FreeBSD Developers' Handbook

Zusammenfassung

Willkommen zum Entwickler-Handbuch. Dieses Handbuch ist *jederzeit unter Bearbeitung* und das Ergebnis der Arbeit vieler Einzelpersonen. Dies kann dazu führen, dass bestimmte Bereiche nicht mehr aktuell sind und auf den neuesten Stand gebracht werden müssen. Bei Unklarheiten empfiehlt es sich daher stets, auch die englische Originalversion des Handbuchs zu lesen.

Wenn Sie bei der Übersetzung dieses Handbuchs mithelfen möchten, senden Sie bitte eine E-Mail an die Mailingliste FreeBSD German Documentation Project <de-bsd-translators@de.FreeBSD.org>.

Die aktuelle Version dieses Handbuchs ist immer auf dem FreeBSD-Webserver verfügbar und kann in verschiedenen Formaten und in komprimierter Form vom FreeBSD-FTP-Server oder einem der zahlreichen Spiegel heruntergeladen werden (ältere Versionen finden Sie hingegen unter http://docs.FreeBSD.org/doc/).

Inhaltsverzeichnis

I: Grundlagen . . . 4
1. Einführung . . . 5
1.1. Unter FreeBSD entwickeln . . . 5
1.2. Die Vision von BSD . . . 5
1.3. Grundlegende Richtlinien . . . 5
1.4. Der Aufbau von /usr/src . . . 6
2. Werkzeuge zur Programmierung . . . 7
2.1. Überblick . . . 7
2.2. Zusammenfassung . . . 7
2.3. Einführung in die Programmierung . . . 7
2.4. Kompilieren mit dem `cc` . . . 10
2.5. Make . . . 18
2.6. Debuggen . . . 23
2.7. Emacs als Entwicklungsumgebung verwenden . . . 27
2.8. Weiterführende Literatur . . . 37
3. Sicheres Programmieren . . . 39
3.1. Zusammenfassung . . . 39
3.2. Methoden des sicheren Entwurfs . . . 39
3.3. Puffer-Überläufe . . . 39
3.4. SetUID-Themen . . . 42
3.5. Die Umgebung ihrer Programme einschränken . . . 42
3.6. Vertrauen . . . 43
3.7. Race-Conditions . . . 44
4. Lokalisierung und Internationalisierung - L10N und I18N . . . 45
4.1. I18N-konforme Anwendungen programmieren . . . 45
4.2. Lokalisierte Nachrichten mit POSIX.1 Native Language Support (NLS) . . . 45
5. Vorgaben und Richtlinien für das Quelltextverzeichnis . . . 50
5.1. Stil-Richtlinien . . . 50
5.2. `MAINTAINER` eines Makefiles . . . 50
5.3. Beigesteuerte Software . . . 51
5.4. Belastende Dateien . . . 57
5.5. Shared-Libraries . . . 58
6. Regressions- und Performance-Tests . . . 59
6.1. Mikro-Benchmark-Checkliste . . . 59
II: Interprozess-Kommunikation . . . 62
7. Sockets . . . 63
8. IPv6 Internals . . . 64
8.1. IPv6/IPsec-Implementierung . . . 64

III: Kernel . . . 84
9. Einen FreeBSD-Kernel bauen und installieren . . . 85
9.1. Einen Kernel auf die "traditionelle" Art und Weise bauen . . . 85
9.2. Einen Kernel auf die "neue" Art und Weise bauen . . . 86
10. Kernel-Fehlersuche . . . 87
10.1. Besorgen eines Speicherauszugs nach einem Kernel-Absturz (Kernel-Crash-Dump) . . . 87
10.2. Fehlersuche in einem Speicherauszug nach einem Kernel-Absturz mit kgdb . . . 89
10.3. Fehlersuche in einem Speicherauszug nach einem Absturz mit DDD . . . 94
10.4. Online-Kernel-Fehlersuche mit DDB . . . 94
10.5. Online-Kernel-Fehlersuche mit GDB auf einem entfernten System . . . 98
10.6. Fehlersuche bei einem Konsolen-Treiber . . . 100
10.7. Fehlersuche bei Deadlocks . . . 100
10.8. Glossar der Kernel-Optionen zur Fehlersuche . . . 100
IV: Architekturen . . . 103
11. x86-Assembler-Programmierung . . . 104
11.1. Synopsis . . . 104
11.2. Die Werkzeuge . . . 104
11.3. Systemaufrufe . . . 105
11.4. Rückgabewerte . . . 107
11.5. Portablen Code erzeugen . . . 109
11.6. Unser erstes Programm . . . 113
11.7. UNIX®-Filter schreiben . . . 115
11.8. Gepufferte Eingabe und Ausgabe . . . 118
11.9. Kommandozeilenparameter . . . 125
11.10. Die UNIX®-Umgebung . . . 130
11.11. Arbeiten mit Dateien . . . 135
11.12. One-Pointed Mind . . . 147
11.13. Die FPU verwenden . . . 156
11.14. Vorsichtsmassnahmen . . . 189
11.15. Danksagungen . . . 191
V: Anhang . . . 192
Literaturverzeichnis . . . 193

Teil I: Grundlagen

Kapitel 1. Einführung

1.1. Unter FreeBSD entwickeln

Hier sind wir also. Ihr System ist vollständig installiert und Sie wollen mit dem Programmieren beginnen. Aber womit sollen Sie anfangen? Was bietet Ihnen FreeBSD? Was kann es für einen Programmierer tun?

Dies sind einige der Fragen, welche dieses Handbuch zu beantworten versucht. Natürlich gibt es, analog zu anderen Berufen, auch bei Programmierern unterschiedliche Leistungsniveaus. Für die einen ist es ein Hobby, für die anderen ist es der Beruf. Die Informationen in diesem Kapitel dürften eher für den Programmieranfänger geeignet sein; allerdings könnte es auch für Programmierer, die bisher nichts mit der FreeBSD-Plattform zu tun hatten, interessante Informationen enthalten.

1.2. Die Vision von BSD

Ziel ist es, das bestmögliche UNIX®-artige Betriebssystempaket zu erstellen, mit dem gebührenden Respekt gegenüber der Ideologie der ursprünglichen Software, sowie der Bedienbarkeit, Leistungsfähigkeit und Stabilität.

1.3. Grundlegende Richtlinien

Unsere Ideologie kann durch die folgenden Leitfäden beschrieben werden.

- Füge keine neue Funktionalität hinzu, solange ein Programmierer diese nicht zur Fertigstellung einer realen Anwendung benötigt.
- Zu entscheiden, was ein System ist, ist genauso wichtig wie zu entscheiden, was ein System nicht ist. Versuchen Sie nicht, alle möglichen Wünsche zu erfüllen; machen Sie lieber das System erweiterbar, so dass zusätzliche Bedürfnisse in einer aufwärtskompatiblen Weise bedient werden können.
- Das Einzige, das schlimmer ist, als von einem Beispiel auf die Allgemeinheit zu schließen, ist, von überhaupt keinem Beispiel auf die Allgemeinheit zu schließen.
- Solange ein Problem nicht vollständig verstanden wurde, ist es besser, keine Lösung bereitzustellen.
- Wenn Sie 90% des gewünschten Effektes bei nur 10% des Aufwands erreichen können, sollten Sie besser die einfachere Lösung verwenden.
- Grenzen Sie Komplexität so gut wie möglich ein.
- Stellen Sie Mechanismen anstelle von Strategien bereit. Überlassen Sie insbesondere Strategien für die Benutzerschnittstelle dem Benutzerprogramm.

Aus Scheifler & Gettys: "X Window System"

1.4. Der Aufbau von /usr/src

Der vollständige Quelltext von FreeBSD ist über unser öffentliches Repository verfügbar. Der Quelltext wird normalerweise in /usr/src abgelegt und enthält die folgenden Unterverzeichnisse:

Verzeichnis	Beschreibung
bin/	Quelldateien für Dateien in /bin
cddl/	Quelldateien für Programme, die unter der Common Development and Distribution License stehen
contrib/	Quelldateien für Dateien von beigesteuerter Software
crypto/	Quelldateien für die Kryptographie
etc/	Quelldateien für Dateien in /etc
games/	Quelldateien für Dateien in /usr/games
gnu/	Programme, die unter der GNU Public License stehen
include/	Quelldateien für Dateien in /usr/include
kerberos5/	Quelldateien für Kerberos Version 5
lib/	Quelldateien für Dateien in /usr/lib
libexec/	Quelldateien für Dateien in /usr/libexec
release/	Dateien, die für die Erstellung eines FreeBSD-Releases nötig sind
rescue/	Bausystem für die /rescue-Programme
sbin/	Quelldateien für Dateien in /sbin
secure/	Quelldateien für FreeSec
share/	Quelldateien für Dateien in /usr/share
sys/	Kernel-Quelldateien
tools/	Programme zum Verwalten und Testen von FreeBSD
usr.bin/	Quelldateien für Dateien in /usr/bin
usr.sbin/	Quelldateien für Dateien in /usr/sbin

Kapitel 2. Werkzeuge zur Programmierung

2.1. Überblick

Dieses Kapitel ist eine Einführung in die Benutzung einiger der Werkzeuge zur Programmierung die mit FreeBSD ausgeliefert werden. Trotzdem ist vieles auch auf verschiedene andere Versionen von UNIX® übertragbar. Dieses Kapitel soll *kein* Versuch sein Programmierung detailliert zu beschreiben. Der größte Teil dieses Kapitels setzt wenige oder gar keine Programmierkenntnisse voraus, dennoch sollten die meisten Programmierer etwas Sinnvolles darin finden.

2.2. Zusammenfassung

FreeBSD bietet eine exzellente Entwicklungsumgebung. Compiler für C und C++, sowie ein Assembler sind im Basissystem enthalten. Natürlich finden sich auch klassische UNIX®-Werkzeuge wie `sed` und `awk`. Sollte das nicht genug sein, finden sich zahlreiche weitere Compiler und Interpreter in der Ports-Sammlung. Der folgende Abschnitt, Einführung in die Programmierung, zählt ein paar der verfügbaren Optionen auf. FreeBSD ist kompatibel zu vielen Standards wie POSIX® und ANSI C, sowie zu seinem eigenen BSD Erbe. So ist es möglich Anwendungen zu schreiben, welche ohne oder zumindest ohne wesentliche Änderungen auf einer großen Zahl an Plattformen kompilieren und laufen werden.

Allerdings können all diese Möglichkeiten anfangs etwas überwältigend sein, wenn Sie vorher nie Programme auf einem UNIX®-System geschrieben haben. Dieses Dokument hat die Zielsetzung ihnen beim Einstieg zu helfen ohne allzu weit in fortgeschrittene Themen vorzudringen. Die Intention ist, daß dieses Dokument ihnen ausreichend Basiswissen vermittelt und die weitergehende Dokumentation sinnvoll nutzen zu können.

Der größte Teil dieses Dokuments erfordert wenige oder gar keine Kenntnisse in der Programmierung, es werden trotzdem Basiswissen im Umgang mit UNIX® und die Bereitschaft zu lernen vorausgesetzt!

2.3. Einführung in die Programmierung

Ein Programm ist eine Zusammenstellung von Anweisungen, die den Computer auffordern verschiedenste Dinge zu tun. Dieser Abschnitt gibt ihnen einen Überblick über die beiden wesentlichen Methoden diese Anweisungen oder "Befehle", wie man diese Anweisungen üblicherweise nennt, zu geben. Die eine Methode nutzt einen *Interpreter*, die andere einen *Compiler*. Da menschliche Sprachen für einen Computer nicht unmissverständlich sind, werden diese Befehle in einer Sprache geschrieben die speziell für diesen Zweck gedacht ist.

2.3.1. Interpreter

Mit einem Interpreter ist die Sprache vielmehr eine Umgebung, in der Sie ein Kommando an der Kommandozeile eingeben welches dann von der Umgebung ausgeführt wird. Für kompliziertere Programme können Sie die Befehle in eine Datei schreiben und den Interpreter dazu bringen diese Datei zu laden und die enthaltenen Befehle auszuführen. Falls etwas schief geht werden viele

Interpreter Sie an einen Debugger weiterleiten.

Der Vorteil hierbei ist, das Sie das Ergebnis ihres Befehls direkt sehen und Fehler sofort korrigiert werden können. Der größte Nachteil bei dieser Methode entsteht, wenn Sie ihr Programm mit jemandem teilen wollen. Diese Person muss den selben Interpreter nutzen wie Sie es tun und Sie muss wissen wie dieser zu bedienen ist. Zudem werden Benutzer es nicht begrüßen sich in einem Debugger wiederzufinden, wenn Sie einmal die falsche Taste drücken! Bei einem Blick auf die Leistungsfähigkeit brauchen Interpreter oftmals viel Speicher und erzeugen den Code nicht so effizient wie Compiler.

Meiner Meinung nach sind interpretierte Sprachen der beste Anfang, wenn Sie bisher noch nicht programmiert haben. Diese Art von Umgebung findet man typischerweise bei Sprachen wie Lisp, Smalltalk, Perl und Basic. Man könnte auch sagen, dass die UNIX® Shell (`sh`, `csh`) für sich bereits einen Interpreter darstellt und viele Leute schreiben tatsächlich Shell "Scripten" um sich bei einigen "Haushaltsaufgaben" auf ihren Maschinen helfen zu lassen. Tatsächlich war es ein wesentlicher Teil der originalen UNIX® Philosophie eine große Zahl an kleinen Hilfsprogrammen zur Verfügung zu stellen, welche mittels eines Shellskripts miteinander kombiniert werden um bestimmte Aufgaben zu übernehmen.

2.3.2. Für FreeBSD verfügbare Interpreter

Im folgenden eine Liste der über die FreeBSD Ports-Sammlung verfügbaren Interpreter einschließlich einer kurzen Erörterung der populären interpretierten Sprachen.

Anleitungen wie man Anwendungen aus der Ports-Sammlung erhält und installiert können Sie dem Kapitel Benutzen der Ports-Sammlung aus dem FreeBSD Handbuch entnehmen.

BASIC

Kurz für Beginner's All-purpose Symbolic Instruction Code. Entwickelt in den 50er Jahren um Studenten in Programmierung zu unterrichten, wurde BASIC in den 80er Jahren mit jedem anständigen Personal Computer ausgeliefert und war für viele Programmierer die erste Programmiersprache. BASIC ist auch die Grundlage für Visual Basic.

Der Bywater Basic Interpreter findet sich in der Ports-Sammlung unter lang/bwbasic und Phil Cockroft's Basic Interpreter (auch bekannt als Rabbit Basic) findet sich unter lang/pbasic.

Lisp

Diese Sprache wurde in den späten 50er Jahren als Alternative zu den, zu dieser Zeit populären, "zahlenverarbeitenden" Sprachen entwickelt. Anstelle auf Zahlen basiert Lisp auf Listen; tatsächlich ist der Name Lisp eine Kurzform für "List Processing" (Listen abarbeiten). Sehr populär fü AI (Artificial Intelligence/ künstliche Intelligez) (Fach-) Kreisen.

Lisp ist eine extrem kraftvolle und durchdachte Sprache, kann aber auch recht groß und unhandlich sein.

Zahlreiche Ausformungen von Lisp, die auf UNIX® Systemen laufen sind über die Ports-Sammlung verfügbar. GNU Common Lisp befindet sich in lang/gcl. CLISP von Bruno Haible und Michael Stoll ist in lang/clisp zu finden. Für CMUCL, welches auch einen hoch-optimierten Compiler enthält, oder einfachere Ausformungen wie SLisp, das die meisten gängigen Lisp

Konstrukte in wenigen hundert Zeilen C Code enthält sind in lang/cmucl und lang/slisp ebenfalls enthalten.

Perl

Unter Systemadministratoren zum Schreiben von Skripten sehr beliebt; wird häufig auch auf World Wide Web Servern verwendet, um CGI-Skripte zu schreiben.

Perl ist in der Ports-Sammlung unter lang/perl5.8 für alle FreeBSD-Versionen verfügbar, und wird im Basissystem von 4.x als `/usr/bin/perl` installiert.

Scheme

Ein Dialekt von Lisp, der kompakter und sauberer als Common Lisp ist. Dieser Dialekt ist an Universitäten sehr beliebt, da er zum einen für den Unterricht im Grundstudium einfach genug ist, und zum anderen ein ausreichend hohes Abstraktionsniveau für den Einsatz in der Forschung bietet.

Scheme ist in der Ports-Sammlung in Form des Elk Scheme Interpreters als lang/elk verfügbar. Den MIT Scheme Interpreter findet man unter lang/mit-scheme, und den SCM Scheme Interpreter unter lang/scm.

Icon

Icon ist eine Hochsprache mit ausgereiften Möglichkeiten zur Verarbeitung von Zeichenketten und Strukturen. Die unter FreeBSD verfügbare Version von Icon steht in der Ports-Sammlung unter lang/icon zur Verfügung.

Logo

Logo ist eine leicht zu erlernende Programmiersprache, welche in vielen Kursen als einführende Programmiersprache gewählt wird. Sie ist ein ideales Arbeitswerkzeug beim Unterricht mit jungen Menschen, da mit ihr die Erstellung komplizierter geometrischer Oberflächen selbst für kleine Kinder einfach ist.

Die für FreeBSD aktuellste, verfügbare Version findet man in der Ports-Sammlung unter lang/logo.

Python

Python ist eine objektorientierte, interpretierte Programmiersprache. Die Verfechter von Python argumentieren, daß sie eine der besten Programmiersprachen für Programmieranfänger sei, da sie einfach zu erlernen ist, und anderen populären interpretierten Programmiersprachen, welche zur Entwicklung großer und komplexer Anwendungen verwendet werden, in nichts nachsteht (Perl und Tcl sind zwei solcher bekannten Programmiersprachen).

Die aktuellste Version von Python ist in der Ports-Sammlung unter lang/python verfügbar.

Ruby

Ruby ist eine interpretierte und rein objektorientierte Programmiersprache. Sie wurde wegen ihrer leicht verständlichen Syntax, ihrer Flexibilität und der Möglichkeit, große und komplexe Programme einfach zu entwickeln und zu pflegen, populär.

Ruby ist in der Ports-Sammlung unter lang/ruby18 verfügbar.

Tcl und Tk

Tcl ist eine einbettbare, interpretierte Programmiersprache, welche aufgrund ihrer Portierbarkeit auf viele unterschiedliche Plattformen eine weite Verbreitung erfahren hat. Sie kann sowohl für die schnelle Entwicklung kleinerer Prototypen, als auch (in Verbindung mit Tk, einem GUI Toolkit) vollwertiger, ausgereifter Programme verwendet werden.

Es sind mehrere Versionen von Tcl als Ports für FreeBSD verfügbar. Die aktuellste Version, Tcl 8.7, ist unter lang/tcl87 verfügbar.

2.3.3. Compiler

Compiler sind eher anders. Zuerst schreibt man seinen Code unter Verwendung eines Editors in eine Datei (oder mehrere Dateien). Anschließend ruft man den Compiler auf um zu sehen, ob dieser das Programm annimmt. Wenn das Programm nicht kompiliert werden konnte, muß man die Zähne zusammenbeissen und wieder zum Editor zurückkehren; falls das Programm kompiliert und eine ausführbare Anwendung erzeugt wurde, kann man diese über eine Eingabeaufforderung oder über einen Debugger aufrufen um zu sehen, ob sie auch funktioniert.

Offensichtlich ist diese Art der Programmierung nicht so direkt wie die Verwendung eines Interpreters. Jedoch sind auf diese Weise viele Dinge möglich, die mit einem Interpreter nur sehr schwer oder überhaupt nicht realisierbar wären, wie z.B. das Schreiben von Code, der sehr eng mit dem Betriebsystem zusammen arbeitet-oder das Schreiben eines eigenen Betriebsystems selbst! Des weiteren ist so das Erzeugen von sehr effizientem Code möglich, da sich der Compiler für die Optimierung Zeit nehmen kann, was bei einem Interpreter inakzeptabel wäre. Ferner ist das Verbreiten von Programmen, welche für einen Compiler geschrieben wurden, einfacher als welche, die für einen Interpreter geschrieben wurden-man muss in ersterem Fall nur die ausführbare Datei verbreiten, vorausgesetzt, daß das gleiche Betriebssystem verwendet wird.

Programmiersprachen, die kompiliert werden, sind unter anderem Pascal, C und C. C und C sind eher unbarmherzige Programmiersprachen und daher eher für erfahrene Programmierer gedacht; Pascal auf der anderen Seite wurde zu Ausbildungszwecken entworfen, und stellt daher eine einsteigerfreundliche Programmiersprache dar. FreeBSD beinhaltet im Basissystem keine Unterstützung für Pascal, stellt jedoch über die Ports-Sammlung den Free Pascal Compiler unter lang/fpc zur Verfügung.

Da der editier-kompilier-ausführ-debug-Kreislauf unter Verwendung mehrerer Programme eher mühsam ist haben viele Hersteller von Compilern integrierte Entwicklungsumgebungen (Integrated Development Environment; auch kurz IDE) entwickelt. FreeBSD bietet zwar im Basissystem keine IDE an, stellt jedoch über die Ports-Sammlung IDEs wie devel/kdevelop oder Emacs zur Verfügung, wobei letztere weit verbreitet ist. Die Verwendung von Emacs als IDE wird unter Emacs als Entwicklungsumgebung verwenden diskutiert.

2.4. Kompilieren mit dem `cc`

Dieser Abschnitt behandelt ausschließlich den GNU Compiler für C und C++, da dieser bereits im Basissystem von FreeBSD enthalten ist. Er kann mittels `cc` oder `gcc` aufgerufen werden. Die Details zur Erstellung einer Anwendung mit einem Interpreter variieren zwischen verschiedenen Interpretern mehr oder weniger stark, und werden meist ausführlich in der zugehörigen

Dokumentation oder Online-Hilfe beschrieben.

Sobald Sie Ihr Meisterwerk fertig geschrieben haben besteht der nächste Schritt darin, dieses (hoffentlich!) unter FreeBSD zum Laufen zu bekommen. Dies beinhaltet üblicherweise mehrere Schritte, wobei jeder einzelne Schritt von einem separaten Programm durchgeführt wird.

1. Aufbereiten Ihres Quelltextes durch Entfernen von Kommentaren, sowie weiteren Tricks wie das Ersetzen von Macros in C.
2. Überprüfen der Syntax Ihres Quelltextes, um die Einhaltung der Sprachregeln sicherzustellen. Wenn Sie diese verletzt haben werden entsprechende Fehlermeldungen Ihnen dies mitteilen!
3. Übersetzen des Quelltextes in Assemblersprache -diese ist dem eigentlichen Maschinencode schon sehr nahe, jedoch immer noch für Menschen lesbar. Angeblich.
4. Übersetzen der Assemblersprache in Maschinencode-genau, wir sprechen hier von Bits und Bytes, Einsen und Nullen.
5. Überprüfen, ob Sie Dinge wie Funktionen und globale Variablen in einheitlicher Weise verwendet haben. Wenn Sie z.B. eine nicht existierende Funktion aufgerufen haben, wird eine entsprechende Fehlermeldung Ihnen dies mitteilen.
6. Wenn aus mehreren Quelltextdateien eine ausführbare Datei erstellt werden soll wird herausgefunden, wie die einzelnen Codeteile zusammengefügt werden müssen.
7. Ausarbeiten, wie das Programm aussehen muss, damit der Lader zur Laufzeit des Systems dieses in den Speicher laden und ausführen kann.
8. Endgültiges Schreiben der ausführbaren Datei in das Dateisystem.

Das Wort *kompilieren* wird häufig für die Schritte 1 bis 4 verwendet-die anderen werden mit dem Wort *verlinken* zusammengefasst. Manchmal wird Schritt 1 auch als *Pre-Processing* und die Schritte 3-4 als *assemblieren* bezeichnet.

Glücklicherweise werden alle diese Details vor Ihnen verborgen, da `cc` ein Frontend ist, welches sich um die Ausführung all dieser Programme mit den richtigen Argumenten für Sie kümmert; einfaches eingeben von

```
% cc foobar.c
```

führt zur Übersetzung von foobar.c durch alle bereits erwähnten Schritte. Wenn Sie mehr als eine Datei übersetzen wollen müssen Sie etwas wie folgt eingeben

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

Beachten Sie, daß die Überprüfung der Syntax genau dies tut-das reine Überprüfen der Syntax. Es findet keine Überprüfung bzgl. logischer Fehler statt, die Sie vielleicht gemacht haben, wie z.B. das Programm in eine Endlosschleife zu versetzen, oder Bubble Sort zu verwenden, wenn Sie eigentlich Binary Sort benutzen wollten.

Es gibt haufenweise Optionen für `cc`, die alle in der zugehörigen Manualpage beschrieben werden. Im Folgenden werden ein paar der wichtigsten Optionen mit Beispielen ihrer Anwendung gezeigt.

`-o filename`

Die Name der Ausgabedatei. Wenn Sie diese Option nicht verwenden erstellt `cc` eine Datei mit dem Namen a.out.

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

`-c`

Dies kompiliert die Datei nur, verlinkt sie jedoch nicht. Nützlich für Spielereien, um die Syntax auf Korrektheit zu überprüfen, oder falls Sie ein Makefile verwenden.

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

Dieser Befehl erzeugt eine *Objektdatei* (nicht ausführbar) mit den Namen foobar.o. Diese kann mit anderen Objektdateien zusammen zu einer ausführbaren Datei verlinkt werden.

`-g`

Diese Option erzeugt die Debug-Version einer ausführbaren Datei. Dabei fügt der Compiler zusätzliche Informationen darüber, welcher Funktionsaufruf zu welcher Zeile im Quelltext gehört, der ausführbaren Datei hinzu. Ein Debugger kann Ihnen mit Hilfe dieser Information den zugehörigen Quelltext anzeigen, während Sie den Programmverlauf schrittweise verfolgen, was *sehr* hilfreich sein kann; der Nachteil dabei ist, daß durch die zusätzlichen Informationen das Programm viel größer wird. Normalerweise verwendet man die Option `-g` während der Entwicklung eines Programms, und für die "Release-Version", wenn man von der Korrektheit des Programms überzeugt ist, kompiliert man das Programm dann ohne diese Option.

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

Mit diesem Befehl wird eine Debug-Version des Programms erzeugt.

