规定服务器写出一个代表当前日期和时间文本字符串到端口13。我们需要使用TCP/IP，所以我们需要指定在地址族域指定AF_INET。AF_INET被定义为2。让我们使用IP地址192.43.244.18，这指向美国联邦政府(time.nist.gov)的服务器。



顺便说一下，域sin_addr被声明为类型struct in_addr，这个类型定义在netinet/in.h之中：

```
/*
 * Internet 地址 (由历史原因而形成的结构)
 */
struct in_addr {
in_addr_t s_addr;
};
```

而in_addr_t是一个32位整数。

192.43.244.18 只是为了表示32位整数的方便写法，按第个8位字节列出，以最高位的字节开始。

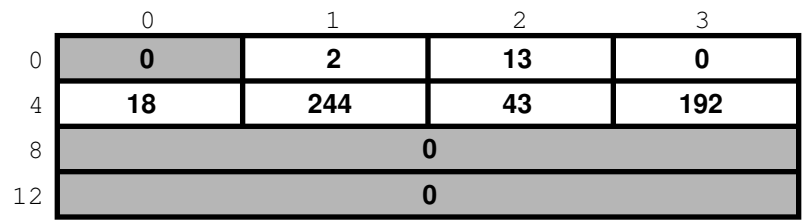
到目前为止，我已经看见了sockaddr。我们的计算机并不将短整数存储为一个16位实体，而是一个2字节序列。同样的，计算机将32位整数存储为4字节序列。

想象我们这样写程序：

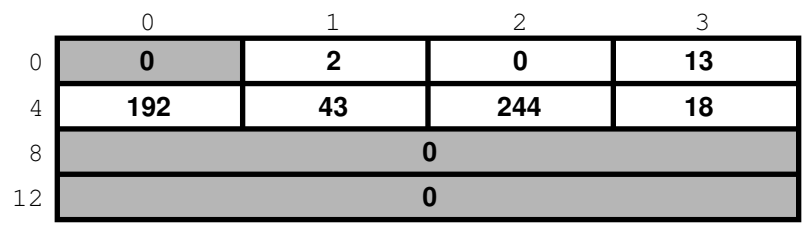
```
sa.sin_family      = AF_INET;
sa.sin_port        = 13;
sa.sin_addr.s_addr = (((((192 << 8) | 43) << 8) | 244) << 8) | 18;
```

结果会是什么样的呢?

好，那当然是要依赖于其它因素的。在Pentium®或其它x86 为基础的计算机上，它会像这样：



在另一个不同的系统上，它可能会是：



And on a PDP it might look different yet. But the above two are the most common ways in use today.

Ordinarily, wanting to write portable code, programmers pretend that these differences do not exist. And they get away with it (except when they code in assembly language). Alas, you cannot get away with it that easily when coding for sockets.

Why?

Because when communicating with another computer, you usually do not know whether it stores data *most significant byte* (MSB) or *least significant byte* (LSB) first.

You might be wondering, *"So, will sockets not handle it for me?"*

It will not.

While that answer may surprise you at first, remember that the general sockets interface only understands the `sa_len` and `sa_family` fields of the `sockaddr` structure. You do not have to worry about the byte order there (of course, on FreeBSD `sa_family` is only 1 byte anyway, but many other UNIX systems do not have `sa_len` and use 2 bytes for `sa_family`, and expect the data in whatever order is native to the computer).

But the rest of the data is just `sa_data[14]` as far as sockets goes. Depending on the *address family*, sockets just forwards that data to its destination.

Indeed, when we enter a port number, it is because we want the other computer to know what service we are asking for. And, when we are the server, we read the port number so we know what service the other computer is expecting from us. Either way, sockets only has to forward the port number as data. It does not interpret it in any way.

Similarly, we enter the IP address to tell everyone on the way where to send our data to. Sockets, again, only forwards it as data.

That is why, we (the *programmers*, not the *sockets*) have to distinguish between the byte order used by our computer and a conventional byte order to send the data in to the other computer.

We will call the byte order our computer uses the *host byte order*, or just the *host order*.

There is a convention of sending the multi-byte data over IP *MSB first*. This, we will refer to as the *network byte order*, or simply the *network order*.

Now, if we compiled the above code for an Intel based computer, our *host byte order* would produce:

	0	1	2	3
0	**0**	**2**	**13**	**0**
4	**18**	**244**	**43**	**192**
8	**0**			
12	**0**			

But the *network byte order* requires that we store the data MSB first:

	0	1	2	3
0	**0**	**2**	**0**	**13**
4	**192**	**43**	**244**	**18**
8	**0**			
12	**0**			

Unfortunately, our *host order* is the exact opposite of the *network order*.

We have several ways of dealing with it. One would be to *reverse* the values in our code:

```
sa.sin_family      = AF_INET;
sa.sin_port        = 13 << 8;
sa.sin_addr.s_addr = (((((18 << 8) | 244) << 8) | 43) << 8) | 192;
```

This will *trick* our compiler into storing the data in the *network byte order*. In some cases, this is exactly the way to do it (e.g., when programming in assembly language). In most cases, however, it can cause a problem.

Suppose, you wrote a sockets-based program in C. You know it is going to run on a Pentium, so you enter all your constants in reverse and force them to the *network byte order*. It works well.

Then, some day, your trusted old Pentium becomes a rusty old Pentium. You replace it with a system whose *host order* is the same as the *network order*. You need to recompile all your software. All of your software continues to perform well, except the one program you wrote.

You have since forgotten that you had forced all of your constants to the opposite of the *host order*. You spend some quality time tearing out your hair, calling the names of all gods you ever heard of (and some you made up), hitting your monitor with a nerf bat, and performing all the other traditional ceremonies of trying to figure out why something that has worked so well is suddenly not working at all.

Eventually, you figure it out, say a couple of swear words, and start rewriting your code.

Luckily, you are not the first one to face the problem. Someone else has created the htons(3) and htonl(3) C functions to convert a `short` and `long` respectively from the *host byte order* to the *network byte order*, and the ntohs(3) and ntohl(3) C functions to go the other way.

On *MSB-first* systems these functions do nothing. On *LSB-first* systems they convert values to the proper order.

So, regardless of what system your software is compiled on, your data will end up in the correct order if you use these functions.

7.5.1.2 Client Functions

Typically, the client initiates the connection to the server. The client knows which server it is about to call: It knows its IP address, and it knows the *port* the server resides at. It is akin to you picking up the phone and dialing the number (the *address*), then, after someone answers, asking for the person in charge of wingdings (the *port*).

7.5.1.2.1 `connect`

Once a client has created a socket, it needs to connect it to a specific port on a remote system. It uses connect(2):

```
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

The `s` argument is the socket, i.e., the value returned by the `socket` function. The `name` is a pointer to `sockaddr`, the structure we have talked about extensively. Finally, `namelen` informs the system how many bytes are in our `sockaddr` structure.

If `connect` is successful, it returns `0`. Otherwise it returns `-1` and stores the error code in `errno`.

There are many reasons why `connect` may fail. For example, with an attempt to an Internet connection, the IP address may not exist, or it may be down, or just too busy, or it may not have a server listening at the specified port. Or it may outright *refuse* any request for specific code.

7.5.1.2.2 Our First Client

We now know enough to write a very simple client, one that will get current time from `192.43.244.18` and print it to `stdout`.

```
/*
 * daytime.c
 *
 * Programmed by G. Adam Stanislav
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main() {
  register int s;
  register int bytes;
  struct sockaddr_in sa;
  char buffer[BUFSIZ+1];

  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return 1;
  }

  bzero(&sa, sizeof sa);

  sa.sin_family = AF_INET;
  sa.sin_port = htons(13);
  sa.sin_addr.s_addr = htonl((((((192 << 8) | 43) << 8) | 244) << 8) | 18);
  if (connect(s, (struct sockaddr *)&sa, sizeof sa) < 0) {
    perror("connect");
    close(s);
    return 2;
  }

  while ((bytes = read(s, buffer, BUFSIZ)) > 0)
    write(1, buffer, bytes);

  close(s);
  return 0;
}
```

Go ahead, enter it in your editor, save it as `daytime.c`, then compile and run it:

```
% cc -O3 -o daytime daytime.c
% ./daytime
```

```
52079 01-06-19 02:29:25 50 0 1 543.9 UTC(NIST) *
%
```

In this case, the date was June 19, 2001, the time was 02:29:25 UTC. Naturally, your results will vary.

7.5.1.3 Server Functions

The typical server does not initiate the connection. Instead, it waits for a client to call it and request services. It does not know when the client will call, nor how many clients will call. It may be just sitting there, waiting patiently, one moment, The next moment, it can find itself swamped with requests from a number of clients, all calling in at the same time.

The sockets interface offers three basic functions to handle this.

7.5.1.3.1 `bind`

Ports are like extensions to a phone line: After you dial a number, you dial the extension to get to a specific person or department.

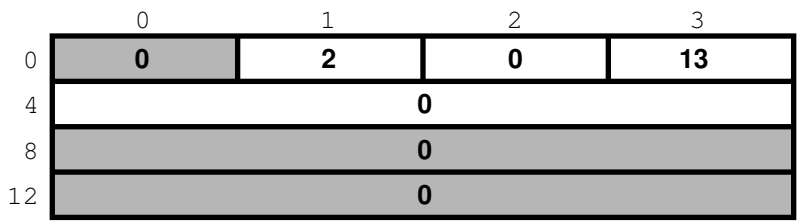
There are 65535 IP ports, but a server usually processes requests that come in on only one of them. It is like telling the phone room operator that we are now at work and available to answer the phone at a specific extension. We use bind(2) to tell sockets which port we want to serve.

```
int bind(int s, const struct sockaddr *addr, socklen_t addrlen);
```

Beside specifying the port in `addr`, the server may include its IP address. However, it can just use the symbolic constant INADDR_ANY to indicate it will serve all requests to the specified port regardless of what its IP address is. This symbol, along with several similar ones, is declared in `netinet/in.h`

```
#define INADDR_ANY (u_int32_t)0x00000000
```

Suppose we were writing a server for the *daytime* protocol over TCP/IP. Recall that it uses port 13. Our `sockaddr_in` structure would look like this:



7.5.1.3.2 `listen`

To continue our office phone analogy, after you have told the phone central operator what extension you will be at, you now walk into your office, and make sure your own phone is plugged in and the ringer is turned on. Plus, you make sure your call waiting is activated, so you can hear the phone ring even while you are talking to someone.

The server ensures all of that with the listen(2) function.

```
int listen(int s, int backlog);
```

In here, the `backlog` variable tells sockets how many incoming requests to accept while you are busy processing the last request. In other words, it determines the maximum size of the queue of pending connections.

7.5.1.3.3 `accept`

After you hear the phone ringing, you accept the call by answering the call. You have now established a connection with your client. This connection remains active until either you or your client hang up.

The server accepts the connection by using the accept(2) function.

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Note that this time `addrlen` is a pointer. This is necessary because in this case it is the socket that fills out `addr`, the `sockaddr_in` structure.

The return value is an integer. Indeed, the `accept` returns a *new socket*. You will use this new socket to communicate with the client.

What happens to the old socket? It continues to listen for more requests (remember the `backlog` variable we passed to `listen`?) until we `close` it.

Now, the new socket is meant only for communications. It is fully connected. We cannot pass it to `listen` again, trying to accept additional connections.

7.5.1.3.4 Our First Server

Our first server will be somewhat more complex than our first client was: Not only do we have more sockets functions to use, but we need to write it as a daemon.

This is best achieved by creating a *child process* after binding the port. The main process then exits and returns control to the **shell** (or whatever program invoked it).

The child calls `listen`, then starts an endless loop, which accepts a connection, serves it, and eventually closes its socket.

```
/*
 * daytimed - a port 13 server
 *
 * Programmed by G. Adam Stanislav
 * June 19, 2001
 */
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BACKLOG 4

int main() {
    register int s, c;
```

```
int b;
struct sockaddr_in sa;
time_t t;
struct tm *tm;
FILE *client;

if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return 1;
}

bzero(&sa, sizeof sa);

sa.sin_family = AF_INET;
sa.sin_port   = htons(13);

if (INADDR_ANY)
    sa.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(s, (struct sockaddr *)&sa, sizeof sa) < 0) {
    perror("bind");
    return 2;
}

switch (fork()) {
    case -1:
        perror("fork");
        return 3;
        break;
    default:
        close(s);
        return 0;
        break;
    case 0:
        break;
}

listen(s, BACKLOG);

for (;;) {
    b = sizeof sa;

    if ((c = accept(s, (struct sockaddr *)&sa, &b)) < 0) {
        perror("daytimed accept");
        return 4;
    }

    if ((client = fdopen(c, "w")) == NULL) {
        perror("daytimed fdopen");
        return 5;
    }

    if ((t = time(NULL)) < 0) {
```

```
            perror("daytimed time");

            return 6;
        }

        tm = gmtime(&t);
        fprintf(client, "%.4i-%.2i-%.2iT%.2i:%.2i:%.2iZ\n",
            tm->tm_year + 1900,
            tm->tm_mon + 1,
            tm->tm_mday,
            tm->tm_hour,
            tm->tm_min,
            tm->tm_sec);

        fclose(client);
    }
}
```

We start by creating a socket. Then we fill out the `sockaddr_in` structure in `sa`. Note the conditional use of INADDR_ANY:

```
    if (INADDR_ANY)
        sa.sin_addr.s_addr = htonl(INADDR_ANY);
```

Its value is `0`. Since we have just used `bzero` on the entire structure, it would be redundant to set it to `0` again. But if we port our code to some other system where INADDR_ANY is perhaps not a zero, we need to assign it to `sa.sin_addr.s_addr`. Most modern C compilers are clever enough to notice that INADDR_ANY is a constant. As long as it is a zero, they will optimize the entire conditional statement out of the code.

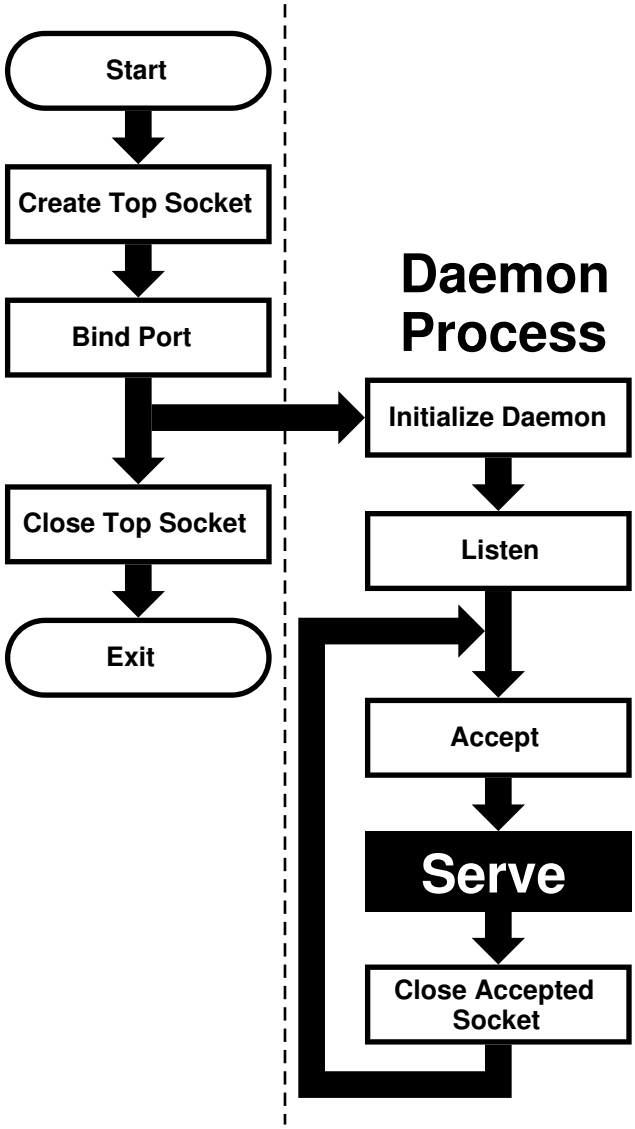
After we have called `bind` successfully, we are ready to become a *daemon*: We use `fork` to create a child process. In both, the parent and the child, the `s` variable is our socket. The parent process will not need it, so it calls `close`, then it returns `0` to inform its own parent it had terminated successfully.

Meanwhile, the child process continues working in the background. It calls `listen` and sets its backlog to `4`. It does not need a large value here because *daytime* is not a protocol many clients request all the time, and because it can process each request instantly anyway.

Finally, the daemon starts an endless loop, which performs the following steps:

1. Call `accept`. It waits here until a client contacts it. At that point, it receives a new socket, `c`, which it can use to communicate with this particular client.
2. It uses the C function `fdopen` to turn the socket from a low-level *file descriptor* to a C-style `FILE` pointer. This will allow the use of `fprintf` later on.
3. It checks the time, and prints it in the *ISO 8601* format to the `client` "file". It then uses `fclose` to close the file. That will automatically close the socket as well.

We can *generalize* this, and use it as a model for many other servers:



This flowchart is good for *sequential servers*, i.e., servers that can serve one client at a time, just as we were able to with our *daytime* server. This is only possible whenever there is no real "conversation" going on between the client and the server: As soon as the server detects a connection to the client, it sends out some data and closes the connection. The entire operation may take nanoseconds, and it is finished.

The advantage of this flowchart is that, except for the brief moment after the parent `fork`s and before it exits, there is always only one *process* active: Our server does not take up much memory and other system resources.

Note that we have added *initialize daemon* in our flowchart. We did not need to initialize our own daemon, but this is a good place in the flow of the program to set up any `signal` handlers, open any files we may need, etc.

Just about everything in the flow chart can be used literally on many different servers. The *serve* entry is the exception. We think of it as a "*black box*", i.e., something you design specifically for your own server, and just "plug it into the rest."

Not all protocols are that simple. Many receive a request from the client, reply to it, then receive another request from the same client. Because of that, they do not know in advance how long they will be serving the client. Such

servers usually start a new process for each client. While the new process is serving its client, the daemon can continue listening for more connections.

Now, go ahead, save the above source code as `daytimed.c` (it is customary to end the names of daemons with the letter `d`). After you have compiled it, try running it:

```
% ./daytimed
bind: Permission denied
%
```

What happened here? As you will recall, the *daytime* protocol uses port 13. But all ports below 1024 are reserved to the superuser (otherwise, anyone could start a daemon pretending to serve a commonly used port, while causing a security breach).

Try again, this time as the superuser:

```
# ./daytimed
#
```

What... Nothing? Let us try again:

```
# ./daytimed

bind: Address already in use
#
```

Every port can only be bound by one program at a time. Our first attempt was indeed successful: It started the child daemon and returned quietly. It is still running and will continue to run until you either kill it, or any of its system calls fail, or you reboot the system.

Fine, we know it is running in the background. But is it working? How do we know it is a proper *daytime* server? Simple:

```
% telnet localhost 13

Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
2001-06-19T21:04:42Z
Connection closed by foreign host.
%
```

telnet tried the new IPv6, and failed. It retried with IPv4 and succeeded. The daemon works.

If you have access to another UNIX system via **telnet**, you can use it to test accessing the server remotely. My computer does not have a static IP address, so this is what I did:

```
% who

whizkid          ttyp0   Jun 19 16:59   (216.127.220.143)
xxx              ttyp1   Jun 19 16:06   (xx.xx.xx.xx)
% telnet 216.127.220.143 13
```

```
Trying 216.127.220.143...
Connected to r47.bfm.org.
Escape character is '^]'.
2001-06-19T21:31:11Z
Connection closed by foreign host.
%
```

Again, it worked. Will it work using the domain name?

```
% telnet r47.bfm.org 13

Trying 216.127.220.143...
Connected to r47.bfm.org.
Escape character is '^]'.
2001-06-19T21:31:40Z
Connection closed by foreign host.
%
```

By the way, **telnet** prints the *Connection closed by foreign host* message after our daemon has closed the socket. This shows us that, indeed, using `fclose(client);` in our code works as advertised.

7.6 辅助函数

FreeBSD C 语言库包含了许多套接字编程的辅助函数。例如，在样例客户端中，我们硬性指定了`time.nist.gov`的IP地址。但是我们并非总是知道IP地址。甚至即使我们知道，允许用户输入IP地址甚至域名将使用我们的软件更有弹性。

7.6.1 `gethostbyname`

域名是不能直接传送给任何套接字函数的，FreeBSD C 语言库携带了函数gethostbyname(3)和gethostbyname2(3)，声明在`netdb.h`中。

```
struct hostent * gethostbyname(const char *name);
struct hostent * gethostbyname2(const char *name, int af);
```

这两个函数都返回`hostent`结构指针，内含有关域的许多信息。对于我们的情况，结构体中的域`h_addr_list[0]`指向长度`h_length`字节的地址，也按网络字节顺序存储。

这允许我们建立一个要有弹性得多的——也要有用得多的——版本的**daytime**程序：

```
/*
 * daytime.c
 *
 * G. Adam Stanislav 编程
 * 2001年6月19日
 */
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int main(int argc, char *argv[]) {
  register int s;
  register int bytes;
  struct sockaddr_in sa;
  struct hostent *he;
  char buf[BUFSIZ+1];
  char *host;

  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return 1;
  }

  bzero(&sa, sizeof sa);

  sa.sin_family = AF_INET;
  sa.sin_port = htons(13);

  host = (argc > 1) ? (char *)argv[1] : "time.nist.gov";

  if ((he = gethostbyname(host)) == NULL) {
    perror(host);
    return 2;
  }

  bcopy(he->h_addr_list[0],&sa.sin_addr, he->h_length);

  if (connect(s, (struct sockaddr *)&sa, sizeof sa) < 0) {
    perror("connect");
    return 3;
  }

  while ((bytes = read(s, buf, BUFSIZ)) > 0)
    write(1, buf, bytes);

  close(s);
  return 0;
}
```

现在我们可以在命令行打一个域名(或者一个IP地址，两种方式都可以)，程序将尝试连接*daytime*服务器。否则，将仍然缺省为`time.nist.gov`。然后，即使在使用缺省值的情形中我们将使用`gethostbyname`而不是硬性指定`192.43.244.18`。这样，即使将来IP地址变更，我们也能找到。

由于从我们的本地计算机获取时间几乎不需要时间，你可以一并运行**daytime**两次：第一次从`time.nist.gov`取得时间，第二次从你自己的系统取得时间。然后你可以比较结果，看看你的系统时钟到底怎么样：

```
% daytime ; daytime localhost
```

```
52080 01-06-20 04:02:33 50 0 0 390.2 UTC(NIST) *
2001-06-20T04:02:35Z
%
```

正如你看见的，我的系统比NIST时间快两秒钟。

7.6.2 getservbyname

有时你不能确定某种服务该用什么端口。函数getservbyname(3)，也声明在netdb.h中，此时就很上手：