`-O`

Diese Option erzeugt eine optimierte Version der ausführbaren Datei. Der Compiler verwendet einige clevere Tricks, um das erzeugte Programm schneller zu machen. Sie können hinter der Option `-O` eine Zahl angeben, um eine höheres Level der Optimierung festzulegen. Dadurch wird jedoch häufig eine fehlerhafte Optimierung seitens des Compilers aufgedeckt. Zum Beispiel erzeugte die Version des `cc`, welche mit dem FreeBSD Release 2.1.0 mitgeliefert wurde, bei Verwendung der Option `-O2` unter bestimmten Umständen falschen Code.

Optimierungen werden normalerweise nur beim Kompilieren von Release-Versionen aktiviert.

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

Durch diesen Befehl wird eine optimierte Version von foobar erzeugt.

Die folgenden drei Flags zwingen den `cc` dazu, Ihren Code auf die Einhaltung der internationalen Standards hin zu überprüfen, welche häufig als ANSI Standards bezeichnet werden, obwohl sie streng genommen zum ISO Standard gehören.

`-Wall`

Aktivieren aller Warnmeldungen, die die Autoren des `cc` für wichtig halten. Trotz des Namens dieser Option werden dadurch nicht sämtliche Warnungen ausgegeben, die der `cc` ausgeben könnte.

`-ansi`

Deaktivieren der meisten, jedoch nicht aller, nicht-ANSI C Eigenschaften, die der `cc` bietet. Trotz des Namens ist durch diese Option nicht sichergestellt, daß Ihr Code diese Standards auch vollständig einhält.

`-pedantic`

Deaktivieren *aller* Eigenschaften des `cc`, welche nicht konform zu ANSI C sind.

Ohne diese Flags wird Ihnen der `cc` die Verwendung eigener Erweiterungen des Standards erlauben. Einige dieser Erweiterungen sind zwar sehr nützlich, werden jedoch nicht von anderen Compilern unterstützt-eigentlich ist eines der Hauptziele des Standards, das Leute Code so schreiben können, daß dieser mit jedem Compiler auf beliebigen Systemen funktioniert. Dies wird häufig als *portabler Code* bezeichnet.

Im Allgemeinen sollten Sie versuchen, Ihren Code so portabel wie möglich zu schreiben, da Sie ansonsten eventuell das gesamte Programm noch einmal neu schreiben müssen, falls dieser in einer anderen Umgebung laufen soll-und wer weiß schon was er in ein paar Jahren verwenden wird?

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

Durch diesen Befehl wird eine ausführbare Datei namens foobar erzeugt, nachdem foobar.c auf die Einhaltung der Standards überprüft wurde.

`-l library`

Mit dieser Option kann eine Bibliothek mit Funktionen angegeben werden, die während des Verlinkens verwendet wird.

Das am häufigsten auftretende Beispiel dieser Option ist die Übersetzung eines Programmes, welches einige der mathematischen Funktionen in C verwendet. Im Gegensatz zu den meisten anderen Plattformen befinden sich diese Funktionen in einer separaten Bibliothek, deren Verwendung Sie dem Compiler explizit mitteilen müssen.

Angenommen eine Bibliothek heißt libirgendwas.a, dann müssen Sie dem `cc` als Argument `-l irgendwas` übergeben. Zum Beispiel heißt die Mathematik-Bibliothek libm.a, und daher müssen Sie dem `cc` als Argument `-lm` übergeben. Ein typisches "Manko" der Mathematik-Bibliothek ist, daß diese immer die letzte Bibliothek auf der Kommandozeile sein muß.

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

Durch diesen Befehl werden die Funktionen aus der Mathematik-Bibliothek in foobar gelinkt.

Wenn Sie c++ -Code kompilieren wollen, müssen Sie -lg++, bzw. -lstdc++ falls Sie FreeBSD 2.2 oder neuer verwenden, zu Ihrer Kommandozeile hinzufügen, um Ihr Programm gegen die Funktionen der C Bibliothek zu linken. Alternativ können Sie anstatt cc auch {c-plus-plus-command} aufrufen, welcher dies für Sie erledigt. C kann unter FreeBSD auch als g++ aufgerufen werden.

```
% cc -o foobar foobar.cc -lg++     Bei FreeBSD 2.1.6 oder älter
% cc -o foobar foobar.cc -lstdc++  Bei FreeBSD 2.2 und neuer
% c++ -o foobar foobar.cc
```

Beide Varianten erzeugen eine ausführbare foobar aus der c++ Quelltextdatei foobar.cc. Beachten Sie bitte, daß auf UNIX® Systemen c++ Quelltextdateien üblicherweise auf .C, .cxx oder .cc enden, und nicht wie bei MS-DOS® auf .cpp (welche schon anderweitig benutzt wurde). Der `gcc` hat normalerweise anhand dieser Information entschieden, welcher Compiler für die Quelltextdatei zum Einsatz kommen soll; allerdings gilt diese Einschränkung jetzt nicht mehr, und Sie können Ihre c++-Dateien ungestraft auf .cpp enden lassen!

2.4.1. Häufig auftretende `cc`-Fragen und -Probleme

2.4.1.1. Ich versuche ein Programm zu schreiben, welches die Funktion sin() verwendet, erhalte jedoch eine Fehlermeldung. Was bedeutet diese?

Wenn Sie mathematische Funktionen wie `sin()` verwenden wollen, müssen Sie den `cc` anweisen, die Mathematik-Bibliothek wie folgt zu verlinken:

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

2.4.1.2. So, ich habe jetzt dieses einfache Programm als Übung für -lm geschrieben. Alles was es macht ist, 2.1 hoch 6 zu berechnen.

Wenn der Compiler Ihren Funktionsaufruf sieht, überprüft er, ob er schon einmal einen Prototypen für diese gesehen hat. Wenn nicht nimmt er als Rückgabewert den Typ int an, was sicherlich nicht das ist, was Sie an dieser Stelle wollen.

2.4.1.3. Wie kann ich das korrigieren?

Die Prototypen der mathematischen Funktionen befinden sich in der Datei math.h. Wenn Sie diese Datei in Ihrem Quelltext includen ist der Compiler in der Lage, den Prototypen zu finden, und wird aufhören, seltsame Dinge mit Ihrer Berechnung zu machen!

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

```
int main() {
...
```

Nach erneutem Compilieren sollte das Folgende bei der Ausführung ausgegeben werden:

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

Wenn Sie irgendwelche mathematischen Funktionen verwenden sollten Sie *immer* die Datei math.h includen und nicht vergessen, Ihr Programm gegen die Mathematik-Bibliothek zu verlinken.

2.4.1.4. Ich habe eine Datei mit dem Namen foobar.c kompiliert, kann jedoch nirgends eine ausführbare Datei namens foobar finden. Wo befindet sich diese?

Denken Sie daran, daß der `cc` die ausführbare Datei a.out nennt, wenn Sie nicht explizit einen Namen angeben. Verwenden Sie in solch einem Fall die Option `-o filename`:

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

2.4.1.5. OK, ich habe eine ausführbare Datei namens foobar, ich kann sie sehen, wenn ich ls aufrufe. Gebe ich jedoch foobar in die Kommandozeile ein wird mir gesagt, daß eine Datei mit diesem Namen nicht existiert. Warum kann die Datei nicht gefunden werden?

Im Gegensatz zu MS-DOS® sucht UNIX® nicht im aktuellen Verzeichnis nach einem ausführbaren Programm, das Sie versuchen auszuführen, solange Sie dies nicht explizit mit angeben. Sie können entweder `./foobar` eingeben, was soviel bedeutet wie "führe eine Datei namens foobar im aktuellen Verzeichnis aus", oder Sie können Ihre Umgebungsvariable `PATH` so erweitern, daß sie ähnlich wie folgt aussieht

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

Der Punkt am Ende bedeutet "siehe im aktuellen Verzeichnis nach, wenn es in keinem der anderen zu finden war".

2.4.1.6. Ich habe meine ausführbare Datei test genannt, allerdings passiert nichts wenn ich diese aufrufe. Was ist hier los?

Bei den meisten UNIX®-Systeme existiert bereits ein Programm mit dem Namen `test` im Verzeichnis /usr/bin, und die Shell nimmt dieses, bevor sie im aktuellen Verzeichnis nachsieht. Sie können entweder den folgenden Befehl eingeben:

```
% ./test
```

oder Sie können einen geeigneteren Namen für Ihr Programm wählen!

2.4.1.7. Ich habe mein Programm kompiliert und bei dessen Aufruf sah zuerst alles gut aus. Jedoch gab es dann eine Fehlermeldung, welche irgendetwas mit core dumped lautete. Was bedeutet das?

Der Name *core dump* stammt noch aus sehr frühen Zeiten von UNIX®, als die Maschinen noch Kernspeicher zum Speichern von Daten verwendeten. Einfach ausgedrückt, wenn bei einem Programm unter bestimmen Bedingungen ein Fehler auftrat, hat das System den Inhalt des Kernspeichers auf der Festplatte in eine Datei namens core geschrieben, welche der Programmierer dann näher untersuchen konnte, um die Ursache des Fehlers herauszufinden.

2.4.1.8. Faszinierendes Zeugs, aber was soll ich jetzt machen?

Verwenden Sie den `gdb`, um das Speicherabbild zu untersuchen (siehe Debuggen).

2.4.1.9. Als mein Programm den core dump erzeugt hat, sagte es etwas von einem segmentation fault. Was ist das?

Diese Meldung heißt im Prinzip, daß Ihr Programm eine illegale Operation mit dem Speicher durchführen wollte; UNIX® wurde so entworfen, daß es das andere Programme und das Betriebssystem selbst vor wildgewordenen Programmen schützt.

Häufige Ursachen hierfür sind:

- Der Versuch, einen NULL-Zeiger zu beschreiben, z.B.

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

- Einen Zeiger zu verwenden, welcher noch nicht initialisiert wurde, z.B.

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

  Der Zeiger hat einen zufälligen Wert, welcher mit etwas Glück in einen Bereich des Speichers zeigt, der für Ihr Programm nicht verfügbar ist, und der Kernel bricht Ihr Programm ab, bevor es irgendwelchen Schaden anrichten kann. Wenn Sie Pech haben zeigt der Zeiger irgendwo mitten in Ihr eigenes Programm, und verändert dort ihre eigenen Datenstrukturen, was zu sehr seltsamen Fehlern Ihres Programmes führt.

- Der Versuch, auf Daten außerhalb eines Arrays zuzugreifen, z.B.

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

- Der Versuch, Daten in eine Speicherbereich zu schreiben, der nur lesbar ist, z.B.

```
char *foo = "My string";
```

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

UNIX®-Compiler speichern häufig feste Zeichenketten wie `"My string"` in nur lesbaren Speicherbereichen ab.

- Wenn man unerlaubte Operationen mit `malloc()` und `free()` ausführt, z.B.

```
char bar[80];
free(bar);
```

oder

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

Einzelne solcher Fehler führen zwar nicht immer zu einem Fehlverhalten des Programms, stellen jedoch immer eine falsche Verwendung dar. Manche Systeme und Compiler sind toleranter als andere, weshalb Programme auf dem einen System einwandfrei laufen, auf dem anderen System jedoch abstürzen.

2.4.1.10. Wenn ich einen core dump erhalte erscheint manchmal die Meldung bus error. In meinem UNIX®-Buch steht, daß die Ursache ein Hardwareproblem sei. Der Computer scheint aber weiterhin zu funktionieren. Ist dies wahr?

Nein, glücklicherweise nicht (es sei denn Sie haben wirklich ein Hardwareproblem...). Üblicherweise ist dies ein Weg Ihnen mitzuteilen, daß Sie auf Speicher in einer Weise zugegriffen haben, in der Sie dies nicht tun sollten.

2.4.1.11. Diese Sache mit den core dumps hört sich sehr nützlich an, wenn ich so etwas selber an beliebiger Stelle bewirken könnte. Kann ich das tun, oder muß ich warten bis ein Fehler auftritt?

Ja, nehmen sie einfach eine andere Konsole oder XTerm und führen Sie

```
% ps
```

aus, um die Prozess-ID Ihres Programms herauszufinden. Führen Sie anschließend

```
% kill -ABRT pid
```

aus, wobei *pid* die Prozess-ID ist, die Sie vorher ermittelt haben.

Dies ist nützlich, wenn sich Ihr Programm z.B. in einer Endlosschleife verfangen hat. Sollte Ihr Programm das Signal SIGABRT abfangen, gibt es noch andere Möglichkeiten, die denselben Effekt haben.

Alternativ können Sie einen core dump aus Ihrem Programm heraus erstellen, indem Sie die Funktion `abort()` aufrufen. Weitere Informationen darüber können Sie in der Manualpage abort(3) nachlesen.

Wenn Sie einen core dump von außerhalb Ihres Programms erzeugen wollen, ohne dabei den Prozess abzubrechen, können Sie das Programm `gcore` verwenden. Weitere Informationen dazu finden Sie in der zugehörigen Manualpage gcore(1).

2.5. Make

2.5.1. Was ist `make`?

Wenn Sie an einem einfachen Programm mit nur einer oder zwei Quelltextdateien arbeiten, ist die Eingabe von

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

zwar nicht aufwendig, wird aber mit zunehmender Anzahl der Quelltextdateien sehr lästig-und auch das Kompilieren kann eine Weile dauern.

Eine Möglichkeit dies zu umgehen besteht in der Verwendung von Objektdateien, wobei man nur die Quelltextdateien neu kompiliert, die verändert wurden. So könnten wir etwa folgendes erhalten:

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

falls wir seit dem letzten Kompiliervorgang nur die Datei file37.c verändert haben. Dadurch könnte der Kompiliervorgang um einiges beschleunigt werden, es muß jedoch immer noch alles von Hand eingegeben werden.

Oder wir könnten uns ein Shell Skript schreiben. Dieses würde jedoch alles immer wieder neu kompilieren, was bei einem großen Projekt sehr ineffizient wäre.

Was ist, wenn wir hunderte von Quelltextdateien hätten? Was ist, wenn wir in einem Team mit anderen Leuten arbeiten würden, die vergessen uns Bescheid zu sagen, falls sie eine der Quelltextdateien verändert haben, die wir ebenfalls benutzen?

Vielleicht könnten wir beide Lösungen kombinieren und etwas wie ein Shell Skript schreiben, welches eine Art magische Regel enthalten würde, die feststellt, welche Quelltextdateien neu kompiliert werden müssten. Alles was wir bräuchten wäre ein Programm, das diese Regeln verstehen könnte, da diese Aufgabe etwas zu kompliziert für eine Shell ist.

Dieses Programm heißt `make`. Es liest eine Datei namens *makefile*, welche ihm sagt, wie unterschiedliche Dateien voneinander abhängen, und berechnet, welche Dateien neu kompiliert werden müssen und welche nicht. Zum Beispiel könnte eine Regel etwas sagen wie "wenn fromboz.o älter als fromboz.c ist, bedeutet dies, daß jemand die Datei fromboz.c verändert haben muß, und diese daher neu kompiliert werden muß." Das makefile enthält außerdem Regeln die

make sagen, *wie* die Quelltextdatei neu kompiliert werden muß, was dieses Tool noch sehr viel mächtiger macht.

Makefiles werden normalerweise im selben Verzeichnis wie die Quelltextdateien abgelegt, zu denen sie gehören, und kann makefile, Makefile oder MAKEFILE heißen. Die meisten Programmierer verwenden den Namen Makefile, da diese Schreibweise dafür sorgt, daß die Datei gut lesbar ganz oben in der Verzeichnisliste aufgeführt wird.

2.5.2. Beispielhafte Verwendung von `make`

Hier ist eine sehr einfache make Datei:

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

Sie besteht aus zwei Zeilen, einer Abhängigkeitszeile und einer Erzeugungszeile.

Die Abhängigkeitszeile hier besteht aus dem Namen des Programms (auch *Ziel* genannt), gefolgt von einem Doppelpunkt und einem Leerzeichen, und anschließend dem Namen der Quelltextdatei. Wenn `make` diese Zeile liest überprüft es die Existenz von foo; falls diese Datei existiert vergleicht es das Datum der letzten Änderung von foo mit der von foo.c. Falls foo nicht existiert, oder älter als foo.c ist, liest es die Erzeugungszeile um herauszufinden, was zu tun ist. Mit anderen Worten, dies ist die Regel die festlegt, wann foo.c neu kompiliert werden muß.

Die Erzeugungszeile beginnt mit einem tab (drücken Sie dazu die `tab`-Taste) gefolgt von dem Befehl, mit dem Sie foo manuell erzeugen würden. Wenn foo veraltet ist, oder nicht existiert, führt `make` diesen Befehl aus, um die Datei zu erzeugen. Mit anderen Worten, dies ist die Regel die make sagt, wie foo.c kompiliert werden muß.

Wenn Sie also `make` eingeben wird dieses sicherstellen, daß foo bzgl. Ihrer letzten Änderungen an foo.c auf dem neuesten Stand ist. Dieses Prinzip kann auf Makefiles mit hunderten von Zielen-es ist bei FreeBSD praktisch möglich, das gesamte Betriebssystem zu kompilieren, indem man nur `make world` im richtigen Verzeichnis eingibt!

Eine weitere nützliche Eigenschaft der makefiles ist, daß die Ziele keine Programme sein müssen. Wir könnten zum Beispiel eine make Datei haben, die wie folgt aussieht:

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

install:
    cp foo /home/me
```

Wir können make sagen welches Ziel wir erzeugt haben wollen, indem wir etwas wie folgt eingeben:

```
% make target
```

`make` wird dann nur dieses Ziel beachten und alle anderen ignorieren. Wenn wir zum Beispiel `make foo` mit dem obigen makefile eingeben, dann wird make das Ziel `install` ignorieren.

Wenn wir nur `make` eingeben wird make immer nur nach dem ersten Ziel suchen und danach mit dem Suchen aufhören. Wenn wir hier also nur `make` eingegeben hätten, würde es nur zu dem Ziel `foo` gehen, gegebenenfalls foo neu kompilieren, und danach einfach aufhören, ohne das Ziel `install` zu beachten.

Beachten Sie, daß das `install`-Ziel von nichts anderem abhängt! Dies bedeutet, daß der Befehl in der nachfolgenden Zeile immer ausgeführt wird, wenn wir dieses Ziel mittels `make install` aufrufen. In diesem Fall wird die Datei foo in das Heimatverzeichnis des Benutzers kopiert. Diese Vorgehensweise wird häufig bei makefiles von Anwendungen benutzt, damit die Anwendung nach erfolgreicher Kompilierung in das richtige Verzeichnis installiert werden kann.

Dieser Teil ist etwas schwierig zu erklären. Wenn Sie immer noch nicht so richtig verstanden haben, wie `make` funktioniert, wäre es das Beste, sie erstellen sich selber ein einfaches Programm wie "hello world" und eine make Datei wie die weiter oben angegebene, und experimentieren damit selber ein bißchen herum. Als nächstes könnten Sie mehrere Quelltextdateien verwenden, oder in Ihrer Quelltextdatei eine Header-Datei includen. Der Befehl `touch` ist an dieser Stelle ganz hilfreich-er verändert das Datum einer Datei, ohne das Sie diese extra editieren müssen.

2.5.3. Make und include-Dateien

C-Code beginnt häufig mit einer Liste von Dateien, die included werden sollen, zum Beispiel stdio.h. Manche dieser Dateien sind include-Dateien des Systems, andere gehören zum aktuellen Projekt, an dem Sie gerade arbeiten:

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

int main(....
```

Um sicherzustellen, daß diese Datei neu kompiliert wird, wenn foo.h verändert wurde, müssen Sie diese Datei Ihrem Makefile hinzufügen:

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

Sobald Ihr Projekt größer wird und Sie mehr und mehr eigene include-Dateien verwalten müssen wird es nur noch sehr schwer möglich sein, die Übersicht über alle include-Dateien und Dateien, die von diesen abhängen, beizubehalten. Falls Sie eine include-Datei verändern, jedoch das erneute Kompilieren aller Dateien, die von dieser Datei abhängen, vergessen, werden die Folgen verheerend sein. Der `gcc` besitzt eine Option, bei der er Ihre Dateien analysiert und eine Liste aller include-Dateien und deren Abhängigkeiten erstellt: `-MM`.

Wenn Sie das Folgende zu Ihrem Makefile hinzufügen:

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

und `make depend` ausführen, wird die Datei .depend mit einer Liste von Objekt-Dateien, C-Dateien und den include-Dateien auftauchen:

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

Falls Sie foo.h verändern werden beim nächsten Aufruf von `make` alle Dateien, die von foo.h abhängen, neu kompiliert.

Vergessen Sie nicht jedes mal `make depend` aufzurufen, wenn Sie eine include-Datei zu einer Ihrer Dateien hinzugefügt haben.

2.5.4. FreeBSD Makefiles

Makefiles können eher schwierig zu schreiben sein. Glücklicherweise kommen BSD-basierende Systeme wie FreeBSD mit einigen sehr mächtigen solcher Dateien als Teil des Systems daher. Ein sehr gutes Beispiel dafür ist das FreeBSD Portssystem. Hier ist der grundlegende Teil eines typischen Makefiles des Portssystems:

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

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

Wenn wir jetzt in das Verzeichnis dieses Ports wechseln und `make` aufrufen, passiert das Folgende:

1. Es wird überprüft, ob sich der Quelltext für diesen Port bereits auf Ihrem System befindet.
2. Falls dies nicht der Fall ist wird eine FTP-Verbindung zu der URL in MASTER_SITES aufgebaut und der Quelltext heruntergeladen.
3. Die Checksumme für den Quelltext wird berechnet und mit der schon bekannten und für sicher und gut empfundenen verglichen. Damit wird sichergestellt, daß der Quelltext bei der Übertragung nicht beschädigt wurde.
4. Sämtliche Anpassungen, die nötig sind, damit der Quelltext unter FreeBSD funktioniert, werden vorgenommen-dieser Vorgang wird auch *patchen* genannt.
5. Alle speziellen Konfigurationen, die am Quelltext nötig sind, werden vorgenommen. (Viele UNIX® Programmdistributionen versuchen herauszufinden, auf welcher UNIX®-Version sie kompiliert werden sollen und welche optionalen UNIX®-Features vorhanden sind-an dieser Stelle erhalten sie die Informationen im FreeBSD Ports Szenario).
6. Der Quelltext für das Programm wird kompiliert. Im Endeffekt wechseln wir in das

Verzeichnis, in das der Quelltext entpackt wurde, und rufen `make` auf-die eigene make-Datei des Programms besitzt die nötigen Informationen um dieses zu bauen.

7. Wir haben jetzt eine kompilierte Version des Programmes. Wenn wir wollen können wir dieses jetzt testen; wenn wir überzeugt vom Programm sind, können wir `make install` eingeben. Dadurch werden das Programm sowie alle zugehörigen Dateien an die richtige Stelle kopiert; es wird auch ein Eintrag in der `Paketdatenbank` erzeugt, sodaß der Port sehr einfach wieder deinstalliert werden kann, falls wir unsere Meinung über dieses geändert haben.

Ich glaube jetzt werden Sie mit mir übereinstimmen, daß dies ziemlich eindrucksvoll für ein Skript mit vier Zeilen ist!

Das Geheimnis liegt in der letzten Zeile, die `make` anweist, in das makefile des Systems mit dem Namen bsd.port.mk zu sehen. Man kann diese Zeile zwar leicht übersehen, aber hierher kommt all das klevere Zeugs-jemand hat ein makefile geschrieben, welches `make` anweist, alle weiter oben beschriebenen Schritte durchzuführen (neben vielen weiteren Dingen, die ich nicht angesprochen habe, einschließlich der Behandlung sämtlicher Fehler, die auftreten könnten) und jeder kann darauf zurückgreifen, indem er eine einzige Zeile in seine eigene make-Datei einfügt!

Falls Sie einen Blick in die makefiles des Systems werfen möchten, finden Sie diese in /usr/shared/mk. Es ist aber wahrscheinlich besser, wenn Sie damit noch warten, bis Sie ein bißchen mehr Praxiserfahrung mit makefiles gesammelt haben, da die dortigen makefiles sehr kompliziert sind (und wenn Sie sich diese ansehen sollten Sie besser eine Kanne starken Kaffee griffbereit haben!)

2.5.5. Fortgeschrittene Verwendung von `make`

`Make` ist ein sehr mächtiges Werkzeug und kann noch sehr viel mehr als die gezeigten einfachen Beispiele weiter oben. Bedauerlicherweise gibt es mehrere verschiedene Versionen von `make`, und sie alle unterscheiden sich beträchtlich voneinander. Der beste Weg herauszufinden was sie können ist wahrscheinlich deren Dokumentation zu lesen-hoffentlich hat diese Einführung Ihnen genügend Grundkenntnisse vermitteln können, damit Sie dies tun können.

Die Version von make, die in FreeBSD enthalten ist, ist Berkeley make; es gibt eine Anleitung dazu in /usr/shared/doc/psd/12.make. Um sich diese anzusehen, müssen Sie

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

in diesem Verzeichnis ausführen.

Viele Anwendungen in den Ports verwenden GNU make, welches einen sehr guten Satz an "info"-Seiten mitbringt. Falls Sie irgendeinen dieser Ports installiert haben wurde GNU make automatisch als `gmake` mit installiert. Es ist auch als eigenständiger Port und Paket verfügbar.

Um sich die Info-Seiten für GNU make anzusehen müssen Sie die Datei dir in /usr/local/info um einen entsprechenden Eintrag erweitern. Dies beinhaltet das Einfügen einer Zeile wie

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

in die Datei. Nachdem Sie dies getan haben können Sie `info` eingeben und dann den Menüeintrag make auswählen (oder Sie können in Emacs die Tastenkombination `C-h i` verwenden).

2.6. Debuggen

2.6.1. Der Debugger

Der Debugger bei FreeBSD heißt `gdb` (GNU debugger). Sie können Ihn durch die Eingabe von

```
% gdb progname
```

starten, wobei viele Leute ihn vorzugsweise innerhalb von Emacs aufrufen. Sie erreichen dies durch die Eingabe von:

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

Die Verwendung eines Debuggers erlaubt Ihnen Ihr Programm unter kontrollierteren Bedingungen ausführen zu können. Typischerweise können Sie so Zeile für Zeile durch Ihr Programm gehen, die Werte von Variablen untersuchen, diese verändern, dem Debugger sagen er soll das Programm bis zu einem bestimmten Punkt ausführen und dann anhalten, und so weiter und so fort. Sie können damit sogar ein schon laufendes Programm untersuchen, oder eine Datei mit einem Kernspeicherabbild laden um herauszufinden, warum das Programm abgestürzt ist. Es ist sogar möglich damit den Kernel zu debuggen, wobei dies etwas trickreicher als bei den Benutzeranwendungen ist, welche wir in diesem Abschnitt behandeln werden.

Der `gdb` besitzt eine recht gute Online-Hilfe, sowie einen Satz von Info-Seiten, weshalb sich dieser Abschnitt auf ein paar grundlegende Befehle beschränken wird.

Falls Sie den textbasierten Kommandozeilen-Stil abstoßend finden gibt es ein graphisches Front-End dafür (devel/xxgdb) in der Ports-Sammlung.

Dieser Abschnitt ist als Einführung in die Verwendung des `gdb` gedacht und beinhaltet nicht spezielle Themen wie das Debuggen des Kernels.

2.6.2. Ein Programm im Debugger ausführen

Sie müssen das Programm mit der Option `-g` kompiliert haben um den `gdb` effektiv einsetzen zu können. Es geht auch ohne diese Option, allerdings werden Sie dann nur den Namen der Funktion sehen, in der Sie sich gerade befinden, anstatt direkt den zugehörigen Quelltext. Falls Sie eine Meldung wie die folgende sehen:

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

wenn der `gdb` gestartet wird, dann wissen Sie, daß das Programm nicht mit der Option `-g` kompiliert wurde.

Geben Sie in der Eingabeaufforderung des `gdb break main` ein. Dies weist den Debugger an, dass Sie nicht daran interessiert sind, den einleitenden Schritten beim Programmstart zuzusehen und dass am Anfang Ihres Codes die Ausführung beginnen soll. Geben Sie nun `run` ein, um das Programm zu starten - es wird starten und beim Aufruf von `main()` vom Debugger angehalten werden. (Falls Sie sich jemals gewundert haben von welcher Stelle `main()` aufgerufen wird, dann wissen Sie es jetzt!).

Sie können nun Schritt für Schritt durch Ihr Programm gehen, indem Sie `n` drücken. Wenn Sie zu einem Funktionsaufruf kommen können Sie diese Funktion durch drücken von `s` betreten. Sobald Sie sich in einem Funktionsaufruf befinden können Sie diesen durch drücken von `f` wieder verlassen. Sie können auch `up` und `down` verwenden, um sich schnell den Aufrufer einer Funktion anzusehen.

Hier ist ein einfaches Beispiel, wie man mit Hilfe des `gdb` einen Fehler in einem Programm findet. Dies ist unser eigenes Programm (mit einem absichtlich eingebauten Fehler):

```
#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;
}
```

Dieses Programm setzt i auf den Wert `5` und übergibt dies einer Funktion `bazz()`, welche den Wert ausgibt, den Sie von uns erhalten hat.

Wenn wir das Programm kompilieren und ausführen erhalten wir

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

Das ist nicht was wir erwartet hatten! Es ist Zeit, daß wir sehen was hier passiert!

```
% 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)
```

Halt mal! Wieso hat denn anint den Wert 4231? Haben wir dieser Variablen nicht in `main()` den Wert 5 zugewiesen? Gehen wir mal zurück zu `main()` und schauen dort nach.

```
(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! Anscheinend haben wir vergessen i zu initialisieren. Wir wollten eigentlich

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

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

schreiben, haben aber die Zeile mit `i=5;` vergessen. Da wir i nicht initialisiert haben hatte diese Variable gerade den Wert, der in dem ihr zugewiesenen Speicherbereich stand als wir das Programm gestartet haben, welcher in diesem Fall 4231 war.

Der `gdb` zeigt jedes mal, wenn wir eine Funktion betreten oder verlassen, den Inhalt des Stack-Rahmens an, selbst wenn wir uns mit `up` und `down` im Aufruf-Stack umher bewegen. Dabei wird der Name der Funktion sowie der übergebenen Argumente angezeigt, was uns dabei hilft, die Übersicht zu behalten. (Der Stack ist ein Speicherbereich, in dem ein Programm Informationen über die an eine Funktion übergebenen Argumente ablegt, sowie die Rücksprungadresse eines Funktionsaufrufes).

2.6.3. Eine Kernspeicherdatei untersuchen

Eine Kernspeicherdatei ist im Prinzip eine Datei, die den vollständigen Zustand eines Prozesses enthält, als dieses abgestürzt ist. In "den guten alten Zeiten" mußten Programmierer hexadezimale Listen der Kernspeicherdatei ausdrucken und über Maschinencodehandbüchern schwitzen, aber heutzutage ist das Leben etwas einfacher geworden. Zufälligerweise wird die Kernspeicherdatei unter FreeBSD und anderen 4.4BSD-Systemen progname.core anstatt einfach nur core genannt, um deutlich zu machen, zu welchem Programm eine Kernspeicherdatei gehört.

Um eine Kernspeicherdatei zu untersuchen müssen Sie den `gdb` wie gewohnt starten. An Stelle von `break` oder `run` müssen Sie das Folgende eingeben

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

Wenn Sie sich nicht in demselben Verzeichnis befinden wie die Kernspeicherdatei müssen Sie zuerst `dir /path/to/core/file` eingeben.

Sie sollten dann etwas wie folgt sehen:

```
% 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 diesem Fall hieß das Programm a.out, weshalb die Kernspeicherdatei den Namen a.out.core trägt. Wie wir sehen können stürzte das Programm in einer Funktion namens `bazz` ab, als es versuchte auf einen Speicherbereich zuzugreifen, der dem Programm nicht zur Verfügung stand.

Manchmal ist es ganz nützlich zu sehen, wie eine Funktion aufgerufen wurde, da bei komplexen Programmen das eigentliche Problem schon sehr viel weiter oben auf dem Aufruf-Stack aufgetreten sein könnte. Der Befehl `bt` veranlaßt den `gdb` dazu, einen Backtrace des Aufruf-Stacks auszugeben:

```
(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)
```

Die Funktion `end()` wird aufgerufen, wenn ein Programm abstürzt; in diesem Fall wurde die

Funktion `bazz()` aus der `main()`-Funktion heraus aufgerufen.

2.6.4. Ein bereits laufendes Programm untersuchen

Eine der tollsten Features des `gdb` ist die Möglichkeit, damit bereits laufende Programme zu untersuchen. Dies bedeutet natürlich, daß Sie die erforderlichen Rechte dafür besitzen. Ein häufig auftretendes Problem ist das Untersuchen eines Programmes, welches sich selber forkt. Vielleicht will man den Kindprozess untersuchen, aber der Debugger erlaubt einem nur den Zugriff auf den Elternprozess.

Was Sie an solch einer Stelle machen ist, Sie starten einen weiteren `gdb`, ermitteln mit Hilfe von `ps` die Prozess-ID des Kindprozesses, und geben

```
(gdb) attach pid
```

im `gdb` ein, und können dann wie üblich mit der Fehlersuche fortfahren.

"Das ist zwar alles sehr schön," werden Sie jetzt vielleicht denken, "aber in der Zeit, in der ich diese Schritte durchführe, ist der Kindprozess schon längst über alle Berge". Fürchtet euch nicht, edler Leser, denn Ihr müßt wie folgt vorgehen (freundlicherweise zur Verfügung gestellt von den Info-Seite des `gdb`):

```
...
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 */
    ...
```

Alles was Sie jetzt noch tun müssen ist, sich an den Kindprozess ranzuhängen, PauseMode auf `0` zu setzen und auf den `sleep()` Funktionsaufruf zu warten, um zurückzukehren!

2.7. Emacs als Entwicklungsumgebung verwenden

2.7.1. Emacs

Leider werden UNIX®-Systeme nicht mit einem alles-was-du-jemals-brauchst-und-vieles-mehr-megapaket an integrierten Entwicklungsumgebungen ausgestattet, die bei anderen Systemen dabei sind. Trotzdem ist es möglich, seine eigene Entwicklungsumgebung aufzusetzen. Diese wird vielleicht nicht so hübsch und integriert sein, aber dafür können Sie sie Ihren eigenen Wünschen anpassen. Und sie ist frei. Und Sie haben die Quelltexte davon.

Der Schlüssel zu all dem ist Emacs. Es gibt zwar ein paar Leute die ihn hassen, es gibt jedoch auch viele die ihn lieben. Falls Sie zu ersteren gehören befürchte ich, daß dieser Abschnitt Ihnen wenig interessantes zu bieten hat. Des weiteren benötigen Sie eine angemessene Menge an freiem Speicher, um ihn zu benutzen-ich würde 8MB für den Textmodus und 16MB unter X als absolutes Minimum empfehlen, um eine halbwegs brauchbare Performance zu erhalten.

Emacs ist im Prinzip ein extrem anpassbarer Editor- in der Tat ist er so stark veränderbar, daß er eher einem Betriebssystem als einem Editor gleicht! Viele Entwickler und Systemadministratoren erledigen praktisch ihre gesamte Arbeit aus Emacs heraus und beenden ihn nur, um sich komplett auszuloggen.

Es ist nicht einmal möglich alles hier zusammenzufassen, was man mit dem Emacs machen kann. Im Folgenden werden einige Features aufgelistet, die für einen Entwickler interessant sein könnten:

- Sehr mächtiger Editor, der suchen-und-ersetzen mit Zeichenfolgen und regulären Ausdrücken (Pattern) sowie das direkte Anspringen von Anfang/Ende von Blockausdrücken erlaubt, etc, etc.
- Pull-Down Menüs und eine Online-Hilfe.
- Sprachunabhängige Syntaxhervorhebung und automatische Einrückung.
- Vollständig konfigurierbar.
- Sie können Programme im Emacs kompilieren und debuggen.
- Bei Kompilationsfehlern können Sie direkt zu der entsprechenden Zeile im Quelltext springen.
- Benutzerfreundliches Front-End für das `info`-Programm, um die GNU Hypertext Dokumentation inklusive der Dokumentation des Emacs selber.
- Benutzerfreundliches Front-End für den `gdb` um sich beim Verfolgen der Programmanweisungen den zugehörigen Quelltext anzeigen zu lassen.
- Sie können E-Mails und News im Usenet lesen, während ihr Programm kompiliert wird.

Und zweifelsfrei viele weitere Punkte, die ich übersehen habe.

Emacs kann unter FreeBSD über den editors/emacs Port installiert werden.

Sobald er installiert ist starten Sie ihn, und geben dann `C-h t` ein, um die Einführung in Emacs zu lesen-d.h. Sie sollen bei gedrückter `Strg`-Taste die `h`-Taste drücken, beide wieder loslassen und anschließend `t` drücken. (Alternativ können Sie mit der Maus den Eintrag Emacs Tutorial aus dem **Hilfe**-Menü auswählen).

Obwohl der Emacs Menüs besitzt ist das Erlernen der Tastaturkombinationen lohnenswert, da man beim Editieren sehr viel schneller Tastenkombinationen eingeben kann, als die Maus zu finden und mit dieser dann an der richtigen Stelle zu klicken. Und wenn Sie sich mit erfahrenen Emacs-Benutzern unterhalten werden Sie feststellen, daß diese häufig nebenbei Ausdrücke wie “M-x replace-s RET foo RET bar RET” verwenden, weshalb das Erlernen dieser sehr nützlich ist. Und Emacs hat auf jeden Fall weit mehr nützliche Funktionen als das diese in der Menüleiste unterzubringen wären.

Zum Glück ist es sehr einfach die jeweiligen Tastaturkombinationen herauszubekommen, da diese

direkt neben den Menüeinträgen stehen. Meine Empfehlung wäre, den Menüeintrag für, sagen wir, das Öffnen einer Datei zu verwenden, bis Sie die Funktionsweise verstanden haben und sie mit dieser vertraut sind, und es dann mit C-x C-f versuchen. Wenn Sie damit zufrieden sind, gehen Sie zum nächsten Menüeintrag.

Falls Sie sich nicht daran erinnern können, was eine bestimmte Tastenkombination macht, wählen Sie Describe Key aus dem **Hilfe**-Menü aus und geben Sie die Tastenkombination ein-Emacs sagt Ihnen dann was diese macht. Sie können ebenfalls den Menüeintrag Command Apropos verwenden, um alle Befehle, die ein bestimmtes Wort enthalten, mit den zugehörigen Tastenkombinationen aufgelistet zu bekommen.

Übrigends bedeutet der Ausdruck weiter oben, bei gedrückter `Meta`-Taste `x` zu drücken, beide wieder loszulassen, `replace-s` einzugeben (Kurzversion für `replace-string`-ein weiteres Feature des Emacs ist, daß Sie Befehle abkürzen können), anschließend die `return`-Taste zu drücken, dann `foo` einzugeben (die Zeichenkette, die Sie ersetzen möchten), dann wieder `return`, dann die Leertaste zu drücken (die Zeichenkette, mit der Sie `foo` ersetzen möchten) und anschließend erneut `return` zu drücken. Emacs wird dann die gewünschte suchen-und-ersetzen-Operation ausführen.

Wenn Sie sich fragen was in aller Welt die `Meta`-Taste ist, das ist eine spezielle Taste die viele UNIX®-Workstations besitzen. Bedauerlicherweise haben PCs keine solche Taste, und daher ist es üblicherweise die `alt`-Taste (oder falls Sie Pech haben die `Esc`-Taste).

Oh, und um den Emacs zu verlassen müssen sie `C-x C-c` (das bedeutet, Sie müssen bei gedrückter `Strg`-Taste zuerst `x` und dann `c` drücken) eingeben. Falls Sie noch irgendwelche ungespeicherten Dateien offen haben wird Emacs Sie fragen ob Sie diese speichern wollen. (Ignorieren Sie bitte die Stelle der Dokumentation, an der gesagt wird, daß `C-z` der übliche Weg ist, Emacs zu verlassen-dadurch wird der Emacs in den Hintergrund geschaltet, was nur nützlich ist, wenn Sie an einem System ohne virtuelle Terminals arbeiten).

2.7.2. Emacs konfigurieren

Emacs kann viele wundervolle Dinge; manche dieser Dinge sind schon eingebaut, andere müssen erst konfiguriert werden.

Anstelle einer proprietären Macrosprache verwendet der Emacs für die Konfiguration eine speziell für Editoren angepaßte Version von Lisp, auch bekannt als Emacs Lisp. Das Arbeiten mit Emacs Lisp kann sehr hilfreich sein, wenn Sie darauf aufbauend etwas wie Common Lisp lernen möchten. Emacs Lisp hat viele Features von Common Lisp obwohl es beträchtlich kleiner ist (und daher auch einfacher zu beherrschen).

Der beste Weg um Emacs Lisp zu erlernen besteht darin, sich das Emacs Tutorial herunterzuladen.

Es ist jedoch keine Kenntnis von Lisp erforderlich, um mit der Konfiguration von Emacs zu beginnen, da ich eine beispielhafte .emacs-Datei hier eingefügt habe, die für den Anfang ausreichen sollte. Kopieren Sie diese einfach in Ihr Heimverzeichnis und starten Sie den Emacs neu, falls dieser bereits läuft; er wird die Befehle aus der Datei lesen und Ihnen (hoffentlich) eine brauchbare Grundeinstellung bieten.

2.7.3. Eine beispielhafte .emacs-Datei

Bedauerlicherweise gibt es hier viel zu viel, um es im Detail zu erklären; es gibt jedoch ein oder zwei Punkte, die besonders erwähnenswert sind.

- Alles was mit einem `;` anfängt ist ein Kommentar und wird von Emacs ignoriert.
- In der ersten Zeile mit `-- Emacs-Lisp --` sorgt dafür, daß wir die Datei .emacs in Emacs selber editieren können und uns damit alle tollen Features zum Editieren von Emacs Lisp zur Verfügung stehen. Emacs versucht dies normalerweise anhand des Dateinamens auszumachen, was vielleicht bei .emacs nicht funktionieren könnte.
- Die `Tab`-Taste ist in manchen Modi an die Einrückungsfunktion gebunden, so daß beim drücken dieser Taste die aktuelle Zeile eingerückt wird. Wenn Sie ein tab-Zeichen in einen Text, welchen auch immer Sie dabei schreiben, einfügen wollen, müssen Sie bei gedrückter `Strg`-Taste die `Tab` -Taste drücken.
- Diese Datei unterstützt Syntax Highlighting für C, C++, Perl, Lisp und Scheme, indem die Sprache anhand des Dateinamens erraten wird.
- Emacs hat bereits eine vordefinierte Funktion mit dem Namen `next-error`. Diese erlaubt es einem, in einem Fenster mit der Kompilierungsausgabe mittels `M-n` von einem zum nächsten Kompilierungsfehler zu springen; wir definieren eine komplementäre Funktion `previous-error`, die es uns erlaubt, mittels `M-p` von einem zum vorherigen Kompilierungsfehler zu springen. Das schönste Feature von allen ist, daß mittels `C-c C-c` die Quelltextdatei, in der der Fehler aufgetreten ist, geöffnet und die betreffende Zeile direkt angesprungen wird.
- Wir aktivieren die Möglichkeit von Emacs als Server zu agieren, so daß wenn Sie etwas außerhalb von Emacs machen und eine Datei editieren möchten, Sie einfach das folgende eingeben können

  ```
  % emacsclient filename
  ```

  und dann die Datei in Ihrem Emacs editieren können!

Beispiel 1. Eine einfache .emacs-Datei

```
;; -*-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. Erweitern des von Emacs unterstützten Sprachbereichs

Das ist jetzt alles sehr schön wenn Sie ausschließlich in einer der Sprachen programmieren wollen, um die wir uns bereits in der .emacs-Datei gekümmert haben (C, C++, Perl, Lisp und Scheme), aber was passiert wenn eine neue Sprache namens "whizbang" herauskommt, mit jeder Menge neuen tollen Features?

Als erstes muß festgestellt werden, ob whizbang mit irgendwelchen Dateien daherkommt, die Emacs etwas über die Sprache sagen. Diese enden üblicherweise auf .el, der Kurzform für "Emacs Lisp". Falls whizbang zum Beispiel ein FreeBSD Port ist, könnten wir diese Dateien mittels

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

finden und durch Kopieren in das Emacs-seitige Lisp-Verzeichnis installieren. Unter FreeBSD ist dies /usr/local/shared/emacs/site-lisp.

Wenn zum Beispiel die Ausgabe des find-Befehls wie folgt war

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

könnten wir das folgende tun

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

Als nächstes müssen wir festlegen, welche Dateiendung Quelltextdateien für whizbang haben. Lassen Sie uns um der Argumente Willen annehmen, die Dateiendung sei .wiz. Wir müssen dann einen Eintrag unserer .emacs-Datei hinzufügen um sicherzustellen, daß Emacs die Informationen in whizbang.el auch verwenden kann.

Suchen Sie den auto-mode-alist Eintrag in der .emacs-Datei und fügen Sie an dieser Stelle eine Zeile wie folgt für whizbang hinzu:

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

Dies bedeutet das Emacs automatisch in den `whizbang-mode` wechseln wird, wenn Sie eine Datei mit der Dateiendung .wiz editieren.

Direkt darunter werden Sie den Eintrag font-lock-auto-mode-list finden. Erweitern Sie den `whizbang-mode` um diesen wie folgt:

```
;; 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-mode)
  "List of modes to always start in font-lock-mode")
```

Dies bedeutet das Emacs immer `font-lock-mode` (z.B. Syntax Highlighting) aktiviert, wenn Sie eine .wiz-Datei editieren.

Und das ist alles was benötigt wird. Falls es weitere Dinge gibt, die automatisch beim Öffnen einer .wiz-Datei ausgeführt werden sollen, können Sie einen `whizbang-mode hook`-Eintrag hinzufügen (für ein einfaches Beispiel, welches `auto-indent` hinzufügt, sehen Sie sich bitte `my-scheme-mode-hook` an).

2.8. Weiterführende Literatur

Für Informationen zum Aufsetzen einer Entwicklungsumgebung, um Fehlerbehebungen an FreeBSD selber beizusteuern sehen Sie sich bitte development(7) an.

- 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

Kapitel 3. Sicheres Programmieren

3.1. Zusammenfassung

Dieses Kapitel beschreibt einige Sicherheitsprobleme, die UNIX®-Programmierer seit Jahrzehnten quälen, und inzwischen verfügbare Werkzeuge, die Programmierern helfen, Sicherheitslücken in ihrem Quelltext zu vermeiden.

3.2. Methoden des sicheren Entwurfs

Sichere Anwendungen zu schreiben erfordert eine sehr skeptische und pessimistische Lebenseinstellung. Anwendungen sollten nach dem Prinzip der "geringsten Privilegien" ausgeführt werden, sodass kein Prozess mit mehr als dem absoluten Minimum an Zugriffsrechten arbeitet, die er zum Erfüllen seiner Aufgabe benötigt. Wo es möglich ist, sollte Quelltext, der bereits überprüft wurde, wiederverwendet werden, um häufige Fehler, die andere schon korrigiert haben, zu vermeiden.

Eine der Stolperfallen der UNIX®-Umgebung ist, dass es sehr einfach ist Annahmen über die Konsistenz der Umgebung zu machen. Anwendungen sollten Nutzereingaben (in allen Formen) niemals trauen, genauso wenig wie den System-Ressourcen, der Inter-Prozess-Kommunikation oder dem zeitlichen Ablauf von Ereignissen. UNIX®-Prozesse arbeiten nicht synchron. Daher sind logische Operationen selten atomar.

3.3. Puffer-Überläufe

Puffer-Überläufe gibt es schon seit den Anfängen der Von-Neuman-Architektur 1. Sie erlangten zum ersten Mal durch den Internetwurm Morris im Jahre 1988 öffentliche Bekanntheit. Unglücklicherweise funktioniert der gleiche grundlegende Angriff noch heute. Die bei weitem häufigste Form eines Puffer-Überlauf-Angriffs basiert darauf, den Stack zu korrumpieren.

Die meisten modernen Computer-Systeme verwenden einen Stack, um Argumente an Prozeduren zu übergeben und lokale Variablen zu speichern. Ein Stack ist ein last-in-first-out-Puffer (LIFO) im hohen Speicherbereich eines Prozesses. Wenn ein Programm eine Funktion aufruft wird ein neuer "Stackframe" erzeugt. Dieser besteht aus den Argumenten, die der Funktion übergeben wurden und einem variabel grossem Bereich für lokale Variablen. Der "Stack-Pointer" ist ein Register, dass die aktuelle Adresse der Stack-Spitze enthält. Da sich dieser Wert oft ändert, wenn neue Werte auf dem Stack abgelegt werden, bieten viele Implementierungen einen "Frame-Pointer", der nahe am Anfang des Stack-Frames liegt und es so leichter macht lokale Variablen relativ zum aktuellen Stackframe zu adressieren. 1 Die Rücksprungadresse der Funktionen werden ebenfalls auf dem Stack gespeichert und das ist der Grund für Stack-Überlauf-Exploits. Denn ein böswilliger Nutzer kann die Rücksprungadresse der Funktion überschreiben indem er eine lokale Variable in der Funktion überlaufen lässt, wodurch es ihm möglich ist beliebigen Code auszuführen.

Obwohl Stack-basierte Angriffe bei weitem die Häufigsten sind, ist es auch möglich den Stack mit einem Heap-basierten (malloc/free) Angriff zu überschreiben.

Die C-Programmiersprache führt keine automatischen Bereichsüberprüfungen bei Feldern oder

Zeigern durch, wie viele andere Sprachen das tun. Außerdem enthält die C-Standardbibliothek eine Handvoll sehr gefährlicher Funktionen.

`strcpy`(char *dest, const char *src)	Kann den Puffer dest überlaufen lassen
`strcat`(char *dest, const char *src)	Kann den Puffer dest überlaufen lassen
`getwd`(char *buf)	Kann den Puffer buf überlaufen lassen
`gets`(char *s)	Kann den Puffer s überlaufen lassen
`[vf]scanf`(const char *format, ...)	Kann sein Argument überlaufen lassen
`realpath`(char *path, char resolved_path[])	Kann den Puffer path überlaufen lassen
`[v]sprintf`(char *str, const char *format, ...)	Kann den Puffer str überlaufen lassen

3.3.1. Puffer-Überlauf Beispiel

Das folgende Quellcode-Beispiel enthält einen Puffer-Überlauf, der darauf ausgelegt ist die Rücksprungadresse zu überschreiben und die Anweisung direkt nach dem Funktionsaufruf zu überspringen. (Inspiriert durch 4)