```
struct servent * getservbyname(const char *name, const char *proto);
```

结构体servent包含s_port，这是正确的端口号，已经按照网络字节顺序存储。

假如我们不知道*daytime*服务的正确端口，我们可以这样找到：

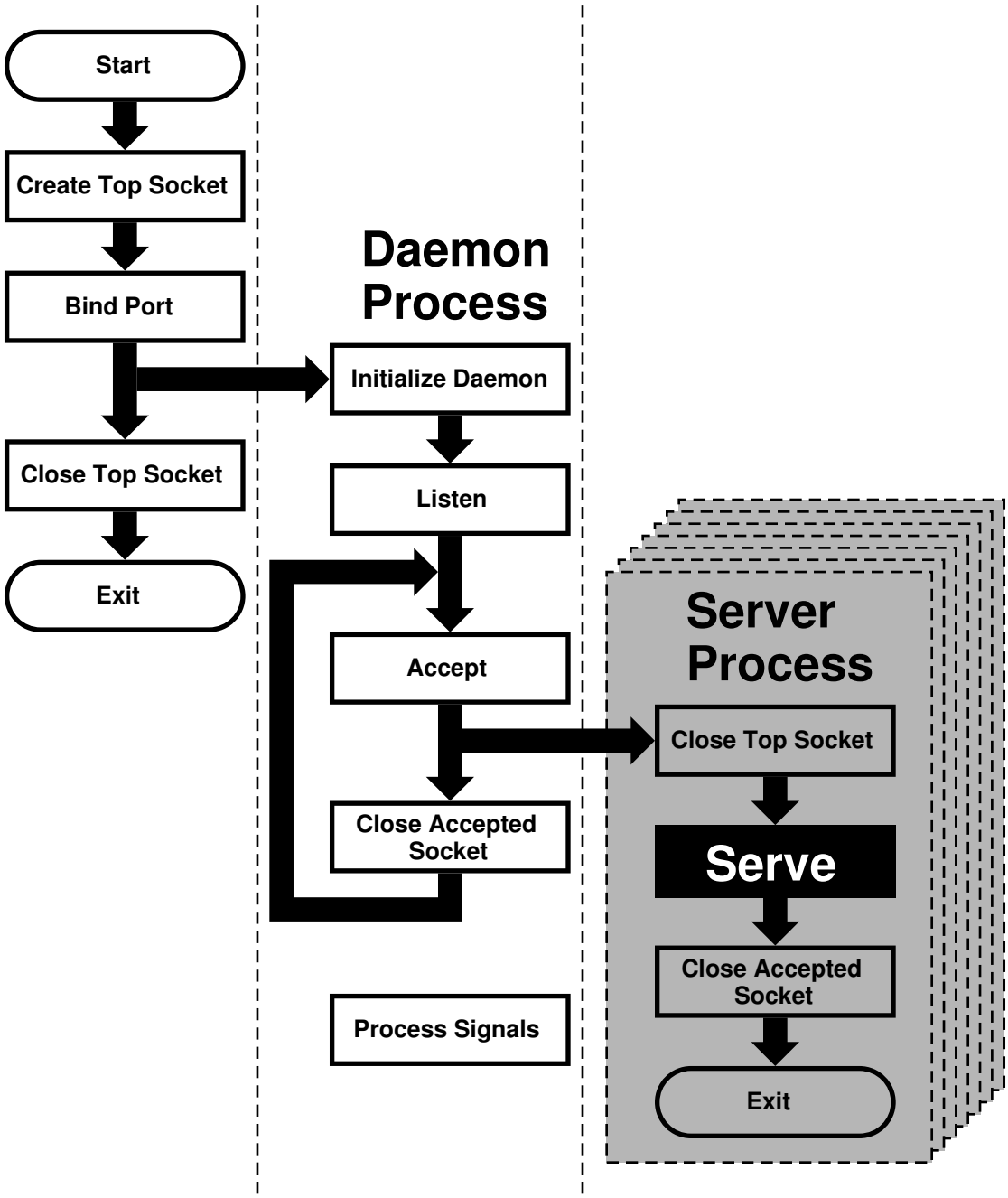
```
struct servent *se;
...
if ((se = getservbyname("daytime", "tcp")) == NULL {
  fprintf(stderr, "Cannot determine which port to use.\n");
  return 7;
}
sa.sin_port = se->s_port;
```

你通常知道端口。但是如何你正开发一个新协议，你可能正在一个非正式端口上测试。有一天，你要注册那个协议和端口(如果不在别处，至少要在你的/etc/services里，那是getservbyname查找的地方)。上面的代码就不再会返回错误，你就可以使用临时端口号。一旦你已经将协议列入/etc/services，你的软件不必重写代码也可以找到端口。

7.7 并发服务器

不同于顺序服务器，并发服务器 就要能在一个时间为多个客户端提供服务。例如，一个聊天服务器可能服务一个特定的客户端数小时——在停止为这个客户端服务之前服务器不能等待，除非是在等待一下个客户端到来之前的间隙才能等待。

这需要在我们的流程图中做一个重要的更改：



我们将提供服务从守护进程移至它自己的服务进程。然而，因为每个子进程都继承所有打开的文件(套接字被像文件一样处理)，新进程不仅继承“*accept()*返回的句柄，” 那是指调用`accept`返回的套接字；新进程也继承顶级套接字，这是顶级进程一开始打开的套接字。

然而，服务进程不需要这个套接字，应该立即关闭它。同样的，守护进程不再需要*accept()*返回的套接字，不仅应该，还必须 关闭它——否则，那迟早会耗尽可用的文件描述符。

在服务进程完成服务之后，它将关闭*accept()*返回的套接字。它不会返回到`accept`，而是退出进程。

在UNIX上，一个进程并不真正的退出，而是返回至父进程。典型情况中，父进程等待(`wait`)子进程，并取得一个返回值。但是，我们的守护进程 不能简单的停止或等待，那有违建立其它进程的整个目的。但

是如果从不使用wait，它的子进程可能会成为僵尸——不再有功用可仍然徘徊着。

出于那样的原因，守护进程 需要在初始化守护进程阶段设置信号处理程序。至少要处理信号SIGCHLD，这样守护进程可以从系统清除僵尸返回值并释放僵尸占用的系统资源。

这是现在我们的流程图包含一个进程信号框的原因，它不与任何其它框相连接。顺便说一句，许多服务器程序也处理SIGHUP，作为超级用户发出的要求重读配置文件的信号。这允许我们不必终止或重启服务器程序就改变设置。

第8章
IPv6 Internals

8.1 IPv6/IPsec Implementation

Contributed by Yoshinobu Inoue.

This section should explain IPv6 and IPsec related implementation internals. These functionalities are derived from KAME project (http://www.kame.net/)

8.1.1 IPv6

8.1.1.1 Conformance

The IPv6 related functions conforms, or tries to conform to the latest set of IPv6 specifications. For future reference we list some of the relevant documents below (*NOTE*: this is not a complete list - this is too hard to maintain...).

For details please refer to specific chapter in the document, RFCs, manual pages, or comments in the source code.

Conformance tests have been performed on the KAME STABLE kit at TAHI project. Results can be viewed at http://www.tahi.org/report/KAME/. We also attended Univ. of New Hampshire IOL tests (http://www.iol.unh.edu/) in the past, with our past snapshots.

RFC1639: FTP Operation Over Big Address Records (FOOBAR)

RFC2428 is preferred over RFC1639. FTP clients will first try RFC2428, then RFC1639 if failed.

RFC1886: DNS Extensions to support IPv6

RFC1933: Transition Mechanisms for IPv6 Hosts and Routers

IPv4 compatible address is not supported.

automatic tunneling (described in 4.3 of this RFC) is not supported.

gif(4) interface implements IPv[46]-over-IPv[46] tunnel in a generic way, and it covers "configured tunnel" described in the spec. See 23.5.1.5 in this document for details.

RFC1981: Path MTU Discovery for IPv6

RFC2080: RIPng for IPv6

usr.sbin/route6d support this.

RFC2292: Advanced Sockets API for IPv6

For supported library functions/kernel APIs, see `sys/netinet6/ADVAPI`.

RFC2362: Protocol Independent Multicast-Sparse Mode (PIM-SM)

RFC2362 defines packet formats for PIM-SM. `draft-ietf-pim-ipv6-01.txt` is written based on this.

RFC2373: IPv6 Addressing Architecture

supports node required addresses, and conforms to the scope requirement.

RFC2374: An IPv6 Aggregatable Global Unicast Address Format

supports 64-bit length of Interface ID.

RFC2375: IPv6 Multicast Address Assignments

Userland applications use the well-known addresses assigned in the RFC.

RFC2428: FTP Extensions for IPv6 and NATs

RFC2428 is preferred over RFC1639. FTP clients will first try RFC2428, then RFC1639 if failed.

RFC2460: IPv6 specification

RFC2461: Neighbor discovery for IPv6

See 23.5.1.2 in this document for details.

RFC2462: IPv6 Stateless Address Autoconfiguration

See 23.5.1.4 in this document for details.

RFC2463: ICMPv6 for IPv6 specification

See 23.5.1.9 in this document for details.

RFC2464: Transmission of IPv6 Packets over Ethernet Networks

RFC2465: MIB for IPv6: Textual Conventions and General Group

Necessary statistics are gathered by the kernel. Actual IPv6 MIB support is provided as a patchkit for ucd-snmp.

RFC2466: MIB for IPv6: ICMPv6 group

Necessary statistics are gathered by the kernel. Actual IPv6 MIB support is provided as patchkit for ucd-snmp.

RFC2467: Transmission of IPv6 Packets over FDDI Networks

RFC2497: Transmission of IPv6 packet over ARCnet Networks

RFC2553: Basic Socket Interface Extensions for IPv6

IPv4 mapped address (3.7) and special behavior of IPv6 wildcard bind socket (3.8) are supported. See 23.5.1.12 in this document for details.

RFC2675: IPv6 Jumbograms

See 23.5.1.7 in this document for details.

RFC2710: Multicast Listener Discovery for IPv6

RFC2711: IPv6 router alert option

`draft-ietf-ipngwg-router-renum-08`: Router renumbering for IPv6

`draft-ietf-ipngwg-icmp-namelookups-02`: IPv6 Name Lookups Through ICMP

`draft-ietf-ipngwg-icmp-name-lookups-03`: IPv6 Name Lookups Through ICMP

`draft-ietf-pim-ipv6-01.txt`: PIM for IPv6

pim6dd(8) implements dense mode. pim6sd(8) implements sparse mode.

`draft-itojun-ipv6-tcp-to-anycast-00`: Disconnecting TCP connection toward IPv6 anycast address

`draft-yamamoto-wideipv6-comm-model-00`

See 23.5.1.6 in this document for details.

`draft-ietf-ipngwg-scopedaddr-format-00.txt` : An Extension of Format for IPv6 Scoped Addresses

8.1.1.2 Neighbor Discovery

Neighbor Discovery is fairly stable. Currently Address Resolution, Duplicated Address Detection, and Neighbor Unreachability Detection are supported. In the near future we will be adding Proxy Neighbor Advertisement support in the kernel and Unsolicited Neighbor Advertisement transmission command as admin tool.

If DAD fails, the address will be marked "duplicated" and message will be generated to syslog (and usually to console). The "duplicated" mark can be checked with ifconfig(8). It is administrators' responsibility to check for and recover from DAD failures. The behavior should be improved in the near future.

Some of the network driver loops multicast packets back to itself, even if instructed not to do so (especially in promiscuous mode). In such cases DAD may fail, because DAD engine sees inbound NS packet (actually from the node itself) and considers it as a sign of duplicate. You may want to look at #if condition marked "heuristics" in sys/netinet6/nd6_nbr.c:nd6_dad_timer() as workaround (note that the code fragment in "heuristics" section is not spec conformant).

Neighbor Discovery specification (RFC2461) does not talk about neighbor cache handling in the following cases:

1. when there was no neighbor cache entry, node received unsolicited RS/NS/NA/redirect packet without link-layer address
2. neighbor cache handling on medium without link-layer address (we need a neighbor cache entry for IsRouter bit)

For first case, we implemented workaround based on discussions on IETF ipngwg mailing list. For more details, see the comments in the source code and email thread started from (IPng 7155), dated Feb 6 1999.

IPv6 on-link determination rule (RFC2461) is quite different from assumptions in BSD network code. At this moment, no on-link determination rule is supported where default router list is empty (RFC2461, section 5.2, last sentence in 2nd paragraph - note that the spec misuse the word "host" and "node" in several places in the section).

To avoid possible DoS attacks and infinite loops, only 10 options on ND packet is accepted now. Therefore, if you have 20 prefix options attached to RA, only the first 10 prefixes will be recognized. If this troubles you, please ask it on FREEBSD-CURRENT mailing list and/or modify nd6_maxndopt in `sys/netinet6/nd6.c`. If there are high demands we may provide sysctl knob for the variable.

8.1.1.3 Scope Index

IPv6 uses scoped addresses. Therefore, it is very important to specify scope index (interface index for link-local address, or site index for site-local address) with an IPv6 address. Without scope index, scoped IPv6 address is ambiguous to the kernel, and kernel will not be able to determine the outbound interface for a packet.

Ordinary userland applications should use advanced API (RFC2292) to specify scope index, or interface index. For similar purpose, sin6_scope_id member in sockaddr_in6 structure is defined in RFC2553. However, the semantics for sin6_scope_id is rather vague. If you care about portability of your application, we suggest you to use advanced API rather than sin6_scope_id.

In the kernel, an interface index for link-local scoped address is embedded into 2nd 16bit-word (3rd and 4th byte) in IPv6 address. For example, you may see something like:

```
fe80:1::200:f8ff:fe01:6317
```

in the routing table and interface address structure (struct in6_ifaddr). The address above is a link-local unicast address which belongs to a network interface whose interface identifier is 1. The embedded index enables us to identify IPv6 link local addresses over multiple interfaces effectively and with only a little code change.

Routing daemons and configuration programs, like route6d(8) and ifconfig(8), will need to manipulate the "embedded" scope index. These programs use routing sockets and ioctls (like SIOCGIFADDR_IN6) and the kernel API will return IPv6 addresses with 2nd 16bit-word filled in. The APIs are for manipulating kernel internal structure. Programs that use these APIs have to be prepared about differences in kernels anyway.

When you specify scoped address to the command line, NEVER write the embedded form (such as ff02:1::1 or fe80:2::fedc). This is not supposed to work. Always use standard form, like ff02::1 or fe80::fedc, with command line option for specifying interface (like `ping6 -I ne0 ff02::1`). In general, if a command does not have command line option to specify outgoing interface, that command is not ready to accept scoped address. This may seem to be opposite from IPv6's premise to support "dentist office" situation. We believe that specifications need some improvements for this.

Some of the userland tools support extended numeric IPv6 syntax, as documented in `draft-ietf-ipngwg-scopedaddr-format-00.txt`. You can specify outgoing link, by using name of the outgoing interface like "fe80::1%ne0". This way you will be able to specify link-local scoped address without much trouble.

To use this extension in your program, you will need to use getaddrinfo(3), and getnameinfo(3) with NI_WITHSCOPEID. The implementation currently assumes 1-to-1 relationship between a link and an interface, which is stronger than what specs say.

8.1.1.4 Plug and Play

Most of the IPv6 stateless address autoconfiguration is implemented in the kernel. Neighbor Discovery functions are implemented in the kernel as a whole. Router Advertisement (RA) input for hosts is implemented in the kernel. Router Solicitation (RS) output for endhosts, RS input for routers, and RA output for routers are implemented in the userland.

8.1.1.4.1 Assignment of link-local, and special addresses

IPv6 link-local address is generated from IEEE802 address (Ethernet MAC address). Each of interface is assigned an IPv6 link-local address automatically, when the interface becomes up (IFF_UP). Also, direct route for the link-local address is added to routing table.

Here is an output of netstat command:

```
Internet6:
Destination                    Gateway                    Flags      Netif Expire
fe80:1::%ed0/64                link#1                     UC          ed0
fe80:2::%ep0/64                link#2                     UC          ep0
```

Interfaces that has no IEEE802 address (pseudo interfaces like tunnel interfaces, or ppp interfaces) will borrow IEEE802 address from other interfaces, such as Ethernet interfaces, whenever possible. If there is no IEEE802 hardware attached, a last resort pseudo-random value, MD5(hostname), will be used as source of link-local address. If it is not suitable for your usage, you will need to configure the link-local address manually.

If an interface is not capable of handling IPv6 (such as lack of multicast support), link-local address will not be assigned to that interface. See section 2 for details.

Each interface joins the solicited multicast address and the link-local all-nodes multicast addresses (e.g. fe80::1:ff01:6317 and ff02::1, respectively, on the link the interface is attached). In addition to a link-local address, the loopback address (::1) will be assigned to the loopback interface. Also, ::1/128 and ff01::/32 are automatically added to routing table, and loopback interface joins node-local multicast group ff01::1.

8.1.1.4.2 Stateless address autoconfiguration on hosts

In IPv6 specification, nodes are separated into two categories: *routers* and *hosts*. Routers forward packets addressed to others, hosts does not forward the packets. net.inet6.ip6.forwarding defines whether this node is router or host (router if it is 1, host if it is 0).

When a host hears Router Advertisement from the router, a host may autoconfigure itself by stateless address autoconfiguration. This behavior can be controlled by net.inet6.ip6.accept_rtadv (host autoconfigures itself if it is set to 1). By autoconfiguration, network address prefix for the receiving interface (usually global address prefix) is added. Default route is also configured. Routers periodically generate Router Advertisement packets. To request an adjacent router to generate RA packet, a host can transmit Router Solicitation. To generate a RS packet at any time, use the *rtsol* command. rtsold(8) daemon is also available. rtsold(8) generates Router Solicitation whenever necessary, and it works great for nomadic usage (notebooks/laptops). If one wishes to ignore Router Advertisements, use sysctl to set net.inet6.ip6.accept_rtadv to 0.

To generate Router Advertisement from a router, use the rtadvd(8) daemon.

Note that, IPv6 specification assumes the following items, and nonconforming cases are left unspecified:

Only hosts will listen to router advertisements

Hosts have single network interface (except loopback)

Therefore, this is unwise to enable net.inet6.ip6.accept_rtadv on routers, or multi-interface host. A misconfigured node can behave strange (nonconforming configuration allowed for those who would like to do some experiments).

To summarize the sysctl knob:

```
accept_rtadv forwarding role of the node
--- --- ---
0 0 host (to be manually configured)
0 1 router
1 0 autoconfigured host
(spec assumes that host has single
interface only, autoconfigured host
with multiple interface is
out-of-scope)
1 1 invalid, or experimental
(out-of-scope of spec)
```

RFC2462 has validation rule against incoming RA prefix information option, in 5.5.3 (e). This is to protect hosts from malicious (or misconfigured) routers that advertise very short prefix lifetime. There was an update from Jim Bound to ipngwg mailing list (look for "(ipng 6712)" in the archive) and it is implemented Jim's update.

See 23.5.1.2 in the document for relationship between DAD and autoconfiguration.

8.1.1.5 Generic tunnel interface

GIF (Generic InterFace) is a pseudo interface for configured tunnel. Details are described in gif(4). Currently

v6 in v6

v6 in v4

v4 in v6

v4 in v4

are available. Use gifconfig(8) to assign physical (outer) source and destination address to gif interfaces. Configuration that uses same address family for inner and outer IP header (v4 in v4, or v6 in v6) is dangerous. It is very easy to configure interfaces and routing tables to perform infinite level of tunneling. *Please be warned*.

gif can be configured to be ECN-friendly. See 23.5.4.5 for ECN-friendliness of tunnels, and gif(4) for how to configure.

If you would like to configure an IPv4-in-IPv6 tunnel with gif interface, read gif(4) carefully. You will need to remove IPv6 link-local address automatically assigned to the gif interface.

8.1.1.6 Source Address Selection

Current source selection rule is scope oriented (there are some exceptions - see below). For a given destination, a source IPv6 address is selected by the following rule:

1. If the source address is explicitly specified by the user (e.g. via the advanced API), the specified address is used.
2. If there is an address assigned to the outgoing interface (which is usually determined by looking up the routing table) that has the same scope as the destination address, the address is used.

   This is the most typical case.
3. If there is no address that satisfies the above condition, choose a global address assigned to one of the interfaces on the sending node.
4. If there is no address that satisfies the above condition, and destination address is site local scope, choose a site local address assigned to one of the interfaces on the sending node.
5. If there is no address that satisfies the above condition, choose the address associated with the routing table entry for the destination. This is the last resort, which may cause scope violation.

For instance, ::1 is selected for ff01::1, fe80:1::200:f8ff:fe01:6317 for fe80:1::2a0:24ff:feab:839b (note that embedded interface index - described in 23.5.1.3 - helps us choose the right source address. Those embedded indices will not be on the wire). If the outgoing interface has multiple address for the scope, a source is selected longest match basis (rule 3). Suppose 3ffe:501:808:1:200:f8ff:fe01:6317 and 3ffe:2001:9:124:200:f8ff:fe01:6317 are given to the outgoing interface. 3ffe:501:808:1:200:f8ff:fe01:6317 is chosen as the source for the destination 3ffe:501:800::1.

Note that the above rule is not documented in the IPv6 spec. It is considered "up to implementation" item. There are some cases where we do not use the above rule. One example is connected TCP session, and we use the address kept in tcb as the source. Another example is source address for Neighbor Advertisement. Under the spec (RFC2461 7.2.2) NA's source should be the target address of the corresponding NS's target. In this case we follow the spec rather than the above longest-match rule.

For new connections (when rule 1 does not apply), deprecated addresses (addresses with preferred lifetime = 0) will not be chosen as source address if other choices are available. If no other choices are available, deprecated address will be used as a last resort. If there are multiple choice of deprecated addresses, the above scope rule will be used to choose from those deprecated addresses. If you would like to prohibit the use of deprecated address for some reason, configure net.inet6.ip6.use_deprecated to 0. The issue related to deprecated address is described in RFC2462 5.5.4 (NOTE: there is some debate underway in IETF ipngwg on how to use "deprecated" address).

8.1.1.7 Jumbo Payload

The Jumbo Payload hop-by-hop option is implemented and can be used to send IPv6 packets with payloads longer than 65,535 octets. But currently no physical interface whose MTU is more than 65,535 is supported, so such payloads can be seen only on the loopback interface (i.e. lo0).

If you want to try jumbo payloads, you first have to reconfigure the kernel so that the MTU of the loopback interface is more than 65,535 bytes; add the following to the kernel configuration file:

```
options "LARGE_LOMTU" #To test jumbo payload
```

and recompile the new kernel.

Then you can test jumbo payloads by the ping6(8) command with -b and -s options. The -b option must be specified to enlarge the size of the socket buffer and the -s option specifies the length of the packet, which should be more than 65,535. For example, type as follows:

```
% ping6 -b 70000 -s 68000 ::1
```

The IPv6 specification requires that the Jumbo Payload option must not be used in a packet that carries a fragment header. If this condition is broken, an ICMPv6 Parameter Problem message must be sent to the sender. specification is followed, but you cannot usually see an ICMPv6 error caused by this requirement.

When an IPv6 packet is received, the frame length is checked and compared to the length specified in the payload length field of the IPv6 header or in the value of the Jumbo Payload option, if any. If the former is shorter than the latter, the packet is discarded and statistics are incremented. You can see the statistics as output of netstat(8) command with '-s -p ip6' option:

```
% netstat -s -p ip6
  ip6:
(snip)
1 with data size < data length
```

So, kernel does not send an ICMPv6 error unless the erroneous packet is an actual Jumbo Payload, that is, its packet size is more than 65,535 bytes. As described above, currently no physical interface with such a huge MTU is supported, so it rarely returns an ICMPv6 error.

TCP/UDP over jumbogram is not supported at this moment. This is because we have no medium (other than loopback) to test this. Contact us if you need this.

IPsec does not work on jumbograms. This is due to some specification twists in supporting AH with jumbograms (AH header size influences payload length, and this makes it real hard to authenticate inbound packet with jumbo payload option as well as AH).

There are fundamental issues in *BSD support for jumbograms. We would like to address those, but we need more time to finalize these. To name a few:

mbuf pkthdr.len field is typed as "int" in 4.4BSD, so it will not hold jumbogram with len > 2G on 32bit architecture CPUs. If we would like to support jumbogram properly, the field must be expanded to hold 4G + IPv6 header + link-layer header. Therefore, it must be expanded to at least int64_t (u_int32_t is NOT enough).

We mistakingly use "int" to hold packet length in many places. We need to convert them into larger integral type. It needs a great care, as we may experience overflow during packet length computation.

We mistakingly check for ip6_plen field of IPv6 header for packet payload length in various places. We should be checking mbuf pkthdr.len instead. ip6_input() will perform sanity check on jumbo payload option on input, and we can safely use mbuf pkthdr.len afterwards.

TCP code needs a careful update in bunch of places, of course.

8.1.1.8 Loop prevention in header processing

IPv6 specification allows arbitrary number of extension headers to be placed onto packets. If we implement IPv6 packet processing code in the way BSD IPv4 code is implemented, kernel stack may overflow due to long function call chain. sys/netinet6 code is carefully designed to avoid kernel stack overflow. Because of this, sys/netinet6 code defines its own protocol switch structure, as "struct ip6protosw" (see `netinet6/ip6protosw.h`). There is no such

update to IPv4 part (sys/netinet) for compatibility, but small change is added to its pr_input() prototype. So "struct ipprotosw" is also defined. Because of this, if you receive IPsec-over-IPv4 packet with massive number of IPsec headers, kernel stack may blow up. IPsec-over-IPv6 is okay. (Off-course, for those all IPsec headers to be processed, each such IPsec header must pass each IPsec check. So an anonymous attacker will not be able to do such an attack.)

8.1.1.9 ICMPv6

After RFC2463 was published, IETF ipngwg has decided to disallow ICMPv6 error packet against ICMPv6 redirect, to prevent ICMPv6 storm on a network medium. This is already implemented into the kernel.

8.1.1.10 Applications

For userland programming, we support IPv6 socket API as specified in RFC2553, RFC2292 and upcoming Internet drafts.

TCP/UDP over IPv6 is available and quite stable. You can enjoy telnet(1), ftp(1), rlogin(1), rsh(1), ssh(1), etc. These applications are protocol independent. That is, they automatically chooses IPv4 or IPv6 according to DNS.

8.1.1.11 Kernel Internals

While ip_forward() calls ip_output(), ip6_forward() directly calls if_output() since routers must not divide IPv6 packets into fragments.

ICMPv6 should contain the original packet as long as possible up to 1280. UDP6/IP6 port unreach, for instance, should contain all extension headers and the *unchanged* UDP6 and IP6 headers. So, all IP6 functions except TCP never convert network byte order into host byte order, to save the original packet.

tcp_input(), udp6_input() and icmp6_input() can not assume that IP6 header is preceding the transport headers due to extension headers. So, in6_cksum() was implemented to handle packets whose IP6 header and transport header is not continuous. TCP/IP6 nor UDP6/IP6 header structures do not exist for checksum calculation.

To process IP6 header, extension headers and transport headers easily, network drivers are now required to store packets in one internal mbuf or one or more external mbufs. A typical old driver prepares two internal mbufs for 96 - 204 bytes data, however, now such packet data is stored in one external mbuf.

`netstat -s -p ip6` tells you whether or not your driver conforms such requirement. In the following example, "cce0" violates the requirement. (For more information, refer to Section 2.)

```
Mbuf statistics:
                317 one mbuf
                two or more mbuf::
                        lo0 = 8
cce0 = 10
                3282 one ext mbuf
                0 two or more ext mbuf
```

Each input function calls IP6_EXTHDR_CHECK in the beginning to check if the region between IP6 and its header is continuous. IP6_EXTHDR_CHECK calls m_pullup() only if the mbuf has M_LOOP flag, that is, the packet comes from the loopback interface. m_pullup() is never called for packets coming from physical network interfaces.

Both IP and IP6 reassemble functions never call m_pullup().

8.1.1.12 IPv4 mapped address and IPv6 wildcard socket

RFC2553 describes IPv4 mapped address (3.7) and special behavior of IPv6 wildcard bind socket (3.8). The spec allows you to:

Accept IPv4 connections by AF_INET6 wildcard bind socket.

Transmit IPv4 packet over AF_INET6 socket by using special form of the address like ::ffff:10.1.1.1.

but the spec itself is very complicated and does not specify how the socket layer should behave. Here we call the former one "listening side" and the latter one "initiating side", for reference purposes.

You can perform wildcard bind on both of the address families, on the same port.

The following table show the behavior of FreeBSD 4.x.

```
listening side          initiating side
                (AF_INET6 wildcard      (connection to ::ffff:10.1.1.1)
                socket gets IPv4 conn.)
                ---                     ---
FreeBSD 4.x     configurable            supported
                default: enabled
```

The following sections will give you more details, and how you can configure the behavior.

Comments on listening side:

It looks that RFC2553 talks too little on wildcard bind issue, especially on the port space issue, failure mode and relationship between AF_INET/INET6 wildcard bind. There can be several separate interpretation for this RFC which conform to it but behaves differently. So, to implement portable application you should assume nothing about the behavior in the kernel. Using getaddrinfo(3) is the safest way. Port number space and wildcard bind issues were discussed in detail on ipv6imp mailing list, in mid March 1999 and it looks that there is no concrete consensus (means, up to implementers). You may want to check the mailing list archives.

If a server application would like to accept IPv4 and IPv6 connections, there will be two alternatives.

One is using AF_INET and AF_INET6 socket (you will need two sockets). Use getaddrinfo(3) with AI_PASSIVE into ai_flags, and socket(2) and bind(2) to all the addresses returned. By opening multiple sockets, you can accept connections onto the socket with proper address family. IPv4 connections will be accepted by AF_INET socket, and IPv6 connections will be accepted by AF_INET6 socket.

Another way is using one AF_INET6 wildcard bind socket. Use getaddrinfo(3) with AI_PASSIVE into ai_flags and with AF_INET6 into ai_family, and set the 1st argument hostname to NULL. And socket(2) and bind(2) to the address returned. (should be IPv6 unspecified addr). You can accept either of IPv4 and IPv6 packet via this one socket.

To support only IPv6 traffic on AF_INET6 wildcard binded socket portably, always check the peer address when a connection is made toward AF_INET6 listening socket. If the address is IPv4 mapped address, you may want to reject the connection. You can check the condition by using IN6_IS_ADDR_V4MAPPED() macro.

To resolve this issue more easily, there is system dependent setsockopt(2) option, IPV6_BINDV6ONLY, used like below.

```
int on;

setsockopt(s, IPPROTO_IPV6, IPV6_BINDV6ONLY,
   (char *)&on, sizeof (on)) < 0));
```

When this call succeed, then this socket only receive IPv6 packets.

Comments on initiating side:

Advise to application implementers: to implement a portable IPv6 application (which works on multiple IPv6 kernels), we believe that the following is the key to the success:

NEVER hardcode AF_INET nor AF_INET6.

Use getaddrinfo(3) and getnameinfo(3) throughout the system. Never use gethostby*(), getaddrby*(), inet_*() or getipnodeby*(). (To update existing applications to be IPv6 aware easily, sometime getipnodeby*() will be useful. But if possible, try to rewrite the code to use getaddrinfo(3) and getnameinfo(3).)

If you would like to connect to destination, use getaddrinfo(3) and try all the destination returned, like telnet(1) does.

Some of the IPv6 stack is shipped with buggy getaddrinfo(3). Ship a minimal working version with your application and use that as last resort.

If you would like to use AF_INET6 socket for both IPv4 and IPv6 outgoing connection, you will need to use getipnodebyname(3). When you would like to update your existing application to be IPv6 aware with minimal effort, this approach might be chosen. But please note that it is a temporal solution, because getipnodebyname(3) itself is not recommended as it does not handle scoped IPv6 addresses at all. For IPv6 name resolution, getaddrinfo(3) is the preferred API. So you should rewrite your application to use getaddrinfo(3), when you get the time to do it.

When writing applications that make outgoing connections, story goes much simpler if you treat AF_INET and AF_INET6 as totally separate address family. {set,get}sockopt issue goes simpler, DNS issue will be made simpler. We do not recommend you to rely upon IPv4 mapped address.

8.1.1.12.1 unified tcp and inpcb code

FreeBSD 4.x uses shared tcp code between IPv4 and IPv6 (from sys/netinet/tcp*) and separate udp4/6 code. It uses unified inpcb structure.

The platform can be configured to support IPv4 mapped address. Kernel configuration is summarized as follows:

By default, AF_INET6 socket will grab IPv4 connections in certain condition, and can initiate connection to IPv4 destination embedded in IPv4 mapped IPv6 address.

You can disable it on entire system with sysctl like below.

```
sysctl net.inet6.ip6.mapped_addr=0
```

8.1.1.12.1.1 listening side

Each socket can be configured to support special AF_INET6 wildcard bind (enabled by default). You can disable it on each socket basis with setsockopt(2) like below.

```
int on;

setsockopt(s, IPPROTO_IPV6, IPV6_BINDV6ONLY,
   (char *)&on, sizeof (on)) < 0));
```

Wildcard AF_INET6 socket grabs IPv4 connection if and only if the following conditions are satisfied:

there is no AF_INET socket that matches the IPv4 connection

the AF_INET6 socket is configured to accept IPv4 traffic, i.e. getsockopt(IPV6_BINDV6ONLY) returns 0.

There is no problem with open/close ordering.

8.1.1.12.1.2 initiating side

FreeBSD 4.x supports outgoing connection to IPv4 mapped address (::ffff:10.1.1.1), if the node is configured to support IPv4 mapped address.

8.1.1.13 sockaddr_storage

When RFC2553 was about to be finalized, there was discussion on how struct sockaddr_storage members are named. One proposal is to prepend "__" to the members (like "__ss_len") as they should not be touched. The other proposal was not to prepend it (like "ss_len") as we need to touch those members directly. There was no clear consensus on it.

As a result, RFC2553 defines struct sockaddr_storage as follows:

```
struct sockaddr_storage {
u_char __ss_len; /* address length */
u_char __ss_family; /* address family */
/* and bunch of padding */
};
```

On the contrary, XNET draft defines as follows:

```
struct sockaddr_storage {
u_char ss_len; /* address length */
u_char ss_family; /* address family */
/* and bunch of padding */
};
```

In December 1999, it was agreed that RFC2553bis should pick the latter (XNET) definition.

Current implementation conforms to XNET definition, based on RFC2553bis discussion.

If you look at multiple IPv6 implementations, you will be able to see both definitions. As an userland programmer, the most portable way of dealing with it is to:

1. ensure ss_family and/or ss_len are available on the platform, by using GNU autoconf,
2. have -Dss_family=__ss_family to unify all occurrences (including header file) into __ss_family, or
3. never touch __ss_family. cast to sockaddr * and use sa_family like:

   ```
   struct sockaddr_storage ss;
   family = ((struct sockaddr *)&ss)->sa_family
   ```

8.1.2 Network Drivers

Now following two items are required to be supported by standard drivers:

1. mbuf clustering requirement. In this stable release, we changed MINCLSIZE into MHLEN+1 for all the operating systems in order to make all the drivers behave as we expect.
2. multicast. If ifmcstat(8) yields no multicast group for a interface, that interface has to be patched.

If any of the drivers do not support the requirements, then the drivers can not be used for IPv6 and/or IPsec communication. If you find any problem with your card using IPv6/IPsec, then, please report it to the FreeBSD 问题报告邮件列表 (http://lists.FreeBSD.org/mailman/listinfo/freebsd-bugs).

(NOTE: In the past we required all PCMCIA drivers to have a call to in6_ifattach(). We have no such requirement any more)

8.1.3 Translator

We categorize IPv4/IPv6 translator into 4 types:

Translator A --- It is used in the early stage of transition to make it possible to establish a connection from an IPv6 host in an IPv6 island to an IPv4 host in the IPv4 ocean.

Translator B --- It is used in the early stage of transition to make it possible to establish a connection from an IPv4 host in the IPv4 ocean to an IPv6 host in an IPv6 island.

Translator C --- It is used in the late stage of transition to make it possible to establish a connection from an IPv4 host in an IPv4 island to an IPv6 host in the IPv6 ocean.

Translator D --- It is used in the late stage of transition to make it possible to establish a connection from an IPv6 host in the IPv6 ocean to an IPv4 host in an IPv4 island.

TCP relay translator for category A is supported. This is called "FAITH". We also provide IP header translator for category A. (The latter is not yet put into FreeBSD 4.x yet.)

8.1.3.1 FAITH TCP relay translator

FAITH system uses TCP relay daemon called faithd(8) helped by the kernel. FAITH will reserve an IPv6 address prefix, and relay TCP connection toward that prefix to IPv4 destination.

For example, if the reserved IPv6 prefix is 3ffe:0501:0200:ffff::, and the IPv6 destination for TCP connection is 3ffe:0501:0200:ffff::163.221.202.12, the connection will be relayed toward IPv4 destination 163.221.202.12.

```
destination IPv4 node (163.221.202.12)
  ^
  | IPv4 tcp toward 163.221.202.12
FAITH-relay dual stack node
  ^
  | IPv6 TCP toward 3ffe:0501:0200:ffff::163.221.202.12
source IPv6 node
```

faithd(8) must be invoked on FAITH-relay dual stack node.

For more details, consult `src/usr.sbin/faithd/README`

8.1.4 IPsec

IPsec is mainly organized by three components.

1. Policy Management
2. Key Management
3. AH and ESP handling

8.1.4.1 Policy Management

The kernel implements experimental policy management code. There are two way to manage security policy. One is to configure per-socket policy using setsockopt(2). In this cases, policy configuration is described in ipsec_set_policy(3). The other is to configure kernel packet filter-based policy using PF_KEY interface, via setkey(8).

The policy entry is not re-ordered with its indexes, so the order of entry when you add is very significant.

8.1.4.2 Key Management

The key management code implemented in this kit (sys/netkey) is a home-brew PFKEY v2 implementation. This conforms to RFC2367.

The home-brew IKE daemon, "racoon" is included in the kit (kame/kame/racoon). Basically you will need to run racoon as daemon, then set up a policy to require keys (like `ping -P 'out ipsec esp/transport//use'`). The kernel will contact racoon daemon as necessary to exchange keys.

8.1.4.3 AH and ESP handling

IPsec module is implemented as "hooks" to the standard IPv4/IPv6 processing. When sending a packet, ip{,6}_output() checks if ESP/AH processing is required by checking if a matching SPD (Security Policy Database) is found. If ESP/AH is needed, {esp,ah}{4,6}_output() will be called and mbuf will be updated accordingly. When a packet is received, {esp,ah}4_input() will be called based on protocol number, i.e. (*inetsw[proto])(). {esp,ah}4_input() will decrypt/check authenticity of the packet, and strips off daisy-chained header and padding for ESP/AH. It is safe to strip off the ESP/AH header on packet reception, since we will never use the received packet in "as is" form.

By using ESP/AH, TCP4/6 effective data segment size will be affected by extra daisy-chained headers inserted by ESP/AH. Our code takes care of the case.

Basic crypto functions can be found in directory "sys/crypto". ESP/AH transform are listed in {esp,ah}_core.c with wrapper functions. If you wish to add some algorithm, add wrapper function in {esp,ah}_core.c, and add your crypto algorithm code into sys/crypto.

Tunnel mode is partially supported in this release, with the following restrictions:

IPsec tunnel is not combined with GIF generic tunneling interface. It needs a great care because we may create an infinite loop between ip_output() and tunnelifp->if_output(). Opinion varies if it is better to unify them, or not.

MTU and Don't Fragment bit (IPv4) considerations need more checking, but basically works fine.

Authentication model for AH tunnel must be revisited. We will need to improve the policy management engine, eventually.

8.1.4.4 Conformance to RFCs and IDs

The IPsec code in the kernel conforms (or, tries to conform) to the following standards:

"old IPsec" specification documented in `rfc182[5-9].txt`

"new IPsec" specification documented in `rfc240[1-6].txt`, `rfc241[01].txt`, `rfc2451.txt` and `draft-mcdonald-simple-ipsec-api-01.txt` (draft expired, but you can take from ftp://ftp.kame.net/pub/internet-drafts/ (ftp://ftp.kame.net/pub/internet-drafts/)). (NOTE: IKE specifications, `rfc241[7-9].txt` are implemented in userland, as "racoon" IKE daemon)

Currently supported algorithms are:

old IPsec AH

null crypto checksum (no document, just for debugging)

keyed MD5 with 128bit crypto checksum (`rfc1828.txt`)

keyed SHA1 with 128bit crypto checksum (no document)

HMAC MD5 with 128bit crypto checksum (`rfc2085.txt`)

HMAC SHA1 with 128bit crypto checksum (no document)

old IPsec ESP

null encryption (no document, similar to `rfc2410.txt`)

DES-CBC mode (`rfc1829.txt`)

new IPsec AH

null crypto checksum (no document, just for debugging)

keyed MD5 with 96bit crypto checksum (no document)

keyed SHA1 with 96bit crypto checksum (no document)

HMAC MD5 with 96bit crypto checksum (`rfc2403.txt`)

HMAC SHA1 with 96bit crypto checksum (`rfc2404.txt`)

new IPsec ESP

null encryption (`rfc2410.txt`)

DES-CBC with derived IV (`draft-ietf-ipsec-ciph-des-derived-01.txt`, draft expired)

DES-CBC with explicit IV (`rfc2405.txt`)

3DES-CBC with explicit IV (`rfc2451.txt`)

BLOWFISH CBC (`rfc2451.txt`)

CAST128 CBC (`rfc2451.txt`)

RC5 CBC (`rfc2451.txt`)

each of the above can be combined with:

ESP authentication with HMAC-MD5(96bit)

ESP authentication with HMAC-SHA1(96bit)

The following algorithms are NOT supported:

old IPsec AH

HMAC MD5 with 128bit crypto checksum + 64bit replay prevention (`rfc2085.txt`)

keyed SHA1 with 160bit crypto checksum + 32bit padding (`rfc1852.txt`)

IPsec (in kernel) and IKE (in userland as "racoon") has been tested at several interoperability test events, and it is known to interoperate with many other implementations well. Also, current IPsec implementation as quite wide coverage for IPsec crypto algorithms documented in RFC (we cover algorithms without intellectual property issues only).

8.1.4.5 ECN consideration on IPsec tunnels

ECN-friendly IPsec tunnel is supported as described in `draft-ipsec-ecn-00.txt`.

Normal IPsec tunnel is described in RFC2401. On encapsulation, IPv4 TOS field (or, IPv6 traffic class field) will be copied from inner IP header to outer IP header. On decapsulation outer IP header will be simply dropped. The decapsulation rule is not compatible with ECN, since ECN bit on the outer IP TOS/traffic class field will be lost.

To make IPsec tunnel ECN-friendly, we should modify encapsulation and decapsulation procedure. This is described in http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt (http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt), chapter 3.

IPsec tunnel implementation can give you three behaviors, by setting net.inet.ipsec.ecn (or net.inet6.ipsec6.ecn) to some value:

RFC2401: no consideration for ECN (sysctl value -1)

ECN forbidden (sysctl value 0)

ECN allowed (sysctl value 1)

Note that the behavior is configurable in per-node manner, not per-SA manner (draft-ipsec-ecn-00 wants per-SA configuration, but it looks too much for me).

The behavior is summarized as follows (see source code for more detail):

```
                encapsulate                     decapsulate
                ---                             ---
RFC2401         copy all TOS bits               drop TOS bits on outer
                from inner to outer.            (use inner TOS bits as is)
```

ECN forbidden	copy TOS bits except for ECN (masked with 0xfc) from inner to outer. set ECN bits to 0.	drop TOS bits on outer (use inner TOS bits as is)
ECN allowed	copy TOS bits except for ECN CE (masked with 0xfe) from inner to outer. set ECN CE bit to 0.	use inner TOS bits with some change. if outer ECN CE bit is 1, enable ECN CE bit on the inner.

General strategy for configuration is as follows:

if both IPsec tunnel endpoint are capable of ECN-friendly behavior, you should better configure both end to "ECN allowed" (sysctl value 1).

if the other end is very strict about TOS bit, use "RFC2401" (sysctl value -1).

in other cases, use "ECN forbidden" (sysctl value 0).

The default behavior is "ECN forbidden" (sysctl value 0).

For more information, please refer to:

http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt (http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt), RFC2481 (Explicit Congestion Notification), src/sys/netinet6/{ah,esp}_input.c

(Thanks goes to Kenjiro Cho <kjc@csl.sony.co.jp> for detailed analysis)

8.1.4.6 Interoperability

Here are (some of) platforms that KAME code have tested IPsec/IKE interoperability in the past. Note that both ends may have modified their implementation, so use the following list just for reference purposes.

Altiga, Ashley-laurent (vpcom.com), Data Fellows (F-Secure), Ericsson ACC, FreeS/WAN, HITACHI, IBM AIX®, IIJ, Intel, Microsoft® Windows NT®, NIST (linux IPsec + plutoplus), Netscreen, OpenBSD, RedCreek, Routerware, SSH, Secure Computing, Soliton, Toshiba, VPNet, Yamaha RT100i

III. 内核

第9章
DMA

9.1 DMA: What it is and How it Works

Copyright © 1995,1997 Frank Durda IV <uhclem@FreeBSD.org>, All Rights Reserved. 10 December 1996. Last Update 8 October 1997.

Direct Memory Access (DMA) is a method of allowing data to be moved from one location to another in a computer without intervention from the central processor (CPU).

The way that the DMA function is implemented varies between computer architectures, so this discussion will limit itself to the implementation and workings of the DMA subsystem on the IBM Personal Computer (PC), the IBM PC/AT and all of its successors and clones.

The PC DMA subsystem is based on the Intel® 8237 DMA controller. The 8237 contains four DMA channels that can be programmed independently and any one of the channels may be active at any moment. These channels are numbered 0, 1, 2 and 3. Starting with the PC/AT, IBM added a second 8237 chip, and numbered those channels 4, 5, 6 and 7.

The original DMA controller (0, 1, 2 and 3) moves one byte in each transfer. The second DMA controller (4, 5, 6, and 7) moves 16-bits from two adjacent memory locations in each transfer, with the first byte always coming from an even-numbered address. The two controllers are identical components and the difference in transfer size is caused by the way the second controller is wired into the system.

The 8237 has two electrical signals for each channel, named DRQ and -DACK. There are additional signals with the names HRQ (Hold Request), HLDA (Hold Acknowledge), -EOP (End of Process), and the bus control signals -MEMR (Memory Read), -MEMW (Memory Write), -IOR (I/O Read), and -IOW (I/O Write).

The 8237 DMA is known as a “fly-by” DMA controller. This means that the data being moved from one location to another does not pass through the DMA chip and is not stored in the DMA chip. Subsequently, the DMA can only transfer data between an I/O port and a memory address, but not between two I/O ports or two memory locations.

> **注意:** The 8237 does allow two channels to be connected together to allow memory-to-memory DMA operations in a non-“fly-by” mode, but nobody in the PC industry uses this scarce resource this way since it is faster to move data between memory locations using the CPU.

In the PC architecture, each DMA channel is normally activated only when the hardware that uses a given DMA channel requests a transfer by asserting the DRQ line for that channel.

9.1.1 A Sample DMA transfer

Here is an example of the steps that occur to cause and perform a DMA transfer. In this example, the floppy disk controller (FDC) has just read a byte from a diskette and wants the DMA to place it in memory at location 0x00123456. The process begins by the FDC asserting the DRQ2 signal (the DRQ line for DMA channel 2) to alert the DMA controller.

The DMA controller will note that the DRQ2 signal is asserted. The DMA controller will then make sure that DMA channel 2 has been programmed and is unmasked (enabled). The DMA controller also makes sure that none of the other DMA channels are active or want to be active and have a higher priority. Once these checks are complete, the DMA asks the CPU to release the bus so that the DMA may use the bus. The DMA requests the bus by asserting the HRQ signal which goes to the CPU.

The CPU detects the HRQ signal, and will complete executing the current instruction. Once the processor has reached a state where it can release the bus, it will. Now all of the signals normally generated by the CPU (-MEMR, -MEMW, -IOR, -IOW and a few others) are placed in a tri-stated condition (neither high or low) and then the CPU asserts the HLDA signal which tells the DMA controller that it is now in charge of the bus.

Depending on the processor, the CPU may be able to execute a few additional instructions now that it no longer has the bus, but the CPU will eventually have to wait when it reaches an instruction that must read something from memory that is not in the internal processor cache or pipeline.

Now that the DMA "is in charge", the DMA activates its -MEMR, -MEMW, -IOR, -IOW output signals, and the address outputs from the DMA are set to 0x3456, which will be used to direct the byte that is about to transferred to a specific memory location.

The DMA will then let the device that requested the DMA transfer know that the transfer is commencing. This is done by asserting the -DACK signal, or in the case of the floppy disk controller, -DACK2 is asserted.

The floppy disk controller is now responsible for placing the byte to be transferred on the bus Data lines. Unless the floppy controller needs more time to get the data byte on the bus (and if the peripheral does need more time it alerts the DMA via the READY signal), the DMA will wait one DMA clock, and then de-assert the -MEMW and -IOR signals so that the memory will latch and store the byte that was on the bus, and the FDC will know that the byte has been transferred.

Since the DMA cycle only transfers a single byte at a time, the FDC now drops the DRQ2 signal, so the DMA knows that it is no longer needed. The DMA will de-assert the -DACK2 signal, so that the FDC knows it must stop placing data on the bus.

The DMA will now check to see if any of the other DMA channels have any work to do. If none of the channels have their DRQ lines asserted, the DMA controller has completed its work and will now tri-state the -MEMR, -MEMW, -IOR, -IOW and address signals.

Finally, the DMA will de-assert the HRQ signal. The CPU sees this, and de-asserts the HOLDA signal. Now the CPU activates its -MEMR, -MEMW, -IOR, -IOW and address lines, and it resumes executing instructions and accessing main memory and the peripherals.

For a typical floppy disk sector, the above process is repeated 512 times, once for each byte. Each time a byte is transferred, the address register in the DMA is incremented and the counter in the DMA that shows how many bytes are to be transferred is decremented.

When the counter reaches zero, the DMA asserts the EOP signal, which indicates that the counter has reached zero and no more data will be transferred until the DMA controller is reprogrammed by the CPU. This event is also called the Terminal Count (TC). There is only one EOP signal, and since only one DMA channel can be active at any instant, the DMA channel that is currently active must be the DMA channel that just completed its task.

If a peripheral wants to generate an interrupt when the transfer of a buffer is complete, it can test for its -DACKn signal and the EOP signal both being asserted at the same time. When that happens, it means the DMA will not transfer any more information for that peripheral without intervention by the CPU. The peripheral can then assert one of the interrupt signals to get the processors' attention. In the PC architecture, the DMA chip itself is not capable of generating an interrupt. The peripheral and its associated hardware is responsible for generating any interrupt that occurs. Subsequently, it is possible to have a peripheral that uses DMA but does not use interrupts.

It is important to understand that although the CPU always releases the bus to the DMA when the DMA makes the request, this action is invisible to both applications and the operating system, except for slight changes in the amount of time the processor takes to execute instructions when the DMA is active. Subsequently, the processor must poll the peripheral, poll the registers in the DMA chip, or receive an interrupt from the peripheral to know for certain when a DMA transfer has completed.

9.1.2 DMA Page Registers and 16Meg address space limitations

You may have noticed earlier that instead of the DMA setting the address lines to 0x00123456 as we said earlier, the DMA only set 0x3456. The reason for this takes a bit of explaining.

When the original IBM PC was designed, IBM elected to use both DMA and interrupt controller chips that were designed for use with the 8085, an 8-bit processor with an address space of 16 bits (64K). Since the IBM PC supported more than 64K of memory, something had to be done to allow the DMA to read or write memory locations above the 64K mark. What IBM did to solve this problem was to add an external data latch for each DMA channel that holds the upper bits of the address to be read to or written from. Whenever a DMA channel is active, the contents of that latch are written to the address bus and kept there until the DMA operation for the channel ends. IBM called these latches "Page Registers".

So for our example above, the DMA would put the 0x3456 part of the address on the bus, and the Page Register for DMA channel 2 would put 0x0012xxxx on the bus. Together, these two values form the complete address in memory that is to be accessed.

Because the Page Register latch is independent of the DMA chip, the area of memory to be read or written must not span a 64K physical boundary. For example, if the DMA accesses memory location 0xffff, after that transfer the DMA will then increment the address register and the DMA will access the next byte at location 0x0000, not 0x10000. The results of letting this happen are probably not intended.

> **注意:** "Physical" 64K boundaries should not be confused with 8086-mode 64K "Segments", which are created by mathematically adding a segment register with an offset register. Page Registers have no address overlap and are mathematically OR-ed together.

To further complicate matters, the external DMA address latches on the PC/AT hold only eight bits, so that gives us 8+16=24 bits, which means that the DMA can only point at memory locations between 0 and 16Meg. For newer computers that allow more than 16Meg of memory, the standard PC-compatible DMA cannot access memory locations above 16Meg.

To get around this restriction, operating systems will reserve a RAM buffer in an area below 16Meg that also does not span a physical 64K boundary. Then the DMA will be programmed to transfer data from the peripheral and into that buffer. Once the DMA has moved the data into this buffer, the operating system will then copy the data from the buffer to the address where the data is really supposed to be stored.

When writing data from an address above 16Meg to a DMA-based peripheral, the data must be first copied from where it resides into a buffer located below 16Meg, and then the DMA can copy the data from the buffer to the hardware. In FreeBSD, these reserved buffers are called "Bounce Buffers". In the MS-DOS world, they are sometimes called "Smart Buffers".

> **注意:** A new implementation of the 8237, called the 82374, allows 16 bits of page register to be specified and enables access to the entire 32 bit address space, without the use of bounce buffers.

9.1.3 DMA Operational Modes and Settings

The 8237 DMA can be operated in several modes. The main ones are:

Single

A single byte (or word) is transferred. The DMA must release and re-acquire the bus for each additional byte. This is commonly-used by devices that cannot transfer the entire block of data immediately. The peripheral will request the DMA each time it is ready for another transfer.

The standard PC-compatible floppy disk controller (NEC 765) only has a one-byte buffer, so it uses this mode.

Block/Demand

Once the DMA acquires the system bus, an entire block of data is transferred, up to a maximum of 64K. If the peripheral needs additional time, it can assert the READY signal to suspend the transfer briefly. READY should not be used excessively, and for slow peripheral transfers, the Single Transfer Mode should be used instead.

The difference between Block and Demand is that once a Block transfer is started, it runs until the transfer count reaches zero. DRQ only needs to be asserted until -DACK is asserted. Demand Mode will transfer one more bytes until DRQ is de-asserted, at which point the DMA suspends the transfer and releases the bus back to the CPU. When DRQ is asserted later, the transfer resumes where it was suspended.

Older hard disk controllers used Demand Mode until CPU speeds increased to the point that it was more efficient to transfer the data using the CPU, particularly if the memory locations used in the transfer were above the 16Meg mark.

Cascade

This mechanism allows a DMA channel to request the bus, but then the attached peripheral device is responsible for placing the addressing information on the bus instead of the DMA. This is also used to implement a technique known as “Bus Mastering”.

When a DMA channel in Cascade Mode receives control of the bus, the DMA does not place addresses and I/O control signals on the bus like the DMA normally does when it is active. Instead, the DMA only asserts the -DACK signal for the active DMA channel.

At this point it is up to the peripheral connected to that DMA channel to provide address and bus control signals. The peripheral has complete control over the system bus, and can do reads and/or writes to any address below 16Meg. When the peripheral is finished with the bus, it de-asserts the DRQ line, and the DMA controller can then return control to the CPU or to some other DMA channel.

Cascade Mode can be used to chain multiple DMA controllers together, and this is exactly what DMA Channel 4 is used for in the PC architecture. When a peripheral requests the bus on DMA channels 0, 1, 2 or 3, the slave DMA controller asserts HLDREQ, but this wire is actually connected to DRQ4 on the primary DMA controller instead of to the CPU. The primary DMA controller, thinking it has work to do on Channel 4, requests the bus from the CPU using HLDREQ signal. Once the CPU grants the bus to the primary DMA controller, -DACK4 is asserted, and that wire is actually connected to the HLDA signal on the slave DMA controller. The slave DMA controller then transfers data for the DMA channel that requested it (0, 1, 2 or 3), or the slave DMA may grant the bus to a peripheral that wants to perform its own bus-mastering, such as a SCSI controller.

Because of this wiring arrangement, only DMA channels 0, 1, 2, 3, 5, 6 and 7 are usable with peripherals on PC/AT systems.

> 注意: DMA channel 0 was reserved for refresh operations in early IBM PC computers, but is generally available for use by peripherals in modern systems.

When a peripheral is performing Bus Mastering, it is important that the peripheral transmit data to or from memory constantly while it holds the system bus. If the peripheral cannot do this, it must release the bus frequently so that the system can perform refresh operations on main memory.

The Dynamic RAM used in all PCs for main memory must be accessed frequently to keep the bits stored in the components "charged". Dynamic RAM essentially consists of millions of capacitors with each one holding one bit of data. These capacitors are charged with power to represent a `1` or drained to represent a `0`. Because all capacitors leak, power must be added at regular intervals to keep the `1` values intact. The RAM chips actually handle the task of pumping power back into all of the appropriate locations in RAM, but they must be told when to do it by the rest of the computer so that the refresh activity will not interfere with the computer wanting to access RAM normally. If the computer is unable to refresh memory, the contents of memory will become corrupted in just a few milliseconds.

Since memory read and write cycles "count" as refresh cycles (a dynamic RAM refresh cycle is actually an incomplete memory read cycle), as long as the peripheral controller continues reading or writing data to sequential memory locations, that action will refresh all of memory.

Bus-mastering is found in some SCSI host interfaces and other high-performance peripheral controllers.

Autoinitialize

This mode causes the DMA to perform Byte, Block or Demand transfers, but when the DMA transfer counter reaches zero, the counter and address are set back to where they were when the DMA channel was originally programmed. This means that as long as the peripheral requests transfers, they will be granted. It is up to the CPU to move new data into the fixed buffer ahead of where the DMA is about to transfer it when doing output operations, and to read new data out of the buffer behind where the DMA is writing when doing input operations.

This technique is frequently used on audio devices that have small or no hardware "sample" buffers. There is additional CPU overhead to manage this "circular" buffer, but in some cases this may be the only way to eliminate the latency that occurs when the DMA counter reaches zero and the DMA stops transfers until it is reprogrammed.

9.1.4 Programming the DMA

The DMA channel that is to be programmed should always be "masked" before loading any settings. This is because the hardware might unexpectedly assert the DRQ for that channel, and the DMA might respond, even though not all of the parameters have been loaded or updated.

Once masked, the host must specify the direction of the transfer (memory-to-I/O or I/O-to-memory), what mode of DMA operation is to be used for the transfer (Single, Block, Demand, Cascade, etc), and finally the address and length of the transfer are loaded. The length that is loaded is one less than the amount you expect the DMA to transfer. The LSB and MSB of the address and length are written to the same 8-bit I/O port, so another port must be

written to first to guarantee that the DMA accepts the first byte as the LSB and the second byte as the MSB of the length and address.

Then, be sure to update the Page Register, which is external to the DMA and is accessed through a different set of I/O ports.

Once all the settings are ready, the DMA channel can be un-masked. That DMA channel is now considered to be "armed", and will respond when the DRQ line for that channel is asserted.

Refer to a hardware data book for precise programming details for the 8237. You will also need to refer to the I/O port map for the PC system, which describes where the DMA and Page Register ports are located. A complete port map table is located below.

9.1.5 DMA Port Map

All systems based on the IBM-PC and PC/AT have the DMA hardware located at the same I/O ports. The complete list is provided below. Ports assigned to DMA Controller #2 are undefined on non-AT designs.

9.1.5.1 0x00–0x1f DMA Controller #1 (Channels 0, 1, 2 and 3)

DMA Address and Count Registers

0x00	write	Channel 0 starting address
0x00	read	Channel 0 current address
0x01	write	Channel 0 starting word count
0x01	read	Channel 0 remaining word count
0x02	write	Channel 1 starting address
0x02	read	Channel 1 current address
0x03	write	Channel 1 starting word count
0x03	read	Channel 1 remaining word count
0x04	write	Channel 2 starting address
0x04	read	Channel 2 current address
0x05	write	Channel 2 starting word count
0x05	read	Channel 2 remaining word count
0x06	write	Channel 3 starting address
0x06	read	Channel 3 current address
0x07	write	Channel 3 starting word count
0x07	read	Channel 3 remaining word count

DMA Command Registers

0x08	write	Command Register
0x08	read	Status Register
0x09	write	Request Register
0x09	read	-
0x0a	write	Single Mask Register Bit

0x0a	read	-
0x0b	write	Mode Register
0x0b	read	-
0x0c	write	Clear LSB/MSB Flip-Flop
0x0c	read	-
0x0d	write	Master Clear/Reset
0x0d	read	Temporary Register (not available on newer versions)
0x0e	write	Clear Mask Register
0x0e	read	-
0x0f	write	Write All Mask Register Bits
0x0f	read	Read All Mask Register Bits (only in Intel 82374)

9.1.5.2 0xc0–0xdf DMA Controller #2 (Channels 4, 5, 6 and 7)

DMA Address and Count Registers

0xc0	write	Channel 4 starting address
0xc0	read	Channel 4 current address
0xc2	write	Channel 4 starting word count
0xc2	read	Channel 4 remaining word count
0xc4	write	Channel 5 starting address
0xc4	read	Channel 5 current address
0xc6	write	Channel 5 starting word count
0xc6	read	Channel 5 remaining word count
0xc8	write	Channel 6 starting address
0xc8	read	Channel 6 current address
0xca	write	Channel 6 starting word count
0xca	read	Channel 6 remaining word count
0xcc	write	Channel 7 starting address
0xcc	read	Channel 7 current address
0xce	write	Channel 7 starting word count
0xce	read	Channel 7 remaining word count

DMA Command Registers

0xd0	write	Command Register
0xd0	read	Status Register
0xd2	write	Request Register
0xd2	read	-

0xd4	write	Single Mask Register Bit
0xd4	read	-
0xd6	write	Mode Register
0xd6	read	-
0xd8	write	Clear LSB/MSB Flip-Flop
0xd8	read	-
0xda	write	Master Clear/Reset
0xda	read	Temporary Register (not present in Intel 82374)
0xdc	write	Clear Mask Register
0xdc	read	-
0xde	write	Write All Mask Register Bits
0xdf	read	Read All Mask Register Bits (only in Intel 82374)

9.1.5.3 0x80–0x9f DMA Page Registers

0x87	r/w	Channel 0 Low byte (23-16) page Register
0x83	r/w	Channel 1 Low byte (23-16) page Register
0x81	r/w	Channel 2 Low byte (23-16) page Register
0x82	r/w	Channel 3 Low byte (23-16) page Register
0x8b	r/w	Channel 5 Low byte (23-16) page Register
0x89	r/w	Channel 6 Low byte (23-16) page Register
0x8a	r/w	Channel 7 Low byte (23-16) page Register
0x8f	r/w	Low byte page Refresh

9.1.5.4 0x400–0x4ff 82374 Enhanced DMA Registers

The Intel 82374 EISA System Component (ESC) was introduced in early 1996 and includes a DMA controller that provides a superset of 8237 functionality as well as other PC-compatible core peripheral components in a single package. This chip is targeted at both EISA and PCI platforms, and provides modern DMA features like scatter-gather, ring buffers as well as direct access by the system DMA to all 32 bits of address space.

If these features are used, code should also be included to provide similar functionality in the previous 16 years worth of PC-compatible computers. For compatibility reasons, some of the 82374 registers must be programmed

after programming the traditional 8237 registers for each transfer. Writing to a traditional 8237 register forces the contents of some of the 82374 enhanced registers to zero to provide backward software compatibility.

0x401	r/w	Channel 0 High byte (bits 23-16) word count
0x403	r/w	Channel 1 High byte (bits 23-16) word count
0x405	r/w	Channel 2 High byte (bits 23-16) word count
0x407	r/w	Channel 3 High byte (bits 23-16) word count
0x4c6	r/w	Channel 5 High byte (bits 23-16) word count
0x4ca	r/w	Channel 6 High byte (bits 23-16) word count
0x4ce	r/w	Channel 7 High byte (bits 23-16) word count
0x487	r/w	Channel 0 High byte (bits 31-24) page Register
0x483	r/w	Channel 1 High byte (bits 31-24) page Register
0x481	r/w	Channel 2 High byte (bits 31-24) page Register
0x482	r/w	Channel 3 High byte (bits 31-24) page Register
0x48b	r/w	Channel 5 High byte (bits 31-24) page Register
0x489	r/w	Channel 6 High byte (bits 31-24) page Register
0x48a	r/w	Channel 6 High byte (bits 31-24) page Register
0x48f	r/w	High byte page Refresh
0x4e0	r/w	Channel 0 Stop Register (bits 7-2)
0x4e1	r/w	Channel 0 Stop Register (bits 15-8)
0x4e2	r/w	Channel 0 Stop Register (bits 23-16)
0x4e4	r/w	Channel 1 Stop Register (bits 7-2)
0x4e5	r/w	Channel 1 Stop Register (bits 15-8)
0x4e6	r/w	Channel 1 Stop Register (bits 23-16)
0x4e8	r/w	Channel 2 Stop Register (bits 7-2)
0x4e9	r/w	Channel 2 Stop Register (bits 15-8)
0x4ea	r/w	Channel 2 Stop Register (bits 23-16)
0x4ec	r/w	Channel 3 Stop Register (bits 7-2)
0x4ed	r/w	Channel 3 Stop Register (bits 15-8)

0x4ee	r/w	Channel 3 Stop Register (bits 23-16)
0x4f4	r/w	Channel 5 Stop Register (bits 7-2)
0x4f5	r/w	Channel 5 Stop Register (bits 15-8)
0x4f6	r/w	Channel 5 Stop Register (bits 23-16)
0x4f8	r/w	Channel 6 Stop Register (bits 7-2)
0x4f9	r/w	Channel 6 Stop Register (bits 15-8)
0x4fa	r/w	Channel 6 Stop Register (bits 23-16)
0x4fc	r/w	Channel 7 Stop Register (bits 7-2)
0x4fd	r/w	Channel 7 Stop Register (bits 15-8)
0x4fe	r/w	Channel 7 Stop Register (bits 23-16)
0x40a	write	Channels 0-3 Chaining Mode Register
0x40a	read	Channel Interrupt Status Register
0x4d4	write	Channels 4-7 Chaining Mode Register
0x4d4	read	Chaining Mode Status
0x40c	read	Chain Buffer Expiration Control Register
0x410	write	Channel 0 Scatter-Gather Command Register
0x411	write	Channel 1 Scatter-Gather Command Register
0x412	write	Channel 2 Scatter-Gather Command Register
0x413	write	Channel 3 Scatter-Gather Command Register
0x415	write	Channel 5 Scatter-Gather Command Register
0x416	write	Channel 6 Scatter-Gather Command Register
0x417	write	Channel 7 Scatter-Gather Command Register
0x418	read	Channel 0 Scatter-Gather Status Register
0x419	read	Channel 1 Scatter-Gather Status Register
0x41a	read	Channel 2 Scatter-Gather Status Register
0x41b	read	Channel 3 Scatter-Gather Status Register
0x41d	read	Channel 5 Scatter-Gather Status Register
0x41e	read	Channel 5 Scatter-Gather Status Register

0x41f	read	Channel 7 Scatter-Gather Status Register
0x420-0x423	r/w	Channel 0 Scatter-Gather Descriptor Table Pointer Register
0x424-0x427	r/w	Channel 1 Scatter-Gather Descriptor Table Pointer Register
0x428-0x42b	r/w	Channel 2 Scatter-Gather Descriptor Table Pointer Register
0x42c-0x42f	r/w	Channel 3 Scatter-Gather Descriptor Table Pointer Register
0x434-0x437	r/w	Channel 5 Scatter-Gather Descriptor Table Pointer Register
0x438-0x43b	r/w	Channel 6 Scatter-Gather Descriptor Table Pointer Register
0x43c-0x43f	r/w	Channel 7 Scatter-Gather Descriptor Table Pointer Register

第10章
调试内核

供稿：*Paul Richards* 和*Jörg Wunsch.*

10.1 如何将内核的崩溃转存数据保存成文件

在极端条件下(比如系统运行在非常高的负载下，数以万计的连接，过多的用户同时登录使用，成百上千的jail(8)) 运行尚有待进一步完善的系统内核(例如FreeBSD-CURRENT)，或在FreeBSD-STABLE 上使用新特性或新驱动(例如PAE) 时，、都有可能使内核崩溃。针对上述可能出现的情况，本章将讲解如何从内核崩溃中获取有价值的信息。

一旦内核崩溃，系统不可避免的要重启。而系统重启将使内存(RAM) 和交换设备上的数据荡然无存。为保存内存中的数据，内核使用交换设备临时储存崩溃前RAM 上的数据。这样做使得FreeBSD 重启后，可从中得到当时内核的镜像，从而为进一步调试提供基础。

注意: 交换设备在被配置成内核存档设备后仍可作为交换设备正常使用。目前尚不支持将其它设备(譬如磁带、CDRW等) 配置成内核崩溃时的转存设备。这里所说的“交换设备” 就是“交换分区。”

要得到可用的核心内存转存，需确保至少有一个容量足以保存保存内存中所有数据的交换分区。当内核崩溃时，内核会在系统重启前查找是否存在配置为用于内核转存的交换设备，如果有，内核就将内存中的全部内容丝毫不变地存入交换设备。

10.1.1 配置内核转存设备

只有在配置内核转存设备之后，内核才会向其写入崩溃时内存中的数据。dumpon(8) 命令告诉内核在何处保存崩溃的内核。dumpon(8) 只能在已经通过swapon(8) 配置好的交换分区上使用。通常，只需在rc.conf(5) 中把`dumpdev` 变量设为交换分区的设备路径既可(推荐用这种方法提取内核转存数据)。

另一种方法是在内核配置文件中，通过config(5) 的`dump` 语句来将转存设备硬编码进内核。这是一种过时的做法，只有当内核会在能够执行dumpon(8) 之前就会发生崩溃时才应考虑采用。

提示: 可以通过查看`/etc/fstab` 或swapinfo(8) 的输出来了解系统中现有的交换设备。

重要: 在分析内核崩溃之前，首先确认一下rc.conf(5) 所设置的`dumpdir` 确实存在！