```
#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;
}
```

Betrachten wir nun, wie das Speicherabbild dieses Prozesses aussehen würde, wenn wir 160 Leerzeichen in unser kleines Programm eingeben, bevor wir Enter drücken.

Offensichtlich kann man durch böswilligere Eingaben bereits kompilierten Programmtext ausführen (wie z.B. exec(/bin/sh)).

3.3.2. Puffer-Überläufe vermeiden

Die direkteste Lösung, um Stack-Überläufe zu vermeiden, ist immer grössenbegrenzten Speicher

und String-Copy-Funktionen zu verwenden. `strncpy` und `strncat` sind Teil der C-Standardbibliothek. Diese Funktionen akzeptieren einen Längen-Parameter. Dieser Wert sollte nicht größer sein als die Länge des Zielpuffers. Die Funktionen kopieren dann bis zu `length` Bytes von der Quelle zum Ziel. Allerdings gibt es einige Probleme. Keine der Funktionen garantiert, dass die Zeichenkette NUL-terminiert ist, wenn die Größe des Eingabepuffers so groß ist wie das Ziel. Außerdem wird der Parameter length zwischen strncpy und strncat inkonsistent definiert, weshalb Programmierer leicht bezüglich der korrekten Verwendung durcheinander kommen können. Weiterhin gibt es einen spürbaren Leistungsverlust im Vergleich zu `strcpy`, wenn eine kurze Zeichenkette in einen großen Puffer kopiert wird. Denn `strncpy` fült den Puffer bis zur angegebenen Länge mit NUL auf.

In OpenBSD wurde eine weitere Möglichkeit zum kopieren von Speicherbereichen implementiert, die dieses Problem umgeht. Die Funktionen `strlcpy` und `strlcat` garantieren, dass das Ziel immer NUL-terminiert wird, wenn das Argument length ungleich null ist. Für weitere Informationen über diese Funktionen lesen Sie bitte 6. Die OpenBSD-Funktionen `strlcpy` und `strlcat` sind seit Version 3.3 auch in FreeBSD verfügbar.

3.3.2.1. Compiler-basierte Laufzeitüberprüfung von Grenzen

Unglücklicherweise gibt es immer noch sehr viel Quelltext, der allgemein verwendet wird und blind Speicher umherkopiert, ohne eine der gerade besprochenen Funktionen, die Begrenzungen unterstützen, zu verwenden. Glücklicherweise gibt es einen Weg, um solche Angriffe zu verhindern - Überprüfung der Grenzen zur Laufzeit, die in verschiedenen C/C++ Compilern eingebaut ist.

ProPolice ist eine solche Compiler-Eigenschaft und ist in den gcc(1) Versionen 4.1 und höher integriert. Es ersetzt und erweitert die gcc(1) StackGuard-Erweiterung von früher.

ProPolice schützt gegen stackbasierte Pufferüberläufe und andere Angriffe durch das Ablegen von Pseudo-Zufallszahlen in Schlüsselbereichen des Stacks bevor es irgendwelche Funktionen aufruft. Wenn eine Funktion beendet wird, werden diese "Kanarienvögel" überprüft und wenn festgestellt wird, dass diese verändert wurden wird das Programm sofort abgebrochen. Dadurch wird jeglicher Versuch, die Rücksprungadresse oder andere Variablen, die auf dem Stack gespeichert werden, durch die Ausführung von Schadcode zu manipulieren, nicht funktionieren, da der Angreifer auch die Pseudo-Zufallszahlen unberührt lassen müsste.

Ihre Anwendungen mit ProPolice neu zu kompilieren ist eine effektive Maßnahme, um sie vor den meisten Puffer-Überlauf-Angriffen zu schützen, aber die Programme können noch immer kompromittiert werden.

3.3.2.2. Bibliotheks-basierte Laufzeitüberprüfung von Grenzen

Compiler-basierte Mechanismen sind bei Software, die nur im Binärformat vertrieben wird, und die somit nicht neu kompiliert werden kann völlig nutzlos. Für diesen Fall gibt es einige Bibliotheken, welche die unsicheren Funktionen der C-Bibliothek (`strcpy`, `fscanf`, `getwd`, etc..) neu implementieren und sicherstellen, dass nicht hinter den Stack-Pointer geschrieben werden kann.

- libsafe
- libverify
- libparanoia

Leider haben diese Bibliotheks-basierten Verteidigungen mehrere Schwächen. Diese Bibliotheken schützen nur vor einer kleinen Gruppe von Sicherheitslücken und sie können das eigentliche Problem nicht lösen. Diese Maßnahmen können versagen, wenn die Anwendung mit -fomit-frame -pointer kompiliert wurde. Außerdem kann der Nutzer die Umgebungsvariablen LD_PRELOAD und LD_LIBRARY_PATH überschreiben oder löschen.

3.4. SetUID-Themen

Es gibt zu jedem Prozess mindestens sechs verschiedene IDs, die diesem zugeordnet sind. Deshalb müssen Sie sehr vorsichtig mit den Zugriffsrechten sein, die Ihr Prozess zu jedem Zeitpunkt besitzt. Konkret bedeutet dass, das alle seteuid-Anwendungen ihre Privilegien abgeben sollten, sobald sie diese nicht mehr benötigen.

Die reale Benutzer-ID kann nur von einem Superuser-Prozess geändert werden. Das Programm login setzt sie, wenn sich ein Benutzer am System anmeldet, und sie wird nur selten geändert.

Die effektive Benutzer-ID wird von der Funktion `exec()` gesetzt, wenn ein Programm das seteuid-Bit gesetzt hat. Eine Anwendung kann `seteuid()` jederzeit aufrufen, um die effektive Benutzer-ID entweder auf die reale Benutzer-ID oder die gespeicherte set-user-ID zu setzen. Wenn eine der `exec()`-Funktionen die effektive Benutzer-ID setzt, wird der vorherige Wert als gespeicherte set-user-ID abgelegt.

3.5. Die Umgebung ihrer Programme einschränken

Die herkömmliche Methode, um einen Prozess einzuschränken, besteht in dem Systemaufruf `chroot()`. Dieser Aufruf ändert das Wurzelverzeichnis, auf das sich alle Pfadangaben des Prozesses und jegliche Kind-Prozesse beziehen. Damit dieser Systemaufruf gelingt, muss der Prozess Ausführungsrechte (Durchsuchungsrechte) für das Verzeichnis haben, auf das er sich bezieht. Die neue Umgebung wird erst wirksam, wenn Sie mittels `chdir()` in Ihre neue Umgebung wechseln. Es sollte erwähnt werden, dass ein Prozess recht einfach aus der chroot-Umgebung ausbrechen kann, wenn er root-Rechte besitzt. Das kann man erreichen, indem man Gerätedateien anlegt, um Kernel-Speicher zu lesen, oder indem man einen Debugger mit einem Prozess außerhalb seiner chroot(8) -Umgebung verbindet, oder auf viele andere kreative Wege.

Das Verhalten des Systemaufrufs `chroot()` kann durch die kern.chroot.allow_open_directories `sysctl`-Variable beeinflusst werden. Wenn diese auf 0 gesetzt ist, wird `chroot()` mit EPERM fehlschlagen, wenn irgendwelche Verzeichnisse geöffnet sind. Wenn die Variable auf den Standardwert 1 gesetzt ist, wird `chroot()` mit EPERM fehlschlagen, wenn irgendwelche Verzeichnisse geöffnet sind und sich der Prozess bereits in einer `chroot()`-Umgebung befindet. Bei jedem anderen Wert wird die Überprüfung auf geöffnete Verzeichnisse komplett umgangen.

3.5.1. Die Jail-Funktionalität in FreeBSD

Das Konzept einer Jail (Gefängnis) erweitert `chroot()`, indem es die Macht des Superusers einschränkt, um einen echten 'virtuellen Server' zu erzeugen. Wenn ein solches Gefängnis einmal eingerichtet ist, muss die gesamte Netzwerkkommunikation über eine bestimmte IP-Adresse erfolgen und die "root-Privilegien" innerhalb der Jail sind sehr stark eingeschränkt.

Solange Sie sich in einer Jail befinden, werden alle Tests auf Superuser-Rechte durch den Aufruf von `suser()` fehlschlagen. Allerdings wurden einige Aufrufe von `suser()` abgeändert, um die neue `suser_xxx()`-Schnittstelle zu implementieren. Diese Funktion ist dafür verantwortlich, festzustellen, ob bestimmte Superuser-Rechte einem eingesperrten Prozess zur Verfügung stehen.

Ein Superuser-Prozess innerhalb einer Jail darf folgendes:

- Berechtigungen verändern mittels: `setuid`, `seteuid`, `setgid`, `setegid`, `setgroups`, `setreuid`, `setregid`, `setlogin`
- Ressourcenbegrenzungen setzen mittels `setrlimit`
- Einige sysctl-Variablen (kern.hostname) verändern
- `chroot()`
- Ein Flag einer vnode setzen: `chflags`, `fchflags`
- Attribute einer vnode setzen wie Dateiberechtigungen, Eigentümer, Gruppe, Größe, Zugriffszeit und Modifikationszeit
- Binden eines Prozesses an einen öffentlichen privilegierten Port (ports 1024)

`Jail`s sind ein mächtiges Werkzeug, um Anwendungen in einer sicheren Umgebung auszuführen, aber sie haben auch ihre Nachteile. Derzeit wurden die IPC-Mechanismen noch nicht an `suser_xxx` angepasst, so dass Anwendungen wie MySQL nicht innerhalb einer Jail ausgeführt werden können. Der Superuser-Zugriff hat in einer Jail nur eine sehr eingeschränkte Bedeutung, aber es gibt keine Möglichkeit zu definieren was "sehr eingeschränkt" heißt.

3.5.2. POSIX®.1e Prozess Capabilities

POSIX® hat einen funktionalen Entwurf (Working Draft) herausgegeben, der Ereignisüberprüfung, Zugriffskontrolllisten, feiner einstellbare Privilegien, Informationsmarkierung und verbindliche Zugriffskontrolle enthält.

Dies ist im Moment in Arbeit und das Hauptziel des TrustedBSD-Projekts. Ein Teil der bisherigen Arbeit wurde in FreeBSD-CURRENT übernommen (cap_set_proc(3)).

3.6. Vertrauen

Eine Anwendung sollte niemals davon ausgehen, dass irgendetwas in der Nutzerumgebung vernünftig ist. Das beinhaltet (ist aber sicher nicht darauf beschränkt): Nutzereingaben, Signale, Umgebungsvariablen, Ressourcen, IPC, mmaps, das Arbeitsverzeichnis im Dateisystem, Dateideskriptoren, die Anzahl geöffneter Dateien, etc..

Sie sollten niemals annehmen, dass Sie jede Art von inkorrekten Eingaben abfangen können, die ein Nutzer machen kann. Stattdessen sollte Ihre Anwendung positive Filterung verwenden, um nur eine bestimmte Teilmenge an Eingaben zuzulassen, die Sie für sicher halten. Ungeeignete Datenüberprüfung ist die Ursache vieler Exploits, besonders für CGI-Skripte im Internet. Bei Dateinamen müssen Sie besonders vorsichtig sein, wenn es sich um Pfade ("../", "/"), symbolische Verknüpfungen und Shell-Escape-Sequenzen handelt.

Perl bietet eine wirklich coole Funktion, den sogenannten "Taint"-Modus, der verwendet werden

kann, um zu verhindern, dass Skripte Daten, die von außerhalb des Programmes stammen, auf unsichere Art und Weise verwenden. Dieser Modus überprüft Kommandozeilenargumente, Umgebungsvariablen, Lokalisierungsinformationen, die Ergebnisse von Systemaufrufen (`readdir()`, `readlink()`, `getpwxxx()`) und alle Dateieingaben.

3.7. Race-Conditions

Eine Race-Condition ist ein unnormales Verhalten, das von einer unerwarteten Abhängigkeit beim Timing von Ereignissen verursacht wird. Mit anderen Worten heißt das, ein Programmierer nimmt irrtümlicher Weise an, dass ein bestimmtes Ereignis immer vor einem anderen stattfindet.

Einige der häufigsten Ursachen für Race-Conditions sind Signale, Zugriffsprüfungen und das Öffnen von Dateien. Signale sind von Natur aus asynchrone Ereignisse, deshalb ist besondere Vorsicht im Umgang damit geboten. Das Prüfen des Zugriffs mittels der Aufrufe `access(2)` gefolgt von `open(2)` ist offensichtlich nicht atomar. Benutzer können zwischen den beiden Aufrufen Dateien verschieben. Stattdessen sollten privilegierte Anwendungen `seteuid()` direkt gefolgt von `open()` aufrufen. Auf die gleiche Art sollte eine Anwendung immer eine korrekte Umask vor dem Aufruf von `open()` setzen, um störende Aufrufe von `chmod()` zu umgehen.

Kapitel 4. Lokalisierung und Internationalisierung - L10N und I18N

4.1. I18N-konforme Anwendungen programmieren

Um Ihre Anwendung verwendbarer für andere Sprachen zu machen, hoffen wir, dass Sie I18N-konform programmieren. Der GNU gcc-Compiler und Bibliotheken für grafische Benutzeroberflächen wie QT und GTK unterstützen I18N durch eine spezielle Verarbeitung von Zeichenketten. Das Erstellen eines I18N-konformen Programms ist sehr einfach und erlaubt anderen Mitwirkenden, Ihre Programme leichter in andere Sprachen zu übersetzen. Lesen Sie die Bibliothek-spezifischen I18N-Dokumentationen für weitere Details.

Im Gegensatz zur allgemeinen Meinung ist I18N-konformer Code einfach zu programmieren. Üblicherweise umfasst dies nur das Einbetten Ihrer Zeichenketten in Bibliothek-spezifische Funktionen. Stellen Sie außerdem bitte sicher, dass Sie Unterstützung für Unicode- und Multibyte-Zeichen vorsehen.

4.1.1. Ein Aufruf, die I18N-Bemühungen zu vereinheitlichen

Wir sind darauf aufmerksam geworden, dass die einzelnen I18N-/L10N-Bemühungen für jedes Land wiederholt wurden. Viele von uns haben somit unproduktiverweise das Rad immer wieder neu erfunden. Wir hoffen, dass die verschiedenen großen Gruppen für I18N Ihre Bemühungen in einer Gruppe vereinen können, ähnlich der Zuständigkeit des Core-Teams.

Derzeit hoffen wir, dass wenn Sie I18N-konforme Programme schreiben oder portieren, diese an die betreffenden FreeBSD-Mailinglisten jedes Landes schicken, um sie testen zu lassen. Wir hoffen in Zukunft, Anwendungen zu entwickeln, die in allen Sprachen direkt und ohne unsaubere Änderungen funktionieren.

Die FreeBSD internationalization-Mailingliste ist eingerichtet worden. Wenn Sie I18N-/L10N-Entwickler sind, schicken Sie bitte Ihre Kommentare, Ideen, Fragen und alles, das Sie mit dem Thema in Verbindung bringen, dorthin.

4.1.2. Perl und Python

Perl und Python bieten Bibliotheken für I18N und zur Behandlung von Unicode-Zeichen. Bitte nutzen Sie diese für I18N-Konformität.

4.2. Lokalisierte Nachrichten mit POSIX.1 Native Language Support (NLS)

Über die Basisfunktionen von I18N hinaus, wie das Bereitstellen von verschiedenen Eingabecodierungen oder die diversen nationalen Konventionen, zum Beispiel die verschiedenen Dezimalpunkte, ist es auf einem höheren Level von I18N möglich, die Ausgabe von Programmen zu lokalisieren. Ein Weg dies zu tun besteht in der Nutzung der POSIX.1 NLS-Funktionen von FreeBSD.

4.2.1. Organisation von lokalisierten Mitteilungen in Katalog Dateien

POSIX.1 NLS basiert auf Katalogdateien, welche die lokalisierten Mitteilungen in der entsprechenden Codierung enthalten. Die Mitteilungen sind in Sets organisiert und jede Mitteilung ist durch eine eindeutige Zahl in dem jeweiligen Set identifiziert. Die Katalogdateien werden nach der Lokale, von den jeweiligen lokalisierten Mitteilungen, die sie enthalten, gefolgt von der `.msg` Endung benannt. Zum Beispiel werden die ungarischen Mitteilungen für das ISO8859-2 Encoding in einer Datei mit dem Dateinamen hu_HU.ISO8859-2 gespeichert.

Diese Katalogdateien sind normale Textdateien, welche die nummerierten Mitteilungen enthalten. Es ist möglich Kommentare in die Dateien zu schreiben, indem Sie ein `$`-Zeichen an den Anfang der Zeile setzen. Das Setzen von Grenzen wird ebenfalls durch spezielle Kommentare möglich wobei das Schlüsselwort `set` direkt nach dem `$`-Zeichen folgen muss. Dem Schlüsselwort `set` folgt dann die Set-Nummer. Ein Beispiel:

```
$set 1
```

Der aktuelle Mitteilungseintrag startet mit der Mitteilungsnummer gefolgt von der lokalisierten Nachricht. Die bekannten Modifikatoren von printf(3) werden akzeptiert:

```
15 "File not found: %s\n"
```

Die Katalogdateien müssen in binärer Form vorliegen, bevor sie von einem Programm benutzt werden können. Dies wird mit dem gencat(1) Tool durchgeführt. Das erste Argument ist der Dateiname des kompilierten Katalogs und die weiteren Argumente sind die Eingabekataloge. Die lokalisierten Mitteilungen können auf mehrere Katalogdateien aufgeteilt sein. Danach werden dann alle auf einmal mit dem gencat(1) Tool kompiliert.

4.2.2. Nutzung der Katalogdateien im Quellcode

Das Benutzen der Katalogdateien ist einfach. Um die relevante Funktion zu nutzen, muss nl_types.h in die Quelldatei eingefügt werden. Bevor ein Katalog benutzt werden kann, muss er mit catopen(3) geöffnet werden. Die Funktion hat 2 Argumente. Der erste Parameter ist der Name des installierten und kompilierten Katalogs. Normalerweise wird der Name des Programmes, zum Beispiel grep, genutzt. Dieser Name wird zum Suchen der kompilierten Katalogdatei benutzt. Der Aufruf von catopen(3) sucht nach dieser Datei in /usr/shared/nls/locale/catname und in /usr/local/shared/nls/locale/catname, wobei `locale` die gesetzte Lokale und `catname` der Katalogname ist. Der zweite Parameter ist eine Konstante, die zwei Werte haben kann:

- `NL_CAT_LOCALE`, hat die Bedeutung, dass die benutzte Katalogdatei auf `LC_MESSAGES` basiert.
- `0`, hat die Bedeutung, dass `LANG` benutzt wird, um die Katalogdatei zu öffnen.

Der catopen(3) Aufruf gibt einen Katalogidentifizierer vom Type `nl_catd` zurück. Sehen Sie in der Manualpage nach, um eine Liste mit möglichen Fehlercodes zu erhalten.

Nach dem Öffnen eines Katalogs, kann catgets(3) benutzt werden, um Mitteilungen zu erhalten. Der erste Parameter ist der Katalogidentifizierer, der von catopen(3) zurück gegeben wurde, das zweite

ist die Nummer des Sets, das dritte die Nummer der Mitteilung und das vierte ist eine Fallbackmitteilung, die angezeigt wird, falls die gewünschte Mitteilung in der Katalogdatei nicht verfügbar ist.

Nach der Nutzung der Katalogdatei, muss sie mit dem Kommando catclose(3), geschlossen werden. Es besitzt ein Argument, die Katalog ID.

4.2.3. Ein Beispiel aus der Praxis

Das folgende Beispiel zeigt einen einfachen Weg wie man NLS-Kataloge flexibel nutzen kann.

Die nachfolgenden Zeilen müssen in eine allgemeine Headerdatei, die in allen Quelldateien vorhanden ist, die lokalisierte Mitteilungen benutzen, eingefügt werden:

```
#ifdef WITHOUT_NLS
#define getstr(n)         nlsstr[n]
#else
#include <nl_types.h>

extern nl_catd            catalog;
#define getstr(n)         catgets(catalog, 1, n, nlsstr[n])
#endif

extern char              *nlsstr[];
```

Als nächstes fügen Sie die folgenden Zeilen in den globalen Deklarationsteil der Hauptquelldatei ein:

```
#ifndef WITHOUT_NLS
#include <nl_types.h>
nl_catd   catalog;
#endif

/*
* Default messages to use when NLS is disabled or no catalog
* is found.
*/
char    *nlsstr[] = {
        "",
/* 1*/  "some random message",
/* 2*/  "some other message"
};
```

Als nächstes kommt der Code der den Katalog öffnet, liest und schließt:

```
#ifndef WITHOUT_NLS
 catalog = catopen("myapp", NL_CAT_LOCALE);
#endif
```

```
...

printf(getstr(1));

...

#ifndef WITHOUT_NLS
 catclose(catalog);
#endif
```

4.2.3.1. Reduzierung von zu lokalisierenden Zeichenketten

Es gibt einen guten Weg, Zeichenketten die lokaliesert werden müssen, durch den Einsatz von libc-Fehlermeldungen zu reduzieren. Dadurch vermeidet man Duplikate und erstellt gleiche Meldungen für häufige Fehlermeldungen, die bei vielen Programmen auftreten können.

Als erstes ist hier ein Beispiel, dass keine libc-Fehlermeldungen benutzt:

```
#include <err.h>
...
if (!S_ISDIR(st.st_mode))
 err(1, "argument is not a directory");
```

Dies kann so abgeändert werden, dass eine Fehlermeldung durch Auslesen der Variabel `errno` ausgegeben wird. Die Fehlermeldung wird entsprechend dem Beispiel ausgegeben:

```
#include <err.h>
#include <errno.h>
...
if (!S_ISDIR(st.st_mode)) {
 errno = ENOTDIR;
 err(1, NULL);
}
```

In diesem Beispiel wurde die benutzerdefinierte Zeichenkette entfernt. Übersetzer haben weniger Arbeit, wenn sie ein Programm lokalisieren und die Benutzer sehen die übliche "";Not a directory";" Fehlermeldung, wenn dieser Fehler auftritt. Diese Meldung wird ihnen wahrscheinlich vertraut erscheinen. Bitte beachten Sie, dass es notwendig ist, errno.h hinzuzufügen um einen direkten Zugriff auf `errno` zu haben.

Es lohnt sich darauf hinzuweisen, dass es Fälle gibt, in denen `errno` automatisch aufgerufen wird, so dass es nicht notwendig ist, es explizit zu tun:

```
#include <err.h>
...
if ((p = malloc(size)) == NULL)
```

```
err(1, NULL);
```

4.2.4. Benutzung von bsd.nls.mk

Das Benutzen von Katalogdateien setzt einige sich wiederholende Schritte, wie das kompilieren und installieren der Kataloge, voraus. Um diese Schritte zu vereinfachen, stellt bsd.nls.mk einige Makros zur Verfügung. Es ist nicht notwendig bsd.nls.mk explizit hinein zu kopieren, es wird automatisch aus den allgemeinen Makefiles wie bsd.prog.mk oder bsd.lib.mk gezogen.

Normalerweise reicht es, NLSNAME zu definieren, die den Namen des Kataloges als erstes Argument von catopen(3) enthalten sollte und die Katalogdateien in NLS ohne ihre Endung .msg auflistet. Hier ist ein Beispiel, das es ermöglicht, NLS mit dem obigen Code zu deaktivieren. Die WITHOUT_NLS Variable von make(1) muss so definiert werden, dass das Programm ohne NLS-Unterstützung gebaut wird.

```
.if !defined(WITHOUT_NLS)
NLS=     es_ES.ISO8859-1
NLS+=    hu_HU.ISO8859-2
NLS+=    pt_BR.ISO8859-1
.else
CFLAGS+= -DWITHOUT_NLS
.endif
```

Normalerweise werden die Katalogdateien in dem nls-Unterverzeichnis abgelegt. Dies ist der Standard von bsd.nls.mk. Es ist möglich, mit der NLSSRCDIR-Variablen von make(1) diese zu überschreiben. Der Standardname der vorkompilierten Katalogdateien folgt den Namenskonventionen, wie oben beschrieben. Er kann durch die NLSNAME-Variablen überschrieben werden. Es gibt noch weitere Optionen, um eine Feinabstimmung zur Verarbeitung der Katalogdateien zu erreichen. Da sie nicht notwendig sind, werden sie hier nicht weiter beschrieben. Weitere Informationen über bsd.nls.mk finden Sie in der Datei selbst. Der Text ist kurz und leicht zu verstehen.

Kapitel 5. Vorgaben und Richtlinien für das Quelltextverzeichnis

Dieses Kapitel dokumentiert verschiedene Vorgaben und Richtlinien für das FreeBSD-Quelltextverzeichnis.

5.1. Stil-Richtlinien

Ein konsistenter Code-Stil ist extrem wichtig, besonders in einem so grossen Projekt wie FreeBSD. Der Code sollte dem FreeBSD Code-Stil entsprechen, welcher in style(9) und style.Makefile(5) genauer beschrieben ist.

5.2. `MAINTAINER` eines Makefiles

Wenn ein bestimmter Bereich der FreeBSD src/-Distribution von einer Person oder Gruppe gepflegt wird, kann dies durch einen Eintrag in die Datei src/MAINTAINERS der Öffentlichkeit mitgeteilt werden. Maintainer eines Ports in der Ports-Sammlung können ihre Verantwortung über den Port durch einen Eintrag in die `MAINTAINER`-Zeile im Makefile des Ports der Welt mitteilen.