```
# mkdir /var/crash
# chmod 700 /var/crash
```

需要注意的是，对外界而言，`/var/crash` 中保存的数据可能包含敏感信息，因为其中可能包含一些机密内容，如用户密码等等。

10.1.2 提取内核转存数据

一旦内核转存到转存设备之后，就需要在下次交换设备挂载之前，将其提取并保存到文件中。要从转存设备中提取内核转存数据，就需要使用savecore(8) 程序。如果在rc.conf(5) 中配置了dumpdev，则崩溃之后的首次多用户方式启动过程中，在配置交换区设备之前便会自动执行savecore(8)。提取出来的内核数据将存放在rc.conf(5) 变量dumpdir 所指定的位置，其默认值为/var/crash，而保存的文件名则是vmcore.0。

若/var/crash 目录下(或dumpdev设置的其它目录)，已存在了名为vmcore.0 的文件，则每次保存内核转存数据时，其末尾的数字会顺次递增(例如vmcore.1) 以避免覆盖之前存档的转存数据。所以，调试内核时，/var/crash 目录下序号最大的vmcore 通常就是希望找的那个vmcore。

提示: 如果正在调试新的内核，但需要从另一能支持系统正常运行的内核启动，就应在屏幕出现启动提示时，使用-s 选项进入单用户模式，再按下列步骤操作:

```
# fsck -p
# mount -a -t ufs       # make sure /var/crash is writable
# savecore /var/crash /dev/ad0s1b
# exit                  # exit to multi-user
```

This instructs savecore(8) to extract a kernel dump from /dev/ad0s1b and place the contents in /var/crash. Do not forget to make sure the destination directory /var/crash has enough space for the dump. Also, do not forget to specify the correct path to your swap device as it is likely different than /dev/ad0s1b!

The recommended, and certainly the easiest way to automate obtaining crash dumps is to use the dumpdev variable in rc.conf(5).

10.2 Debugging a Kernel Crash Dump with kgdb

注意: This section covers kgdb(1) as found in FreeBSD 5.3 and later. In previous versions, one must use gdb -k to read a core dump file.

Once a dump has been obtained, getting useful information out of the dump is relatively easy for simple problems. Before launching into the internals of kgdb(1) to debug the crash dump, locate the debug version of your kernel (normally called kernel.debug) and the path to the source files used to build your kernel (normally /usr/obj/usr/src/sys/*KERNCONF*, where *KERNCONF* is the ident specified in a kernel config(5)). With those two pieces of info, let the debugging commence!

To enter into the debugger and begin getting information from the dump, the following steps are required at a minimum:

```
# cd /usr/obj/usr/src/sys/KERNCONF
# kgdb kernel.debug /var/crash/vmcore.0
```

You can debug the crash dump using the kernel sources just like you can for any other program.

This first dump is from a 5.2-BETA kernel and the crash comes from deep within the kernel. The output below has been modified to include line numbers on the left. This first trace inspects the instruction pointer and obtains a back trace. The address that is used on line 41 for the list command is the instruction pointer and can be found on

line line 17. Most developers will request having at least this information sent to them if you are unable to debug the problem yourself. If, however, you do solve the problem, make sure that your patch winds its way into the source tree via a problem report, mailing lists, or by being able to commit it!

```
 1:# cd /usr/obj/usr/src/sys/KERNCONF
 2:# kgdb kernel.debug /var/crash/vmcore.0
 3:GNU gdb 5.2.1 (FreeBSD)
 4:Copyright 2002 Free Software Foundation, Inc.
 5:GDB is free software, covered by the GNU General Public License, and you are
 6:welcome to change it and/or distribute copies of it under certain conditions.
 7:Type "show copying" to see the conditions.
 8:There is absolutely no warranty for GDB.  Type "show warranty" for details.
 9:This GDB was configured as "i386-undermydesk-freebsd"...
10:panic: page fault
11:panic messages:
12:---
13:Fatal trap 12: page fault while in kernel mode
14:cpuid = 0; apic id = 00
15:fault virtual address   = 0x300
16:fault code:             = supervisor read, page not present
17:instruction pointer     = 0x8:0xc0713860
18:stack pointer           = 0x10:0xdc1d0b70
19:frame pointer           = 0x10:0xdc1d0b7c
20:code segment            = base 0x0, limit 0xfffff, type 0x1b
21:                        = DPL 0, pres 1, def32 1, gran 1
22:processor eflags        = resume, IOPL = 0
23:current process         = 14394 (uname)
24:trap number             = 12
25:panic: page fault
26      cpuid = 0;
27:Stack backtrace:
28
29:syncing disks, buffers remaining... 2199 2199 panic: mi_switch: switch in a critical section
30:cpuid = 0;
31:Uptime: 2h43m19s
32:Dumping 255 MB
33: 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240
34:---
35:Reading symbols from /boot/kernel/snd_maestro3.ko...done.
36:Loaded symbols for /boot/kernel/snd_maestro3.ko
37:Reading symbols from /boot/kernel/snd_pcm.ko...done.
38:Loaded symbols for /boot/kernel/snd_pcm.ko
39:#0  doadump () at /usr/src/sys/kern/kern_shutdown.c:240
40:240             dumping++;
41:(kgdb) list *0xc0713860
42:0xc0713860 is in lapic_ipi_wait (/usr/src/sys/i386/i386/local_apic.c:663).
43:658                     incr = 0;
44:659                     delay = 1;
45:660             } else
46:661                     incr = 1;
47:662             for (x = 0; x < delay; x += incr) {
48:663                     if ((lapic->icr_lo & APIC_DELSTAT_MASK) == APIC_DELSTAT_IDLE)
49:664                             return (1);
```

```
50:665                      ia32_pause();
51:666              }
52:667              return (0);
53:(kgdb) backtrace
54:#0  doadump () at /usr/src/sys/kern/kern_shutdown.c:240
55:#1  0xc055fd9b in boot (howto=260) at /usr/src/sys/kern/kern_shutdown.c:372
56:#2  0xc056019d in panic () at /usr/src/sys/kern/kern_shutdown.c:550
57:#3  0xc0567ef5 in mi_switch () at /usr/src/sys/kern/kern_synch.c:470
58:#4  0xc055fa87 in boot (howto=256) at /usr/src/sys/kern/kern_shutdown.c:312
59:#5  0xc056019d in panic () at /usr/src/sys/kern/kern_shutdown.c:550
60:#6  0xc0720c66 in trap_fatal (frame=0xdc1d0b30, eva=0)
61:    at /usr/src/sys/i386/i386/trap.c:821
62:#7  0xc07202b3 in trap (frame=
63:      {tf_fs = -1065484264, tf_es = -1065484272, tf_ds = -1065484272, tf_edi = 1, tf_esi = 0, tf_
64:    at /usr/src/sys/i386/i386/trap.c:250
65:#8  0xc070c9f8 in calltrap () at {standard input}:94
66:#9  0xc07139f3 in lapic_ipi_vectored (vector=0, dest=0)
67:    at /usr/src/sys/i386/i386/local_apic.c:733
68:#10 0xc0718b23 in ipi_selected (cpus=1, ipi=1)
69:    at /usr/src/sys/i386/i386/mp_machdep.c:1115
70:#11 0xc057473e in kseq_notify (ke=0xcc05e360, cpu=0)
71:    at /usr/src/sys/kern/sched_ule.c:520
72:#12 0xc0575cad in sched_add (td=0xcbcf5c80)
73:    at /usr/src/sys/kern/sched_ule.c:1366
74:#13 0xc05666c6 in setrunqueue (td=0xcc05e360)
75:    at /usr/src/sys/kern/kern_switch.c:422
76:#14 0xc05752f4 in sched_wakeup (td=0xcbcf5c80)
77:    at /usr/src/sys/kern/sched_ule.c:999
78:#15 0xc056816c in setrunnable (td=0xcbcf5c80)
79:    at /usr/src/sys/kern/kern_synch.c:570
80:#16 0xc0567d53 in wakeup (ident=0xcbcf5c80)
81:    at /usr/src/sys/kern/kern_synch.c:411
82:#17 0xc05490a8 in exit1 (td=0xcbcf5b40, rv=0)
83:    at /usr/src/sys/kern/kern_exit.c:509
84:#18 0xc0548011 in sys_exit () at /usr/src/sys/kern/kern_exit.c:102
85:#19 0xc0720fd0 in syscall (frame=
86:      {tf_fs = 47, tf_es = 47, tf_ds = 47, tf_edi = 0, tf_esi = -1, tf_ebp = -1077940712, tf_isp
87:    at /usr/src/sys/i386/i386/trap.c:1010
88:#20 0xc070ca4d in Xint0x80_syscall () at {standard input}:136
89:---Can't read userspace from dump, or kernel process---
90:(kgdb) quit
```

This next trace is an older dump from the FreeBSD 2 time frame, but is more involved and demonstrates more of the features of `gdb`. Long lines have been folded to improve readability, and the lines are numbered for reference. Despite this, it is a real-world error trace taken during the development of the pcvt console driver.

```
 1:Script started on Fri Dec 30 23:15:22 1994
 2:# cd /sys/compile/URIAH
 3:# gdb -k kernel /var/crash/vmcore.1
 4:Reading symbol data from /usr/src/sys/compile/URIAH/kernel
...done.
 5:IdlePTD 1f3000
 6:panic: because you said to!
```

```
 7:current pcb at 1e3f70
 8:Reading in symbols for ../../i386/i386/machdep.c...done.
 9:(kgdb) backtrace
10:#0  boot (arghowto=256) (../../i386/i386/machdep.c line 767)
11:#1  0xf0115159 in panic ()
12:#2  0xf01955bd in diediedie () (../../i386/i386/machdep.c line 698)
13:#3  0xf010185e in db_fncall ()
14:#4  0xf0101586 in db_command (-266509132, -266509516, -267381073)
15:#5  0xf0101711 in db_command_loop ()
16:#6  0xf01040a0 in db_trap ()
17:#7  0xf0192976 in kdb_trap (12, 0, -272630436, -266743723)
18:#8  0xf019d2eb in trap_fatal (...)
19:#9  0xf019ce60 in trap_pfault (...)
20:#10 0xf019cb2f in trap (...)
21:#11 0xf01932a1 in exception:calltrap ()
22:#12 0xf0191503 in cnopen (...)
23:#13 0xf0132c34 in spec_open ()
24:#14 0xf012d014 in vn_open ()
25:#15 0xf012a183 in open ()
26:#16 0xf019d4eb in syscall (...)
27:(kgdb) up 10
28:Reading in symbols for ../../i386/i386/trap.c...done.
29:#10 0xf019cb2f in trap (frame={tf_es = -260440048, tf_ds = 16, tf_\
30:edi = 3072, tf_esi = -266445372, tf_ebp = -272630356, tf_isp = -27\
31:2630396, tf_ebx = -266427884, tf_edx = 12, tf_ecx = -266427884, tf\
32:_eax = 64772224, tf_trapno = 12, tf_err = -272695296, tf_eip = -26\
33:6672343, tf_cs = -266469368, tf_eflags = 66066, tf_esp = 3072, tf_\
34:ss = -266427884}) (../../i386/i386/trap.c line 283)
35:283                              (void) trap_pfault(&frame, FALSE);
36:(kgdb) frame frame->tf_ebp frame->tf_eip
37:Reading in symbols for ../../i386/isa/pcvt/pcvt_drv.c...done.
38:#0  0xf01ae729 in pcopen (dev=3072, flag=3, mode=8192, p=(struct p\
39:roc *) 0xf07c0c00) (../../i386/isa/pcvt/pcvt_drv.c line 403)
40:403             return ((*linesw[tp->t_line].l_open)(dev, tp));
41:(kgdb) list
42:398
43:399             tp->t_state |= TS_CARR_ON;
44:400             tp->t_cflag |= CLOCAL;  /* cannot be a modem (:-) */
45:401
46:402     #if PCVT_NETBSD || (PCVT_FREEBSD >= 200)
47:403             return ((*linesw[tp->t_line].l_open)(dev, tp));
48:404     #else
49:405             return ((*linesw[tp->t_line].l_open)(dev, tp, flag));
50:406     #endif /* PCVT_NETBSD || (PCVT_FREEBSD >= 200) */
51:407     }
52:(kgdb) print tp
53:Reading in symbols for ../../i386/i386/cons.c...done.
54:$1 = (struct tty *) 0x1bae
55:(kgdb) print tp->t_line
56:$2 = 1767990816
57:(kgdb) up
58:#1  0xf0191503 in cnopen (dev=0x00000000, flag=3, mode=8192, p=(st\
59:ruct proc *) 0xf07c0c00) (../../i386/i386/cons.c line 126)
```

```
60:       return ((*cdevsw[major(dev)].d_open)(dev, flag, mode, p));
61:(kgdb) up
62:#2  0xf0132c34 in spec_open ()
63:(kgdb) up
64:#3  0xf012d014 in vn_open ()
65:(kgdb) up
66:#4  0xf012a183 in open ()
67:(kgdb) up
68:#5  0xf019d4eb in syscall (frame={tf_es = 39, tf_ds = 39, tf_edi =\
69: 2158592, tf_esi = 0, tf_ebp = -272638436, tf_isp = -272629788, tf\
70:_ebx = 7086, tf_edx = 1, tf_ecx = 0, tf_eax = 5, tf_trapno = 582, \
71:tf_err = 582, tf_eip = 75749, tf_cs = 31, tf_eflags = 582, tf_esp \
72:= -272638456, tf_ss = 39}) (../../i386/i386/trap.c line 673)
73:673             error = (*callp->sy_call)(p, args, rval);
74:(kgdb) up
75:Initial frame selected; you cannot go up.
76:(kgdb) quit
```

Comments to the above script:

line 6:

This is a dump taken from within DDB (see below), hence the panic comment “because you said to!”, and a rather long stack trace; the initial reason for going into DDB has been a page fault trap though.

line 20:

This is the location of function `trap()` in the stack trace.

line 36:

Force usage of a new stack frame; this is no longer necessary. The stack frames are supposed to point to the right locations now, even in case of a trap. From looking at the code in source line 403, there is a high probability that either the pointer access for “tp” was messed up, or the array access was out of bounds.

line 52:

The pointer looks suspicious, but happens to be a valid address.

line 56:

However, it obviously points to garbage, so we have found our error! (For those unfamiliar with that particular piece of code: `tp->t_line` refers to the line discipline of the console device here, which must be a rather small integer number.)

提示: If your system is crashing regularly and you are running out of disk space, deleting old `vmcore` files in `/var/crash` could save a considerable amount of disk space!

10.3 Debugging a Crash Dump with DDD

Examining a kernel crash dump with a graphical debugger like `ddd` is also possible (you will need to install the `devel/ddd` port in order to use the `ddd` debugger). Add the `-k` option to the `ddd` command line you would use normally. For example;

```
# ddd -k /var/crash/kernel.0 /var/crash/vmcore.0
```

You should then be able to go about looking at the crash dump using `ddd`'s graphical interface.

10.4 Post-Mortem Analysis of a Dump

What do you do if a kernel dumped core but you did not expect it, and it is therefore not compiled using `config -g`? Not everything is lost here. Do not panic!

Of course, you still need to enable crash dumps. See above for the options you have to specify in order to do this.

Go to your kernel config directory (`/usr/src/sys/`*`arch`*`/conf`) and edit your configuration file. Uncomment (or add, if it does not exist) the following line:

```
makeoptions    DEBUG=-g                #Build kernel with gdb(1) debug symbols
```

Rebuild the kernel. Due to the time stamp change on the Makefile, some other object files will be rebuilt, for example `trap.o`. With a bit of luck, the added `-g` option will not change anything for the generated code, so you will finally get a new kernel with similar code to the faulting one but with some debugging symbols. You should at least verify the old and new sizes with the size(1) command. If there is a mismatch, you probably need to give up here.

Go and examine the dump as described above. The debugging symbols might be incomplete for some places, as can be seen in the stack trace in the example above where some functions are displayed without line numbers and argument lists. If you need more debugging symbols, remove the appropriate object files, recompile the kernel again and repeat the `gdb -k` session until you know enough.

All this is not guaranteed to work, but it will do it fine in most cases.

10.5 On-Line Kernel Debugging Using DDB

While `gdb -k` as an off-line debugger provides a very high level of user interface, there are some things it cannot do. The most important ones being breakpointing and single-stepping kernel code.

If you need to do low-level debugging on your kernel, there is an on-line debugger available called DDB. It allows setting of breakpoints, single-stepping kernel functions, examining and changing kernel variables, etc. However, it cannot access kernel source files, and only has access to the global and static symbols, not to the full debug information like `gdb` does.

To configure your kernel to include DDB, add the option line

```
options DDB
```

to your config file, and rebuild. (See The FreeBSD Handbook (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/handbook/index.html) for details on configuring the FreeBSD kernel).

注意: If you have an older version of the boot blocks, your debugger symbols might not be loaded at all. Update the boot blocks; the recent ones load the DDB symbols automatically.

Once your DDB kernel is running, there are several ways to enter DDB. The first, and earliest way is to type the boot flag `-d` right at the boot prompt. The kernel will start up in debug mode and enter DDB prior to any device probing. Hence you can even debug the device probe/attach functions.

The second scenario is to drop to the debugger once the system has booted. There are two simple ways to accomplish this. If you would like to break to the debugger from the command prompt, simply type the command:

```
# sysctl debug.enter_debugger=ddb
```

Alternatively, if you are at the system console, you may use a hot-key on the keyboard. The default break-to-debugger sequence is **Ctrl+Alt+ESC**. For syscons, this sequence can be remapped and some of the distributed maps out there do this, so check to make sure you know the right sequence to use. There is an option available for serial consoles that allows the use of a serial line BREAK on the console line to enter DDB (`options BREAK_TO_DEBUGGER` in the kernel config file). It is not the default since there are a lot of serial adapters around that gratuitously generate a BREAK condition, for example when pulling the cable.

The third way is that any panic condition will branch to DDB if the kernel is configured to use it. For this reason, it is not wise to configure a kernel with DDB for a machine running unattended.

The DDB commands roughly resemble some `gdb` commands. The first thing you probably need to do is to set a breakpoint:

```
b function-name
b address
```

Numbers are taken hexadecimal by default, but to make them distinct from symbol names; hexadecimal numbers starting with the letters `a-f` need to be preceded with `0x` (this is optional for other numbers). Simple expressions are allowed, for example: `function-name + 0x103`.

To continue the operation of an interrupted kernel, simply type:

```
c
```

To get a stack trace, use:

```
trace
```

注意: Note that when entering DDB via a hot-key, the kernel is currently servicing an interrupt, so the stack trace might be not of much use to you.

If you want to remove a breakpoint, use

```
del
del address-expression
```

The first form will be accepted immediately after a breakpoint hit, and deletes the current breakpoint. The second form can remove any breakpoint, but you need to specify the exact address; this can be obtained from:

```
show b
```

To single-step the kernel, try:

```
s
```

This will step into functions, but you can make DDB trace them until the matching return statement is reached by:

```
n
```

注意: This is different from `gdb`'s `next` statement; it is like `gdb`'s `finish`.

To examine data from memory, use (for example):

```
x/wx 0xf0133fe0,40
x/hd db_symtab_space
x/bc termbuf,10
x/s stringbuf
```

for word/halfword/byte access, and hexadecimal/decimal/character/ string display. The number after the comma is the object count. To display the next 0x10 items, simply use:

```
x ,10
```

Similarly, use

```
x/ia foofunc,10
```

to disassemble the first 0x10 instructions of `foofunc`, and display them along with their offset from the beginning of `foofunc`.

To modify memory, use the write command:

```
w/b termbuf 0xa 0xb 0
w/w 0xf0010030 0 0
```

The command modifier (`b`/`h`/`w`) specifies the size of the data to be written, the first following expression is the address to write to and the remainder is interpreted as data to write to successive memory locations.

If you need to know the current registers, use:

```
show reg
```

Alternatively, you can display a single register value by e.g.

```
p $eax
```

and modify it by:

```
set $eax new-value
```

Should you need to call some kernel functions from DDB, simply say:

```
call func(arg1, arg2, ...)
```

The return value will be printed.

For a ps(1) style summary of all running processes, use:

```
ps
```

Now you have examined why your kernel failed, and you wish to reboot. Remember that, depending on the severity of previous malfunctioning, not all parts of the kernel might still be working as expected. Perform one of the following actions to shut down and reboot your system:

```
panic
```

This will cause your kernel to dump core and reboot, so you can later analyze the core on a higher level with `gdb`. This command usually must be followed by another `continue` statement.

```
call boot(0)
```

Which might be a good way to cleanly shut down the running system, `sync()` all disks, and finally reboot. As long as the disk and filesystem interfaces of the kernel are not damaged, this might be a good way for an almost clean shutdown.