```
MAINTAINER= email-addresses
```

Für andere Teile des Repositories oder andere Abschnitte, die noch keinen Maintainer aufweisen, oder falls Sie sich nicht sicher sind, wer der Maintainer ist, sehen Sie sich die Commit-Historie des betreffenden Ports an. Es ist recht häufig der Fall, dass ein Maintainer nicht explizit aufgeführt ist, aber trotzdem diejenigen Personen, die den Port seit den letzten paar Jahren aktiv betreuen, daran interessiert sind, Änderungen zu begutachten. Selbst wenn dies nicht explizit in der Dokumentation oder im Quellcode erwähnt ist, wird es trotzdem als höfliche Geste angesehen, wenn man nach einer Überprüfung der eigenen Änderungen fragt.

Die Rolle eines Maintainers ist die folgende:

- Der Maintainer ist verantwortlich für diesen Code. Er oder sie muss einerseits für die Behebung von Fehlern und das Beantworten von Problemberichten für diesen Code die Verantwortung tragen und andererseits, falls es sich um beigesteuerte Software handelt, neue Versionen verfolgen und bereitstellen.
- Änderungen an Verzeichnissen, die ein Maintainer definiert hat, sollten an den Maintainer für eine Überprüfung gesendet werden, bevor diese committet werden. Nur wenn der Maintainer in einer inakzeptablen Zeitspanne auf mehrere E-Mails nicht antwortet, können die Änderungen, die mit dem Commit in Kraft treten, auch ohne Überprüfung durch den Maintainer vollzogen werden. Dennoch wird empfohlen, dass die Änderungen, falls möglich, von jemand anderem überprüft werden.
- Es ist natürlich nicht akzeptabel, einer Person oder Gruppe den Status eines Maintainers zu

geben, so lange sie nicht zustimmt, diese Pflicht auf sich zu nehmen. Andererseits muss es kein einzelner Mensch sein. Eine Gruppe von Menschen ist genauso in Ordnung.

5.3. Beigesteuerte Software

Einige Teile der FreeBSD-Distribution enthalten Software, die aktiv außerhalb des FreeBSD-Projektes gepflegt wird. Aus historischen Gründen nennen wir dies *contributed* Software. Beispiele dafür sind sendmail, gcc und patch.

Über die Jahre wurden verschiedene Methoden genutzt, um solche Software zu verwalten, und jede hat Vor- wie auch Nachteile. So hat sich kein eindeutiger Gewinner herauskristallisiert.

Es wurde viel über diesen Umstand diskutiert und eine Methode als die "offizielle" vorgestellt, um in Zukunft diese Art der Software zu importieren. Ferner wird dringend geraten, dass sich existierende, beigesteuerte Software diesem Modell annähert, da es signifikante Vorteile gegenüber der alten Methode gibt. Dazu gehört auch, dass jeder einfach Diffs bezüglich der "offiziellen" Quelltext-Versionen erzeugen kann (auch ohne direkten Repository-Zugang). Dies wird es deutlich vereinfachen, Änderungen an die Hauptentwickler zurückfließen zu lassen.

Letztendlich kommt es jedoch auf die Menschen an, welche die Arbeit leisten. Wenn die Durchführung dieses Modells bei einem Paket mal nicht möglich ist, können Ausnahmen dieser Regeln nur mit Genehmigung des Core-Teams und der Übereinstimmung der anderen Entwickler gewährt werden. Die Fähigkeit, dieses Paket auch in Zukunft pflegen zu können, ist eine der Schlüsselfragen bei dieser Entscheidung.

Durch einige bedauernswerte Einschränkungen des RCS-Dateiformats und die Handhabung von Herstellerzweigen ist von unwesentlichen, trivialen und/oder kosmetischen Änderungen an Dateien *dringend abzuraten*, die dem Herstellerzweig folgen. "Grammatikalische oder sprachliche Fehlerbehebungen" sind explizit unter der "Kosmetik"-Kategorie einzuordnen und sollten vermieden werden. Das Repository kann sich durch Änderungen einzelner Zeichen dramatisch aufblähen.

5.3.1. Herstellerimports mit CVS

Das file-Werkzeug soll als Beispiel dienen, wie dieses Modell funktioniert:

src/contrib/file enthält den Quelltext so, wie vom Maintainer dieses Pakets bereitgestellt. Teile, die unter FreeBSD gänzlich unnutzbar sind, können entfernt werden. Im Fall von file(1) wurde u.a. das Unterverzeichnis python und Dateien mit dem Präfix lt vor dem Import entfernt.

src/lib/libmagic enthält ein Makefile im bmake-Stil, das die Regeln des Standard-Makefiles bsd.lib.mk nutzt, um die Bibliothek zu erstellen und die Dokumentation zu installieren.

src/usr.bin/file enthält ein Makefile im bmake-Stil, welches das `file`-Programm erstellt und installiert, ebenso die dazugehörigen Manualpages, welche die Regeln von bsd.prog.mk nutzen.

Das Entscheidende ist hier das src/contrib/file-Verzeichnis, welches nach den folgenden Regeln erstellt wird: Es muss den Quelltext aus dem Original enthalten (ohne RCS-Schlüsselworte und im

korrekten Herstellerzweig) mit so wenig FreeBSD-spezifischen Änderungen wie möglich. Sollte es Zweifel geben, wie hier zu verfahren ist, unbedingt zuerst nachfragen und nicht auf gut Glück etwas probieren in der vagen Hoffnung, dass es "irgendwie funktioniert".

Aufgrund der eingangs schon erwähnten Einschränkungen bei Herstellerzweigen ist es erforderlich, dass "offizielle" Fehlerbehebungen vom Hersteller in die Originalquellen der Distribution einfließen und als Resultat wieder in den Herstellerzweig importiert werden. Offizielle Fehlerbehebungen sollten nie direkt in FreeBSD eingepflegt und "committet" werden, da dies den Herstellerzweig zerstören würde und der Import von zukünftigen Versionen wäre um ein Vielfaches schwerer, da es zu Konflikten kommen würde.

Da einige Pakete Dateien enthalten, die zur Kompatibilität mit anderen Architekturen und Umgebungen als FreeBSD gedacht sind, ist es zulässig, diese Teile zu löschen, wenn sie für FreeBSD nicht von Interesse sind, und so Speicherplatz zu sparen. Dateien, die ein Copyright und Release-artige Informationen zu den vorhandenen Dateien enthalten, sollten *nicht* gelöscht werden.

Falls es einfacher erscheint, können die `bmake`-Makefiles vom Verzeichnisbaum durch einige Dienstprogramme automatisch erstellt werden, was es hoffentlich sogar noch einfacher macht, eine Version zu aktualisieren. Ist dies geschehen, so stellen Sie bitte sicher, diese Werkzeuge in das Verzeichnis src/tools gleich mit dem Port an sich einzuchecken, sodass es für zukünftige Maintainer verfügbar ist.

Im Verzeichnis src/contrib/file sollte eine Datei mit dem Namen FREEBSD-upgrade hinzugefügt werden und sie sollte den Stand wie folgt anzeigen:

- Welche Dateien ausgelassen wurden.
- Von wo die Original-Distribution stammt und/oder wo die offizielle Hauptseite zu finden ist.
- Wohin Fehlerbehebungen an den Originalautor gesendet werden können.
- Möglicherweise eine Übersicht, welche FreeBSD-spezifischen Änderungen vorgenommen wurden.

Ein Beispielinhalt von src/contrib/groff/FREEBSD-upgrade ist hier aufgelistet:

```
$FreeBSD: src/contrib/groff/FREEBSD-upgrade,v 1.5.12.1 2005/11/15 22:06:18 ru Exp $

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.

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

        2. Use the command:
                cvs import -m 'Virgin import of FSF groff v<version>' \
                        src/contrib/groff FSF v<version>

           For example, to do the import of version 1.19.2, I typed:
```

```
                cvs import -m 'Virgin import of FSF groff v1.19.2' \
                        src/contrib/groff FSF v1_19_2

        3. Follow the instructions printed out in step 2 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 groff, simply patch and commit to the main
branch (aka HEAD).  Never make local changes on the FSF branch.

All local changes should be submitted to Werner Lemberg <wl@gnu.org> or
Ted Harding <ted.harding@nessie.mcc.ac.uk> for inclusion in the next
vendor release.

ru@FreeBSD.org - 20 October 2005
```

Eine weitere Möglichkeit ist es, eine Liste von Dateien, die nicht enthalten sein sollen zu pflegen, was besonders dann sehr hilfreich sein kann, wenn die Liste ziemlich gross oder kompliziert ist bzw. Imports sehr häufig stattfinden. Durch erstellen einer Datei namens FREEBSD-Xlist im gleichen Verzeichnis, in welches das Herstellerverzeichnis importiert werden soll, die eine Liste von auszuschliessenden Dateinamen-Mustern pro Zeile enthält, können zukünftige Imports folgendermassen durchgeführt werden:

```
% tar -X FREEBSD-Xlist -xzf vendor-source.tgz
```

Als Beispiel einer FREEBSD-Xlist-Datei wird hier diejenige von src/contrib/tcsh gezeigt:

```
*/BUGS
*/config/a*
*/config/bs2000
*/config/bsd
*/config/bsdreno
*/config/[c-z]*
*/tests
*/win32
```

Bitte importieren Sie weder FREEBSD-upgrade noch FREEBSD-Xlist mit den beigesteuerten Quellen. Stattdessen sollten Sie diese Dateien nach dem initialen Import hinzufügen.

5.3.2. Herstellerimports mit SVN

Dieser Abschnitt beschreibt die Prozedur für Herstellerimports mit Subversion im Detail.

1. Vorbereiten des Quellbaums

Wenn dies Ihr erster Import nach dem Wechsel zu SVN ist, sollen Sie den Herstellerbaum aufräumen, verflachen und die Merge-Historie in den Hauptzweig vorbereiten. Falls das nicht Ihr erster Import ist, können Sie diesen Schritt ohne Probleme überspringen.

Während der Konvertierung von CVS zu SVN wurden Herstellerzweige mit der gleichen Struktur wie der Hauptzweig importiert. Beispielsweise wurden die foo Herstellerquellen in vendor/foo/dist/contrib/foo abgelegt, jedoch ist dies unpraktisch und zwecklos. Was wir wirklich wollen, ist dass die Herstellerquellen direkt in vendor/foo/dist liegen, beispielsweise so:

```
% cd vendor/foo/dist/contrib/foo
% svn move $(svn list) ../..
% cd ../..
% svn remove contrib
% svn propdel -R svn:mergeinfo
% svn commit
```

Beachten Sie, dass das `propdel`-Bit notwendig ist, da mit Subversion 1.5 automatisch `svn:mergeinfo` zu jedem Verzeichnis hinzugefügt wird, das Sie kopieren oder verschieben. In diesem Fall brauchen Sie diese Informationen nicht, da Sie nichts in den Zweig mergen werden, den Sie gelöscht haben.

Sie werden wahrscheinlich die Tags genauso verflachen wollen. Die Prozedur dafür ist die selbe. Wenn Sie dies tun, sollten Sie den Commit bis zum Schluss aufschieben.

Prüfen Sie den dist-Baum und führen Sie alle nötigen Aufräumarbeiten durch, die Sie für sinnvoll erachten. Sie werden möglicherweise die Erweiterung von Schlüsselwörtern deaktivieren wollen, da dies auf unmodifizierten Quellen keinen Sinn ergibt. In machen Fällen kann dies sogar schädlich sein.

```
% svn propdel svn:keywords -R .
% svn commit
```

Bootstrappen der `svn:mergeinfo` auf dem Zielverzeichnis (des Hauptzweiges) auf die Revision die mit der letzten Änderung, die im Herstellerzweig vor dem Import der neuen Quellen durchgeführt wurde, korrespondiert, wird ebenso benötigt:

```
% cd head/contrib/foo
% svn merge --record-only svn_base/vendor/foo/dist@12345678 .
% svn commit
```

Dabei entspricht *svn_base* dem Basisverzeichnis Ihres SVN-Repositories, z.B. `svn+ssh://svn.FreeBSD.org/base`.

2. Neue Quellen importieren

Bereiten Sie einen kompletten, sauberen Baum mit Herstellerquellen vor. Mit SVN können wir eine komplette Distribution in dem Herstellerzweig aufbewahren, ohne den Hauptzweig aufzublähen. Importieren Sie alles, aber mergen Sie nur das, was wirklich benötigt wird.

Beachten Sie, dass Sie alle Dateien, die seit dem letzten Herstellerimport hinzugefügt wurden, auch einbeziehen und diejenigen, welche entfernt wurden, auch löschen müssen. Um dies zu bewerkstelligen, sollten Sie sortierte Listen der Bestandteile des Herstellerbaums und von den Quellen, Sie die vorhaben zu importieren, vorbereiten:

```
% cd vendor/foo/dist
% svn list -R | grep -v '/$' | sort > ../old
% cd ../foo-9.9
% find . -type f | cut -c 3- | sort > ../new
```

Mit diesen beiden Dateien, wird Ihnen das folgende Kommando alle Dateien auflisten, die entfernt wurden (nur die Dateien in old):

```
% comm -23 ../old ../new
```

Der folgende Befehl wird die hinzugefügten Dateien auflisten (nur diejenigen Dateien in new):

```
% comm -13 ../old ../new
```

Wir führen dies nun zusammen:

```
% cd vendor/foo/foo-9.9
% tar cf - . | tar xf - -C ../dist
% cd ../dist
% comm -23 ../old ../new | xargs svn remove
% comm -13 ../old ../new | xargs svn add
```

Wenn in der neuen Version neue Verzeichnisse hinzugekommen sind, wird dieser letzte Befehl fehlschlagen. Sie müssen diese Verzeichnisse hinzufügen und anschliessend den Befehl erneut ausführen. Genauso müssen Sie Verzeichnisse, die entfernt wurden, händisch löschen.

Prüfen Sie die Eigenschaften jeder neuen Datei:

- Alle Textdateien sollten `svn:eol-style` auf den Wert `native` gesetzt haben.
- Alle Binärdateien sollten `svn:mime-type` auf `application/octet-stream` gesetzt haben, ausser es existiert ein passenderer Medientyp.
- Ausführbare Dateien sollten `svn:executable` auf * gesetzt haben.

- Es sollten keine anderen Eigenschaften auf den Dateien im Baum gesetzt sein.

Sie sind bereit, zu committen, jedoch sollten Sie zuerst die Ausgabe von `svn stat` und `svn diff` überprüfen, um sicher zu gehen, dass alles in Ordnung ist.

Sobald Sie den die neue Release-Version des Herstellers committed haben, sollten Sie Ihn für zukünftige Referenzen taggen. Die beste und schnellste Methode ist, dies direkt im Repository zu tun:

```
% svn copy svn_base/vendor/foo/dist svn_base/vendor/foo/9.9
```

Um den neuen Tag zu bekommen, brauchen Sie nur ihre Arbeitskopie von vendor/foo zu aktualisieren.

Wenn Sie lieber die Kopie in der ausgecheckten Kopie durchführen wollen, vergessen Sie nicht, die generierte `svn:mergeinfo` wie oben beschrieben zu entfernen.

3. Mit *-HEAD* mergen

Nachdem Sie Ihren Import vorbereitet haben, wird es Zeit zu mergen. Die Option `--accept=postpone` weist SVN an, noch keine merge-Konflikte aufzulösen, weil wir uns um diese manuell kümmern werden:

```
% cd head/contrib/foo
% svn update
% svn merge --accept=postpone svn_base/vendor/foo/dist
```

Lösen Sie die Konflikte und stellen Sie sicher, dass alle Dateien, die im Herstellerzweig hinzugefügt oder entfernt wurden, auch sauber im Hauptzweig hinzugefügt bzw. gelöscht wurden. Es ist immer ratsam, diese Unterschiede gegen den Herstellerbaum zu prüfen:

```
% svn diff --no-diff-deleted --old=svn_base/vendor/foo/dist --new=.
```

Die Option `--no-diff-deleted` weist SVN an, keine Dateien zu prüfen, die sich zwar im Herstellerbaum, aber nicht im Hauptzweig befinden.

Bei SVN gibt es das Konzept von innerhalb und ausserhalb des Herstellerbaums nicht. Wenn eine Datei, die zuvor eine lokale Änderung hatte, aber nun keine mehr besitzt, entfernen Sie einfach das was übrig ist, wie FreeBSD Versionstags, damit diese nicht länger in den diffs gegen den Herstellerbaum erscheinen.

Wenn irgendwelche Änderungen notwendig sind, um die Welt mit den neuen Quellen zu

bauen, machen Sie diese jetzt und testen Sie diese bis Sie sicher sind, dass alles korrekt gebaut wird und richtig funktionert.

4. Commit

   Nun sind Sie bereit für den Commit. Stellen Sie sicher, dass Sie alles in einem einzigen Schritt durchführen. Idealerweise sollten Sie alle diese Schritte in einem sauberen Baum durchgeführt haben. Falls dies der Fall ist, können Sie einfach aus dem obersten Verzeichnis dieses Baums committen. Dies ist der beste Weg, um Überraschungen zu vermeiden. Wenn Sie dies korrekt durchführen, wird der Baum atomar von einem konsistenten Zustand mit dem alten Code in einen neuen konsistenten Zustand mit dem neuen Code überführt.

5.4. Belastende Dateien

Es kann gelegentlich notwendig sein, belastende Dateien in den FreeBSD-Quelltextbaum zu integrieren. Braucht ein Gerät zum Beispiel ein Stück binären Code, der zuerst geladen werden muss, bevor das Gerät funktioniert, und wir haben keine Quellen zu diesem Code, dann wird die binäre Datei als belastend bezeichnet. Die folgenden Richtlinien sind beim Aufnehmen von belastenden Dateien in den FreeBSD-Quelltextbaum zu beachten.

1. Jede Datei, die durch die System-CPU(s) ausgeführt wird und nicht als Quelltext vorliegt, ist belastend.
2. Jede Datei, deren Lizenz restriktiver ist als die BSD- oder GNU-Lizenz, ist belastend.
3. Eine Datei, die herunterladbare Binär-Daten enthält, ist nur belastend, wenn (1) oder (2) zutreffen. Sie muss in einem ASCII-Format gespeichert werden, das Architektur-neutral ist (file2c oder uuencoding wird empfohlen).
4. Jede belastende Datei braucht eine spezielle Genehmigung vom Core-Team, bevor diese in das Repository aufgenommen werden darf.
5. Belastende Dateien liegen unter src/contrib oder src/sys/contrib.
6. Das komplette Modul sollte auch am Stück aufbewahrt werden. Es gibt keinen Grund, dieses zu teilen, außer es gibt einen Code-Austausch mit Quelltext, der nicht belastend ist.
7. Objekt-Dateien werden wie folgt benannt: arch/filename.o.uu>.
8. Kernel-Dateien:
   a. Sollten immer nach conf/files.* verweisen (um den Bau einfach zu halten).
   b. Sollten sich immer in LINT befinden, jedoch entscheidet das Core-Team je nach Fall, ob es auskommentiert wird oder nicht. Das Core-Team kann sich zu einem späteren Zeitpunkt immer noch anders entscheiden.
   c. Der *Release-Engineer* entscheidet, ob es in ein Release aufgenommen wird oder nicht.
9. Userland-Dateien:
   a. Das Core-Team entscheidet, ob der Code von `make world` gebaut wird oder nicht.
   b. Der Release-Engineer entscheidet, ob es in das Release aufgenommen wird oder nicht.

5.5. Shared-Libraries

Sollten Sie die Unterstützung für Shared-Libraries bei einem Port oder einem Stück Software, das dies nicht hat, hinzufügen, sollten die Versionsnummern dessen Regeln folgen. Im Allgemeinen hat die sich daraus resultierende Nummer nichts mit der Release-Version der Software zu tun.

Die drei Grundsätze zum Erstellen von Shared-Libraries sind:

- Sie beginnen mit `1.0`.
- Gibt es eine Änderung, die abwärtskompatibel ist, so springen Sie zur nächsten Minor-Version (beachten Sie, dass ELF-Systeme die Minor-Version ignorieren).
- Gibt es eine inkompatible Änderung, so springen Sie bitte zur nächsten Major-Version.

Zum Beispiel wird beim Hinzufügen von Funktionen und oder Fehlerbehebungen zur nächsten Minor-Version gesprungen, beim Löschen einer Funktion, Ändern von Funktionsaufrufen usw. ändert sich die Major-Version.

Bleiben Sie bei Versionsnummern in der Form major.minor (*x.y*). Unser dynamischer Linker a.out kann mit Versionsnummern in der Form *x.y.z* nicht gut umgehen. Jede Versionsnummer nach dem *y* (die dritte Zahl) wird völlig ignoriert, wenn Versionsnummern der Shared-Libraries verglichen werden, um zu bestimmen, mit welcher Bibliothek eine Anwendung verlinkt wird. Sind zwei Shared-Libraries vorhanden, die sich nur in der "micro"-Revision unterscheiden, so wird `ld.so` zu der höheren verlinken. Dies bedeutet, dass wenn Sie mit libfoo.so.3.3.3 verlinken, der Linker nur `3.3` in den Header aufnimmt und alles linkt, was mit *libfoo.so.3 .(irgendetwas >= 3).(höchste verfügbare Nummer)* beginnt.

`ld.so` wird immer die höchste "Minor"-Revision benutzen. Beispielsweise wird es die libc.so.2.2 bevorzugen gegenüber der libc.so.2.0, auch dann, wenn das Programm ursprünglich mit libc.so.2.0 verlinkt war.

Unser dynamischer ELF-Linker kann keine Minor-Versionen handhaben. Dennoch sollten die Major- und Minor-Versionen genutzt werden, da unsere Makefiles "das Richtige machen" bezogen auf den Systemtyp.

Für nicht-Port-Bibliotheken lautet die Richtlinie, die Shared-Library-Versionsnummer nur einmal zwischen den Releases zu ändern. Weiterhin ist es vorgeschrieben, die Major-Version der Shared-Libraries nur bei Major-OS-Releases zu ändern (beispielsweise von 6.0 auf 7.0). Wenn Sie also eine Änderung an einer Systembibliothek vornehmen, die eine neue Versionsnummer benötigt, überprüfen Sie die Commit-Logs des Makefiles. Es liegt in der Verantwortung des Committers, dass sich eine erste solche Änderung seit dem letzten Release in der aktualisierten Versionsnummer der Shared-Library im Makefile äußert, folgende Änderungen werden nicht berücksichtigt.

Kapitel 6. Regressions- und Performance-Tests

Regressions-Tests werden durchgeführt, um zu überprüfen, ob ein bestimmter Teil des Systems wie erwartet funktioniert, und um sicherzustellen, dass bereits beseitigte Fehler nicht wieder eingebaut werden.

Die FreeBSD-Regressions-Testwerkzeuge finden Sie im FreeBSD-Quelltextbaum unter src/tools/regression.

6.1. Mikro-Benchmark-Checkliste

Dieser Abschnitt enthält Tipps, wie ordnungsgemäße Mikro-Benchmarks unter FreeBSD oder für FreeBSD selbst erstellt werden.

Es ist nicht möglich, immer alle der folgenden Vorschläge zu berücksichtigen, aber je mehr davon, desto besser wird der Benchmark kleine Unterschiede nachweisen können.

- Schalten Sie APM und alles andere, das den Systemtakt beeinflusst, ab (ACPI?).
- Starten Sie in den Single-User-Modus. cron(8) und andere Systemdienste verursachen nur Störungen. Genauso der sshd(8)-Systemdienst. Falls während des Tests SSH-Zugriff benötigt wird, schalten Sie entweder die Neuerstellung des SSHv1-Schlüssels ab oder beenden Sie den `sshd`-Elternprozess während der Tests.
- Beenden Sie ntpd(8).
- Falls syslog(3)-Ereignisse erzeugt werden, starten Sie syslogd(8) mit leerer /etc/syslogd.conf oder beenden Sie es.
- Sorgen Sie für möglichst wenig Disk-I/O; vermeiden Sie es ganz wenn möglich.
- Hängen Sie keine Dateisysteme ein, die Sie nicht benötigen.
- Hängen Sie /, /usr und die anderen Dateisysteme nur lesbar ein wenn möglich. Dies verhindert, dass atime-Aktualisierungen auf der Festplatte (usw.) das Ergebnis verfälschen.
- Initialisieren Sie das beschreibbare Test-Dateisystem mit newfs(8) neu und füllen Sie es aus einer tar(1)- oder dump(8)-Datei vor jedem Lauf. Hängen Sie es aus und wieder ein, bevor Sie den Test starten. Dies sorgt für einen konsistenten Dateisystemaufbau. Bei einem "worldstone"-Test bezieht sich dies auf /usr/obj (Initialisieren Sie es einfach mit `newfs` neu und hängen Sie es ein). Um absolut reproduzierbare Ergebnisse zu bekommen, füllen Sie das Dateisystem aus einer dd(1)-Datei (d.h. `dd if=myimage of=/dev/ad0s1h bs=1m`).
- Benutzen Sie malloc-gestützte oder vorbelastete md(4)-Partitionen.
- Starten Sie zwischen den einzelnen Durchläufen neu, dies sichert einen konsistenteren Zustand.
- Entfernen Sie alle nicht unbedingt benötigten Gerätetreiber aus dem Kernel. Wenn z.B. USB für den Test nicht benötigt wird, entfernen Sie es aus dem Kernel. Gerätetreiber, die sich Hardware zuteilen, haben oft "tickende" Timeouts.
- Konfigurieren Sie nicht Hardware, die nicht benutzt wird. Entfernen Sie Festplatten mit

atacontrol(8) und camcontrol(8), wenn diese für den Test nicht gebraucht werden.

- Konfigurieren Sie nicht das Netzwerk, es sei denn es wird getestet, oder warten Sie, bis der Test fertig ist, wenn Sie das Ergebnis auf einen anderen Rechner übertragen wollen.

  Falls das System an ein öffentliches Netzwerk angeschlossen sein muss, achten Sie auf Spitzen im Broadcast-Verkehr. Obwohl dieser kaum auffällt, wird er CPU-Zyklen brauchen. Ähnliches gilt für Multicast.

- Legen Sie jedes Dateisystem auf eine eigene Festplatte. Dies minimiert Jitter durch Optimierungen von Lesekopfbewegungen.
- Minimieren Sie Ausgaben auf serielle oder VGA-Konsolen. Ausgabenumleitung in Dateien ergibt weniger Jitter (serielle Konsolen werden leicht zum Flaschenhals). Benutzen Sie die Tastatur nicht, während der Test läuft, sogar `space` oder `back-space` wirken sich auf die Ergebnisse aus.
- Stellen Sie sicher, dass der Test lang genug läuft, aber nicht zu lange. Wenn er zu kurz ist, sind Zeitstempel ein Problem. Wenn er zu lang ist, werden Temperaturänderungen und Drift die Frequenz von Quarzkristallen im Rechner beeinflussen. Daumenregel: mehr als eine Minute, weniger als eine Stunde.
- Versuchen Sie, die Temperatur in der Umgebung des Rechners so stabil wie möglich zu halten. Diese beeinflusst sowohl Quarzkristalle als auch Festplatten-Algorithmen. Um einen wirklich stabilen Takt zu erhalten, wäre es auch möglich, einen stabilisierten Takt anzuschließen. D.h. besorgen Sie sich einen OCXO + PLL und koppeln Sie das Ausgangssignal mit den Taktgeberschaltkreisen anstelle des Quarzkristalls der Hauptplatine. Wenden Sie sich an Poul-Henning Kamp <phk@FreeBSD.org>, wenn Sie mehr Informationen hierüber benötigen.
- Lassen Sie den Test mindestens drei Mal laufen, besser mehr als 20 Mal, sowohl für "vor" als auch für "nach" dem Code. Versuchen Sie abzuwechseln (d.h. nicht erst 20 Mal "vorher" und dann 20 Mal "nachher"), dies ermöglicht, umgebungsbedingte Effekte zu erkennen. Wechseln Sie nicht 1:1 ab, sondern 3:3; dies erlaubt, Wechselwirkungseffekte zu erkennen.

  Ein gutes Muster ist: `bababa{bbbaaa}*`. Dies gibt Hinweise nach den ersten 1+1-Läufen (sodass Sie den Test stoppen können, falls er völlig daneben geht), Sie können die Standardabweichung nach den ersten 3+3-Läufen überprüfen (zeigt an, ob sich ein längerer Lauf lohnt), später Trends und Wechselwirkungen.

- Benutzen Sie ministat(1), um festzustellen, ob Ihre Ergebnisse signifikant sind. Überlegen Sie sich, das Buch "Cartoon guide to statistics" ISBN: 0062731025 zu kaufen. Es ist sehr empfehlenswert, falls Sie Dinge wie Standardabweichung und Studentsche t-Verteilung vergessen oder nie gelernt haben.
- Benutzen Sie keinen Hintergrund-fsck(8), wenn Sie ihn nicht selbst testen wollen. Schalten Sie auch `background_fsck` in /etc/rc.conf aus, es sei denn der Benchmark wird nicht mindestens 60+"Laufzeit von `fsck`" Sekunden nach Systemstart gestartet, da rc(8) startet und prüft, ob `fsck` auf irgendeinem der Dateisysteme laufen muss, wenn Hintergrund-`fsck` eingeschaltet ist. Stellen Sie ebenfalls sicher, dass keine Snapshots herumliegen, falls der Benchmark nicht ein Test mit Snapshots ist.
- Falls der Benchmark unerwartet schlechte Performance zeigt, überprüfen Sie Dinge wie große Mengen Interrupts von unerwarteten Quellen. Es gibt Berichte, dass einige ACPI-Versionen sich "daneben benehmen" und ein Übermaß an Interrupts erzeugen. Um zu helfen, ungewöhnliche

Testergebnisse zu diagnostizieren, machen Sie ein paar Momentaufnahmen von `vmstat -i` und suchen Sie nach Ungewöhnlichem.

- Gehen Sie mit Parametern zur Optimierung von Kernel, Userland und Fehlersuche vorsichtig um. Es passiert schnell, irgendetwas durchrutschen zu lassen und dann später festzustellen, dass der Test nicht das gleiche verglichen hat.
- Erstellen Sie nie Benchmarks unter Verwendung der Kernel-Optionen `WITNESS` und `INVARIANTS`, wenn der Test nicht diese Merkmale selbst untersuchen soll. `WITNESS` kann zu 400% und mehr Performance-Abnahme führen. Ähnliches gilt für Userland-malloc(3)-Parameter, Voreinstellungen hierbei unterscheiden sich bei -CURRENT von denen bei Production-Releases.

Teil II: Interprozess-Kommunikation

Kapitel 7. Sockets

Dieses Kapitel ist noch nicht übersetzt. Lesen Sie bitte das Original in englischer Sprache. Wenn Sie helfen wollen, dieses Kapitel zu übersetzen, senden Sie bitte eine E-Mail an die Mailingliste FreeBSD German Documentation Project <de-bsd-translators@de.FreeBSD.org>.

Kapitel 8. IPv6 Internals

8.1. IPv6/IPsec-Implementierung

Dieser Abschnitt erklärt die von der IPv6- und IPsec-Implementierung abhängigen Internas. Die Funktionalitäten wurden vom KAME-Projekt abgeleitet

8.1.1. IPv6

8.1.1.1. Konformität

Die IPv6 abhängigen Funktionen richten sich nach, oder versuchen sich nach den neuesten IPv6-Spezifikationen zu richten. (*Achtung*: Dies ist keine vollständige Liste - es wäre zu aufwändig, diese zu pflegen...).

Für weitere Details beachten sie bitte die entsprechenden Kapitel, RFCs, manual pages, oder Kommentare in den Quelltexten.

Konformitätsprüfungen wurden basierend auf KAME-STABLE-Kit des TAHI-Projekts durchgeführt. Die Ergebnisse können unter http://www.tahi.org/report/KAME/ eingesehen werden. In der Vergangenheit begleiteten wir auch Tests mit unseren älteren "Snapshots" an der Univ. of New Hampshire IOL (http://www.iol.unh.edu/).

- RFC1639: FTP Operation Over Big Address Records (FOOBAR)
  - RFC2428 wird gegenüber RFC1639 bevorzugt. FTP-Clients versuchen zuerst RFC2428, dann im Fehlerfall RFC1639.
- RFC1886: DNS Extensions to support IPv6
- RFC1933: Transition Mechanisms for IPv6 Hosts and Routers
  - IPv4 kompatible Adressen werden nicht unterstützt.
  - Automatisches Tunneln (beschrieben in 4.3 dieses RFC) wird nicht unterstützt.
  - Die gif(4)-Schnittstelle implementiert einen IPv[46]-over-IPv[46] Tunnel in einer allgemeinen Art und Weise und es umfaßt "configured tunnel" wie in der Spezifikation beschrieben. Siehe auch 23.5.1.5 in diese Dokument für weitere Details.
- RFC1981: Path MTU Discovery for IPv6
- RFC2080: RIPng for IPv6
  - usr.sbin/route6d unterstützt dies.
- RFC2292: Advanced Sockets API for IPv6
  - Unterstützte Bibliotheksfunktionen bzw. Kernel-APIs, siehe auch sys/netinet6/ADVAPI.
- RFC2362: Protocol Independent Multicast-Sparse Mode (PIM-SM)
  - RFC2362 definiert Paketformate für PIM-SM. draft-ietf-pim-ipv6-01.txt wurde basierend auf diesem RFC verfaßt.
- RFC2373: IPv6 Addressing Architecture

    - Unterstützt vom Knoten erforderliche Adressen und richtet sich nach den Erfordernissen des Bereichs.
- RFC2374: An IPv6 Aggregatable Global Unicast Address Format
    - Unterstützt die 64-Bit-Breite einer Interface ID.
- RFC2375: IPv6 Multicast Address Assignments
    - Userland-Applikationen nutzen die bekannten Adressen, die in den RFC festgelegt sind.
- RFC2428: FTP Extensions for IPv6 and NATs
    - RFC2428 wird gegenüber RFC1639 bevorzugt. FTP-Clients versuchen zuerst RFC2428, dann im Fehlerfall RFC1639.
- RFC2460: IPv6 specification
- RFC2461: Neighbor discovery for IPv6
    - Siehe auch 23.5.1.2 in diesem Dokument für weitere Details.
- RFC2462: IPv6 Stateless Address Autoconfiguration
    - Siehe auch 23.5.1.4 in diesem Dokument für weitere Details.
- RFC2463: ICMPv6 for IPv6 specification
    - Siehe auch 23.5.1.9 in diesem Dokument für weitere Details.
- RFC2464: Transmission of IPv6 Packets over Ethernet Networks
- RFC2465: MIB for IPv6: Textual Conventions and General Group
    - Notwendige Statistiken werden vom Kernel gesammelt. Die aktuelle IPv6-MIB-Unterstützung wird als Patch-Sammlung für ucd-snmp bereitgestellt.
- RFC2466: MIB for IPv6: ICMPv6 group
    - Notwendige Statistiken werden vom Kernel gesammelt. Die aktuelle IPv6-MIB-Unterstützung wird als Patch-Sammlung für ucd-snmp bereitgestellt.
- 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 Jumbogramms
    - Siehe auch 23.5.1.7 in diesem Dokument für weitere 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) implementiert dense mode. pim6sd(8) implementiert sparse mode.
- draft-itojun-ipv6-tcp-to-anycast-00: Unterbrechen einer TCP-Verbindung toward IPv6 anycast address
- draft-yamamoto-wideipv6-comm-model-00
    - Beachte 23.5.1.6 in deisem Dokument für weitere Deatils.
- draft-ietf-ipngwg-scopedaddr-format-00.txt: Eine Erweiterung des Format for IPv6 Scoped Addresses

8.1.1.2. Neighbor Discovery

Neighbor Discovery ist weitestgehend stabil. Zur Zeit werden Addressauflösung, Duplicated Address Detection (DAD), und Neighbor Unreachability Detection (NUD) unterstützt. In der näheren Zukunft werden wir Proxy Neighbor Advertisement Unterstützung in den Kernel einbauen und Unsolicited Neighbor Advertisement Übertragungskommandos als Verwaltungsprogramm zur Verfügung stellen.

Falls DAD versagt, wird die Adresse als "duplicated" markiert und eine Nachricht wird erzeugt, die an Syslog gesandt wird (und für gewöhnlich an die Konsole). Die "duplicated"-Markierung kann mit ifconfig(8) überprüft werden. Es liegt in der Verantwortung des Administrators, auf DAD-Fehler zu achten und diese zu beheben. Dieses Verhalten sollte in der näheren Zukunft verbessert werden.

Manche Netzwerktreiber verbinden Multicast-Pakete mit sich selbst, sogar, wenn es vorgeschrieben ist, es nicht zu tun (vor allem im Promiscuous-Modus). In solchen Fällen könnte DAD versagen, weil die DAD-Steuerung ein inbound NS packet sieht (eigentlich vom Knoten selber) und betrachtet es als ein Duplikat. Sie könnten sich die #if-Bedingung ansehen, die in sys/netinet6/nd6_nbr.c:nd6_dad_timer() als "Workaround" mit "heuristics" markiert ist (Beachte, dass das Kodefragment im Abschnitt "heuristics" nicht der Spezifikation entspricht).

Neighbor Discovery specification (RFC2461) kommuniziert in den folgenden Fällen nicht über neighbor cache handling:

1. Der Knoten empfing ein unverlangtes RS/NS/NA/redirect-Paket ohne Link-Layer-Adresse, wenn kein neighbor cache-Eintrag vorhanden ist.
2. neighbor cache handling bei Geräten ohne Link-Layer-Adresse (wir benötigen einen neighbor cache Eintrag für das IsRouter-Bit)

Im ersten Fall implemenierten wir einen Workaround basierend auf Diskussionen in der IETF-Ipngwg-Mailing-Liste. Für weitere Details beachten Sie die Kommentare im Quelltext und im Email-Thread, der bei (IPng 7155) mit dem Datum vom 6. Feb 1999 gestartet wurde.

IPv6 on-link Erkennungsregel (RFC2461) ist recht unterschiedlich zu Übernahmen im BSD-Netzwerkkode. Zur Zeit wird keine on-link Erkennungsregel unterstützt, bei der die Defaultrouter-Liste leer ist (RFC2461, Abschnitt 5.2, letzter Satz im zweiten Absatz - beachte, dass die Spezifikation das Wort "host" und "Knoten" an mehreren Stellen im Abschnitt mißbraucht).

Um mögliche DoS-Attacken und unendliche Schleifen zu verhindern, werden bis jetzt nur 10 Optionen bei ND-Paketen akzeptiert. Deshalb werden nur die ersten 10 Präfixe berücksichtigt, wenn man 20-Präfixoptionen zu RA hinzugefügt hat. Falls das zu Schwierigkeiten führen sollte,

dann sollte in der FREEBSD-CURRENT-Mailing-Liste gefragt werden und/oder die Variable nd6_maxndopt in sys/netinet6/nd6.c modifizieren. Falls die Nachfrage groß genug ist, könnte man einen sysctl-Knopf für die Variable vorsehen.

8.1.1.3. Bereichsindex

IPv6 benutzt Adressbereiche (Scoped Addresses). Deshalb ist es sehr wichtig, mit einer IPv6-Adresse einen Bereichsindex anzugeben (Schnittstellenindex für link-local-Adresse, oder einen Lageindex für site-local-Adressen). Ohne einen Bereichsindex ist ein IPv6-Adressbereich für den Kernel zweideutig und dem Kernel ist es nicht möglich, die Ausgabeschnittstelle für ein Paket festzustellen.

Gewöhnliche Userland-Anwendungen sollten die erweiterte Programmierschnittstelle (RFC2292) benutzen, um den Bereichsindex oder Schnittstellenindex festzulegen. Für ähnliche Zwecke wurde in RFC2553 sin6_scope_id member in der sockaddr_in6-Struktur definiert. Wie auch immer, die Semantik für sin6_scope_id ist ziemlich wage. Wenn man auf Portierbarkeit der Anwendung achten muß, dann schlagen wir vor, die erweiterte Programmierschnittstelle anstelle von sin6_scope_id zu benutzen.

Im Kernel ist ein Schnittstellenindex für link-local scoped-Adressen in das zweite 16bit-Wort (drittes und viertes Byte) der IPv6-Adresse eingebettet. Zum Beispiel sieht man folgendes

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

in der Routing-Tabelle und in der Schnittstellenadress-Struktur (structin6_ifaddr). Oben genannte Adresse ist eine "link-local unicast address" die zu einer Netzwerkschnittstelle gehört, deren Schnittstellenbezeichner 1 (eins) ist. Der eingebettete Index ermöglicht es, IPv6 link local-Adressen über mehrere Schnittstellen hinweg effektiv und mit wenig Änderungen am Kode zu identifizieren.

Routing-Dämonen und Konfigurationsprogramme wie route6d(8) und ifconfig(8) werden den "eingebetteten" Bereichsindex verändern müssen. Diese Programme benutzen routing sockets und ioctls (wie SIOCGIFADDR_IN6) und die Kernel-Programmierschnittstelle wird IPv6-Adressen, dessen zweites 16-Bit-Word gesetzt ist, zurückgeben. Diese Programmierschnittstellen dienen zur Änderung der Kernel-internen Struktur. Programme, die diese Programmierschnittstellen benutzen, müssen ohnehin auf Unterschiede in den Kerneln vorbereitet sein.

Wenn man einen Adressbereich in der Kommandozeile angibt, schreibt man niemals die eingebettete Form (so etwas wie ff02:1::1 or fe80:2::fedc). Man erwartet nicht, dass es funktioniert. Man benutzt immer die Standardform wie ff02::1 oder fe80::fedc, zusammen mit der Kommandozeilenoption, die die Schnittstelle festlegt (wie `ping6 -I ne0 ff02::1`). Allgemein gilt, wenn ein Kommando keine Kommandozeilenoption hat, um die Ausgabeschnittstelle zu definieren, ist dieses Kommando noch nicht für Adressbereiche bereit. Dies scheint der Prämisse von IPv6 entgegenzustehen. Wir glauben, dass die Spezifikationen einige Verbesserungen benötigen.

Einige der Userland-Werkzeuge unterstützen die erweiterte numerische IPv6-Syntax wie sie in draft-ietf-ipngwg-scopedaddr-format-00.txt beschrieben ist. Man kann die ausgehende Verbindung angeben, indem man den Namen der ausgehenden Schnittstelle wie folgt benutzt: "fe80::1%ne0". Auf diese Art und Weise ist man in der Lage, eine link-local scoped Adresse ohne viele Schwierigkeiten anzugeben.

Um die Erweiterungen im eigenen Programm zu nutzen, muss man getaddrinfo(3) und getnameinfo(3) mit NI_WITHSCOPEID verwenden. Die Implementierung setzt im Moment eine 1-zu-1 Beziehung zwischen einer Verbindung und einer Schnittstelle voraus, die stärker ist, als es die Spezifikationen beschreiben.

8.1.1.4. Plug and Play

Der grösste Teil der statuslosen IPv6-Adress-Autokonfiguration ist im Kernel implementiert. Neighbor-Discovery-Funktionen sind als ganzes im Kernel implementiert. Router-Advertisement (RA) Eingabe für Hosts ist im Kernel implementiert. Router-Solicitation (RS) Ausgabe für Hosts, RS-Eingabe für Router und RA-Ausgabe für Router ist im Userland implementiert.

8.1.1.4.1. Zuweisung von link-local und speziellen Adressen

Die IPv6 link-local-Adresse wird aus einer IEEE802-Adresse (Ethernet MAC address) erzeugt. Jeder Schnittstelle wird automatisch eine IPv6 link-local-Adresse zugewiesen, sobald die Schnittstelle aktiv ist (IFF_UP). Ebenso wird eine direkte Route für die link-local-Adresse zur Routing-Tabelle hinzugefügt.

Hier ist eine Ausgabe des netstat-Kommandos:

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

Schnittstellen, die keine IEEE802-Adresse haben (Pseudo-Schnittstellen wie Tunnel-Schnittstellen oder ppp-Schnittstellen), borgen sich eine IEEE802-Adresse von anderen Schnittstellen wie Ethernet-Schnittstellen aus, wann immer das möglich ist. Wenn keine IEEE802-Geräte eingebaut sind, wird als letzte Möglichkeit eine Pseudo-Zufallszahl - MD5(hostname) - als Quelle für eine link-local-Adresse benutzt. Falls diese für den Einsatz nicht geeignet sein sollte, dann muss man eine link-local-Adresse manuell konfigurieren.

Falls eine Schnittstelle nicht imstande ist, IPv6-Adressen zu handhaben (wie fehlende Unterstützung des multicast), wird keine link-local-Adresse der Schnittstelle zugewiesen. Siehe Abschnitt 2 für weitere Details.

Jede Schnittstelle verbindet die solicited multicast Adresse und link-local all-nodes multicast-Adressen (z.B. fe80::1:ff01:6317 und ff02::1, jeweils zu der Verbindung, an die die Schnittstelle verbunden ist). zusätzlich zu einer link-local-Adresse wird eine loopback-Adresse (::1) einer loopback-Schnittstelle zugewiesen. Außerdem werden ::1/128 und ff01::/32 automatisch zur Routing-Tabelle hinzugefügt und die loopback-Schnittstelle verbindet sich mit der node-local multicast Gruppe ff01::1.

8.1.1.4.2. Stateless address autoconfiguration beim Host

In der IPv6-Spezifikation werden Knoten in zwei Kategorien unterteilt: *Router* und *Hosts*. Router leiten Pakete, die an andere adressiert sind, weiter, Hosts leiten Pakete nicht weiter. net.inet6.ip6.forwarding definiert, ob dieser Knoten ein Router oder ein Host ist (Router falls es 1

ist, Host, falls es 0 ist).

Sobald ein Host ein Router-Advertisement vom Router hört, kann er sich selbst mit statusloser automatischer Adressen konfigurieren. Dieses Verhalten kann mit net.inet6.ip6.accept_rtadv (der Host konfiguriert sich selber, wenn es auf 1 gesetzt ist) beeinflusst werden. Bei einer automatischen Konfiguration wird das Netzwerkadresspräfix für die empfangende Schnittstelle (für gewöhnlich das globale Adresspräfix) hinzugefügt. Die Standard-Route wird ebenso konfiguriert. Router erzeugen periodisch Router-Advertisement-Pakete. Um einen benachbarten Router aufzufordern, ein RA-Paket zu erzeugen, kann eine Host-Router-Solicitation übertragen werden. Um jederzeit ein RS-Paket zu erzeugen, benutzt man das *rtsol*-Kommando. Ein rtsold(8)-Dämon ist ebenso verfügbar. rtsold(8) erzeugt Router-Solicitation, wann immer es notwendig ist und es funktioniert großartig "bei normadischem Einsatz" (Notebooks/Laptops). Falls jemand Router-Advertisements zu ignorieren wünscht, setzt man mit sysctl et.inet6.ip6.accept_rtadv auf 0.

Um Router-Advertisement von einem Router aus zu erzeugen, benutzt man den rtadvd(8)-Dämon.

Beachte, dass die IPv6-Spezifikation von folgenden Punkte ausgeht und nicht konforme Fälle werden als nicht spezifiziert ausgelassen:

- Nur Hosts hören auf Router-Angebote
- Hosts haben eine einzige Netzwerk-Schnittstelle (außer loopback)

Deshalb ist es unklug, net.inet6.ip6.accept_rtadv bei Routern oder bei Hosts mit mehreren Schnittstellen einzuschalten. Ein falsch konfigurierter Knoten kann sich seltsam verhalten (nicht konforme Konfiguration ist für diejenigen erlaubt, die Experimente durchführen möchten).

Eine Zusammenfassung des sysctl-Angaben:

```
    accept_rtadv forwarding Rolle des Knotens
   ---     ---    ---
    0       0    Host (wird manuell konfiguriert)
    0       1    Router
    1       0    automatisch konfigurierter Host
                 (Die Spezifikation setzt voraus, dass der Host nur eine einzelne
Schnittstelle hat, ein automatisch konfigurierter Host mit mehreren Schnittstellen ist
außerhalb der Betrachtung)
    1       1    ungültig, oder für Experimentierzwecke (außerhalb der Spezifikation)
```

RFC2462 hat eine Überprüfungsregel gegen eingehende RA-prefix-information-option, in 5.5.3 (e). Dies dient zum Schutz des Hosts vor schlecht oder falsch konfigurierten Routern, die eine sehr kurze Präfixlebenszeit ankündigen. Es gab Aktualisierungen von Jim Bound in der ipngwg-Mailing-Liste (suche nach "(ipng 6712)" im Archive) und es wurde Jims Aktualisierung implementiert.

Siehe auch 23.5.1.2 im Dokument für das Verhältnis zwischen DAD und autoconfiguration.

8.1.1.5. Generische Tunnel-Schnittstelle

GIF (Generische Schnittstelle) ist eine Pseudoschnittstelle für konfigurierte Tunnel. Details sind in gif(4) beschrieben. Im Moment sind

- v6 in v6
- v6 in v4
- v4 in v6
- v4 in v4

verfügbar. Benutze gifconfig(8), um die physikalische (außerhalb liegende) Quelle und die Zieladresse den gif-Schnittstellen zuzuweisen. Eine Konfiguration, die die selbe Adressfamilie für innere und äußere IP-Header (v4 in v4, oder v6 in v6) benutzt, ist gefährlich. Es ist sehr leicht, Schnittstellen und Routing-Tabellen so zu konfigurieren, dass eine unendliche Ebene von Tunneln ausgeführt wird. *Seien Sie also gewarnt*.

gif kann ECN-freundlich konfiguriert werden. Beachte 23.5.4.5 für eine ECN-Freundlichkeit von Tunneln und gif(4) wie man sie konfiguriert.

Falls man einen IPv4-in-IPv6-Tunnel mit einer gif-Schnittstelle konfigurieren möchte, sollte man gif(4) sorgfältig lesen. Man muss die IPv6 link-local Adresse, die automatisch der gif-Schnittstelle zugewiesen wird, entfernen.

8.1.1.6. Source Address Selection

Im Moment ist die Regel zur Auswahl der Quelle bereichsorientiert (es gibt einige Ausnahmen - siehe unten). Für ein gegebenes Ziel wird eine Quell-IPv6-Adresse durch folgende Regel ausgewählt:

1. Falls die Quelladresse explizit durch den Benutzer angegeben ist (z.B. über das erweiterte API), dann wird die angegebene Adresse benutzt.
2. Falls eine Adresse der ausgehenden Schnittstelle zugewiesen wird, die den selben Bereich wie die Zieladresse hat (was normalerweise durch einen Blick in die Routing-Tabelle festgestellt werden kann), dann wird diese Adresse benutzt.

   Dies ist ein typischer Fall.
3. Falls keine Adresse der obigen Bedingung genügt, dann wählt man eine globale Adresse, die einer der Schnittstellen des sendenden Knotens zugewiesen ist.
4. Falls keine Adresse der obigen Bedingung genügt und die Zieladresse ist im site local-Bereich, dann wählt man eine eine site local-Adresse, die einer der Schnittstellen des sendenden Knotens zugewiesen ist.
5. Falls keine Adresse der obigen Bedingung genügt, dann wählt man eine Adresse, die mit einem Eintrag in der Routing-Tabelle für das Ziel verbunden ist. Dies ist die letzte Möglichkeit, die eine Bereichsverletzung verursachen könnte.

Zum Beispiel, ::1 ist ausgewählt für ff01::1, fe80:1::200:f8ff:fe01:6317 für fe80:1::2a0:24ff:feab:839b (beachte den eingebetteten Schnittstelleindex - beschrieben in 23.5.1.3 - er hilft uns, die richtige Quelladresse auszuwählen. Diese eingebetteten Indexe werden nicht übertragen). Falls die ausgehende Schnittstelle mehrere Adressen für einen Bereich hat, wird die Quelle gewählt, die die breiteste passende Basis hat (Regel 3). Angenommen 2001:0DB8:808:1:200:f8ff:fe01:6317 und 2001:0DB8:9:124:200:f8ff:fe01:6317 sind einer ausgehenden Schnittstelle zugewiesen. 2001:0DB8:808:1:200:f8ff:fe01:6317 wird als Quelle für das Ziel 2001:0DB8:800::1 ausgewählt.

Beachte, dass obige Regel nicht in der IPv6-Spezifikation dokumentiert ist. Es wird als "up to implementation"-Punkt betrachtet. Es gibt einige Fälle, bei denen die obige Regel nicht benutzt werden soll. Ein Beispiel ist die verbundene TCP-Sitzung und man benutzt die Adresse, die in tcb als Quelle gehalten wird. Ein anderes Beispiel ist die Quelladresse für Neighbor Advertisement. Laut Spezifikation (RFC2461 7.2.2) sollte die Quelle des NA die Zieladresse des korrespondierenden Ziel des NS sein. In diesem Fall folgen wir eher der Spezifikation, als der obigen longest-match-Regel.

Für neue Verbindungen werden (wenn Regel eins nicht zutrifft) abgelehnte Adressen (Adressen mit bevorzugter Lebenszeit = 0) nicht ausgewählt, wenn andere Auswahlmöglichkeiten bestehen. Wenn keine anderen Auswahlmöglichkeiten bestehen, werden abgelehnte Adressen als letzte Möglichkeit benutzt. Falls mehrere Auswahlmöglichkeiten für abgelehnte Adressen bestehen, dann wird ogige Regel verwendet, um aus diesen abgelehnten Adressen auszuwählen. Falls man aus bestimmten Gründen die Benutzung abgelehnter Adressen unterbinden möchte, dann setzt man net.inet6.ip6.use_deprecated auf 0. Der Punkt bezüglich der abgelehnten Adressen ist in RFC2462 5.5.4 beschrieben (Beachte: Im Moment wird in der IETF ipngwg darüber debatiert, wie angelehnte Adressen benutzt werden sollen).

8.1.1.7. Jumbo Payload

Die Jumbo-Payload hop-by-hop-Option ist implementiert und kann benutzt werden, um IPv6-Pakete mit Datenpaketen größer als 65.535 Oktette. Aber im Moment wird keine physikalische Schnittstelle unterstützt, deren MTU größer ist als 65.536, so dass diese Datenpakete nur bei den loopback-Schnittstellen zu finden sind (z.B. lo0).

Falls man die Jumbo Payloads testen möchte, muss man zunächst den Kernel rekonfigurieren, so dass die MTU der loopback-Schnittstelle grösser 65.535 Bytes sein kann. Füge folgende Zeile zur Kernel-Konfiguration hinzu:

```
options "LARGE_LOMTU" #Um Jumbo Payload zu testen
```

und dann kompiliere den Kernel neu.

Dann kann man die Jumbo-Payloads mittels ping6(8)-Kommando mit den Optionen -b und -s testen. Die Option -b muss angegeben werden, um die Größe des Socket-Puffers zu erhön, und die Option -s gibt die Größe des Pakets an, die größer als 65.535 sein sollte. Beispielsweise gibt man folgendes ein:

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

Die IPv6-Spezifikation verlangt, dass die Jumbo-Payload-Option nicht in einem Paket verwendet werden darf, das einen fragmentierten Header hat. Falls diese Bedingung nicht zutrifft, dann muss eine ICMPv6-Parameter-Problem-Nachricht an den Absender geschickt werden. Die Spezifikation ist befolgt, aber man kann normalerweise nicht einen ICMPv6-Fehler sehen, der durch diese Forderung hervorgerufen wird.

Wenn ein IPv6-Paket empfangen wird, dann wird die Rahmenlänge geprüft und sie wird mit der Größe verglichen, die im Datenfeld für die Paketgröße des IPv6-Headers oder im Wert für die Jumbo-Payload-Option angegeben ist, sofern vorhanden. Falls ersterer kleiner als letzterer ist, dann wird das Paket abgelehnt und die Statistiken werden erhöht. Man kann die Statistik als Ausgabe des

netstat(8)-Kommandos mit der `-s -p ip6`-Option sehen:

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

So, der Kernel sendet keinen ICMPv6-Fehler, außer das fehlerhafte Paket ist ein aktuelles Jumbo-Payload, dessen Paketgröße größer als 65,535 Bytes ist. Wie oben beschrieben, gibt es momentan keine physikalische Schnittstelle, die eine so riesige MTU unterstützt, daher gibt es so selten einen ICMPv6-Fehler.

TCP/UDP over Jumbogramm wird im Moment nicht unterstützt. Dies kommt daher, weil wir kein Medium (außer loopback) haben, dies zu testen. Melden Sie sich, falls Sie es benötigen.

IPsec funktioniert nicht mit Jumbogramm. Dies ist bedingt durch einige Änderungen an der Spezifikation, welche die Unterstützung von AH mit Jumbogramm betrifft (AH-Header-Größe beeinflusst die Länge des Datenpakets und das macht es richtig schwierig, ein eingehendes Paket mit Jumbo-Payload-Option so gut zu authentifizieren wie ein AH).

Es gibt grundlegende Punkte in der *BSD-Unterstützung für Jumbogramms. Wir würden jene gerne ansprechen, aber wir benötigen mehr Zeit diese fertig zu stellen. Um ein paar zu benennen:

- mbuf pkthdr.len-Feld ist in 4.4BSD typisiert als "int", so dass es kein Jumbogramm mit len > 2G bei 32Bit-Architekturen aufnehmen kann. Wenn wir Jumbogramme geeignet unterstützen wollten, dann muss das Feld erweitert werden, damit es 4G + IPv6-Header + link-layer-Header aufnehmen kann. Deshalb muss es schließlich auf int64_t (u_int32_t ist NICHT genug) erweitert werden.
- Irrigerweise benutzen wir "int" an vielen Stellen, um die Paketlänge aufzunehmen. Wir müssen sie in einen größeren ganzzahligen Typ konvertieren. Es braucht große Vorsicht, weil wir sonst einen Überlauf während der Berechnung der Paketlänge erleben können.
- Irrigerweise prüfen wir das ip6_plen-Feld des IPv6-Header für packet payload length an verschiedenen Stellen. Wir sollten mbuf pkthdr.len stattdessen prüfen. ip6_input() wird bei der Eingabe eine Prüfung der Jumbo -Payload-Option durchführen und wir können danach mbuf pkthdr.len sicher benutzen.
- Natürlich braucht der TCP-Kode an einigen Stellen eine sorgfältige Aktualisierung.

8.1.1.8. Verhindern von Schleifen beim Verarbeiten von Headern

Die IPv6-Spezifikation erlaubt eine willkürliche Zahl von Erweiterungs-Headern, die in einem Paket platziert werden können. Wenn wir IPv6-Kode für die Paketverarbeitung auf die Art und Weise implementieren wie wir es beim BSD-IPv4-Kode geschehen ist, dann würde wegen einer lange Kette von Funktionsaufrufen der Kernel-Stack überlaufen. sys/netinet6-Kode ist behutsam entwickelt wurden, um einen Überlauf des Kernel-Stacks zu verhindern. Deswegen definiert der sys/netinet6-Kode seine eigene Protocol-Switch-Struktur "struct ip6protosw" (siehe auch netinet6/ip6protosw.h). Aus Gründen der Kompatibilität gibt es keine solche Aktualisierung im IPv4-Teil (sys/netinet), aber eine kleine Änderung ist zum pr_input()-Prototyp hinzugefügt worden. So ist "struct ipprotosw"

ebenso definiert. Deswegen kann der Kernel-Stack sich aufblähen, wenn man ein IPsec-over-IPv4-Paket mit einer massiven Zahl von IPSec-Header empfängt. IPsec-over-IPv6 ist in Ordnung. (Natürlich muss für all diese zu verarbeitenden IPSec-Header jeder einzelne IPSec-Header jede IPSec-Prüfung durchlaufen. So wird es einem anonymen Angreifer unmöglich gemacht eine Attacke durchzuführen.)

8.1.1.9. ICMPv6

Nachdem RFC2463 veröffentlicht worden war, hat die IETF-ipngwg beschlossen ICMPv6-Fehler-Pakete gegen ICMPv6 umzuleiten, um einen ICMPv6-Sturm auf einem Netzwerkmedium zu unterbinden. Dies ist bereits im Kernel implementiert.

8.1.1.10. Anwendungen

Für Programmierung des Userland unterstützen wir das IPv6-Socket-API wie es in RFC2553, RFC2292 und in aufkommenden Internet-Konzepten beschrieben ist.

TCP/UDP über IPv6 ist verfügbar und ziemlich stabil. Man kann sich an telnet(1), ftp(1), rlogin(1), rsh(1), ssh(1), usw. erfreuen. Diese Anwendungen sind unabhängig vom Protokoll. Das liegt daran, weil diese Programme automatisch IPv4 oder IPv6 entsprechend des DNS auswählen.

8.1.1.11. Kernel Interna

Während ip_forward() ip_output() aufruft, ruft ip6_forward() direkt if_output() auf, da Router IPv6-Pakete nicht in Fragmente teilen dürfen.

ICMPv6 sollte das original Paket so lang wie möglich bis maximal 1280 halten. UDP6/IP6 port unreach, zum Beispiel, sollte alle Erweiterungs-Header und die unveränderten UDP6- und IP6-Header enthalten. Um das originale Paket zu erhalten, konvertieren alle IP6-Funktionen außer TCP niemals Network-Byte-Order in Host-Byte-Order.

tcp_input(), udp6_input() und icmp6_input() können nicht voraussetzen, dass der IP6-Header vor dem Transport-Header, der zum Extension-Header gehört, kommt. Deshalb wurde in6_cksum() implementiert, um Pakete, deren IP6-Header und Transport-Header nicht fortlaufend ist, zu behandeln. Weder TCP/IP6- noch UDP6/IP6-Header-Strukturen existieren, um eine Prüsumme zu bilden.

Um IP6-Header, Extension-Header und Transport-Headers leichter verarbeiten zu können, werden nun Netzwerktreiber benötigt, die Pakete in einem internen mbuf oder in einem oder mehreren externen mbuf speichern können. Ein typischer alter Treiber legt zwei interne mbuf für 96 - 204 Bytes an Daten an, wie auch immer wird ein solches Paket jetzt in einem externen mbuf gespeichert.

`netstat -s -p ip6` ermittelt, ob der Treiber sich nach solchen Erfordernissen richtet, oder nicht. Im folgenden Beispiel verletzt "cce0" dies Erfordernisse (Für weitere Informationen, siehe Abschnitt 2.).

```
Mbuf statistics:
                317 one mbuf
                two or more mbuf::
```

```
                        lo0 = 8
             cce0 = 10
                 3282 one ext mbuf
                 0 two or more ext mbuf
```

Jede Eingabefunktion ruft IP6_EXTHDR_CHECK am Anfang auf, um zu prüfen, ob der Bereich zwischen IP6 und seinen Header durchgehend ist. IP6_EXTHDR_CHECK ruft m_pullup() nur dann auf, wenn mbuf das M_LOOP-Flag gestzt hat, weil das Paket von der Loopback-Schnittstelle kommt. m_pullup() wird niemals aufgerufen, wenn Pakete von physikalischen Netzwerkschnittstellen kommen.

IP- und IP6-Reassemble-Funktionen rufen niemals m_pullup() auf.

8.1.1.12. IPv4-Mapped-Address und IPv6-Wildcard-Socket

RFC2553 beschreibt IPv4-Mapped-Address (3.7) und die spezielle Verhaltensweise des IPv6-Wildcard-Bind-Socket (3.8). Die Spezifikation gestattet es:

- IPv4-Verbindungen von AF_INET6-Wildcard-Bind-Socket zu erlauben.
- IPv4-Pakete über AF_INET6-Socket zu transportieren, indem eine spezielle Form der Adresse wie ::ffff:10.1.1.1 benutzt wird.

Aber die Spezifikation ist sehr kompliziert und spezifiziert nicht, wie der Socket-Layer sich verhalten soll. Darauf Bezug nehmend nennen wir hier ersteren "hörende Seite" und letzteren "beginnende Seite".

Man kann einen Wildcard-Bind auf demselben Port bei beiden Adressfamilien durchführen.

Die folgende Tabelle zeigt das Verhalten von FreeBSD 4.x.

```
Hörende Seite          Beginnende Seite
               (AF_INET6-Wildcard-      (Verbindung zu ::ffff:10.1.1.1)
               Socket erreicht IPv4 Verb.)
               ---                      ---
FreeBSD 4.x    Konfigurierbar             unterstützt
               Standard: erlaubt
```

Die folgende Abschnitte zeigen mehr Details und wie man das Verhalten konfigurieren kann.

Kommentare auf der hörenden Seite:

Es sieht so aus, dass RFC2553 zu wenig zu den Punkten über Wildcard-Bind erläutert, speziell zum Punkt über Port-Space, Fehler-Modus und Beziehung zwischen AF_INET/INET6 wildcard bind. Es kann mehrere unterschiedliche Interpretationen zu diesem RFC geben, die sich nach diesen richten, aber sich unterschiedlich verhalten. Um eine portable Anwendung zu implementieren, sollte man deshalb nicht ein bestimmtes Verhalten des Kernels voraussetzen. Der Einsatz von getaddrinfo(3) ist der sicherste Weg. Port number space und wildcard bind issues wurden Mitte Mai 1999 detailliert in der Ipv6imp-Mailing-Liste diskutiert und es sieht so aus, als ob es keinen

konkreten Konsens gab (means, up to implementers). Vielleicht sollte man die Archive der Mailing-Liste prüfen.

Wenn eine Server-Anwendung IPv4- und IPv6-Verbindungen annehmen möchte, dann gibt es zwei Alternativen.

Eine benutzt AF_INET- und AF_INET6-Socket (man benötigt zwei Sockets). Benutze getaddrinfo(3) mit gesetztem AI_PASSIVE-Bit in ai_flags, socket(2) und bind(2) für alle zurückgegebenen Adressen. Mit dem öffnen mehrerer Sockets kann man Verbindungen an dem Socket mit der richtigen Adressfamilie annehmen. IPv4-Verbindungen werden vom AF_INET-Socket und IPv6-Verbindungen vom AF_INET6-Socket angenommen.

Ein anderer Weg ist einen AF_INET6 wildcard bind-Socket zu verwenden. Man benutzt getaddrinfo(3) mit AI_PASSIVE in ai_flags, mit AF_INET6 in ai_family, man setzt das erste Argument hostname auf NULL, socket(2) und bind(2) auf die zurückgegebene Adresse (es sollte eine unspezifizierte IPv6-Adresse sein). Man kann IPv4- und IPv6-Paket über diesen Socket annehmen.

Um nur IPv6-Datenverkehr portabel an AF_INET6 wildcard gebundenen Socket zu unterstützen, prüft man, sobald die Verbindung Zustande gekommen ist, immer die Peer-Adresse gegen den hörenden AF_INET6-Socket. Wenn die Adresse eine IPv4-Mapped-Adresse ist, dann sollte man die Verbindung zurückweisen. Man kann die Bedingung mit dem IN6_IS_ADDR_V4MAPPED()-Makro prüfen.

Um diesen Punkt leichter lösen zu können, gibt es für setsockopt(2) die System abhängige Option IPV6_BINDV6ONLY, die wie folgt benutzt wird.

```
    int on;

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

Wenn der Aufruf erfolgreich ist, dann empfängt dieser Socket nur IPv6-Pakete.

Kommentare zur sendenden Seite:

Ratschlag an Anwendungsentwickler: um eine portable IPv6-Anwendung zu implementieren (die mit verschiedenen IPv6-Kerneln funktioniert), ist das Folgende der Schlüssel zum Erfolg wie wir glauben:

- NIEMALS AF_INET oder AF_INET6 hart kodieren.
- Benutze getaddrinfo(3) und getnameinfo(3) überall im System. Benutze niemals gethostby*(), getaddrby*(), inet_*() oder getipnodeby*() (Um bestehende Applikationen leicht IPv6 fähig zu machen, wird getipnodeby*() manchmal nützlich sein. Falss es aber möglich sein sollte, versuche den Kode neu zu schreiben und getaddrinfo(3) und getnameinfo(3) zu benutzen)
- Wenn man sich an ein Ziel verbinden möchte, benutze getaddrinfo(3) und versuche alle zurückgegebenen Ziele, wie telnet(1) es macht.
- Einige IPv6-Stacks sind mit fehlerhafter getaddrinfo(3) verschickt worden. Man verschickt als letzte Möglichkeit eine minimal arbeitende Version der Anwendung.

Wenn man einen AF_INET6-Socket für jeweils eine ausgehende IPv4- und IPv6-Verbingung benutzen möchte, dann muss man getipnodebyname(3) benutzen. Wenn man seine existierende Anwendung mit wenig Aufwand IPv6-fähig machen möchte, dann sollte dieser Versuch gewählt werden. Aber beachte bitte, dass dies eine temporäre Lösung ist, weil getipnodebyname(3) selber noch zu empfehlen ist, da es noch keine Adressbereiche verarbeitet. Für eine IPv6-NAmensauflösung ist getaddrinfo(3) das bevorzugte API. Deshalb sollte man seine Anwendung so umschreiben, dass getaddrinfo(3) benutzt wird, wann man Zeit dazu hat.

Wenn man Anwendungen schreibt, die ausgehende Verbindungen herstellen, wird die Geschichte viel einfacher, wenn man AF_INET und AF_INET6 als total getrennte Adressfamilien behandelt. {set,get}sockopt funktioniert viel einfacher, DNS-Angelegenheiten werden einfacher gemacht. Wir empfehlen sich nicht auf IPv4-Mapped-Adressen zu verlassen.

8.1.1.12.1. Einheitlicher TCP-und INPCB-Kode

FreeBSD 4.x benutzt shared TCP-Kode zwischen IPv4 und IPv6 (von sys/netinet/tcp*) und separaten udp4/6-Kode. Es benutzt eine vereinheitlichte inpcb-Struktur.

Die Plattform kann für eine Unterstützung von IPv4-mapped-Adressen konfiguriert werden. Die Kernel-Konfiguration läßt sich wie folgt zusammenfassen:

- 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.
- Man kann es wie unten beschrieben abschalten.

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

8.1.1.12.1.1. Hörende Seite

Jeder Socket kann für eine Unterstützung eines speziellen AF_INET6 wildcard bind (Standardmäßig eingeschaltet) konfiguriert werden. Man kann es auf Socket-Basis mit setsockopt(2) wie unten beschrieben abschalten.

```
    int on;

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

Wildcard-AF_INET6-Socket schnappt sich die IPv4-Verbindung, wenn, und nur wenn folgende Bedingungen erfüllt sind

- Es gibt keinen AF_INET-Socket, der zu einer IPv4-Verbindung passt
- Der AF_INET6-Socket ist so konfiguriert, dass er IPv4-Datenverkehr akzeptiert, z.B. gibt getsockopt(IPV6_BINDV6ONLY) 0 zurück.

Es gibt kein Problem mit der Öffnen/Schließen-Reihenfolge.

8.1.1.12.1.2. initiating side

FreeBSD 4.x unterstützt ausgehende Verbindungen zu IPv4 mapped Adressen (::ffff:10.1.1.1), falls der Knoten so konfiguriert ist, dass er IPv4 mapped Adressen unterstützt.

8.1.1.13. sockaddr_storage

Als RFC2553 kurz vor der Vollendung stand, gab es eine Diskussion, wie struct sockaddr_storage Mitglieder benannt werden sollten. Ein Vorschlag war "" *den Mitgliedern (wie* "ss_len") voranzustellen und es sollten sie nicht verändert werden. Der andere Vorschlag war, nichts voranzustellen (wie "ss_len") also mußten wir solche Mitglieder direkt verändern. Es gab keinen klaren Konsens.

Als Ergebnis definiert RFC2553 die Struktur sockaddr_storage wie folgt:

```
    struct sockaddr_storage {
        u_char  __ss_len;   /* address length */
        u_char  __ss_family;    /* address family */
        /* and bunch of padding */
    };
```

Im Gegensatz dazu definiert der XNET-Entwurf die Struktur wie folgt:

```
    struct sockaddr_storage {
        u_char  ss_len;     /* address length */
        u_char  ss_family;  /* address family */
        /* and bunch of padding */
    };
```

Im Dezember 1999 kam man überein, dass RFC2553bis letztere Definition (XNET) aufnehmen sollte.

Die aktuelle Implementierung ist konform zur XNET-Definition basierend auf der RFC2553bis Diskussion.

Wenn man mehrere IPv6-Implementierungen betrachtet, wird man beide Definitionen sehen. Für Userland-Programmierer ist der folgende Weg der meist portable um damit umzugehen:

1. Man versichert sich, dass ss_family und/oder ss_len für die Plattform verfügbar sind, indem man GNU autoconf verwendet,
2. Man benutzet -Dss_family=*ss_family um alle Vorkommen (einschließlich der Header-Files) zu* ss_family zu vereinheitlichen, oder
3. Man benutzt niemals __ss_family. Man führe einen Typecast nach sockaddr * durch und verwendet sa_family wie folgt:

   ```
       struct sockaddr_storage ss;
       family = ((struct sockaddr *)&ss)->sa_family
   ```

8.1.2. Netzwerktreiber

Die beiden folgenden Dinge müssen zwingend von Standardtreibern unterstützt werden:

1. Mbuf-Clustering-Erfordernis. In diesem stabilen Release haben wir für alle Betriebssystem MINCLSIZE in MHLEN+1 geändert, damit sich alle Treiber wie erwartet verhalten.
2. Multicast. Falls ifmcstat(8) keine Multicast-Gruppe für die Schnittstelle liefert, dann muss diese Schnittstelle überarbeitet werden.

Falls keiner der Treiber die Erfordernisse erfüllt, dann können die Treiber nicht für IPv6/IPSec-Kommunikation verwendet werden. Falls man ein Problem beim Einsatz von IPv6/IPSec mit seiner Karte hat, dann melde es bitte bei FreeBSD problem reports.

(Beachte: In der Vergangenheit haben wir gefordert, dass alle PCMCIA-Treiber einen Aufruf nach in6_ifattach() haben. Inzwischen haben wir keine solche Forderung mehr)

8.1.3. Translator

Wir kategorisieren einen IPv4/IPv6-Translator in 4 Typen:

- *Translator A* --- Er wird im frühen Stadium des Übergangs benutzt um es zu ermöglichen, dass eine Verbindung von einem IPv6-Host auf einer IPv6-Insel zu einem IPv4-Host im IPv4-Ozean hergestellt wird.
- *Translator B* --- Er wird im frühen Stadium des Übergangs benutzt um es zu ermöglichen, dass eine Verbindung von einem IPv4-Host im IPv4-Ozean zu einem IPv6-Host auf einer IPv6-Insel hergestellt wird.
- *Translator C* --- Er wird im frühen Stadium des Übergangs benutzt um es zu ermöglichen, dass eine Verbindung von einem IPv4-Host auf einer IPv4-Insel zu einem IPv6-Host im IPv6-Ozean hergestellt wird.
- *Translator D* --- Er wird im frühen Stadium des Übergangs benutzt um es zu ermöglichen, dass eine Verbindung von einem IPv6-Host im IPv6-Ozean zu einem IPv4-Host auf einer IPv4-Insel hergestellt wird.

Ein TCP-Relay-Translator der Kategorie A wird unterstützt. Er wird "FAITH" genannt. Wir stellen ebenso einen IP-Header-Translator der Kataegorie A zur Verfügung (Letzterer ist noch nicht in FreeBSD 4.x übernommen).

8.1.3.1. FAITH TCP-Relay-Translator

Das FAITH-System benutzt mit Hilfe des Kernels den faithd(8) genannten TCP-Relay-Daemon. FAITH wird einen IPv6-Adress-Präfix reservieren und eine TCP-Verbindungen an diesen Präfix zum IPv4-Ziel weiterleiten.

Wenn beispielsweise der IPv6-Präfix 2001:0DB8:0200:ffff

ist und das IPv6-Ziel für TCP-Verbindungen 2001:0DB8:0200:ffff::163.221.202.12 ist, dann wird die Verbindung an das IPv4-Ziel 163.221.202.12 weitergeleitet.

```
    IPv4-Ziel-Knoten (163.221.202.12)
```

```
      ^
      | IPv4 tcp toward 163.221.202.12
    FAITH-relay dual stack node
      ^
      | IPv6 TCP toward 2001:0DB8:0200:ffff::163.221.202.12
    source IPv6 node
```

faithd(8) muss auf FAITH-relay dual stack node aufgerufen werden.

Für weitere Details siehe src/usr.sbin/faithd/README

8.1.4. IPsec

IPsec besteht hauptsächlich aus drei Komponenten.

1. Policy Management
2. Key Management
3. AH und ESP Behandlung

8.1.4.1. Regel Management

Im Kernel ist experimenteller Kode für Regel-Management implementiert. Es gibt zwei Wege eine Sicherheitsregel zu handhaben. Einer ist eine Regel für jeden Socket mithilfe von setsockopt(2) zu konfigurieren. Für diesen Fall ist die Konfiguration der Regel in ipsec_set_policy(3) beschrieben. Der andere Weg ist eine auf einem Kernel-Packet-Filter basierende Regel mithilfe der PF_KEY-Schnittstelle mittels setkey(8) zu konfigurieren.

Der Regeleintrag mit seinen Indices wird nicht sortiert, so dass es sehr wichtig ist, wann ein Eintrag hinzugefügt wird.

8.1.4.2. Key Management

Der in dieser Bibliothek (sys/netkey) implementierte Kode für das key management ist eine Eigenentwicklung der PFKEYv2-Implementierung. Er ist konform zu RFC2367.

Die Eigenentwicklung des IKE-Daemons "racoon" ist in der Bibliothek (kame/kame/racoon) implementiert. Grundsätzlich muss man racoon als Dämonprozess laufen lassen, dann setzt man eine Regel auf, die Schlüssel erwartet (ähnlich wie `ping -P 'out ipsec esp/transport//use'`). Der Kernel wird den racoon-Dämon wegen des notwendigen Austauschs der Schlüssel kontaktieren.

8.1.4.3. AH- und ESP-Handhabung

Das IPsec-Modul ist als "hook" in die Standard-IPv4/IPv6-Verarbeitung implementiert. Sobald ein Paket gesendet wird, prüft ip{,6_output(), ob eine ESP/AH-Verarbeitung notwendig ist. Es findet eine Überprüfung statt, ob eine passende SPD (Security Policy Database) gefunden wurde. Wenn ESP/AH benötigt wird, dann wird {esp,ah}{4,6}_output() aufgerufen und mbuf wird folglich aktualisiert. Wenn ein Paket empfangen wird, dann wird {esp,ah}4_input() basierend auf der Protokollnummer aufgerufen, z.B. (*inetsw[proto])(). {esp,ah}4_input() entschlüsselt/prüft die Authentizität des Pakets und entfernt den daisy-chained-Header und das Padding des ESP/AH. Es ist sicherer den ESP/AH-

Header beim Empfang zu entfernen, weil man das empfangene Paket niemals so wie es ist benutzt.

Mit der Verwendung von ESP/AH wird die effektive TCP4/6-Datensegmentgröße durch weitere von ESP/AH eingefügte Daisy-chained-Headers beeinflußt. Unser Kode berücksichtigt dies.

Grundlegende Crypto-Funktionen sind im Verzeichnis "sys/crypto" zu finden. ESP/AH-Umformungen sind zusammen mit den Wrapper-Funktionen in {esp,ah}_core.c gelistet. Wenn man einige Algorithmen hinzufügen möchte, dann fügt man in {esp,ah}_core.c eine Wrapper-Funktion hinzu und trägt seinen Crypto-Algorithmus in sys/crypto ein.

Der Tunnel-Modus wird in diesem Release teilweise mit den folgenden Restriktionen unterstützt:

- Der IPsec-Tunnel ist nicht mit der generischen Tunnelschnittstelle kombiniert. Man muss sehr vorsichtig sein, weil man sonst eine Endlosschleife zwischen ip_output() und tunnelifp→if_output() aufbaut. Die Meinungen gehen auseinander, ob es besser ist dies zu vereinheitlichen, oder nicht.
- Die Betrachtung von MTU und des "Don't Fragment"-Bits (IPv4) müssen mehr geprüft werden, aber grundsätzlichen arbeiten sie gut.
- Das Authentifizierungsmodel für einen AH-Tunnel muss überarbeitet werden. Man muss eventuell die "policy management engine" überarbeiten.

8.1.4.4. Konformität zu RFCs und IDs

Der IPsec-Kode im Kernel ist konform (oder versucht konform zu sein) zu den folgenden Standards:

Die "alte IPsec"-Spezifikation, die in rfc182[5-9].txt dokumentiert ist

Die "neue IPsec"-Spezifikation, die rfc240[1-6].txt, rfc241[01].txt, rfc2451.txt und draft-mcdonald-simple-ipsec-api-01.txt (Der Entwurf ist erloschen, aber man kann ihn sich von ftp://ftp.kame.net/pub/internet -drafts/ holen) dokumentiert ist (Beachte: Die IKE-Spezifikationen rfc241[7-9].txt sind im Userland als "racoon"-IKE-Daemon implementiert).

Aktuell werden folgende Algorithmen unterstützt:

- altes IPsec-AH
  - null crypto Prüfsumme (Kein Dokument, nur für Debug-Zwecke)
  - keyed MD5 mit 128bit crypto Prüfsumme (rfc1828.txt)
  - keyed SHA1 mit 128bit crypto Prüfsumme (kein Document)
  - HMAC MD5 mit 128bit crypto Prüfsumme (rfc2085.txt)
  - HMAC SHA1 mit 128bit crypto Prüfsumme (kein Dokument)
- altes IPsec-ESP
  - null encryption (kein Dokument, ähnlich zu rfc2410.txt)
  - DES-CBC-Modus (rfc1829.txt)
- neues IPsec-AH
  - null crypto Prüfsumme (kein Dokument, nur für Debug-Zwecke)

    - keyed MD5 mit 96bit crypto Prüfsumme (kein Dokument)
    - keyed SHA1 mit 96bit crypto Prüfsumme (kein Dokument)
    - HMAC MD5 mit 96bit crypto Prüfsumme (rfc2403.txt)
    - HMAC SHA1 mit 96bit crypto Prüfsumme (rfc2404.txt)
- neues IPsec-ESP
    - null encryption (rfc2410.txt)
    - DES-CBC mit abgeleiteter IV (draft-ietf-ipsec-ciph-des-derived-01.txt, Entwurf abgelaufen)
    - DES-CBC mit expliziter IV (rfc2405.txt)
    - 3DES-CBC mit expliziter IV (rfc2451.txt)
    - BLOWFISH CBC (rfc2451.txt)
    - CAST128 CBC (rfc2451.txt)
    - RC5 CBC (rfc2451.txt)
    - Jeder Algorithmus kann kombiniert werden mit:
        - ESP-Beglaubigung mit HMAC-MD5(96bit)
        - ESP-Beglaubigung mit HMAC-SHA1(96bit)

Die folgenden Algorithmen werden NICHT unterstützt:

- altes IPsec-AH
    - HMAC MD5 mit 128bit crypto Prüfsumme + 64bit replay prevention (rfc2085.txt)
    - keyed SHA1 mit 160bit crypto Prüfsumme + 32bit padding (rfc1852.txt)

IPsec (im Kernel) und IKE (im Userland als "racoon") wurden bei unterschiedlichen Interoperabilitätstests geprüft und es ist bekannt, dass es mit vielen anderen Implementierungen gut zusammenarbeitet. Außerdem wurde die IPsec-Implementierung sowie die breite Abdeckung mit IPsec-Crypto-Algorithmen, die in den RFCs dokumentiert sind, geprüft (es werden nur Algorithmen ohne intellektuelle Besitzansprüche behandelt).

8.1.4.5. ECN-Betrachtung von IPsec-Tunneln

ECN-freundliche IPsec-Tunnel werden unterstützt wie es in draft-ipsec-ecn-00.txt beschrieben ist.

Normale IPsec-Tunnel sind in RFC2401 beschrieben. Für eine Kapselung wird das IPv4-TOS-Feld (oder das IPv6-Traffic-Class-Feld) vom inneren in den äußeren IP-Header kopiert. Für eine Entkapselung wird der ässere IP-Header einfach verworfen. Die Entkapselungsregel ist nicht mit ECN kompatibel, sobald das ECN-Bit im äußeren IP-TOS/Traffic-Class-Feld verloren geht.

Um einen IPsec-Tunnel ECN-freundlich zu machen, sollte man die Kapselungs- und Entkapselungsprozeduren modifizieren. Dies ist in http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt, Kapitel 3, beschrieben.

Die IPsec-Tunnel-Implementierung kann drei Zustände annehmen, indem man net.inet.ipsec.ecn (oder net.inet6.ipsec6.ecn) auf diese Werte setzt:

- RFC2401: Keine Betrachtung von ECN (Sysctl-Wert -1)
- ECN verboten (Sysctl-Wert 0)
- ECN erlaubt (Sysctl-Wert 1)

Beachte, dass dieses Verhalten per-node konfigurierbar ist und nicht per-SA (draft-ipsec-ecn-00 möchte per-SA Konfiguration).

Das Verhalten ist wie folgt zusammengefaßt (man beachte auch den Quelltext für weitere Details):

```
               encapsulate                     decapsulate
               ---                             ---
RFC2401        kopiere alle TOS-Bits              lösche TOS-Bits im äußeren
               von innen nach außen.            (benutze innere TOS-Bits so wie sie
sind)

ECN verboten   kopiere TOS-Bits außer für ECN    lösche TOS-Bits im äußeren
               (maskiert mit 0xfc) von innen   (benutze innere TOS-Bits so wie sie
sind)
               nach außen.  Setze ECN-Bits auf 0.