```
call cpu_reset()
```

This is the final way out of disaster and almost the same as hitting the Big Red Button.

If you need a short command summary, simply type:

```
help
```

However, it is highly recommended to have a printed copy of the ddb(4) manual page ready for a debugging session. Remember that it is hard to read the on-line manual while single-stepping the kernel.

10.6 On-Line Kernel Debugging Using Remote GDB

This feature has been supported since FreeBSD 2.2, and it is actually a very neat one.

GDB has already supported *remote debugging* for a long time. This is done using a very simple protocol along a serial line. Unlike the other methods described above, you will need two machines for doing this. One is the host providing the debugging environment, including all the sources, and a copy of the kernel binary with all the symbols in it, and the other one is the target machine that simply runs a similar copy of the very same kernel (but stripped of the debugging information).

You should configure the kernel in question with `config -g`, include `DDB` into the configuration, and compile it as usual. This gives a large binary, due to the debugging information. Copy this kernel to the target machine, strip the debugging symbols off with `strip -x`, and boot it using the `-d` boot option. Connect the serial line of the target machine that has "flags 080" set on its sio device to any serial line of the debugging host. Now, on the debugging machine, go to the compile directory of the target kernel, and start `gdb`:

```
% gdb -k kernel
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i386-unknown-freebsd),
Copyright 1996 Free Software Foundation, Inc...
(kgdb)
```

Initialize the remote debugging session (assuming the first serial port is being used) by:

```
(kgdb) target remote /dev/cuaa0
```

Now, on the target host (the one that entered DDB right before even starting the device probe), type:

```
Debugger("Boot flags requested debugger")
Stopped at Debugger+0x35: movb $0, edata+0x51bc
db> gdb
```

DDB will respond with:

```
Next trap will enter GDB remote protocol mode
```

Every time you type `gdb`, the mode will be toggled between remote GDB and local DDB. In order to force a next trap immediately, simply type `s` (step). Your hosting GDB will now gain control over the target kernel:

```
Remote debugging using /dev/cuaa0
Debugger (msg=0xf01b0383 "Boot flags requested debugger")
    at ../../i386/i386/db_interface.c:257
(kgdb)
```

You can use this session almost as any other GDB session, including full access to the source, running it in gud-mode inside an Emacs window (which gives you an automatic source code display in another Emacs window), etc.

10.7 Debugging Loadable Modules Using GDB

When debugging a panic that occurred within a module, or using remote GDB against a machine that uses dynamic modules, you need to tell GDB how to obtain symbol information for those modules.

First, you need to build the module(s) with debugging information:

```
# cd /sys/modules/linux
# make clean; make COPTS=-g
```

If you are using remote GDB, you can run `kldstat` on the target machine to find out where the module was loaded:

```
# kldstat
Id Refs Address    Size     Name
 1    4 0xc0100000 1c1678   kernel
 2    1 0xc0a9e000 6000     linprocfs.ko
 3    1 0xc0ad7000 2000     warp_saver.ko
 4    1 0xc0adc000 11000    linux.ko
```

If you are debugging a crash dump, you will need to walk the `linker_files` list, starting at `linker_files->tqh_first` and following the `link.tqe_next` pointers until you find the entry with the `filename` you are looking for. The `address` member of that entry is the load address of the module.

Next, you need to find out the offset of the text section within the module:

```
# objdump --section-headers /sys/modules/linux/linux.ko | grep text
  3 .rel.text     000016e0  000038e0  000038e0  000038e0  2**2
 10 .text         00007f34  000062d0  000062d0  000062d0  2**2
```

The one you want is the `.text` section, section 10 in the above example. The fourth hexadecimal field (sixth field overall) is the offset of the text section within the file. Add this offset to the load address of the module to obtain the relocation address for the module's code. In our example, we get 0xc0adc000 + 0x62d0 = 0xc0ae22d0. Use the `add-symbol-file` command in GDB to tell the debugger about the module:

```
(kgdb) add-symbol-file /sys/modules/linux/linux.ko 0xc0ae22d0
add symbol table from file "/sys/modules/linux/linux.ko" at text_addr = 0xc0ae22d0?
(y or n) y
Reading symbols from /sys/modules/linux/linux.ko...done.
(kgdb)
```

You should now have access to all the symbols in the module.

10.8 Debugging a Console Driver

Since you need a console driver to run DDB on, things are more complicated if the console driver itself is failing. You might remember the use of a serial console (either with modified boot blocks, or by specifying `-h` at the `Boot:` prompt), and hook up a standard terminal onto your first serial port. DDB works on any configured console driver, including a serial console.

IV. 系统结构

第11章
x86 汇编语言

本章节由*G. Adam Stanislav* <adam@redprince.net> 撰写。翻译：*sunlecn@gmail.com*。

11.1 概述

涉及UNIX 下的汇编语言的相关资料很少。通常我们都假设几乎没有人希望用到它，因为不同的UNIX 运行在不同的处理器上，所以为了移植性的考虑所有的东西都应该用C 来实现。

实际上，C 语言的移植性是非常神秘的。即使C 程序在不同的UNIX 之间移植的时候需要进行修改，但这些修改绝对和在什么处理器上运行无关。特别明显地是，这样的程序充满了在编译时对于系统依赖的条件描述。

即使我们希望所有的UNIX 软件都应该用C 语言或者是其他的高级语言编写，但是我们仍然需要掌握汇编语言的程序员：不然谁将完成C 语言函数库中直接访问内核的部分?

在本章节，我将尝试着说明如果用汇编语言编写UNIX 程序，特别是在FreeBSD 上的程序。

这个章节不会讲解汇编语言的基础知识。现在已经有很多相关的资料了（如果你想寻找完整的关于汇编语言的在线课程，请看Randall Hyde 的《汇编语言艺术》 (http://webster.cs.ucr.edu/)；如果你想选择一本书，那么去看看Jeff Duntemann 的《循序渐进学汇编》 (http://www.int80h.org/cgi-bin/isbn?isbn=0471375233)）不过，一旦你掌握了这个章节的内容，任何汇编语言的程序员，将能够在FreeBSD 上高效、快速地编写程序。

Copyright © 2000-2001 G. Adam Stanislav. All rights reserved.

11.2 工具

11.2.1 汇编器

汇编语言编程最重要的工具是汇编器，它将汇编语言代码转换成机器语言。

在FreeBSD 中有两个完全不同的汇编器。一个是as(1), 使用传统的UNIX汇编语法，它是随系统发布的。

另外一个是**/usr/ports/devel/nasm**。它使用Intel 的语法规范，其最大的好处是可以在许多操作系统上汇编代码。它需要你单独安装，不过它是完全免费的。

本章节使用**nasm** 的语法规范，这是因为许多从其他操作系统移植到FreeBSD 的汇编语言可以更好地被理解。不过，坦白地说，这是因为我更习惯这样的语法规范。

11.2.2 链接器

像任何编译器一样，汇编器的输出需要被链接后才能成为可执行文件。

标准的链接器ld(1) 随FreeBSD 发布。它可以和以上任何一个汇编器一起完成代码组装。

11.3 系统调用

11.3.1 默认的调用规范

通常，FreeBSD 的内核使用C 语言的调用规范。此外，虽然我们使用`int 80h` 来访问内核，但是我们常常通过调用一个函数来执行`int 80h`，而不是直接访问。

这个规范是非常方便的，比Microsoft 的MS-DOS 上使用的规范更加优越。为什么呢？因为UNIX 的规范允许任何语言所写的程序访问内核。

汇编语言也可以这样做，比如，我们可以编写一个文件：

```
kernel:
int 80h ; Call kernel
ret

open:
push dword mode
push dword flags
push dword path
mov eax, 5
call kernel
add esp, byte 12
ret
```

这是一种非常清晰而易于移植的编码方法。如果你需要将代码移植到一个使用和FreeBSD 完全不同中断或参数传递方式的UNIX 系统，那么你所需要做的仅仅是改变那一段内核程序。

但是汇编程序员喜欢使用一些技巧来削减程序执行所需的时钟周期数。以上的例子需要一个`call/ret` 组合，我们可以通过压栈`push` 一个额外的双字节来去除它。

```
open:
push dword mode
push dword flags
push dword path
mov eax, 5
push eax ; Or any other dword
int 80h
add esp, byte 16
```

在上一个`open` 的例子中，我们放在`EAX` 中的数字`5` 表示了一个内核函数。

11.3.2 另一种调用规范

FreeBSD 是一个非常灵活的系统。它提供了访问内核的其他方式。但是，如果你要用到它，你的系统必须安装Linux emulation。

Linux 是一个类UNIX 操作系统。但是，它的内核在传递参数的时候，使用和MS-DOS 相同系统调用规范。比如在UNIX 的规范中，代表内核函数的数字存放在`EAX` 中。但是在Linux 中，参数不进行压栈而是存放在`EBX, ECX, EDX, ESI, EDI, EBP`：

```
open:
mov eax, 5
```

```
mov ebx, path
mov ecx, flags
mov edx, mode
int 80h
```

这种做法与UNIX 的常规做法相比有一个严重的缺点，至少现在汇编语言程序需要注意的：每次进行系统调用，你必须对寄存器的值进行压栈和出栈操作，这让你的程序变得冗长而效率低下。因此，FreeBSD 提供给你了一个其他的选择。

如果你确实选择了Linux 的规范，你必要让你的系统知道这一点。在你的程序完成汇编和连接之后，你需要对可执行文件进行标识：

```
% brandelf -f Linux filename
```

11.3.3 你该选择哪个规范?

如果你专门为FreeBSD 写代码，你必须使用UNIX 的规范：因为它快速，你可以在寄存器中存取全局变量，而且不需要进行可执行的标识，更不需要在你的系统上安装Linux emulation。

如果你想做一些可以在Linux 上运行的程序，并且你打算尽可能地给FreeBSD 用户提供最有效率地程序。我将在我讲解完汇编基础之后，告诉你怎样完成这样地程序。

11.3.4 系统调用号

要通知内核你调用了什么系统服务，将代表调用的数字放入EAX 中。当然，你需要知道那些系统调用号代表了什么。

11.3.4.1 syscalls 文件

系统调用号列在了syscalls 文件里面。locate syscalls 去寻找这个文件的不同格式，它们都是从syscalls.master 中自动生成的。

你可以在/usr/src/sys/kern/syscalls.master 下找到这个描述默认情况下UNIX 系统调用规范的文件。如果你需要使用在Linux emulation 中使用的规范，请参阅/usr/src/sys/i386/linux/syscalls.master。

注意: FreeBSD 和Linux 不仅仅是使用不同的调用规范，有时候它们也使用不同的系统调用号来表示相同函数。

syscalls.master 描述了如何使用这些调用：

```
0 STD NOHIDE { int nosys(void); } syscall nosys_args int
1 STD NOHIDE { void exit(int rval); } exit rexit_args void
2 STD POSIX { int fork(void); }
3 STD POSIX { ssize_t read(int fd, void *buf, size_t nbyte); }
4 STD POSIX { ssize_t write(int fd, const void *buf, size_t nbyte); }
5 STD POSIX { int open(char *path, int flags, int mode); }
6 STD POSIX { int close(int fd); }
etc...
```

最左边的一列告诉我们需要放入EAX 的数字。

最右边的一列告诉我们那些需要被压栈push 的参数，它们的压栈push 顺序是从右到左。

例如，要打开open 一个文件，我们需要先首先压栈push 模式字mode，然后是压栈标志字flags，然后是压栈保存路径path 的地址。

11.4 返回值

如果系统调用没有返回一些数值，那么在很多情况下并没有太多用处。这些返回值包括：一个打开的文件的文件描述符、一个从缓存里读取的字节，或者系统时间等等。

此外，如果错误出现，系统需要通知我们。这些错误包括：文件不存在、系统资源耗尽，或者我们传递了一个错误的参数等等。

11.4.1 联机手册

传统情况下，寻找UNIX 下不同系统调用的地方是手册页。FreeBSD 会在手册页的第2节描述系统调用，有的时候会在第三节。

例如，open(2) 所描述的:

> 如果执行成功，open() 将返回一个表示文件描述符的非负整数。如果执行失败，它会返回-1 并且设置errno 来标识错误。

刚刚接触UNIX 和FreeBSD的汇编程序员或许会马上提出一个难题：那些errno 在哪里？我怎么才能找得到它们?

注意: 这些在手册页中展示的信息仅针对C 语言。汇编程序员则需要更多的信息。

11.4.2 返回值在哪里?

很不幸，它有依赖性。。。大部分系统调用都使用EAX 但不是所有的系统调用都回这样。不过当第一次使用系统调用的时候，直接在EAX 检查返回值是个不错的查找方法。如果它不在那里，你再需要深入调查。

注意: 我知道一个SYS_fork 的系统调用将返回值存放在EDX 中。而我曾经用过的其他调用都使用EAX 存放返回值，但是我目前还没有用过所有的系统调用。

提示: 如果你哪里都找不到答案，去学习**libc** 代码吧，去看看它如何访问内核的。

11.4.3 哪里可以找到errno?

实际上，没什么地方。。。

errno 不属于UNIX 内核，而是C 语言中的一部分。当直接访问内核服务的时候，错误代码返回到EAX 中，也就是那个通常用来保存返回值得寄存器。

这个处理方式非常有意义。如果这里没有错误，这里就没有错误代码。如果这里有错误，这里就没有返回值。使用一个寄存器可以涵盖以上任意一种情况。

11.4.4 判断错误的出现

当使用标准的FreeBSD 调用规范时，标志位carry flag 在程序运行成功时被清零，在失败时被置位。

当使用Linux emulation 模式的时候，EAX 中的有符号数反映了执行结果。当为非负数时，表示执行成功，并包含了返回值。如果失败，返回值即为负数，如-errno。

11.5 建立可移植的代码

一般说来，可移植性并非汇编语言的长项。然而，写出能够在不同平台上执行的汇编代码仍然是可能的事情，特别是在使用**nasm** 的时候。我曾经写过一个汇编语言函数库，可以在Windows 或FreeBSD 这样不同的操作系统下进行汇编。

所以，让你的代码在两种不同但是又基于相似的结构的平台上运行是完全可能的。

比如，FreeBSD 是UNIX 操作系统，Linux 是类UNIX 操作系统。从一个汇编语言程序员的观点来看，我只说明三个两者不同的地方：调用方式，功能号，以及返回值的传递方式。

11.5.1 功能号的处理

许多情况下，两个平台下的功能号是相同的。当然，即使它们不一样的时候，问题也一样容易解决。方法很简单，就是在代码中用常量替代数字，这样你可以根据不同的系统结构进行不同的声明：

```
%ifdef LINUX
%define SYS_execve 11
%else
%define SYS_execve 59
%endif
```

11.5.2 编程规范的处理

调用规范和返回值（errno 相关的问题）可以通过使用宏一起得到解决：

```
%ifdef LINUX

%macro system 0
call kernel
%endmacro

align 4
```

```
kernel:
push ebx
push ecx
push edx
push esi
push edi
push ebp

mov ebx, [esp+32]
mov ecx, [esp+36]
mov edx, [esp+40]
mov esi, [esp+44]
mov ebp, [esp+48]
int 80h

pop ebp
pop edi
pop esi
pop edx
pop ecx
pop ebx

or eax, eax
js .errno
clc
ret

.errno:
neg eax
stc
ret

%else

%macro system 0
int 80h
%endmacro

%endif
```

11.5.3 与移植相关的其他问题处理

以上的方法可以解决FreeBSD 和Linux 之间的代码移植过程。尽管如此，一些内核服务之间的差异还是很大。

如果那样的话，你需要针对那些特殊的调用编写代码，并且使用针对环境的条件汇编。不过，幸运的是，你的代码所进行地大部分工作不是在调用内核，所以通常情况下，你只需要在你的代码中增加一些针对环境的条件片段就可以了。

11.5.4 使用函数库

你可以通过为系统调用编写函数库来完全避免你主程序中的移植性问题。所以，为FreeBSD 建议一个独立的函数库吧，为Linux 建立另外的一个，再为其他的操作系统建立这样的函数库。

在你的函数库中，为每一个系统调用编写独立的函数。（如果你习惯于传统的汇编语言术语，我们也可以称之为程序）使用C 语言传递参数的方式，但是依然使用EAX 来传递功能号。如果那样，你的FreeBSD 函数库将非常简单，因为许多看似不同的函数，实际上都只是同一段代码的不同标签：

```
sys.open:
sys.close:
[etc...]
int 80h
ret
```

你的Linux 函数库需要更多彼此不同的函数。然而，尽管如此，仍可以将系统调用按其参数的个数进行分组。

```
sys.exit:
sys.close:
[etc... one-parameter functions]
push ebx
mov ebx, [esp+12]
int 80h
pop ebx
jmp sys.return

...

sys.return:
or eax, eax
js sys.err
clc
ret

sys.err:
neg eax
stc
ret
```

使用函数库的方法起初会看起来很不方便，因为它需要你去建立一个你程序所依赖地独立的文件。但是它又有很多优点：首先，你只需要编写一次，就可以在你所有的程序中使用。甚至你可以让其他的汇编程序员或者其他程序使用。不过，使用函数库的最大的好处在于：仅仅需要增加一个新的函数库，你的程序就可以被任何人移植到其他系统上了。

如果你对使用函数库没有任何概念，你至少可以将你所有的系统调用放置在一个独立的汇编语言文件中，然后将它和你的主程序连接。这里，再次强调，所有的移植程序的程序员所做的，就是建立一个新的对象文件，然后连接到你的主程序中。

11.5.5 使用头文件

如果你发布的软件中包含代码，你可以在你包含代码的地方使用宏，将它们放置在一个独立的文件中。

移植你软件的人只需要简单地写一个头文件，不需要外部的对象文件。你的程序将不加修改地被移植。

注意: 这个就是我们将在本章中使用的方法。我们将把我们的头文件命名为`system.inc`，然后在我们使用新的系统调用时增加它。

我们从声明标准文件描述符开始，编写我们的`system.inc`：

```
%define stdin 0
%define stdout 1
%define stderr 2
```

接下来，为每个系统调用指定符号名：

```
%define SYS_nosys 0
%define SYS_exit 1
%define SYS_fork 2
%define SYS_read 3
%define SYS_write 4
; [etc...]
```

接下来增加一个短小的非全局子程序，并给它其一个够长的名字，以避免我们不慎在代码中使用同样的名字。

```
section .text
align 4
access.the.bsd.kernel:
int 80h
ret
```

我们建立了一个带有一个参数的宏，其系统调用号为：

```
%macro system 1
mov eax, %1
call access.the.bsd.kernel
%endmacro
```

最后，我们为每一个系统调用建立了一个宏。这些宏不带有任何参数。

```
%macro sys.exit 0
system SYS_exit
%endmacro

%macro sys.fork 0
system SYS_fork
%endmacro

%macro sys.read 0
```

```
system SYS_read
%endmacro

%macro sys.write 0
system SYS_write
%endmacro

; [etc...]
```

继续，把它添加到你的编辑器中，然后把它保存为`system.inc`。我们将随着讨论得深入，将更多的系统调用添加进来。

11.6 编写第一个程序

We are now ready for our first program, the mandatory **Hello, World!**

```
 1: %include 'system.inc'
 2:
 3: section .data
 4: hello db 'Hello, World!', 0Ah
 5: hbytes equ $-hello
 6:
 7: section .text
 8: global _start
 9: _start:
10: push dword hbytes
11: push dword hello
12: push dword stdout
13: sys.write
14:
15: push dword 0
16: sys.exit
```

Here is what it does: Line 1 includes the defines, the macros, and the code from `system.inc`.

Lines 3-5 are the data: Line 3 starts the data section/segment. Line 4 contains the string "Hello, World!" followed by a new line (`0Ah`). Line 5 creates a constant that contains the length of the string from line 4 in bytes.

Lines 7-16 contain the code. Note that FreeBSD uses the *elf* file format for its executables, which requires every program to start at the point labeled `_start` (or, more precisely, the linker expects that). This label has to be global.

Lines 10-13 ask the system to write `hbytes` bytes of the `hello` string to `stdout`.

Lines 15-16 ask the system to end the program with the return value of `0`. The `SYS_exit` syscall never returns, so the code ends there.

> **注意:** If you have come to UNIX from MS-DOS assembly language background, you may be used to writing directly to the video hardware. You will never have to worry about this in FreeBSD, or any other flavor of UNIX. As far as you are concerned, you are writing to a file known as `stdout`. This can be the video screen, or a **telnet** terminal, or an actual file, or even the input of another program. Which one it is, is for the system to figure out.

11.6.1 Assembling the Code

Type the code (except the line numbers) in an editor, and save it in a file named `hello.asm`. You need **nasm** to assemble it.

11.6.1.1 Installing nasm

If you do not have **nasm**, type:

```
% su
Password:your root password
# cd /usr/ports/devel/nasm
# make install
# exit
%
```

You may type **`make install clean`** instead of just **`make install`** if you do not want to keep **nasm** source code.

Either way, FreeBSD will automatically download **nasm** from the Internet, compile it, and install it on your system.

> **注意:** If your system is not FreeBSD, you need to get **nasm** from its home page (http://www.web-sites.co.uk/nasm/). You can still use it to assemble FreeBSD code.

Now you can assemble, link, and run the code:

```
% nasm -f elf hello.asm
% ld -s -o hello hello.o
% ./hello
Hello, World!
%
```

11.7 Writing UNIX® Filters

A common type of UNIX application is a filter——a program that reads data from the `stdin`, processes it somehow, then writes the result to `stdout`.

In this chapter, we shall develop a simple filter, and learn how to read from `stdin` and write to `stdout`. This filter will convert each byte of its input into a hexadecimal number followed by a blank space.

```
%include 'system.inc'

section .data
hex db '0123456789ABCDEF'
buffer db 0, 0, ' '

section .text
global _start
```

```
_start:
; read a byte from stdin
push dword 1
push dword buffer
push dword stdin
sys.read
add esp, byte 12
or eax, eax
je .done

; convert it to hex
movzx eax, byte [buffer]
mov edx, eax
shr dl, 4
mov dl, [hex+edx]
mov [buffer], dl
and al, 0Fh
mov al, [hex+eax]
mov [buffer+1], al

; print it
push dword 3
push dword buffer
push dword stdout
sys.write
add esp, byte 12
jmp short _start

.done:
push dword 0
sys.exit
```

In the data section we create an array called `hex`. It contains the 16 hexadecimal digits in ascending order. The array is followed by a buffer which we will use for both input and output. The first two bytes of the buffer are initially set to `0`. This is where we will write the two hexadecimal digits (the first byte also is where we will read the input). The third byte is a space.

The code section consists of four parts: Reading the byte, converting it to a hexadecimal number, writing the result, and eventually exiting the program.

To read the byte, we ask the system to read one byte from `stdin`, and store it in the first byte of the `buffer`. The system returns the number of bytes read in `EAX`. This will be `1` while data is coming, or `0`, when no more input data is available. Therefore, we check the value of `EAX`. If it is `0`, we jump to `.done`, otherwise we continue.

注意: For simplicity sake, we are ignoring the possibility of an error condition at this time.

The hexadecimal conversion reads the byte from the `buffer` into `EAX`, or actually just `AL`, while clearing the remaining bits of `EAX` to zeros. We also copy the byte to `EDX` because we need to convert the upper four bits (nibble) separately from the lower four bits. We store the result in the first two bytes of the buffer.

Next, we ask the system to write the three bytes of the buffer, i.e., the two hexadecimal digits and the blank space, to `stdout`. We then jump back to the beginning of the program and process the next byte.

Once there is no more input left, we ask the system to exit our program, returning a zero, which is the traditional value meaning the program was successful.

Go ahead, and save the code in a file named `hex.asm`, then type the following (the **^D** means press the control key and type **D** while holding the control key down):

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A ^D %
```

> **注意:** If you are migrating to UNIX from MS-DOS, you may be wondering why each line ends with `0A` instead of `0D 0A`. This is because UNIX does not use the cr/lf convention, but a "new line" convention, which is `0A` in hexadecimal.

Can we improve this? Well, for one, it is a bit confusing because once we have converted a line of text, our input no longer starts at the beginning of the line. We can modify it to print a new line instead of a space after each `0A`:

```
%include 'system.inc'

section .data
hex db '0123456789ABCDEF'
buffer db 0, 0, ' '

section .text
global _start
_start:
mov cl, ' '

.loop:
; read a byte from stdin
push dword 1
push dword buffer
push dword stdin
sys.read
add esp, byte 12
or eax, eax
je .done

; convert it to hex
movzx eax, byte [buffer]
mov [buffer+2], cl
cmp al, 0Ah
jne .hex
mov [buffer+2], al

.hex:
mov edx, eax
shr dl, 4
```

```
mov dl, [hex+edx]
mov [buffer], dl
and al, 0Fh
mov al, [hex+eax]
mov [buffer+1], al

; print it
push dword 3
push dword buffer
push dword stdout
sys.write
add esp, byte 12
jmp short .loop

.done:
push dword 0
sys.exit
```

We have stored the space in the `CL` register. We can do this safely because, unlike Microsoft Windows, UNIX system calls do not modify the value of any register they do not use to return a value in.

That means we only need to set `CL` once. We have, therefore, added a new label `.loop` and jump to it for the next byte instead of jumping at `_start`. We have also added the `.hex` label so we can either have a blank space or a new line as the third byte of the `buffer`.

Once you have changed `hex.asm` to reflect these changes, type:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D %
```

That looks better. But this code is quite inefficient! We are making a system call for every single byte twice (once to read it, another time to write the output).

11.8 Buffered Input and Output

We can improve the efficiency of our code by buffering our input and output. We create an input buffer and read a whole sequence of bytes at one time. Then we fetch them one by one from the buffer.

We also create an output buffer. We store our output in it until it is full. At that time we ask the kernel to write the contents of the buffer to `stdout`.

The program ends when there is no more input. But we still need to ask the kernel to write the contents of our output buffer to `stdout` one last time, otherwise some of our output would make it to the output buffer, but never be sent out. Do not forget that, or you will be wondering why some of your output is missing.

```
%include 'system.inc'
```

```
%define BUFSIZE 2048

section .data
hex db '0123456789ABCDEF'

section .bss
ibuffer resb BUFSIZE
obuffer resb BUFSIZE

section .text
global _start
_start:
sub eax, eax
sub ebx, ebx
sub ecx, ecx
mov edi, obuffer

.loop:
; read a byte from stdin
call getchar

; convert it to hex
mov dl, al
shr al, 4
mov al, [hex+eax]
call putchar

mov al, dl
and al, 0Fh
mov al, [hex+eax]
call putchar

mov al, ' '
cmp dl, 0Ah
jne .put
mov al, dl

.put:
call putchar
jmp short .loop

align 4
getchar:
or ebx, ebx
jne .fetch

call read

.fetch:
lodsb
dec ebx
ret
```

```
read:
push dword BUFSIZE
mov esi, ibuffer
push esi
push dword stdin
sys.read
add esp, byte 12
mov ebx, eax
or eax, eax
je .done
sub eax, eax
ret

align 4
.done:
call write ; flush output buffer
push dword 0
sys.exit

align 4
putchar:
stosb
inc ecx
cmp ecx, BUFSIZE
je write
ret

align 4
write:
sub edi, ecx ; start of buffer
push ecx
push edi
push dword stdout
sys.write
add esp, byte 12
sub eax, eax
sub ecx, ecx ; buffer is empty now
ret
```

We now have a third section in the source code, named `.bss`. This section is not included in our executable file, and, therefore, cannot be initialized. We use `resb` instead of `db`. It simply reserves the requested size of uninitialized memory for our use.

We take advantage of the fact that the system does not modify the registers: We use registers for what, otherwise, would have to be global variables stored in the `.data` section. This is also why the UNIX convention of passing parameters to system calls on the stack is superior to the Microsoft convention of passing them in the registers: We can keep the registers for our own use.

We use `EDI` and `ESI` as pointers to the next byte to be read from or written to. We use `EBX` and `ECX` to keep count of the number of bytes in the two buffers, so we know when to dump the output to, or read more input from, the system.

Let us see how it works now:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
Here I come!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D %
```

Not what you expected? The program did not print the output until we pressed **^D**. That is easy to fix by inserting three lines of code to write the output every time we have converted a new line to `0A`. I have marked the three lines with > (do not copy the > in your `hex.asm`).

```
%include 'system.inc'

%define BUFSIZE 2048

section .data
hex db '0123456789ABCDEF'

section .bss
ibuffer resb BUFSIZE
obuffer resb BUFSIZE

section .text
global _start
_start:
sub eax, eax
sub ebx, ebx
sub ecx, ecx
mov edi, obuffer

.loop:
; read a byte from stdin
call getchar

; convert it to hex
mov dl, al
shr al, 4
mov al, [hex+eax]
call putchar

mov al, dl
and al, 0Fh
mov al, [hex+eax]
call putchar

mov al, ' '
cmp dl, 0Ah
jne .put
mov al, dl

.put:
```

```
call putchar
> cmp al, 0Ah
> jne .loop
> call write
jmp short .loop

align 4
getchar:
or ebx, ebx
jne .fetch

call read

.fetch:
lodsb
dec ebx
ret

read:
push dword BUFSIZE
mov esi, ibuffer
push esi
push dword stdin
sys.read
add esp, byte 12
mov ebx, eax
or eax, eax
je .done
sub eax, eax
ret

align 4
.done:
call write ; flush output buffer
push dword 0
sys.exit

align 4
putchar:
stosb
inc ecx
cmp ecx, BUFSIZE
je write
ret

align 4
write:
sub edi, ecx ; start of buffer
push ecx
push edi
push dword stdout
sys.write
add esp, byte 12
```

```
sub eax, eax
sub ecx, ecx ; buffer is empty now
ret
```

Now, let us see how it works:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D %
```

Not bad for a 644-byte executable, is it!

注意: This approach to buffered input/output still contains a hidden danger. I will discuss——and fix——it later, when I talk about the dark side of buffering.

11.8.1 How to Unread a Character

警告: This may be a somewhat advanced topic, mostly of interest to programmers familiar with the theory of compilers. If you wish, you may skip to the next section, and perhaps read this later.

While our sample program does not require it, more sophisticated filters often need to look ahead. In other words, they may need to see what the next character is (or even several characters). If the next character is of a certain value, it is part of the token currently being processed. Otherwise, it is not.

For example, you may be parsing the input stream for a textual string (e.g., when implementing a language compiler): If a character is followed by another character, or perhaps a digit, it is part of the token you are processing. If it is followed by white space, or some other value, then it is not part of the current token.

This presents an interesting problem: How to return the next character back to the input stream, so it can be read again later?

One possible solution is to store it in a character variable, then set a flag. We can modify `getchar` to check the flag, and if it is set, fetch the byte from that variable instead of the input buffer, and reset the flag. But, of course, that slows us down.

The C language has an `ungetc()` function, just for that purpose. Is there a quick way to implement it in our code? I would like you to scroll back up and take a look at the `getchar` procedure and see if you can find a nice and fast solution before reading the next paragraph. Then come back here and see my own solution.

The key to returning a character back to the stream is in how we are getting the characters to start with:

First we check if the buffer is empty by testing the value of `EBX`. If it is zero, we call the `read` procedure.

If we do have a character available, we use `lodsb`, then decrease the value of `EBX`. The `lodsb` instruction is effectively identical to:

```
mov al, [esi]
inc esi
```

The byte we have fetched remains in the buffer until the next time `read` is called. We do not know when that happens, but we do know it will not happen until the next call to `getchar`. Hence, to "return" the last-read byte back to the stream, all we have to do is decrease the value of `ESI` and increase the value of `EBX`:

```
ungetc:
dec esi
inc ebx
ret
```

But, be careful! We are perfectly safe doing this if our look-ahead is at most one character at a time. If we are examining more than one upcoming character and call `ungetc` several times in a row, it will work most of the time, but not all the time (and will be tough to debug). Why?

Because as long as `getchar` does not have to call `read`, all of the pre-read bytes are still in the buffer, and our `ungetc` works without a glitch. But the moment `getchar` calls `read`, the contents of the buffer change.

We can always rely on `ungetc` working properly on the last character we have read with `getchar`, but not on anything we have read before that.

If your program reads more than one byte ahead, you have at least two choices:

If possible, modify the program so it only reads one byte ahead. This is the simplest solution.

If that option is not available, first of all determine the maximum number of characters your program needs to return to the input stream at one time. Increase that number slightly, just to be sure, preferably to a multiple of 16——so it aligns nicely. Then modify the `.bss` section of your code, and create a small "spare" buffer right before your input buffer, something like this:

```
section .bss
resb 16 ; or whatever the value you came up with
ibuffer resb BUFSIZE
obuffer resb BUFSIZE
```

You also need to modify your `ungetc` to pass the value of the byte to unget in `AL`:

```
ungetc:
dec esi
inc ebx
mov [esi], al
ret
```

With this modification, you can call `ungetc` up to 17 times in a row safely (the first call will still be within the buffer, the remaining 16 may be either within the buffer or within the "spare").

11.9 Command Line Arguments

Our **hex** program will be more useful if it can read the names of an input and output file from its command line, i.e., if it can process the command line arguments. But... Where are they?

Before a UNIX system starts a program, it `push`es some data on the stack, then jumps at the `_start` label of the program. Yes, I said jumps, not calls. That means the data can be accessed by reading `[esp+offset]`, or by simply `pop`ping it.

The value at the top of the stack contains the number of command line arguments. It is traditionally called `argc`, for "argument count."

Command line arguments follow next, all `argc` of them. These are typically referred to as `argv`, for "argument value(s)." That is, we get `argv[0]`, `argv[1]`, `...`, `argv[argc-1]`. These are not the actual arguments, but pointers to arguments, i.e., memory addresses of the actual arguments. The arguments themselves are NUL-terminated character strings.

The `argv` list is followed by a NULL pointer, which is simply a `0`. There is more, but this is enough for our purposes right now.

> **注意:** If you have come from the MS-DOS programming environment, the main difference is that each argument is in a separate string. The second difference is that there is no practical limit on how many arguments there can be.

Armed with this knowledge, we are almost ready for the next version of `hex.asm`. First, however, we need to add a few lines to `system.inc`:

First, we need to add two new entries to our list of system call numbers:

```
%define SYS_open 5
%define SYS_close 6
```

Then we add two new macros at the end of the file:

```
%macro sys.open 0
system SYS_open
%endmacro

%macro sys.close 0
system SYS_close
%endmacro
```

Here, then, is our modified source code:

```
%include 'system.inc'

%define BUFSIZE 2048

section .data
fd.in dd stdin
fd.out dd stdout
hex db '0123456789ABCDEF'

section .bss
ibuffer resb BUFSIZE
obuffer resb BUFSIZE

section .text
```

```
align 4
err:
push dword 1 ; return failure
sys.exit

align 4
global _start
_start:
add esp, byte 8 ; discard argc and argv[0]

pop ecx
jecxz .init ; no more arguments

; ECX contains the path to input file
push dword 0 ; O_RDONLY
push ecx
sys.open
jc err ; open failed

add esp, byte 8
mov [fd.in], eax

pop ecx
jecxz .init ; no more arguments

; ECX contains the path to output file
push dword 420 ; file mode (644 octal)
push dword 0200h | 0400h | 01h
; O_CREAT | O_TRUNC | O_WRONLY
push ecx
sys.open
jc err

add esp, byte 12
mov [fd.out], eax

.init:
sub eax, eax
sub ebx, ebx
sub ecx, ecx
mov edi, obuffer

.loop:
; read a byte from input file or stdin
call getchar

; convert it to hex
mov dl, al
shr al, 4
mov al, [hex+eax]
call putchar

mov al, dl
```

```
and al, 0Fh
mov al, [hex+eax]
call putchar

mov al, ' '
cmp dl, 0Ah
jne .put
mov al, dl

.put:
call putchar
cmp al, dl
jne .loop
call write
jmp short .loop

align 4
getchar:
or ebx, ebx
jne .fetch

call read

.fetch:
lodsb
dec ebx
ret

read:
push dword BUFSIZE
mov esi, ibuffer
push esi
push dword [fd.in]
sys.read
add esp, byte 12
mov ebx, eax
or eax, eax
je .done
sub eax, eax
ret

align 4
.done:
call write ; flush output buffer

; close files
push dword [fd.in]
sys.close

push dword [fd.out]
sys.close

; return success
```

```
push dword 0
sys.exit

align 4
putchar:
stosb
inc ecx
cmp ecx, BUFSIZE
je write
ret

align 4
write:
sub edi, ecx ; start of buffer
push ecx
push edi
push dword [fd.out]
sys.write
add esp, byte 12
sub eax, eax
sub ecx, ecx ; buffer is empty now
ret
```

In our `.data` section we now have two new variables, `fd.in` and `fd.out`. We store the input and output file descriptors here.

In the `.text` section we have replaced the references to `stdin` and `stdout` with `[fd.in]` and `[fd.out]`.

The `.text` section now starts with a simple error handler, which does nothing but exit the program with a return value of `1`. The error handler is before `_start` so we are within a short distance from where the errors occur.

Naturally, the program execution still begins at `_start`. First, we remove `argc` and `argv[0]` from the stack: They are of no interest to us (in this program, that is).

We pop `argv[1]` to `ECX`. This register is particularly suited for pointers, as we can handle NULL pointers with `jecxz`. If `argv[1]` is not NULL, we try to open the file named in the first argument. Otherwise, we continue the program as before: Reading from `stdin`, writing to `stdout`. If we fail to open the input file (e.g., it does not exist), we jump to the error handler and quit.

If all went well, we now check for the second argument. If it is there, we open the output file. Otherwise, we send the output to `stdout`. If we fail to open the output file (e.g., it exists and we do not have the write permission), we, again, jump to the error handler.

The rest of the code is the same as before, except we close the input and output files before exiting, and, as mentioned, we use `[fd.in]` and `[fd.out]`.

Our executable is now a whopping 768 bytes long.

Can we still improve it? Of course! Every program can be improved. Here are a few ideas of what we could do:

Have our error handler print a message to `stderr`.

Add error handlers to the `read` and `write` functions.

Close `stdin` when we open an input file, `stdout` when we open an output file.

Add command line switches, such as `-i` and `-o`, so we can list the input and output files in any order, or perhaps read from `stdin` and write to a file.

Print a usage message if command line arguments are incorrect.

I shall leave these enhancements as an exercise to the reader: You already know everything you need to know to implement them.

11.10 UNIX Environment

An important UNIX concept is the environment, which is defined by *environment variables*. Some are set by the system, others by you, yet others by the **shell**, or any program that loads another program.

11.10.1 How to Find Environment Variables

I said earlier that when a program starts executing, the stack contains `argc` followed by the NULL-terminated `argv` array, followed by something else. The "something else" is the *environment*, or, to be more precise, a NULL-terminated array of pointers to *environment variables*. This is often referred to as `env`.

The structure of `env` is the same as that of `argv`, a list of memory addresses followed by a NULL (`0`). In this case, there is no `"envc"`——we figure out where the array ends by searching for the final NULL.

The variables usually come in the `name=value` format, but sometimes the `=value` part may be missing. We need to account for that possibility.

11.10.2 webvars

I could just show you some code that prints the environment the same way the UNIX **env** command does. But I thought it would be more interesting to write a simple assembly language CGI utility.

11.10.2.1 CGI: A Quick Overview

I have a detailed CGI tutorial (http://www.whizkidtech.redprince.net/cgi-bin/tutorial) on my web site, but here is a very quick overview of CGI:

The web server communicates with the CGI program by setting *environment variables*.

The CGI program sends its output to `stdout`. The web server reads it from there.

It must start with an HTTP header followed by two blank lines.

It then prints the HTML code, or whatever other type of data it is producing.

> **注意:** While certain *environment variables* use standard names, others vary, depending on the web server. That makes **webvars** quite a useful diagnostic tool.

11.10.2.2 The Code

Our **webvars** program, then, must send out the HTTP header followed by some HTML mark-up. It then must read the *environment variables* one by one and send them out as part of the HTML page.

The code follows. I placed comments and explanations right inside the code:

```
;;;;;;; webvars.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Copyright (c) 2000 G. Adam Stanislav
; All rights reserved.
;
; Redistribution and use in source and binary forms, with or without
; modification, are permitted provided that the following conditions
; are met:
; 1. Redistributions of source code must retain the above copyright
;    notice, this list of conditions and the following disclaimer.
; 2. Redistributions in binary form must reproduce the above copyright
;    notice, this list of conditions and the following disclaimer in the
;    documentation and/or other materials provided with the distribution.
;
; THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
; ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
; IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
; ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
; FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
; DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
; OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
; HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
; LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
; OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
; SUCH DAMAGE.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Version 1.0
;
; Started:  8-Dec-2000
; Updated:  8-Dec-2000
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%include 'system.inc'

section .data
http db 'Content-type: text/html', 0Ah, 0Ah
db '<?xml version="1.0" encoding="UTF-8"?>', 0Ah
db '<!DOCTYPE html PUBLIC "-//W3C/DTD XHTML Strict//EN" '
db '"DTD/xhtml1-strict.dtd">', 0Ah
db '<html xmlns="http://www.w3.org/1999/xhtml" '
db 'xml.lang="en" lang="en">', 0Ah
db '<head>', 0Ah
db '<title>Web Environment</title>', 0Ah
db '<meta name="author" content="G. Adam Stanislav" />', 0Ah
db '</head>', 0Ah, 0Ah
db '<body bgcolor="#ffffff" text="#000000" link="#0000ff" '
```

```
db 'vlink="#840084" alink="#0000ff">', 0Ah
db '<div class="webvars">', 0Ah
db '<h1>Web Environment</h1>', 0Ah
db '<p>The following <b>environment variables</b> are defined '
db 'on this web server:</p>', 0Ah, 0Ah
db '<table align="center" width="80" border="0" cellpadding="10" '
db 'cellspacing="0" class="webvars">', 0Ah
httplen equ $-http
left db '<tr>', 0Ah
db '<td class="name"><tt>'
leftlen equ $-left
middle db '</tt></td>', 0Ah
db '<td class="value"><tt><b>'
midlen equ $-middle
undef db '<i>(undefined)</i>'
undeflen equ $-undef
right db '</b></tt></td>', 0Ah
db '</tr>', 0Ah
rightlen equ $-right
wrap db '</table>', 0Ah
db '</div>', 0Ah
db '</body>', 0Ah
db '</html>', 0Ah, 0Ah
wraplen equ $-wrap

section .text
global _start
_start:
; First, send out all the http and xhtml stuff that is
; needed before we start showing the environment
push dword httplen
push dword http
push dword stdout
sys.write

; Now find how far on the stack the environment pointers
; are. We have 12 bytes we have pushed before "argc"
mov eax, [esp+12]

; We need to remove the following from the stack:
;
; The 12 bytes we pushed for sys.write
; The  4 bytes of argc
; The EAX*4 bytes of argv
; The  4 bytes of the NULL after argv
;
; Total:
; 20 + eax * 4
;
; Because stack grows down, we need to ADD that many bytes
; to ESP.
lea esp, [esp+20+eax*4]
cld ; This should already be the case, but let's be sure.
```

```
; Loop through the environment, printing it out
.loop:
pop edi
or edi, edi ; Done yet?
je near .wrap

; Print the left part of HTML
push dword leftlen
push dword left
push dword stdout
sys.write

; It may be tempting to search for the '=' in the env string next.
; But it is possible there is no '=', so we search for the
; terminating NUL first.
mov esi, edi ; Save start of string
sub ecx, ecx
not ecx ; ECX = FFFFFFFF
sub eax, eax
repne scasb
not ecx ; ECX = string length + 1
mov ebx, ecx ; Save it in EBX

; Now is the time to find '='
mov edi, esi ; Start of string
mov al, '='
repne scasb
not ecx
add ecx, ebx ; Length of name

push ecx
push esi
push dword stdout
sys.write

; Print the middle part of HTML table code
push dword midlen
push dword middle
push dword stdout
sys.write

; Find the length of the value
not ecx
lea ebx, [ebx+ecx-1]

; Print "undefined" if 0
or ebx, ebx
jne .value

mov ebx, undeflen
mov edi, undef
```

```
.value:
push ebx
push edi
push dword stdout
sys.write

; Print the right part of the table row
push dword rightlen
push dword right
push dword stdout
sys.write

; Get rid of the 60 bytes we have pushed
add esp, byte 60

; Get the next variable
jmp .loop

.wrap:
; Print the rest of HTML
push dword wraplen
push dword wrap
push dword stdout
sys.write

; Return success
push dword 0
sys.exit
```

This code produces a 1,396-byte executable. Most of it is data, i.e., the HTML mark-up we need to send out.

Assemble and link it as usual:

```
% nasm -f elf webvars.asm
% ld -s -o webvars webvars.o
```

To use it, you need to upload `webvars` to your web server. Depending on how your web server is set up, you may have to store it in a special `cgi-bin` directory, or perhaps rename it with a `.cgi` extension.

Then you need to use your browser to view its output. To see its output on my web server, please go to `http://www.int80h.org/webvars/`. If curious about the additional environment variables present in a password protected web directory, go to `http://www.int80h.org/private/`, using the name **`asm`** and password **`programmer`**.

11.11 Working with Files

We have already done some basic file work: We know how to open and close them, how to read and write them using buffers. But UNIX offers much more functionality when it comes to files. We will examine some of it in this section, and end up with a nice file conversion utility.

Indeed, let us start at the end, that is, with the file conversion utility. It always makes programming easier when we know from the start what the end product is supposed to do.

One of the first programs I wrote for UNIX was **tuc** (ftp://ftp.int80h.org/unix/tuc/), a text-to-UNIX file converter. It converts a text file from other operating systems to a UNIX text file. In other words, it changes from different kind of line endings to the newline convention of UNIX. It saves the output in a different file. Optionally, it converts a UNIX text file to a DOS text file.

I have used **tuc** extensively, but always only to convert from some other OS to UNIX, never the other way. I have always wished it would just overwrite the file instead of me having to send the output to a different file. Most of the time, I end up using it like this:

```
% tuc myfile tempfile
% mv tempfile myfile
```

It would be nice to have a **ftuc**, i.e., *fast tuc*, and use it like this:

```
% ftuc myfile
```

In this chapter, then, we will write **ftuc** in assembly language (the original **tuc** is in C), and study various file-oriented kernel services in the process.

At first sight, such a file conversion is very simple: All you have to do is strip the carriage returns, right?

If you answered yes, think again: That approach will work most of the time (at least with MS DOS text files), but will fail occasionally.

The problem is that not all non UNIX text files end their line with the carriage return / line feed sequence. Some use carriage returns without line feeds. Others combine several blank lines into a single carriage return followed by several line feeds. And so on.

A text file converter, then, must be able to handle any possible line endings:

carriage return / line feed

carriage return

line feed / carriage return

line feed

It should also handle files that use some kind of a combination of the above (e.g., carriage return followed by several line feeds).

11.11.1 Finite State Machine

The problem is easily solved by the use of a technique called *finite state machine*, originally developed by the designers of digital electronic circuits. A *finite state machine* is a digital circuit whose output is dependent not only on its input but on its previous input, i.e., on its state. The microprocessor is an example of a *finite state machine*: Our assembly language code is assembled to machine language in which some assembly language code produces a single byte of machine language, while others produce several bytes. As the microprocessor fetches the bytes from the memory one by one, some of them simply change its state rather than produce some output. When all the bytes of the op code are fetched, the microprocessor produces some output, or changes the value of a register, etc.

Because of that, all software is essentially a sequence of state instructions for the microprocessor. Nevertheless, the concept of *finite state machine* is useful in software design as well.

Our text file converter can be designed as a *finite state machine* with three possible states. We could call them states 0-2, but it will make our life easier if we give them symbolic names:

ordinary

cr

lf

Our program will start in the ordinary state. During this state, the program action depends on its input as follows:

If the input is anything other than a carriage return or line feed, the input is simply passed on to the output. The state remains unchanged.

If the input is a carriage return, the state is changed to cr. The input is then discarded, i.e., no output is made.

If the input is a line feed, the state is changed to lf. The input is then discarded.

Whenever we are in the cr state, it is because the last input was a carriage return, which was unprocessed. What our software does in this state again depends on the current input:

If the input is anything other than a carriage return or line feed, output a line feed, then output the input, then change the state to ordinary.

If the input is a carriage return, we have received two (or more) carriage returns in a row. We discard the input, we output a line feed, and leave the state unchanged.

If the input is a line feed, we output the line feed and change the state to ordinary. Note that this is not the same as the first case above – if we tried to combine them, we would be outputting two line feeds instead of one.

Finally, we are in the lf state after we have received a line feed that was not preceded by a carriage return. This will happen when our file already is in UNIX format, or whenever several lines in a row are expressed by a single carriage return followed by several line feeds, or when line ends with a line feed / carriage return sequence. Here is how we need to handle our input in this state:

If the input is anything other than a carriage return or line feed, we output a line feed, then output the input, then change the state to ordinary. This is exactly the same action as in the cr state upon receiving the same kind of input.

If the input is a carriage return, we discard the input, we output a line feed, then change the state to ordinary.

If the input is a line feed, we output the line feed, and leave the state unchanged.

11.11.1.1 The Final State

The above *finite state machine* works for the entire file, but leaves the possibility that the final line end will be ignored. That will happen whenever the file ends with a single carriage return or a single line feed. I did not think of it when I wrote **tuc**, just to discover that occasionally it strips the last line ending.

This problem is easily fixed by checking the state after the entire file was processed. If the state is not ordinary, we simply need to output one last line feed.

注意: Now that we have expressed our algorithm as a *finite state machine*, we could easily design a dedicated digital electronic circuit (a "chip") to do the conversion for us. Of course, doing so would be considerably more expensive than writing an assembly language program.

11.11.1.2 The Output Counter

Because our file conversion program may be combining two characters into one, we need to use an output counter. We initialize it to `0`, and increase it every time we send a character to the output. At the end of the program, the counter will tell us what size we need to set the file to.

11.11.2 Implementing FSM in Software

The hardest part of working with a *finite state machine* is analyzing the problem and expressing it as a *finite state machine*. That accomplished, the software almost writes itself.

In a high-level language, such as C, there are several main approaches. One is to use a `switch` statement which chooses what function should be run. For example,

```
switch (state) {
default:
case REGULAR:
regular(inputchar);
break;
case CR:
cr(inputchar);
break;
case LF:
lf(inputchar);
break;
}
```

Another approach is by using an array of function pointers, something like this:

```
(output[state])(inputchar);
```

Yet another is to have `state` be a function pointer, set to point at the appropriate function:

```
(*state)(inputchar);
```

This is the approach we will use in our program because it is very easy to do in assembly language, and very fast, too. We will simply keep the address of the right procedure in `EBX`, and then just issue:

```
call ebx
```

This is possibly faster than hardcoding the address in the code because the microprocessor does not have to fetch the address from the memory——it is already stored in one of its registers. I said *possibly* because with the caching modern microprocessors do, either way may be equally fast.

11.11.3 Memory Mapped Files

Because our program works on a single file, we cannot use the approach that worked for us before, i.e., to read from an input file and to write to an output file.

UNIX allows us to map a file, or a section of a file, into memory. To do that, we first need to open the file with the appropriate read/write flags. Then we use the `mmap` system call to map it into the memory. One nice thing about `mmap` is that it automatically works with virtual memory: We can map more of the file into the memory than we have physical memory available, yet still access it through regular memory op codes, such as `mov`, `lods`, and `stos`. Whatever changes we make to the memory image of the file will be written to the file by the system. We do not even have to keep the file open: As long as it stays mapped, we can read from it and write to it.

The 32-bit Intel microprocessors can access up to four gigabytes of memory – physical or virtual. The FreeBSD system allows us to use up to a half of it for file mapping.

For simplicity sake, in this tutorial we will only convert files that can be mapped into the memory in their entirety. There are probably not too many text files that exceed two gigabytes in size. If our program encounters one, it will simply display a message suggesting we use the original **tuc** instead.

If you examine your copy of `syscalls.master`, you will find two separate syscalls named `mmap`. This is because of evolution of UNIX: There was the traditional BSD `mmap`, syscall 71. That one was superseded by the POSIX `mmap`, syscall 197. The FreeBSD system supports both because older programs were written by using the original BSD version. But new software uses the POSIX version, which is what we will use.

The `syscalls.master` file lists the POSIX version like this:

```
197 STD BSD { caddr_t mmap(caddr_t addr, size_t len, int prot, \
    int flags, int fd, long pad, off_t pos); }
```

This differs slightly from what mmap(2) says. That is because mmap(2) describes the C version.

The difference is in the `long pad` argument, which is not present in the C version. However, the FreeBSD syscalls add a 32-bit pad after `push`ing a 64-bit argument. In this case, `off_t` is a 64-bit value.

When we are finished working with a memory-mapped file, we unmap it with the `munmap` syscall:

提示: For an in-depth treatment of `mmap`, see W. Richard Stevens' Unix Network Programming, Volume 2, Chapter 12 (http://www.int80h.org/cgi-bin/isbn?isbn=0130810819).

11.11.4 Determining File Size

Because we need to tell `mmap` how many bytes of the file to map into the memory, and because we want to map the entire file, we need to determine the size of the file.

We can use the `fstat` syscall to get all the information about an open file that the system can give us. That includes the file size.

Again, `syscalls.master` lists two versions of `fstat`, a traditional one (syscall 62), and a POSIX one (syscall 189). Naturally, we will use the POSIX version:

```
189 STD POSIX { int fstat(int fd, struct stat *sb); }
```

This is a very straightforward call: We pass to it the address of a stat structure and the descriptor of an open file. It will fill out the contents of the stat structure.

I do, however, have to say that I tried to declare the stat structure in the `.bss` section, and `fstat` did not like it: It set the carry flag indicating an error. After I changed the code to allocate the structure on the stack, everything was working fine.

11.11.5 Changing the File Size

Because our program may combine carriage return / line feed sequences into straight line feeds, our output may be smaller than our input. However, since we are placing our output into the same file we read the input from, we may have to change the size of the file.

The `ftruncate` system call allows us to do just that. Despite its somewhat misleading name, the `ftruncate` system call can be used to both truncate the file (make it smaller) and to grow it.

And yes, we will find two versions of `ftruncate` in `syscalls.master`, an older one (130), and a newer one (201). We will use the newer one:

```
201 STD BSD { int ftruncate(int fd, int pad, off_t length); }
```

Please note that this one contains a `int pad` again.

11.11.6 ftuc

We now know everything we need to write **ftuc**. We start by adding some new lines in `system.inc`. First, we define some constants and structures, somewhere at or near the beginning of the file:

```
;;;;;;; open flags
%define O_RDONLY 0
%define O_WRONLY 1
%define O_RDWR 2

;;;;;;; mmap flags
%define PROT_NONE 0
%define PROT_READ 1
%define PROT_WRITE 2
%define PROT_EXEC 4
;;
%define MAP_SHARED 0001h
%define MAP_PRIVATE 0002h

;;;;;;; stat structure
struc stat
st_dev resd 1 ; = 0
st_ino resd 1 ; = 4
st_mode resw 1 ; = 8, size is 16 bits
st_nlink resw 1 ; = 10, ditto
st_uid resd 1 ; = 12
st_gid resd 1 ; = 16
st_rdev resd 1 ; = 20
st_atime resd 1 ; = 24
```

```
st_atimensec resd 1 ; = 28
st_mtime resd 1 ; = 32
st_mtimensec resd 1 ; = 36
st_ctime resd 1 ; = 40
st_ctimensec resd 1 ; = 44
st_size resd 2 ; = 48, size is 64 bits
st_blocks resd 2 ; = 56, ditto
st_blksize resd 1 ; = 64
st_flags resd 1 ; = 68
st_gen resd 1 ; = 72
st_lspare resd 1 ; = 76
st_qspare resd 4 ; = 80
endstruc
```

We define the new syscalls:

```
%define SYS_mmap 197
%define SYS_munmap 73
%define SYS_fstat 189
%define SYS_ftruncate 201
```

We add the macros for their use:

```
%macro sys.mmap 0
system SYS_mmap
%endmacro

%macro sys.munmap 0
system SYS_munmap
%endmacro

%macro sys.ftruncate 0
system SYS_ftruncate
%endmacro

%macro sys.fstat 0
system SYS_fstat
%endmacro
```

And here is our code:

```
;;;;;;; Fast Text-to-Unix Conversion (ftuc.asm) ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Started: 21-Dec-2000
;; Updated: 22-Dec-2000
;;
;; Copyright 2000 G. Adam Stanislav.
;; All rights reserved.
;;
;;;;;;; v.1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%include 'system.inc'

section .data
db 'Copyright 2000 G. Adam Stanislav.', 0Ah
```

```
db 'All rights reserved.', 0Ah
usg db 'Usage: ftuc filename', 0Ah
usglen equ $-usg
co db "ftuc: Can't open file.", 0Ah
colen equ $-co
fae db 'ftuc: File access error.', 0Ah
faelen equ $-fae
ftl db 'ftuc: File too long, use regular tuc instead.', 0Ah
ftllen equ $-ftl
mae db 'ftuc: Memory allocation error.', 0Ah
maelen equ $-mae

section .text

align 4
memerr:
push dword maelen
push dword mae
jmp short error

align 4
toolong:
push dword ftllen
push dword ftl
jmp short error

align 4
facerr:
push dword faelen
push dword fae
jmp short error

align 4
cantopen:
push dword colen
push dword co
jmp short error

align 4
usage:
push dword usglen
push dword usg

error:
push dword stderr
sys.write

push dword 1
sys.exit

align 4
global _start
_start:
```

```
pop eax ; argc
pop eax ; program name
pop ecx ; file to convert
jecxz usage

pop eax
or eax, eax ; Too many arguments?
jne usage

; Open the file
push dword O_RDWR
push ecx
sys.open
jc cantopen

mov ebp, eax ; Save fd

sub esp, byte stat_size
mov ebx, esp

; Find file size
push ebx
push ebp ; fd
sys.fstat
jc facerr

mov edx, [ebx + st_size + 4]

; File is too long if EDX != 0 ...
or edx, edx
jne near toolong
mov ecx, [ebx + st_size]
; ... or if it is above 2 GB
or ecx, ecx
js near toolong

; Do nothing if the file is 0 bytes in size
jecxz .quit

; Map the entire file in memory
push edx
push edx ; starting at offset 0
push edx ; pad
push ebp ; fd
push dword MAP_SHARED
push dword PROT_READ | PROT_WRITE
push ecx ; entire file size
push edx ; let system decide on the address
sys.mmap
jc near memerr

mov edi, eax
mov esi, eax
```

```
push ecx ; for SYS_munmap
push edi

; Use EBX for state machine
mov ebx, ordinary
mov ah, 0Ah
cld

.loop:
lodsb
call ebx
loop .loop

cmp ebx, ordinary
je .filesize

; Output final lf
mov al, ah
stosb
inc edx

.filesize:
; truncate file to new size
push dword 0 ; high dword
push edx ; low dword
push eax ; pad
push ebp
sys.ftruncate

; close it (ebp still pushed)
sys.close

add esp, byte 16
sys.munmap

.quit:
push dword 0
sys.exit

align 4
ordinary:
cmp al, 0Dh
je .cr

cmp al, ah
je .lf

stosb
inc edx
ret

align 4
.cr:
```

```
mov ebx, cr
ret

align 4
.lf:
mov ebx, lf
ret

align 4
cr:
cmp al, 0Dh
je .cr

cmp al, ah
je .lf

xchg al, ah
stosb
inc edx

xchg al, ah
; fall through

.lf:
stosb
inc edx
mov ebx, ordinary
ret

align 4
.cr:
mov al, ah
stosb
inc edx
ret

align 4
lf:
cmp al, ah
je .lf

cmp al, 0Dh
je .cr

xchg al, ah
stosb
inc edx

xchg al, ah
stosb
inc edx
mov ebx, ordinary
ret
```

```
align 4
.cr:
mov ebx, ordinary
mov al, ah
; fall through

.lf:
stosb
inc edx
ret
```

> **警告:** Do not use this program on files stored on a disk formated by MS-DOS or Windows. There seems to be a subtle bug in the FreeBSD code when using `mmap` on these drives mounted under FreeBSD: If the file is over a certain size, `mmap` will just fill the memory with zeros, and then copy them to the file overwriting its contents.

11.12 One-Pointed Mind

As a student of Zen, I like the idea of a one-pointed mind: Do one thing at a time, and do it well.

This, indeed, is very much how UNIX works as well. While a typical Windows application is attempting to do everything imaginable (and is, therefore, riddled with bugs), a typical UNIX program does only one thing, and it does it well.

The typical UNIX user then essentially assembles his own applications by writing a shell script which combines the various existing programs by piping the output of one program to the input of another.

When writing your own UNIX software, it is generally a good idea to see what parts of the problem you need to solve can be handled by existing programs, and only write your own programs for that part of the problem that you do not have an existing solution for.

11.12.1 CSV

I will illustrate this principle with a specific real-life example I was faced with recently:

I needed to extract the 11th field of each record from a database I downloaded from a web site. The database was a CSV file, i.e., a list of *comma-separated values*. That is quite a standard format for sharing data among people who may be using different database software.

The first line of the file contains the list of various fields separated by commas. The rest of the file contains the data listed line by line, with values separated by commas.

I tried **awk**, using the comma as a separator. But because several lines contained a quoted comma, **awk** was extracting the wrong field from those lines.

Therefore, I needed to write my own software to extract the 11th field from the CSV file. However, going with the UNIX spirit, I only needed to write a simple filter that would do the following:

Remove the first line from the file;

Change all unquoted commas to a different character;

Remove all quotation marks.

Strictly speaking, I could use **sed** to remove the first line from the file, but doing so in my own program was very easy, so I decided to do it and reduce the size of the pipeline.

At any rate, writing a program like this took me about 20 minutes. Writing a program that extracts the 11th field from the CSV file would take a lot longer, and I could not reuse it to extract some other field from some other database.

This time I decided to let it do a little more work than a typical tutorial program would:

It parses its command line for options;

It displays proper usage if it finds wrong arguments;

It produces meaningful error messages.

Here is its usage message:

```
Usage: csv [-t<delim>] [-c<comma>] [-p] [-o <outfile>] [-i <infile>]
```

All parameters are optional, and can appear in any order.

The *-t* parameter declares what to replace the commas with. The `tab` is the default here. For example, *-t;* will replace all unquoted commas with semicolons.

I did not need the *-c* option, but it may come in handy in the future. It lets me declare that I want a character other than a comma replaced with something else. For example, *-c@* will replace all at signs (useful if you want to split a list of email addresses to their user names and domains).

The *-p* option preserves the first line, i.e., it does not delete it. By default, we delete the first line because in a CSV file it contains the field names rather than data.

The *-i* and *-o* options let me specify the input and the output files. Defaults are `stdin` and `stdout`, so this is a regular UNIX filter.

I made sure that both *-i filename* and *-ifilename* are accepted. I also made sure that only one input and one output files may be specified.

To get the 11th field of each record, I can now do:

```
% csv '-t;' data.csv | awk '-F;' '{print $11}'
```

The code stores the options (except for the file descriptors) in `EDX`: The comma in `DH`, the new separator in `DL`, and the flag for the *-p* option in the highest bit of `EDX`, so a check for its sign will give us a quick decision what to do.

Here is the code:

```
;;;;;;; csv.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Convert a comma-separated file to a something-else separated file.
;
; Started: 31-May-2001
; Updated:  1-Jun-2001
;
; Copyright (c) 2001 G. Adam Stanislav
```

```
; All rights reserved.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%include 'system.inc'

%define BUFSIZE 2048

section .data
fd.in dd stdin
fd.out dd stdout
usg db 'Usage: csv [-t<delim>] [-c<comma>] [-p] [-o <outfile>] [-i <infile>]', 0Ah
usglen equ $-usg
iemsg db "csv: Can't open input file", 0Ah
iemlen equ $-iemsg
oemsg db "csv: Can't create output file", 0Ah
oemlen equ $-oemsg

section .bss
ibuffer resb BUFSIZE
obuffer resb BUFSIZE

section .text
align 4
ierr:
push dword iemlen
push dword iemsg
push dword stderr
sys.write
push dword 1 ; return failure
sys.exit

align 4
oerr:
push dword oemlen
push dword oemsg
push dword stderr
sys.write
push dword 2
sys.exit

align 4
usage:
push dword usglen
push dword usg
push dword stderr
sys.write
push dword 3
sys.exit

align 4
global _start
_start:
```

```
add esp, byte 8 ; discard argc and argv[0]
mov edx, (',' << 8) | 9

.arg:
pop ecx
or ecx, ecx
je near .init ; no more arguments

; ECX contains the pointer to an argument
cmp byte [ecx], '-'
jne usage

inc ecx
mov ax, [ecx]

.o:
cmp al, 'o'
jne .i

; Make sure we are not asked for the output file twice
cmp dword [fd.out], stdout
jne usage

; Find the path to output file - it is either at [ECX+1],
; i.e., -ofile --
; or in the next argument,
; i.e., -o file

inc ecx
or ah, ah
jne .openoutput
pop ecx
jecxz usage

.openoutput:
push dword 420 ; file mode (644 octal)
push dword 0200h | 0400h | 01h
; O_CREAT | O_TRUNC | O_WRONLY
push ecx
sys.open
jc near oerr

add esp, byte 12
mov [fd.out], eax
jmp short .arg

.i:
cmp al, 'i'
jne .p

; Make sure we are not asked twice
cmp dword [fd.in], stdin
jne near usage
```

```
; Find the path to the input file
inc ecx
or ah, ah
jne .openinput
pop ecx
or ecx, ecx
je near usage

.openinput:
push dword 0 ; O_RDONLY
push ecx
sys.open
jc near ierr ; open failed

add esp, byte 8
mov [fd.in], eax
jmp .arg

.p:
cmp al, 'p'
jne .t
or ah, ah
jne near usage
or edx, 1 << 31
jmp .arg

.t:
cmp al, 't' ; redefine output delimiter
jne .c
or ah, ah
je near usage
mov dl, ah
jmp .arg

.c:
cmp al, 'c'
jne near usage
or ah, ah
je near usage
mov dh, ah
jmp .arg

align 4
.init:
sub eax, eax
sub ebx, ebx
sub ecx, ecx
mov edi, obuffer

; See if we are to preserve the first line
or edx, edx
js .loop
```

```
.firstline:
; get rid of the first line
call getchar
cmp al, 0Ah
jne .firstline

.loop:
; read a byte from stdin
call getchar

; is it a comma (or whatever the user asked for)?
cmp al, dh
jne .quote

; Replace the comma with a tab (or whatever the user wants)
mov al, dl

.put:
call putchar
jmp short .loop

.quote:
cmp al, '"'
jne .put

; Print everything until you get another quote or EOL. If it
; is a quote, skip it. If it is EOL, print it.
.qloop:
call getchar
cmp al, '"'
je .loop

cmp al, 0Ah
je .put

call putchar
jmp short .qloop

align 4
getchar:
or ebx, ebx
jne .fetch

call read

.fetch:
lodsb
dec ebx
ret

read:
jecxz .read
```

```
call write

.read:
push dword BUFSIZE
mov esi, ibuffer
push esi
push dword [fd.in]
sys.read
add esp, byte 12
mov ebx, eax
or eax, eax
je .done
sub eax, eax
ret

align 4
.done:
call write ; flush output buffer

; close files
push dword [fd.in]
sys.close

push dword [fd.out]
sys.close

; return success
push dword 0
sys.exit

align 4
putchar:
stosb
inc ecx
cmp ecx, BUFSIZE
je write
ret

align 4
write:
jecxz .ret ; nothing to write
sub edi, ecx ; start of buffer
push ecx
push edi
push dword [fd.out]
sys.write
add esp, byte 12
sub eax, eax
sub ecx, ecx ; buffer is empty now
.ret:
ret
```

Much of it is taken from `hex.asm` above. But there is one important difference: I no longer call `write` whenever I am outputting a line feed. Yet, the code can be used interactively.

I have found a better solution for the interactive problem since I first started writing this chapter. I wanted to make sure each line is printed out separately only when needed. After all, there is no need to flush out every line when used non-interactively.

The new solution I use now is to call `write` every time I find the input buffer empty. That way, when running in the interactive mode, the program reads one line from the user's keyboard, processes it, and sees its input buffer is empty. It flushes its output and reads the next line.

11.12.1.1 The Dark Side of Buffering

This change prevents a mysterious lockup in a very specific case. I refer to it as the *dark side of buffering*, mostly because it presents a danger that is not quite obvious.

It is unlikely to happen with a program like the **csv** above, so let us consider yet another filter: In this case we expect our input to be raw data representing color values, such as the *red*, *green*, and *blue* intensities of a pixel. Our output will be the negative of our input.

Such a filter would be very simple to write. Most of it would look just like all the other filters we have written so far, so I am only going to show you its inner loop:

```
.loop:
call getchar
not al ; Create a negative
call putchar
jmp short .loop
```

Because this filter works with raw data, it is unlikely to be used interactively.

But it could be called by image manipulation software. And, unless it calls `write` before each call to `read`, chances are it will lock up.

Here is what might happen:

1. The image editor will load our filter using the C function `popen()`.
2. It will read the first row of pixels from a bitmap or pixmap.
3. It will write the first row of pixels to the *pipe* leading to the `fd.in` of our filter.
4. Our filter will read each pixel from its input, turn it to a negative, and write it to its output buffer.
5. Our filter will call `getchar` to fetch the next pixel.
6. `getchar` will find an empty input buffer, so it will call `read`.
7. `read` will call the `SYS_read` system call.
8. The *kernel* will suspend our filter until the image editor sends more data to the pipe.
9. The image editor will read from the other pipe, connected to the `fd.out` of our filter so it can set the first row of the output image *before* it sends us the second row of the input.
10. The *kernel* suspends the image editor until it receives some output from our filter, so it can pass it on to the image editor.

At this point our filter waits for the image editor to send it more data to process, while the image editor is waiting for our filter to send it the result of the processing of the first row. But the result sits in our output buffer.

The filter and the image editor will continue waiting for each other forever (or, at least, until they are killed). Our software has just entered a race condition.

This problem does not exist if our filter flushes its output buffer *before* asking the *kernel* for more input data.

11.13 Using the FPU

Strangely enough, most of assembly language literature does not even mention the existence of the FPU, or *floating point unit*, let alone discuss programming it.

Yet, never does assembly language shine more than when we create highly optimized FPU code by doing things that can be done *only* in assembly language.

11.13.1 Organization of the FPU

The FPU consists of 8 80–bit floating–point registers. These are organized in a stack fashion——you can `push` a value on TOS (*top of stack*) and you can `pop` it.

That said, the assembly language op codes are not `push` and `pop` because those are already taken.

You can `push` a value on TOS by using `fld`, `fild`, and `fbld`. Several other op codes let you `push` many common *constants*——such as *pi*——on the TOS.

Similarly, you can `pop` a value by using `fst`, `fstp`, `fist`, `fistp`, and `fbstp`. Actually, only the op codes that end with a *p* will literally `pop` the value, the rest will `store` it somewhere else without removing it from the TOS.

We can transfer the data between the TOS and the computer memory either as a 32–bit, 64–bit, or 80–bit *real*, a 16–bit, 32–bit, or 64–bit *integer*, or an 80–bit *packed decimal*.

The 80–bit *packed decimal* is a special case of *binary coded decimal* which is very convenient when converting between the ASCII representation of data and the internal data of the FPU. It allows us to use 18 significant digits.

No matter how we represent data in the memory, the FPU always stores it in the 80–bit *real* format in its registers.

Its internal precision is at least 19 decimal digits, so even if we choose to display results as ASCII in the full 18–digit precision, we are still showing correct results.

We can perform mathematical operations on the TOS: We can calculate its *sine*, we can *scale* it (i.e., we can multiply or divide it by a power of 2), we can calculate its base–2 *logarithm*, and many other things.

We can also *multiply* or *divide* it by, *add* it to, or *subtract* it from, any of the FPU registers (including itself).

The official Intel op code for the TOS is `st`, and for the *registers* `st(0)–st(7)`. `st` and `st(0)`, then, refer to the same register.

For whatever reasons, the original author of **nasm** has decided to use different op codes, namely `st0–st7`. In other words, there are no parentheses, and the TOS is always `st0`, never just `st`.

11.13.1.1 The Packed Decimal Format

The *packed decimal* format uses 10 bytes (80 bits) of memory to represent 18 digits. The number represented there is always an *integer*.

提示: You can use it to get decimal places by multiplying the TOS by a power of 10 first.

The highest bit of the highest byte (byte 9) is the *sign bit*: If it is set, the number is *negative*, otherwise, it is *positive*. The rest of the bits of this byte are unused/ignored.

The remaining 9 bytes store the 18 digits of the number: 2 digits per byte.

The *more significant digit* is stored in the high *nibble* (4 bits), the *less significant digit* in the low *nibble*.

That said, you might think that `-1234567` would be stored in the memory like this (using hexadecimal notation):

```
80 00 00 00 00 00 01 23 45 67
```

Alas it is not! As with everything else of Intel make, even the *packed decimal* is *little–endian*.

That means our `-1234567` is stored like this:

```
67 45 23 01 00 00 00 00 00 80
```

Remember that, or you will be pulling your hair out in desperation!

注意: The book to read——if you can find it——is Richard Startz' 8087/80287/80387 for the IBM PC & Compatibles (http://www.int80h.org/cgi-bin/isbn?isbn=013246604X). Though it does seem to take the fact about the little–endian storage of the *packed decimal* for granted. I kid you not about the desperation of trying to figure out what was wrong with the filter I show below *before* it occurred to me I should try the little–endian order even for this type of data.

11.13.2 Excursion to Pinhole Photography

To write meaningful software, we must not only understand our programming tools, but also the field we are creating software for.

Our next filter will help us whenever we want to build a *pinhole camera*, so, we need some background in *pinhole photography* before we can continue.

11.13.2.1 The Camera

The easiest way to describe any camera ever built is as some empty space enclosed in some lightproof material, with a small hole in the enclosure.

The enclosure is usually sturdy (e.g., a box), though sometimes it is flexible (the bellows). It is quite dark inside the camera. However, the hole lets light rays in through a single point (though in some cases there may be several). These light rays form an image, a representation of whatever is outside the camera, in front of the hole.

If some light sensitive material (such as film) is placed inside the camera, it can capture the image.

The hole often contains a *lens*, or a lens assembly, often called the *objective*.

11.13.2.2 The Pinhole

But, strictly speaking, the lens is not necessary: The original cameras did not use a lens but a *pinhole*. Even today, *pinholes* are used, both as a tool to study how cameras work, and to achieve a special kind of image.

The image produced by the *pinhole* is all equally sharp. Or *blurred*. There is an ideal size for a pinhole: If it is either larger or smaller, the image loses its sharpness.

11.13.2.3 Focal Length

This ideal pinhole diameter is a function of the square root of *focal length*, which is the distance of the pinhole from the film.

```
D = PC * sqrt(FL)
```

In here, `D` is the ideal diameter of the pinhole, `FL` is the focal length, and `PC` is a pinhole constant. According to Jay Bender, its value is `0.04`, while Kenneth Connors has determined it to be `0.037`. Others have proposed other values. Plus, this value is for the daylight only: Other types of light will require a different constant, whose value can only be determined by experimentation.

11.13.2.4 The F–Number

The f–number is a very useful measure of how much light reaches the film. A light meter can determine that, for example, to expose a film of specific sensitivity with f5.6 may require the exposure to last 1/1000 sec.

It does not matter whether it is a 35–mm camera, or a 6x9cm camera, etc. As long as we know the f–number, we can determine the proper exposure.

The f–number is easy to calculate:

```
F = FL / D
```

In other words, the f–number equals the focal length divided by the diameter of the pinhole. It also means a higher f–number either implies a smaller pinhole or a larger focal distance, or both. That, in turn, implies, the higher the f–number, the longer the exposure has to be.

Furthermore, while pinhole diameter and focal distance are one–dimensional measurements, both, the film and the pinhole, are two–dimensional. That means that if you have measured the exposure at f–number `A` as `t`, then the exposure at f–number `B` is:

```
t * (B / A)?
```

11.13.2.5 Normalized F–Number

While many modern cameras can change the diameter of their pinhole, and thus their f–number, quite smoothly and gradually, such was not always the case.

To allow for different f–numbers, cameras typically contained a metal plate with several holes of different sizes drilled to them.

Their sizes were chosen according to the above formula in such a way that the resultant f–number was one of standard f–numbers used on all cameras everywhere. For example, a very old Kodak Duaflex IV camera in my possession has three such holes for f–numbers 8, 11, and 16.

A more recently made camera may offer f–numbers of 2.8, 4, 5.6, 8, 11, 16, 22, and 32 (as well as others). These numbers were not chosen arbitrarily: They all are powers of the square root of 2, though they may be rounded somewhat.

11.13.2.6 The F–Stop

A typical camera is designed in such a way that setting any of the normalized f–numbers changes the feel of the dial. It will naturally *stop* in that position. Because of that, these positions of the dial are called f–stops.

Since the f–numbers at each stop are powers of the square root of 2, moving the dial by 1 stop will double the amount of light required for proper exposure. Moving it by 2 stops will quadruple the required exposure. Moving the dial by 3 stops will require the increase in exposure 8 times, etc.

11.13.3 Designing the Pinhole Software

We are now ready to decide what exactly we want our pinhole software to do.

11.13.3.1 Processing Program Input

Since its main purpose is to help us design a working pinhole camera, we will use the *focal length* as the input to the program. This is something we can determine without software: Proper focal length is determined by the size of the film and by the need to shoot "regular" pictures, wide angle pictures, or telephoto pictures.

Most of the programs we have written so far worked with individual characters, or bytes, as their input: The **hex** program converted individual bytes into a hexadecimal number, the **csv** program either let a character through, or deleted it, or changed it to a different character, etc.

One program, **ftuc** used the state machine to consider at most two input bytes at a time.

But our **pinhole** program cannot just work with individual characters, it has to deal with larger syntactic units.

For example, if we want the program to calculate the pinhole diameter (and other values we will discuss later) at the focal lengths of `100 mm`, `150 mm`, and `210 mm`, we may want to enter something like this:

```
100, 150, 210
```

Our program needs to consider more than a single byte of input at a time. When it sees the first `1`, it must understand it is seeing the first digit of a decimal number. When it sees the `0` and the other `0`, it must know it is seeing more digits of the same number.

When it encounters the first comma, it must know it is no longer receiving the digits of the first number. It must be able to convert the digits of the first number into the value of `100`. And the digits of the second number into the value of `150`. And, of course, the digits of the third number into the numeric value of `210`.

We need to decide what delimiters to accept: Do the input numbers have to be separated by a comma? If so, how do we treat two numbers separated by something else?

Personally, I like to keep it simple. Something either is a number, so I process it. Or it is not a number, so I discard it. I do not like the computer complaining about me typing in an extra character when it is *obvious* that it is an extra character. Duh!

Plus, it allows me to break up the monotony of computing and type in a query instead of just a number:

```
What is the best pinhole diameter for the focal length of 150?
```

There is no reason for the computer to spit out a number of complaints:

```
Syntax error: What
Syntax error: is
Syntax error: the
Syntax error: best
```

Et cetera, et cetera, et cetera.

Secondly, I like the `#` character to denote the start of a comment which extends to the end of the line. This does not take too much effort to code, and lets me treat input files for my software as executable scripts.

In our case, we also need to decide what units the input should come in: We choose *millimeters* because that is how most photographers measure the focus length.

Finally, we need to decide whether to allow the use of the decimal point (in which case we must also consider the fact that much of the world uses a decimal *comma*).

In our case allowing for the decimal point/comma would offer a false sense of precision: There is little if any noticeable difference between the focus lengths of `50` and `51`, so allowing the user to input something like `50.5` is not a good idea. This is my opinion, mind you, but I am the one writing this program. You can make other choices in yours, of course.

11.13.3.2 Offering Options

The most important thing we need to know when building a pinhole camera is the diameter of the pinhole. Since we want to shoot sharp images, we will use the above formula to calculate the pinhole diameter from focal length. As experts are offering several different values for the `PC` constant, we will need to have the choice.

It is traditional in UNIX programming to have two main ways of choosing program parameters, plus to have a default for the time the user does not make a choice.

Why have two ways of choosing?

One is to allow a (relatively) *permanent* choice that applies automatically each time the software is run without us having to tell it over and over what we want it to do.

The permanent choices may be stored in a configuration file, typically found in the user's home directory. The file usually has the same name as the application but is started with a dot. Often *"rc"* is added to the file name. So, ours could be `~/.pinhole` or `~/.pinholerc`. (The `~/` means current user's home directory.)

The configuration file is used mostly by programs that have many configurable parameters. Those that have only one (or a few) often use a different method: They expect to find the parameter in an *environment variable*. In our case, we might look at an environment variable named `PINHOLE`.

Usually, a program uses one or the other of the above methods. Otherwise, if a configuration file said one thing, but an environment variable another, the program might get confused (or just too complicated).

Because we only need to choose *one* such parameter, we will go with the second method and search the environment for a variable named `PINHOLE`.

The other way allows us to make *ad hoc* decisions: *"Though I usually want you to use 0.039, this time I want 0.03872."* In other words, it allows us to *override* the permanent choice.

This type of choice is usually done with command line parameters.

Finally, a program *always* needs a *default*. The user may not make any choices. Perhaps he does not know what to choose. Perhaps he is "just browsing." Preferably, the default will be the value most users would choose anyway. That way they do not need to choose. Or, rather, they can choose the default without an additional effort.

Given this system, the program may find conflicting options, and handle them this way:

1. If it finds an *ad hoc* choice (e.g., command line parameter), it should accept that choice. It must ignore any permanent choice and any default.
2. *Otherwise*, if it finds a permanent option (e.g., an environment variable), it should accept it, and ignore the default.
3. *Otherwise*, it should use the default.

We also need to decide what *format* our `PC` option should have.

At first site, it seems obvious to use the `PINHOLE=0.04` format for the environment variable, and *`-p0.04`* for the command line.

Allowing that is actually a security risk. The `PC` constant is a very small number. Naturally, we will test our software using various small values of `PC`. But what will happen if someone runs the program choosing a huge value?

It may crash the program because we have not designed it to handle huge numbers.

Or, we may spend more time on the program so it can handle huge numbers. We might do that if we were writing commercial software for computer illiterate audience.

Or, we might say, *"Tough! The user should know better.""*

Or, we just may make it impossible for the user to enter a huge number. This is the approach we will take: We will use an *implied 0.* prefix.

In other words, if the user wants `0.04`, we will expect him to type *`-p04`*, or set `PINHOLE=04` in his environment. So, if he says *`-p9999999`*, we will interpret it as `0.9999999`——still ridiculous but at least safer.

Secondly, many users will just want to go with either Bender's constant or Connors' constant. To make it easier on them, we will interpret *`-b`* as identical to *`-p04`*, and *`-c`* as identical to *`-p037`*.

11.13.3.3 The Output

We need to decide what we want our software to send to the output, and in what format.

Since our input allows for an unspecified number of focal length entries, it makes sense to use a traditional database–style output of showing the result of the calculation for each focal length on a separate line, while separating all values on one line by a `tab` character.

Optionally, we should also allow the user to specify the use of the CSV format we have studied earlier. In this case, we will print out a line of comma–separated names describing each field of every line, then show our results as before, but substituting a `comma` for the `tab`.

We need a command line option for the CSV format. We cannot use `-c` because that already means *use Connors' constant*. For some strange reason, many web sites refer to CSV files as *"Excel spreadsheet"* (though the CSV format predates Excel). We will, therefore, use the `-e` switch to inform our software we want the output in the CSV format.

We will start each line of the output with the focal length. This may sound repetitious at first, especially in the interactive mode: The user types in the focal length, and we are repeating it.

But the user can type several focal lengths on one line. The input can also come in from a file or from the output of another program. In that case the user does not see the input at all.

By the same token, the output can go to a file which we will want to examine later, or it could go to the printer, or become the input of another program.

So, it makes perfect sense to start each line with the focal length as entered by the user.

No, wait! Not as entered by the user. What if the user types in something like this:

```
00000000150
```

Clearly, we need to strip those leading zeros.

So, we might consider reading the user input as is, converting it to binary inside the FPU, and printing it out from there.

But...

What if the user types something like this:

```
1745976572345235345353453535353053053456350730967676442 3
```

Ha! The packed decimal FPU format lets us input 18–digit numbers. But the user has entered more than 18 digits. How do we handle that?

Well, we *could* modify our code to read the first 18 digits, enter it to the FPU, then read more, multiply what we already have on the TOS by 10 raised to the number of additional digits, then `add` to it.

Yes, we could do that. But in *this* program it would be ridiculous (in a different one it may be just the thing to do): Even the circumference of the Earth expressed in millimeters only takes 11 digits. Clearly, we cannot build a camera that large (not yet, anyway).

So, if the user enters such a huge number, he is either bored, or testing us, or trying to break into the system, or playing games——doing anything but designing a pinhole camera.

What will we do?

We will slap him in the face, in a manner of speaking:

```
1745976572345235345353453535353053053456350730967676442 3 ??? ??? ??? ??? ???
```

To achieve that, we will simply ignore any leading zeros. Once we find a non–zero digit, we will initialize a counter to `0` and start taking three steps:

1. Send the digit to the output.

2. Append the digit to a buffer we will use later to produce the packed decimal we can send to the FPU.
3. Increase the counter.

Now, while we are taking these three steps, we also need to watch out for one of two conditions:

If the counter grows above 18, we stop appending to the buffer. We continue reading the digits and sending them to the output.

If, or rather *when*, the next input character is not a digit, we are done inputting for now.

Incidentally, we can simply discard the non–digit, unless it is a `#`, which we must return to the input stream. It starts a comment, so we must see it after we are done producing output and start looking for more input.

That still leaves one possibility uncovered: If all the user enters is a zero (or several zeros), we will never find a non–zero to display.

We can determine this has happened whenever our counter stays at `0`. In that case we need to send `0` to the output, and perform another "slap in the face":

```
0 ??? ??? ??? ??? ???
```

Once we have displayed the focal length and determined it is valid (greater than `0` but not exceeding 18 digits), we can calculate the pinhole diameter.

It is not by coincidence that *pinhole* contains the word *pin*. Indeed, many a pinhole literally is a *pin hole*, a hole carefully punched with the tip of a pin.

That is because a typical pinhole is very small. Our formula gets the result in millimeters. We will multiply it by `1000`, so we can output the result in *microns*.

At this point we have yet another trap to face: *Too much precision.*

Yes, the FPU was designed for high precision mathematics. But we are not dealing with high precision mathematics. We are dealing with physics (optics, specifically).

Suppose we want to convert a truck into a pinhole camera (we would not be the first ones to do that!). Suppose its box is `12` meters long, so we have the focal length of `12000`. Well, using Bender's constant, it gives us square root of `12000` multiplied by `0.04`, which is `4.381780460` millimeters, or `4381.780460` microns.

Put either way, the result is absurdly precise. Our truck is not *exactly* `12000` millimeters long. We did not measure its length with such a precision, so stating we need a pinhole with the diameter of `4.381780460` millimeters is, well, deceiving. `4.4` millimeters would do just fine.

注意: I "only" used ten digits in the above example. Imagine the absurdity of going for all 18!

We need to limit the number of significant digits of our result. One way of doing it is by using an integer representing microns. So, our truck would need a pinhole with the diameter of `4382` microns. Looking at that number, we still decide that `4400` microns, or `4.4` millimeters is close enough.

Additionally, we can decide that no matter how big a result we get, we only want to display four significant digits (or any other number of them, of course). Alas, the FPU does not offer rounding to a specific number of digits (after all, it does not view the numbers as decimal but as binary).

We, therefore, must devise an algorithm to reduce the number of significant digits.

Here is mine (I think it is awkward——if you know a better one, *please*, let me know):

1. Initialize a counter to `0`.
2. While the number is greater than or equal to `10000`, divide it by `10` and increase the counter.
3. Output the result.
4. While the counter is greater than `0`, output `0` and decrease the counter.

> **注意:** The `10000` is only good if you want *four* significant digits. For any other number of significant digits, replace `10000` with `10` raised to the number of significant digits.

We will, then, output the pinhole diameter in microns, rounded off to four significant digits.

At this point, we know the *focal length* and the *pinhole diameter*. That means we have enough information to also calculate the *f–number*.

We will display the f–number, rounded to four significant digits. Chances are the f–number will tell us very little. To make it more meaningful, we can find the nearest *normalized f–number*, i.e., the nearest power of the square root of 2.

We do that by multiplying the actual f–number by itself, which, of course, will give us its `square`. We will then calculate its base–2 logarithm, which is much easier to do than calculating the base–square–root–of–2 logarithm! We will round the result to the nearest integer. Next, we will raise 2 to the result. Actually, the FPU gives us a good shortcut to do that: We can use the `fscale` op code to "scale" 1, which is analogous to `shift`ing an integer left. Finally, we calculate the square root of it all, and we have the nearest normalized f–number.

If all that sounds overwhelming——or too much work, perhaps——it may become much clearer if you see the code. It takes 9 op codes altogether:

```
fmul st0, st0
fld1
fld st1
fyl2x
frndint
fld1
fscale
fsqrt
fstp st1
```

The first line, `fmul st0, st0`, squares the contents of the TOS (top of the stack, same as `st`, called `st0` by **nasm**). The `fld1` pushes `1` on the TOS.

The next line, `fld st1`, pushes the square back to the TOS. At this point the square is both in `st` and `st(2)` (it will become clear why we leave a second copy on the stack in a moment). `st(1)` contains `1`.

Next, `fyl2x` calculates base–2 logarithm of `st` multiplied by `st(1)`. That is why we placed `1` on `st(1)` before.

At this point, `st` contains the logarithm we have just calculated, `st(1)` contains the square of the actual f–number we saved for later.

`frndint` rounds the TOS to the nearest integer. `fld1` pushes a `1`. `fscale` shifts the `1` we have on the TOS by the value in `st(1)`, effectively raising 2 to `st(1)`.

Finally, `fsqrt` calculates the square root of the result, i.e., the nearest normalized f–number.

We now have the nearest normalized f–number on the TOS, the base–2 logarithm rounded to the nearest integer in `st(1)`, and the square of the actual f–number in `st(2)`. We are saving the value in `st(2)` for later.

But we do not need the contents of `st(1)` anymore. The last line, `fstp st1`, places the contents of `st` to `st(1)`, and pops. As a result, what was `st(1)` is now `st`, what was `st(2)` is now `st(1)`, etc. The new `st` contains the normalized f–number. The new `st(1)` contains the square of the actual f–number we have stored there for posterity.

At this point, we are ready to output the normalized f–number. Because it is normalized, we will not round it off to four significant digits, but will send it out in its full precision.

The normalized f-number is useful as long as it is reasonably small and can be found on our light meter. Otherwise we need a different method of determining proper exposure.

Earlier we have figured out the formula of calculating proper exposure at an arbitrary f–number from that measured at a different f–number.

Every light meter I have ever seen can determine proper exposure at f5.6. We will, therefore, calculate an *"f5.6 multiplier,"* i.e., by how much we need to multiply the exposure measured at f5.6 to determine the proper exposure for our pinhole camera.

From the above formula we know this factor can be calculated by dividing our f–number (the actual one, not the normalized one) by `5.6`, and squaring the result.

Mathematically, dividing the square of our f–number by the square of `5.6` will give us the same result.

Computationally, we do not want to square two numbers when we can only square one. So, the first solution seems better at first.

But...

`5.6` is a *constant*. We do not have to have our FPU waste precious cycles. We can just tell it to divide the square of the f–number by whatever `5.6?` equals to. Or we can divide the f–number by `5.6`, and then square the result. The two ways now seem equal.

But, they are not!

Having studied the principles of photography above, we remember that the `5.6` is actually square root of 2 raised to the fifth power. An *irrational* number. The square of this number is *exactly* `32`.

Not only is `32` an integer, it is a power of 2. We do not need to divide the square of the f–number by `32`. We only need to use `fscale` to shift it right by five positions. In the FPU lingo it means we will `fscale` it with `st(1)` equal to `-5`. That is *much faster* than a division.

So, now it has become clear why we have saved the square of the f–number on the top of the FPU stack. The calculation of the f5.6 multiplier is the easiest calculation of this entire program! We will output it rounded to four significant digits.

There is one more useful number we can calculate: The number of stops our f–number is from f5.6. This may help us if our f–number is just outside the range of our light meter, but we have a shutter which lets us set various speeds, and this shutter uses stops.

Say, our f–number is 5 stops from f5.6, and the light meter says we should use 1/1000 sec. Then we can set our shutter speed to 1/1000 first, then move the dial by 5 stops.

This calculation is quite easy as well. All we have to do is to calculate the base-2 logarithm of the f5.6 multiplier we had just calculated (though we need its value from before we rounded it off). We then output the result

rounded to the nearest integer. We do not need to worry about having more than four significant digits in this one: The result is most likely to have only one or two digits anyway.

11.13.4 FPU Optimizations

In assembly language we can optimize the FPU code in ways impossible in high languages, including C.

Whenever a C function needs to calculate a floating–point value, it loads all necessary variables and constants into FPU registers. It then does whatever calculation is required to get the correct result. Good C compilers can optimize that part of the code really well.

It "returns" the value by leaving the result on the TOS. However, before it returns, it cleans up. Any variables and constants it used in its calculation are now gone from the FPU.

It cannot do what we just did above: We calculated the square of the f–number and kept it on the stack for later use by another function.

We *knew* we would need that value later on. We also knew we had enough room on the stack (which only has room for 8 numbers) to store it there.

A C compiler has no way of knowing that a value it has on the stack will be required again in the very near future.

Of course, the C programmer may know it. But the only recourse he has is to store the value in a memory variable.

That means, for one, the value will be changed from the 80-bit precision used internally by the FPU to a C *double* (64 bits) or even *single* (32 bits).

That also means that the value must be moved from the TOS into the memory, and then back again. Alas, of all FPU operations, the ones that access the computer memory are the slowest.

So, whenever programming the FPU in assembly language, look for the ways of keeping intermediate results on the FPU stack.

We can take that idea even further! In our program we are using a *constant* (the one we named `PC`).

It does not matter how many pinhole diameters we are calculating: 1, 10, 20, 1000, we are always using the same constant. Therefore, we can optimize our program by keeping the constant on the stack all the time.

Early on in our program, we are calculating the value of the above constant. We need to divide our input by `10` for every digit in the constant.

It is much faster to multiply than to divide. So, at the start of our program, we divide `10` into `1` to obtain `0.1`, which we then keep on the stack: Instead of dividing the input by `10` for every digit, we multiply it by `0.1`.

By the way, we do not input `0.1` directly, even though we could. We have a reason for that: While `0.1` can be expressed with just one decimal place, we do not know how many *binary* places it takes. We, therefore, let the FPU calculate its binary value to its own high precision.

We are using other constants: We multiply the pinhole diameter by `1000` to convert it from millimeters to microns. We compare numbers to `10000` when we are rounding them off to four significant digits. So, we keep both, `1000` and `10000`, on the stack. And, of course, we reuse the `0.1` when rounding off numbers to four digits.

Last but not least, we keep `-5` on the stack. We need it to scale the square of the f–number, instead of dividing it by `32`. It is not by coincidence we load this constant last. That makes it the top of the stack when only the constants

are on it. So, when the square of the f–number is being scaled, the `-5` is at `st(1)`, precisely where `fscale` expects it to be.

It is common to create certain constants from scratch instead of loading them from the memory. That is what we are doing with `-5`:

```
fld1 ; TOS =  1
fadd st0, st0 ; TOS =  2
fadd st0, st0 ; TOS =  4
fld1 ; TOS =  1
faddp st1, st0 ; TOS =  5
fchs ; TOS = -5
```

We can generalize all these optimizations into one rule: *Keep repeat values on the stack!*

提示: *PostScript®* is a stack–oriented programming language. There are many more books available about PostScript than about the FPU assembly language: Mastering PostScript will help you master the FPU.

11.13.5 pinhole——The Code

```
;;;;;;; pinhole.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Find various parameters of a pinhole camera construction and use
;
; Started:  9-Jun-2001
; Updated: 10-Jun-2001
;
; Copyright (c) 2001 G. Adam Stanislav
; All rights reserved.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%include 'system.inc'

%define BUFSIZE 2048

section .data
align 4
ten dd 10
thousand dd 1000
tthou dd 10000
fd.in dd stdin
fd.out dd stdout
envar db 'PINHOLE=' ; Exactly 8 bytes, or 2 dwords long
pinhole db '04,',  ; Bender's constant (0.04)
connors db '037', 0Ah ; Connors' constant
usg db 'Usage: pinhole [-b] [-c] [-e] [-p <value>] [-o <outfile>] [-i <infile>]', 0Ah
usglen equ $-usg
iemsg db "pinhole: Can't open input file", 0Ah
iemlen equ $-iemsg
```

```
oemsg db "pinhole: Can't create output file", 0Ah
oemlen equ $-oemsg
pinmsg db "pinhole: The PINHOLE constant must not be 0", 0Ah
pinlen equ $-pinmsg
toobig db "pinhole: The PINHOLE constant may not exceed 18 decimal places", 0Ah
biglen equ $-toobig
huhmsg db 9, '???'
separ db 9, '???'
sep2 db 9, '???'
sep3 db 9, '???'
sep4 db 9, '???', 0Ah
huhlen equ $-huhmsg
header db 'focal length in millimeters,pinhole diameter in microns,'
db 'F-number,normalized F-number,F-5.6 multiplier,stops '
db 'from F-5.6', 0Ah
headlen equ $-header

section .bss
ibuffer resb BUFSIZE
obuffer resb BUFSIZE
dbuffer resb 20 ; decimal input buffer
bbuffer resb 10 ; BCD buffer

section .text
align 4
huh:
call write
push dword huhlen
push dword huhmsg
push dword [fd.out]
sys.write
add esp, byte 12
ret

align 4
perr:
push dword pinlen
push dword pinmsg
push dword stderr
sys.write
push dword 4 ; return failure
sys.exit

align 4
consttoobig:
push dword biglen
push dword toobig
push dword stderr
sys.write
push dword 5 ; return failure
sys.exit

align 4
```

```
ierr:
push dword iemlen
push dword iemsg
push dword stderr
sys.write
push dword 1 ; return failure
sys.exit

align 4
oerr:
push dword oemlen
push dword oemsg
push dword stderr
sys.write
push dword 2
sys.exit

align 4
usage:
push dword usglen
push dword usg
push dword stderr
sys.write
push dword 3
sys.exit

align 4
global _start
_start:
add esp, byte 8 ; discard argc and argv[0]
sub esi, esi

.arg:
pop ecx
or ecx, ecx
je near .getenv ; no more arguments

; ECX contains the pointer to an argument
cmp byte [ecx], '-'
jne usage

inc ecx
mov ax, [ecx]
inc ecx

.o:
cmp al, 'o'
jne .i

; Make sure we are not asked for the output file twice
cmp dword [fd.out], stdout
jne usage
```

```
; Find the path to output file - it is either at [ECX+1],
; i.e., -ofile --
; or in the next argument,
; i.e., -o file

or ah, ah
jne .openoutput
pop ecx
jecxz usage

.openoutput:
push dword 420 ; file mode (644 octal)
push dword 0200h | 0400h | 01h
; O_CREAT | O_TRUNC | O_WRONLY
push ecx
sys.open
jc near oerr

add esp, byte 12
mov [fd.out], eax
jmp short .arg

.i:
cmp al, 'i'
jne .p

; Make sure we are not asked twice
cmp dword [fd.in], stdin
jne near usage

; Find the path to the input file
or ah, ah
jne .openinput
pop ecx
or ecx, ecx
je near usage

.openinput:
push dword 0 ; O_RDONLY
push ecx
sys.open
jc near ierr ; open failed

add esp, byte 8
mov [fd.in], eax
jmp .arg

.p:
cmp al, 'p'
jne .c
or ah, ah
jne .pcheck
```

```
pop ecx
or ecx, ecx
je near usage

mov ah, [ecx]

.pcheck:
cmp ah, '0'
jl near usage
cmp ah, '9'
ja near usage
mov esi, ecx
jmp .arg

.c:
cmp al, 'c'
jne .b
or ah, ah
jne near usage
mov esi, connors
jmp .arg

.b:
cmp al, 'b'
jne .e
or ah, ah
jne near usage
mov esi, pinhole
jmp .arg

.e:
cmp al, 'e'
jne near usage
or ah, ah
jne near usage
mov al, ','
mov [huhmsg], al
mov [separ], al
mov [sep2], al
mov [sep3], al
mov [sep4], al
jmp .arg

align 4
.getenv:
; If ESI = 0, we did not have a -p argument,
; and need to check the environment for "PINHOLE="
or esi, esi
jne .init

sub ecx, ecx

.nextenv:
```

```
pop esi
or esi, esi
je .default ; no PINHOLE envar found

; check if this envar starts with 'PINHOLE='
mov edi, envar
mov cl, 2 ; 'PINHOLE=' is 2 dwords long
rep cmpsd
jne .nextenv

; Check if it is followed by a digit
mov al, [esi]
cmp al, '0'
jl .default
cmp al, '9'
jbe .init
; fall through

align 4
.default:
; We got here because we had no -p argument,
; and did not find the PINHOLE envar.
mov esi, pinhole
; fall through

align 4
.init:
sub eax, eax
sub ebx, ebx
sub ecx, ecx
sub edx, edx
mov edi, dbuffer+1
mov byte [dbuffer], '0'

; Convert the pinhole constant to real
.constloop:
lodsb
cmp al, '9'
ja .setconst
cmp al, '0'
je .processconst
jb .setconst

inc dl

.processconst:
inc cl
cmp cl, 18
ja near consttoobig
stosb
jmp short .constloop

align 4
```

```
.setconst:
or dl, dl
je near perr

finit
fild dword [tthou]

fld1
fild dword [ten]
fdivp st1, st0

fild dword [thousand]
mov edi, obuffer

mov ebp, ecx
call bcdload

.constdiv:
fmul st0, st2
loop .constdiv

fld1
fadd st0, st0
fadd st0, st0
fld1
faddp st1, st0
fchs

; If we are creating a CSV file,
; print header
cmp byte [separ], ','
jne .bigloop

push dword headlen
push dword header
push dword [fd.out]
sys.write

.bigloop:
call getchar
jc near done

; Skip to the end of the line if you got '#'
cmp al, '#'
jne .num
call skiptoeol
jmp short .bigloop

.num:
; See if you got a number
cmp al, '0'
jl .bigloop
cmp al, '9'
```

```
ja .bigloop

; Yes, we have a number
sub ebp, ebp
sub edx, edx

.number:
cmp al, '0'
je .number0
mov dl, 1

.number0:
or dl, dl ; Skip leading 0's
je .nextnumber
push eax
call putchar
pop eax
inc ebp
cmp ebp, 19
jae .nextnumber
mov [dbuffer+ebp], al

.nextnumber:
call getchar
jc .work
cmp al, '#'
je .ungetc
cmp al, '0'
jl .work
cmp al, '9'
ja .work
jmp short .number

.ungetc:
dec esi
inc ebx

.work:
; Now, do all the work
or dl, dl
je near .work0

cmp ebp, 19
jae near .toobig

call bcdload

; Calculate pinhole diameter

fld st0 ; save it
fsqrt
fmul st0, st3
fld st0
```

```
fmul st5
sub ebp, ebp

; Round off to 4 significant digits
.diameter:
fcom st0, st7
fstsw ax
sahf
jb .printdiameter
fmul st0, st6
inc ebp
jmp short .diameter

.printdiameter:
call printnumber ; pinhole diameter

; Calculate F-number

fdivp st1, st0
fld st0

sub ebp, ebp

.fnumber:
fcom st0, st6
fstsw ax
sahf
jb .printfnumber
fmul st0, st5
inc ebp
jmp short .fnumber

.printfnumber:
call printnumber ; F number

; Calculate normalized F-number
fmul st0, st0
fld1
fld st1
fyl2x
frndint
fld1
fscale
fsqrt
fstp st1

sub ebp, ebp
call printnumber

; Calculate time multiplier from F-5.6

fscale
fld st0
```

```
; Round off to 4 significant digits
.fmul:
fcom st0, st6
fstsw ax
sahf

jb .printfmul
inc ebp
fmul st0, st5
jmp short .fmul

.printfmul:
call printnumber ; F multiplier

; Calculate F-stops from 5.6

fld1
fxch st1
fyl2x

sub ebp, ebp
call printnumber

mov al, 0Ah
call putchar
jmp .bigloop

.work0:
mov al, '0'
call putchar

align 4
.toobig:
call huh
jmp .bigloop

align 4
done:
call write ; flush output buffer

; close files
push dword [fd.in]
sys.close

push dword [fd.out]
sys.close

finit

; return success
push dword 0
sys.exit
```

```
align 4
skiptoeol:
; Keep reading until you come to cr, lf, or eof
call getchar
jc done
cmp al, 0Ah
jne .cr
ret

.cr:
cmp al, 0Dh
jne skiptoeol
ret

align 4
getchar:
or ebx, ebx
jne .fetch

call read

.fetch:
lodsb
dec ebx
clc
ret

read:
jecxz .read
call write

.read:
push dword BUFSIZE
mov esi, ibuffer
push esi
push dword [fd.in]
sys.read
add esp, byte 12
mov ebx, eax
or eax, eax
je .empty
sub eax, eax
ret

align 4
.empty:
add esp, byte 4
stc
ret

align 4
putchar:
```

```
stosb
inc ecx
cmp ecx, BUFSIZE
je write
ret

align 4
write:
jecxz .ret ; nothing to write
sub edi, ecx ; start of buffer
push ecx
push edi
push dword [fd.out]
sys.write
add esp, byte 12
sub eax, eax
sub ecx, ecx ; buffer is empty now
.ret:
ret

align 4
bcdload:
; EBP contains the number of chars in dbuffer
push ecx
push esi
push edi

lea ecx, [ebp+1]
lea esi, [dbuffer+ebp-1]
shr ecx, 1

std

mov edi, bbuffer
sub eax, eax
mov [edi], eax
mov [edi+4], eax
mov [edi+2], ax

.loop:
lodsw
sub ax, 3030h
shl al, 4
or al, ah
mov [edi], al
inc edi
loop .loop

fbld [bbuffer]

cld
pop edi
pop esi
```

```
pop ecx
sub eax, eax
ret

align 4
printnumber:
push ebp
mov al, [separ]
call putchar

; Print the integer at the TOS
mov ebp, bbuffer+9
fbstp [bbuffer]

; Check the sign
mov al, [ebp]
dec ebp
or al, al
jns .leading

; We got a negative number (should never happen)
mov al, '-'
call putchar

.leading:
; Skip leading zeros
mov al, [ebp]
dec ebp
or al, al
jne .first
cmp ebp, bbuffer
jae .leading

; We are here because the result was 0.
; Print '0' and return
mov al, '0'
jmp putchar

.first:
; We have found the first non-zero.
; But it is still packed
test al, 0F0h
jz .second
push eax
shr al, 4
add al, '0'
call putchar
pop eax
and al, 0Fh

.second:
add al, '0'
call putchar
```

```
.next:
cmp ebp, bbuffer
jb .done

mov al, [ebp]
push eax
shr al, 4
add al, '0'
call putchar
pop eax
and al, 0Fh
add al, '0'
call putchar

dec ebp
jmp short .next

.done:
pop ebp
or ebp, ebp
je .ret

.zeros:
mov al, '0'
call putchar
dec ebp
jne .zeros

.ret:
ret
```

The code follows the same format as all the other filters we have seen before, with one subtle exception:

We are no longer assuming that the end of input implies the end of things to do, something we took for granted in the *character–oriented* filters.

This filter does not process characters. It processes a *language* (albeit a very simple one, consisting only of numbers).

When we have no more input, it can mean one of two things:

We are done and can quit. This is the same as before.

The last character we have read was a digit. We have stored it at the end of our ASCII–to–float conversion buffer. We now need to convert the contents of that buffer into a number and write the last line of our output.

For that reason, we have modified our `getchar` and our `read` routines to return with the `carry flag` *clear* whenever we are fetching another character from the input, or the `carry flag` *set* whenever there is no more input.

Of course, we are still using assembly language magic to do that! Take a good look at `getchar`. It *always* returns with the `carry flag` *clear*.

Yet, our main code relies on the `carry flag` to tell it when to quit——and it works.

The magic is in `read`. Whenever it receives more input from the system, it just returns to `getchar`, which fetches a character from the input buffer, *clears* the `carry flag` and returns.

But when `read` receives no more input from the system, it does *not* return to `getchar` at all. Instead, the `add esp, byte 4` op code adds `4` to `ESP`, *sets* the `carry flag`, and returns.

So, where does it return to? Whenever a program uses the `call` op code, the microprocessor `push`es the return address, i.e., it stores it on the top of the stack (not the FPU stack, the system stack, which is in the memory). When a program uses the `ret` op code, the microprocessor `pop`s the return value from the stack, and jumps to the address that was stored there.

But since we added `4` to `ESP` (which is the stack pointer register), we have effectively given the microprocessor a minor case of *amnesia*: It no longer remembers it was `getchar` that `call`ed `read`.

And since `getchar` never `push`ed anything before `call`ing `read`, the top of the stack now contains the return address to whatever or whoever `call`ed `getchar`. As far as that caller is concerned, he `call`ed `getchar`, which `ret`urned with the `carry flag` set!

Other than that, the `bcdload` routine is caught up in the middle of a Lilliputian conflict between the Big–Endians and the Little–Endians.

It is converting the text representation of a number into that number: The text is stored in the big–endian order, but the *packed decimal* is little–endian.

To solve the conflict, we use the `std` op code early on. We cancel it with `cld` later on: It is quite important we do not `call` anything that may depend on the default setting of the *direction flag* while `std` is active.

Everything else in this code should be quite clear, providing you have read the entire chapter that precedes it.

It is a classical example of the adage that programming requires a lot of thought and only a little coding. Once we have thought through every tiny detail, the code almost writes itself.

11.13.6 Using pinhole

Because we have decided to make the program *ignore* any input except for numbers (and even those inside a comment), we can actually perform *textual queries*. We do not *have to*, but we *can*.

In my humble opinion, forming a textual query, instead of having to follow a very strict syntax, makes software much more user friendly.

Suppose we want to build a pinhole camera to use the 4x5 inch film. The standard focal length for that film is about 150mm. We want to *fine–tune* our focal length so the pinhole diameter is as round a number as possible. Let us also suppose we are quite comfortable with cameras but somewhat intimidated by computers. Rather than just have to type in a bunch of numbers, we want to *ask* a couple of questions.

Our session might look like this:

```
% pinhole

Computer,

What size pinhole do I need for the focal length of 150?
150 490 306 362 2930 12
Hmmm... How about 160?
160 506 316 362 3125 12
Let's make it 155, please.
155 498 311 362 3027 12
Ah, let's try 157...
157 501 313 362 3066 12
156?
```

```
156 500 312 362 3047 12
That's it! Perfect! Thank you very much!
^D
```

We have found that while for the focal length of 150, our pinhole diameter should be 490 microns, or 0.49 mm, if we go with the almost identical focal length of 156 mm, we can get away with a pinhole diameter of exactly one half of a millimeter.

11.13.7 Scripting

Because we have chosen the `#` character to denote the start of a comment, we can treat our **pinhole** software as a *scripting language*.

You have probably seen **shell** *scripts* that start with:

```
#! /bin/sh
```

...or...

```
#!/bin/sh
```

...because the blank space after the `#!` is optional.

Whenever UNIX is asked to run an executable file which starts with the `#!`, it assumes the file is a script. It adds the command to the rest of the first line of the script, and tries to execute that.

Suppose now that we have installed **pinhole** in **/usr/local/bin/**, we can now write a script to calculate various pinhole diameters suitable for various focal lengths commonly used with the 120 film.

The script might look something like this:

```
#! /usr/local/bin/pinhole -b -i
# Find the best pinhole diameter
# for the 120 film

### Standard
80

### Wide angle
30, 40, 50, 60, 70

### Telephoto
100, 120, 140
```

Because 120 is a medium size film, we may name this file **medium**.

We can set its permissions to execute, and run it as if it were a program:

```
% chmod 755 medium
% ./medium
```

UNIX will interpret that last command as:

```
% /usr/local/bin/pinhole -b -i ./medium
```

It will run that command and display:

```
80 358 224 256 1562 11
30 219 137 128 586 9
40 253 158 181 781 10
50 283 177 181 977 10
60 310 194 181 1172 10
70 335 209 181 1367 10
100 400 250 256 1953 11
120 438 274 256 2344 11
140 473 296 256 2734 11
```

Now, let us enter:

```
% ./medium -c
```

UNIX will treat that as:

```
% /usr/local/bin/pinhole -b -i ./medium -c
```

That gives it two conflicting options: *-b* and *-c* (Use Bender's constant and use Connors' constant). We have programmed it so later options override early ones——our program will calculate everything using Connors' constant:

```
80 331 242 256 1826 11
30 203 148 128 685 9
40 234 171 181 913 10
50 262 191 181 1141 10
60 287 209 181 1370 10
70 310 226 256 1598 11
100 370 270 256 2283 11
120 405 296 256 2739 11
140 438 320 362 3196 12
```

We decide we want to go with Bender's constant after all. We want to save its values as a comma–separated file:

```
% ./medium -b -e > bender
% cat bender
focal length in millimeters,pinhole diameter in microns,F-number,normalized F-number,F-5.6 multiplie
80,358,224,256,1562,11
30,219,137,128,586,9
40,253,158,181,781,10
50,283,177,181,977,10
60,310,194,181,1172,10
70,335,209,181,1367,10
100,400,250,256,1953,11
120,438,274,256,2344,11
140,473,296,256,2734,11
%
```

11.14 Caveats

Assembly language programmers who "grew up" under MS-DOS and Windows often tend to take shortcuts. Reading the keyboard scan codes and writing directly to video memory are two classical examples of practices which, under MS-DOS are not frowned upon but considered the right thing to do.

The reason? Both the PC BIOS and MS-DOS are notoriously slow when performing these operations.

You may be tempted to continue similar practices in the UNIX environment. For example, I have seen a web site which explains how to access the keyboard scan codes on a popular UNIX clone.

That is generally a *very bad idea* in UNIX environment! Let me explain why.

11.14.1 UNIX Is Protected

For one thing, it may simply not be possible. UNIX runs in protected mode. Only the kernel and device drivers are allowed to access hardware directly. Perhaps a particular UNIX clone will let you read the keyboard scan codes, but chances are a real UNIX operating system will not. And even if one version may let you do it, the next one may not, so your carefully crafted software may become a dinosaur overnight.

11.14.2 UNIX Is an Abstraction

But there is a much more important reason not to try accessing the hardware directly (unless, of course, you are writing a device driver), even on the UNIX like systems that let you do it:

UNIX is an abstraction!

There is a major difference in the philosophy of design between MS-DOS and UNIX. MS-DOS was designed as a single-user system. It is run on a computer with a keyboard and a video screen attached directly to that computer. User input is almost guaranteed to come from that keyboard. Your program's output virtually always ends up on that screen.

This is NEVER guaranteed under UNIX. It is quite common for a UNIX user to pipe and redirect program input and output:

```
% program1 | program2 | program3 > file1
```

If you have written **program2**, your input does not come from the keyboard but from the output of **program1**. Similarly, your output does not go to the screen but becomes the input for **program3** whose output, in turn, goes to `file1`.

But there is more! Even if you made sure that your input comes from, and your output goes to, the terminal, there is no guarantee the terminal is a PC: It may not have its video memory where you expect it, nor may its keyboard be producing PC-style scan codes. It may be a Macintosh, or any other computer.

Now you may be shaking your head: My software is in PC assembly language, how can it run on a Macintosh? But I did not say your software would be running on a Macintosh, only that its terminal may be a Macintosh.

Under UNIX, the terminal does not have to be directly attached to the computer that runs your software, it can even be on another continent, or, for that matter, on another planet. It is perfectly possible that a Macintosh user in Australia connects to a UNIX system in North America (or anywhere else) via **telnet**. The software then runs on one computer, while the terminal is on a different computer: If you try to read the scan codes, you will get the wrong input!

Same holds true about any other hardware: A file you are reading may be on a disk you have no direct access to. A camera you are reading images from may be on a space shuttle, connected to you via satellites.

That is why under UNIX you must never make any assumptions about where your data is coming from and going to. Always let the system handle the physical access to the hardware.

> **注意:** These are caveats, not absolute rules. Exceptions are possible. For example, if a text editor has determined it is running on a local machine, it may want to read the scan codes directly for improved control. I am not mentioning these caveats to tell you what to do or what not to do, just to make you aware of certain pitfalls that await you if you have just arrived to UNIX form MS-DOS. Of course, creative people often break rules, and it is OK as long as they know they are breaking them and why.

11.15 Acknowledgements

This tutorial would never have been possible without the help of many experienced FreeBSD programmers from the FreeBSD 技术讨论邮件列表 (http://lists.FreeBSD.org/mailman/listinfo/freebsd-hackers), many of whom have patiently answered my questions, and pointed me in the right direction in my attempts to explore the inner workings of UNIX system programming in general and FreeBSD in particular.

Thomas M. Sommers opened the door for me. His How do I write "Hello, world" in FreeBSD assembler? (http://user.nj.net/~tms/hello.html) web page was my first encounter with an example of assembly language programming under FreeBSD.

Jake Burkholder has kept the door open by willingly answering all of my questions and supplying me with example assembly language source code.

Copyright © 2000-2001 G. Adam Stanislav. All rights reserved.

V. 附录

参考书目

[1] Dave A Patterson 和John L Hennessy, 1998, 1-55860-428-6, Morgan Kaufmann Publishers, Inc., *Computer Organization and Design: The Hardware / Software Interface, 1-2.*

[2] W. Richard Stevens, 1993, 0-201-56317-7, Addison Wesley Longman, Inc., *Advanced Programming in the Unix Environment, 1-2.*

[3] (美)W. Richard Stevens, 尤晋元等, 2000 年2 月, 7-111-07579-X, 机械工业出版社, *UNIX*环境高级编程.

[4] Marshall Kirk McKusick、Keith Bostic、Michael J Karels 和John S Quarterman, 1996, 0-201-54979-4, Addison-Wesley Publishing Company, Inc., *The Design and Implementation of the 4.4 BSD Operating System, 1-2.*

[5] Aleph One, *Phrack 49; "Smashing the Stack for Fun and Profit".*

[6] Chrispin Cowan、Calton Pu 和Dave Maier, *StackGuard; Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.*

[7] Todd Miller 和Theo de Raadt, *strlcpy and strlcat -- consistent, safe string copy and concatenation..*

索引

A

arguments(参数), 30

B

bounds checking(边界检查)
  compiler-based(基于编译器的), 32
  library-based(基于库的), 32
buffer overflow(缓冲区溢出), 30, 32

C

CERT
  security advisories(安全通告), 30
chroot(), 32
contributed software, 36
core team, 39

D

data validation(数据合法性确认), 33

F

frame pointer(帧指针), 30

G

gcc, 32
GTK, 35

J

jail, 33

L

LIFO(后进先出), 30

M

Morris 互联网蠕虫, 30

N

NUL 结束符, 31

O

OpenBSD, 31

P

Perl, 35
Perl Taint模式, 34
ports maintainer, 36
positive filtering(积极过滤), 33
POSIX.1e 处理能力, 33
process image(进程映象)
  frame pointer(帧指针), 30
  stack pointer(栈指针), 30
Python, 35

Q

Qt, 35

R

race conditions(竞态条件)
  access checks(存取检查), 34
  file opens(打开文件操作), 34
  signals(信号), 34
release engineer, 39
return address(返回地址), 30

S

seteuid, 32
stack frame(栈帧), 30
stack pointer(栈指针), 30
stack(栈), 30
stack-overflow(栈溢出), 30
StackGuard, 32
string copy functions(字符串复制函数)
  strlcat, 31
  strlcpy, 31
  strncat, 31
  strncpy, 31

T

TrustedBSD, 33

U

user IDs(用户ID)
  effective user ID(有效用户ID), 32
  real user ID(实际用户ID), 32

V

Von-Neuman(冯诺依曼), 30