ECN erlaubt    kopiere TOS-Bits außer für ECN    benutze innere TOS-Bits mit einigen
Änderungen.
               CE (maskiert mit 0xfe) von      Wenn das äußere ECN-CE-Bit 1 ist,
               innen nach außen.                 setze das ECN-CE-Bit im
               Setze ECN-CE-Bit auf 0.            Inneren.
```

Allgemeine Strategie zur Konfiguration:

- Wenn beide IPsec-Tunnel-Endpunkte ein ECN-freundliches Verhalten beherrschen, dann sollte man besser beide Endpunkte auf "ECN allowed" (Sysctl-Wert 1) setzen.
- Wenn das andere Ende das TOS-Bit sehr strikt handhabt, dann benutzt man "RFC2401" (Sysctl-Wert -1).
- in den anderen Fällen benutzt man "ECN verboten" (Sysctl-Wert 0).

Der Standard ist "ECN verboten" (Sysctl-Wert 0).

Für weitere Informationen siehe auch:

http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt, RFC2481 (Explicit Congestion Notification), src/sys/netinet6/{ah,esp}_input.c

(Dank gebührt Kenjiro Cho kjc@csl.sony.co.jp für seine detailliert Analyse)

8.1.4.6. Interoperabilität

Hier sind einige Plattformen angegeben, die in der Vergangenheit die IPsec/IKE-Interoperabilität mit dem KAME-Kode getestet haben. Beachte, dass beide Enden vielleicht ihre Implementierung verändert haben, deshalb sollte man folgende Liste nur für Referenzzwecke benutzen.

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

Teil III: Kernel

Kapitel 9. Einen FreeBSD-Kernel bauen und installieren

Ein Kernelentwickler muss wissen, wie der Bau eines angepassten Kernels funktioniert, da das Debuggen des FreeBSD-Kernels nur durch den Bau eines neuen Kernels möglich ist. Es gibt zwei Wege, einen angepassten Kernel zu bauen:

- Den "traditionellen" Weg
- Den "neuen" Weg

Die folgenden Ausführungen setzen voraus, dass Sie den Abschnitt Erstellen und Installation eines angepassten Kernels des FreeBSD-Handbuchs gelesen haben und daher wissen, wie man einen FreeBSD-Kernel baut.

9.1. Einen Kernel auf die "traditionelle" Art und Weise bauen

Bis FreeBSD 4.X wurde dieser Weg zum Bau eines angepassten Kernels empfohlen. Sie können Ihren Kernel nach wie vor auf diese Art und Weise bauen (anstatt das Target "buildkernel" der Makefiles im Verzeichnis /usr/src/ zu verwenden). Dies kann beispielsweise sinnvoll sein, wenn Sie am Kernel-Quellcode arbeiten. Haben Sie nur ein oder zwei Optionen der Kernelkonfigurationsdatei geändert, ist dieser Weg in der Regel schneller als der "neue" Weg. Andererseits kann es aber auch zu unerwarteten Fehlern beim Bau des Kernels kommen, wenn Sie Ihren Kernel unter aktuellen FreeBSD-Versionen auf diese Art und Weise bauen.

1. Erzeugen Sie den Kernel-Quellcode mit config(8):

```
# /usr/sbin/config MYKERNEL
```

2. Wechseln Sie in das Build-Verzeichnis. config(8) gibt den Namen dieses Verzeichnisses aus, wenn die Erzeugung des Kernel-Quellcodes im vorherigen Schritt erfolgreich abgeschlossen wurde.

```
# cd ../compile/MYKERNEL
```

3. Kompilieren Sie den neuen Kernel:

```
# make depend
# make
```

4. Installieren Sie den neuen Kernel:

```
# make install
```

9.2. Einen Kernel auf die "neue" Art und Weise bauen

Dieser Weg wird für alle aktuellen FreeBSD-Versionen empfohlen. Lesen Sie bitte den Abschnitt Erstellen und Installation eines angepassten Kernels des FreeBSD-Handbuchs, wenn Sie Ihren Kernel auf diese Art und Weise bauen wollen.

Kapitel 10. Kernel-Fehlersuche

10.1. Besorgen eines Speicherauszugs nach einem Kernel-Absturz (Kernel-Crash-Dump)

Wenn ein Entwicklungs-Kernel (z.B. FreeBSD-CURRENT) wie zum Beispiel ein Kernel unter Extrembedingungen (z.B. sehr hohe Belastungsraten (Load), eine äußerst hohe Anzahl an gleichzeitigen Benutzern, Hunderte jail(8)s usw.) eingesetzt oder eine neue Funktion oder ein neuer Gerätetreiber in FreeBSD-STABLE verwendet wird (z.B. PAE), tritt manchmal eine Kernel-Panic ein. In einem solchen Fall zeigt dieses Kapitel, wie dem Absturz nützliche Informationen entnommen werden können.

Bei Kernel-Panics ist ein Neustart unvermeidlich. Nachdem ein System neu gestartet wurde, ist der Inhalt des physikalischen Speichers (RAM), genauso wie jedes Bit, das sich vor der Panic auf dem Swap-Gerät befand, verloren. Um die Bits im physikalischen Speicher zu erhalten, zieht der Kernel Nutzen aus dem Swap-Gerät als vorübergehenden Ablageort, wo die Bits, welche sich im RAM befinden, auch nach einem Neustart nach einem Absturz verfügbar sind. Durch diese Vorgehensweise kann ein Kernel-Abbild, wenn FreeBSD nach einem Absturz startet, abgezogen und mit der Fehlersuche begonnen werden.

Ein Swap-Gerät, das als Ausgabegerät (Dump-Device) konfiguriert wurde, verhält sich immer noch wie ein Swap-Gerät. Die Ausgabe auf Nicht-Swap-Geräte (wie zum Beispiel Bänder oder CDRWs) wird zur Zeit nicht unterstützt. Ein "Swap-Gerät" ist gleichbedeutend mit einer "Swap-Partition".

Es stehen verschiedene Arten von Speicherabzügen zur Verfügung: komplette Speicherabzüge (full memory dumps), welche den gesamten Inhalt des physischen Speichers beinhalten, Miniauszüge (minidumps), die nur die gerade verwendeten Speicherseiten des Kernels enthalten (FreeBSD 6.2 und höhere Versionen) und Textauszüge (textdumps), welche geskriptete oder Debugger-Ausgaben enthalten (FreeBSD 7.1 und höher). Miniauszüge sind der Standardtyp der Abzüge seit FreeBSD 7.0 und fangen in den meisten Fällen alle nötigen Informationen ein, die in einem kompletten Kernel-Speicherabzug enthalten sind, da die meisten Probleme nur durch den Zustand des Kernels isoliert werden können.

10.1.1. Konfigurieren des Ausgabegeräts

Bevor der Kernel den Inhalt seines physikalischen Speichers auf einem Ausgabegerät ablegt, muss ein solches konfiguriert werden. Ein Ausgabegerät wird durch Benutzen des dumpon(8)-Befehls festgelegt, um dem Kernel mitzuteilen, wohin die Speicherauszüge bei einem Kernel-Absturz gesichert werden sollen. Das dumpon(8)-Programm muss aufgerufen werden, nachdem die Swap-Partition mit swapon(8) konfiguriert wurde. Dies wird normalerweise durch Setzen der `dumpdev` -Variable in rc.conf(5) auf den Pfad des Swap-Geräts (der empfohlene Weg, um einen Kernel-Speicherauszug zu entnehmen) bewerkstelligt, oder über `AUTO`, um die erste konfigurierte Swap-Partition zu verwenden. In HEAD ist die Standardeinstellung für `dumpdev`AUTO und änderte sich in den RELENG_*-Zweigen (mit Ausnahme von RELENG_7, bei dem `AUTO` beibehalten wurde) auf `NO`. In FreeBSD 9.0-RELEASE und späteren Versionen fragt bsdinstall, ob Speicherauszüge für das

Zielsystem während des Installationsvorgangs aktiviert werden sollen.

Vergleichen Sie /etc/fstab oder swapinfo(8) für eine Liste der Swap-Geräte.

Stellen Sie sicher, dass das in rc.conf(5) festgelegte `dumpdir` vor einem Kernel-Absturz vorhanden ist.

```
# mkdir /var/crash
# chmod 700 /var/crash
```

Denken Sie auch daran, dass der Inhalt von /var/crash heikel ist und sehr wahrscheinlich vertrauliche Informationen wie Passwörter enthält.

10.1.2. Entnehmen eines Kernel-Speicherauszugs (Kernel-Dump)

Sobald ein Speicherauszug auf ein Ausgabegerät geschrieben wurde, muss er entnommen werden, bevor das Swap-Gerät eingehängt wird. Um einen Speicherauszug aus einem Ausgabegerät zu entnehmen, benutzen Sie das savecore(8)-Programm. Falls `dumpdev` in rc.conf(5) gesetzt wurde, wird savecore(8) automatisch beim ersten Start in den Multiuser-Modus nach dem Absturz und vor dem Einhängen des Swap-Geräts aufgerufen. Der Speicherort des entnommenen Kernels ist im rc.conf(5)-Wert `dumpdir`, standardmäßig /var/crash, festgelegt und der Dateiname wird vmcore.0 sein.

In dem Fall, dass bereits eine Datei mit dem Namen vmcore.0 in /var/crash (oder auf was auch immer `dumpdir` gesetzt ist) vorhanden ist, erhöht der Kernel die angehängte Zahl bei jedem Absturz um eins und verhindert damit, dass ein vorhandener vmcore (z.B. vmcore.1) überschrieben wird. Während der Fehlersuche, möchten Sie höchst wahrscheinlich den vmcore mit der höchsten Version in /var/crash benutzen, wenn Sie den passenden vmcore suchen.

Falls Sie einen neuen Kernel testen, aber einen anderen starten müssen, um Ihr System wieder in Gang zu bringen, starten Sie es nur in den Singleuser-Modus, indem Sie das `-s`-Flag an der Boot-Eingabeaufforderung benutzen, und nehmen dann folgende Schritte vor:

```
# fsck -p
# mount -a -t ufs       # make sure /var/crash is writable
# savecore /var/crash /dev/ad0s1b
# exit                  # exit to multi-user
```

Dies weist savecore(8) an, einen Kernel-Speicherauszug aus /dev/ad0s1b zu entnehmen und den Inhalt in /var/crash abzulegen. Vergessen Sie nicht sicherzustellen, dass das Zielverzeichnis /var/crash genug freien Speicherplatz für den Speicherauszug zur Verfügung hat. Vergessen Sie auch nicht, den korrekten Pfad des Swap-Geräts anzugeben, da es sehr wahrscheinlich anders als /dev/ad0s1b lautet!

10.2. Fehlersuche in einem Speicherauszug nach einem Kernel-Absturz mit kgdb

Dieser Abschnitt deckt kgdb(1) ab, wie es in FreeBSD 5.3 und später zu finden ist. In früheren Versionen muss `gdb -k` benutzt werden, um einen Kernspeicherauszug auszulesen.

Sobald ein Speicherauszug zur Verfügung steht, ist es recht einfach nützliche Informationen für einfache Probleme daraus zu bekommen. Bevor Sie sich auf die Interna von kgdb(1) stürzen, um die Fehler im Kernspeicherauszug zu suchen und zu beheben, machen Sie die Debug-Version Ihres Kernels (normalerweise kernel.debug genannt) und den Pfad der Quelldateien, die zum Bau Ihres Kernels verwendet wurden (normalerweise /usr/obj/usr/src/sys/KERNCONF, wobei KERNCONF das in einer Kernel-config(5) festgelegte `ident` ist), ausfindig. Mit diesen beiden Informationen kann die Fehlersuche beginnen.

Um in den Debugger zu gelangen und mit dem Informationserhalt aus dem Speicherauszug zu beginnen, sind zumindest folgende Schritte nötig:

```
# cd /usr/obj/usr/src/sys/KERNCONF
# kgdb kernel.debug /var/crash/vmcore.0
```

Sie können Fehler im Speicherauszug nach dem Absturz suchen, indem Sie die Kernel-Quellen benutzen, genauso wie Sie es bei jedem anderen Programm können.

Dieser erste Speicherauszug ist aus einem 5.2-BETA-Kernel und der Absturz ist tief im Kernel begründet. Die Ausgabe unten wurde dahingehend bearbeitet, dass sie nun Zeilennummern auf der linken Seite einschließt. Diese erste Ablaufverfolgung (Trace) untersucht den Befehlszeiger (Instruction-Pointer) und beschafft eine Zurückverfolgung (Back-Trace). Die Adresse, die in Zeile 41 für den `list`-Befehl benutzt wird, ist der Befehlszeiger und kann in Zeile 17 gefunden werden. Die meisten Entwickler wollen zumindest dies zugesendet bekommen, falls Sie das Problem nicht selber untersuchen und beheben können. Falls Sie jedoch das Problem lösen, stellen Sie sicher, dass Ihr Patch seinen Weg in den Quellbaum mittels eines Fehlerberichts, den Mailinglisten oder ihres Privilegs, zu committen, findet!

```
 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_ebp = -602076292, tf_isp = -602076324, tf_ebx = 0, tf_edx = 0, tf_ecx =
1000000, tf_eax = 243, tf_trapno = 12, tf_err = 0, tf_eip = -1066321824, tf_cs = 8,
tf_eflags = 65671, tf_esp = 243, tf_ss = 0})
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 = -602075788, tf_ebx = 672411944, tf_edx = 10, tf_ecx = 672411600,
tf_eax = 1, tf_trapno = 12, tf_err = 2, tf_eip = 671899563, tf_cs = 31, tf_eflags =
642, tf_esp = -1077940740, tf_ss = 47})
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
```

Diese nächste Ablaufverfolgung ist ein älterer Speicherauszug aus FreeBSD 2-Zeiten, aber ist komplizierter und zeigt mehr der gdb-Funktionen. Lange Zeilen wurden gefaltet, um die Lesbarkeit zu verbessern, und die Zeilen wurden zur Verweisung nummeriert. Trotzdem ist es eine reale Fehlerverfolgung (Error-Trace), die während der Entwicklung des pcvt-Konsolentreibers entstanden ist.

```
 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.
```

```
(kgdb) backtrace
#0  boot (arghowto=256) (../../i386/i386/machdep.c line 767)
#1  0xf0115159 in panic ()
#2  0xf01955bd in diediedie () (../../i386/i386/machdep.c line 698)
#3  0xf010185e in db_fncall ()
#4  0xf0101586 in db_command (-266509132, -266509516, -267381073)
#5  0xf0101711 in db_command_loop ()
#6  0xf01040a0 in db_trap ()
#7  0xf0192976 in kdb_trap (12, 0, -272630436, -266743723)
#8  0xf019d2eb in trap_fatal (...)
#9  0xf019ce60 in trap_pfault (...)
#10 0xf019cb2f in trap (...)
#11 0xf01932a1 in exception:calltrap ()
#12 0xf0191503 in cnopen (...)
#13 0xf0132c34 in spec_open ()
#14 0xf012d014 in vn_open ()
#15 0xf012a183 in open ()
#16 0xf019d4eb in syscall (...)
(kgdb) up 10
Reading in symbols for ../../i386/i386/trap.c...done.
#10 0xf019cb2f in trap (frame={tf_es = -260440048, tf_ds = 16, tf_\
edi = 3072, tf_esi = -266445372, tf_ebp = -272630356, tf_isp = -27\
2630396, tf_ebx = -266427884, tf_edx = 12, tf_ecx = -266427884, tf\
_eax = 64772224, tf_trapno = 12, tf_err = -272695296, tf_eip = -26\
6672343, tf_cs = -266469368, tf_eflags = 66066, tf_esp = 3072, tf_\
ss = -266427884}) (../../i386/i386/trap.c line 283)
283                             (void) trap_pfault(&frame, FALSE);
(kgdb) frame frame->tf_ebp frame->tf_eip
Reading in symbols for ../../i386/isa/pcvt/pcvt_drv.c...done.
#0  0xf01ae729 in pcopen (dev=3072, flag=3, mode=8192, p=(struct p\
roc *) 0xf07c0c00) (../../i386/isa/pcvt/pcvt_drv.c line 403)
403             return ((*linesw[tp->t_line].l_open)(dev, tp));
(kgdb) list
398
399             tp->t_state |= TS_CARR_ON;
400             tp->t_cflag |= CLOCAL;  /* cannot be a modem (:-) */
401
402     #if PCVT_NETBSD || (PCVT_FREEBSD >= 200)
403             return ((*linesw[tp->t_line].l_open)(dev, tp));
404     #else
405             return ((*linesw[tp->t_line].l_open)(dev, tp, flag));
406     #endif /* PCVT_NETBSD || (PCVT_FREEBSD >= 200) */
407     }
(kgdb) print tp
Reading in symbols for ../../i386/i386/cons.c...done.
$1 = (struct tty *) 0x1bae
(kgdb) print tp->t_line
$2 = 1767990816
(kgdb) up
#1  0xf0191503 in cnopen (dev=0x00000000, flag=3, mode=8192, p=(st\
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
```

Kommentare zum Skript oben:

Zeile 6

Dies ist ein Speicherauszug, der innerhalb von DDB genommen wurde (siehe unten), deswegen der Kommentar zur Panic "because you said to!" und die eher lange Stack-Ablaufverfolgung (Stack-Trace); der anfängliche Grund für das Starten von DDB war jedoch ein Seitenfehler-Trap (Page-Fault-Trap).

Zeile 20

Dies ist die Position der Funktion `trap()` in der Stack-Ablaufverfolgung.

Zeile 36

Erzwingt die Benutzung eines neuen Stack-Frames; dies ist nicht mehr notwendig. Die Stack-Frames sollen jetzt an die richtige Stelle im Speicher zeigen, selbst im Falle eines Traps. Nach einem Blick auf den Code in Zeile 403 ergibt sich mit hoher Wahrscheinlichkeit, dass entweder der Zeigerzugriff auf "tp" fehlerbehaftet oder der Array-Zugriff unerlaubt war.

Zeile 52

Der Zeiger scheint verdächtig, aber besitzt zufällig eine gültige Adresse.

Zeile 56

Jedoch zeigt er offensichtlich auf nichts und so haben wir unseren Fehler gefunden! (Für diejenigen, die nichts mit diesem speziellen Stück Code anfangen können: `tp→t_line` verweist hier auf das Zeilenverhalten (Line-Discipline) des Konsolen-Geräts, was eine ziemlich kleine Ganzzahl (Integer) sein muss.)

Falls Ihr System regelmäßig abstürzt und und Sie bald keinen freien Speicherplatz mehr zur Verfügung haben, könnte das Löschen alter vmcore-Dateien in /var/core einen beträchtlichen Betrag an Speicherplatz einsparen.

10.3. Fehlersuche in einem Speicherauszug nach einem Absturz mit DDD

Die Untersuchung eines Speicherauszugs nach einem Kernel-Absturz mit einem grafischen Debugger wie `ddd` ist auch möglich (Sie müssen den devel/ddd-Port installieren, um den `ddd` -Debugger benutzen zu können). Nehmen Sie die `-k` mit in die `ddd`-Kommandozeile auf, die Sie normalerweise benutzen würden. Zum Beispiel:

```
# ddd --debugger kgdb kernel.debug /var/crash/vmcore.0
```

Sie sollten nun in der Lage sein, die Untersuchung des Speicherauszugs nach dem Absturz unter Benutzung der grafischen Schnittstelle von `ddd` anzugehen.

10.4. Online-Kernel-Fehlersuche mit DDB

Während `kgdb` als Offline-Debugger eine Benutzerschnittstelle auf höchster Ebene bietet, gibt es einige Dinge, die es nicht kann. Die wichtigsten sind das Setzen von Breakpoints und das Abarbeiten des Kernel-Codes in Einzelschritten (Single-Stepping).

Falls Sie eine systemnahe Fehlersuche an Ihrem Kernel vorhaben, steht Ihnen ein Online-Debugger mit dem Namen DDB zur Verfügung. Er erlaubt Ihnen das Setzen von Breakpoints, die Abarbeitung von Kernel-Funktionen in Einzelschritten, das Untersuchen und Verändern von Kernel-Variablen usw. Jedoch hat er keinen Zugriff auf Kernel-Quelldateien, sondern kann nur, im Gegensatz zu `gdb`, welches auf die ganzen Informationen zur Fehlersuche zurückgreifen kann, auf globale und statische Symbole zugreifen.

Um DDB in Ihren Kernel einzubinden, fügen Sie die Optionen

```
options KDB
```

```
options DDB
```

Ihrer Konfigurationsdatei hinzu und bauen Sie den Kernel neu. (Details zur Konfiguration des FreeBSD-Kernels finden Sie im FreeBSD-Handbuch).

Falls Sie eine ältere Version des Boot-Blocks haben, könnte es sein, dass Ihre Symbole zur Fehlersuche noch nicht einmal geladen werden. Aktualisieren Sie den Boot-Block; aktuelle Versionen laden die DDB-Symbole automatisch.

Sobald Ihr Kernel mit DDB startet, gibt es mehrere Wege, um in DDB zu gelangen. Der erste und früheste Weg ist, das Boot-Flag `-d` gleich an der Boot-Eingabeaufforderung einzugeben. Der Kernel startet dann in den Debug-Modus und betritt DDB noch vor jedweder Gerätesuche. Somit können Sie Funktionen zur Gerätesuche/-bereitstellung auf Fehler untersuchen. FreeBSD-CURRENT-Benutzer müssen die sechste Option im Boot-Menü auswählen, um an eine Eingabeaufforderung zu

gelangen.

Das zweite Szenario ist der Gang in den Debugger, sobald das System schon gestartet ist. Es gibt zwei einfache Wege dies zu erreichen. Falls Sie von der Eingabeaufforderung aus in den Debugger gelangen möchten, geben Sie einfach folgenden Befehl ab:

```
# sysctl debug.kdb.enter=1
```

Um eine schnelle Panic zu erzwingen, geben Sie das folgende Kommando ein:

```
# sysctl debug.kdb.panic=1
```

Anderenfalls können Sie ein Tastenkürzel auf der Tastatur benutzen, wenn Sie an der Systemkonsole sind. Die Voreinstellung für die break-to-debugger-Sequenz ist `Ctrl` + `Alt` + `ESC`. In syscons kann diese Sequenz auf eine andere Tastenkombination gelegt werden (remap) und manche der verfügbaren Tastaturlayouts tun dies, stellen Sie also sicher, dass Sie die richtige Sequenz kennen, die benutzt werden soll. Für serielle Konsolen ist eine Option vorhanden, die die Benutzung einer Unterbrechung der seriellen Verbindung (BREAK) auf der Kommandozeile erlaubt, um in DDB zu gelangen (`options BREAK_TO_DEBUGGER` in der Kernel-Konfigurationsdatei). Dies ist jedoch nicht der Standard, da viele serielle Adapter in Verwendung sind, die grundlos eine BREAK-Bedingung erzeugen, zum Beispiel bei Ziehen des Kabels.

Die dritte Möglichkeit ist, dass jede Panic-Bedingung in DDB springt, falls der Kernel hierfür konfiguriert ist. Aus diesem Grund ist es nicht sinnvoll einen Kernel mit DDB für ein unbeaufsichtigtes System zu konfigurieren.

Um die unbeaufsichtigte Funktionsweise zu erreichen fügen Sie:

```
options KDB_UNATTENDED
```

der Kernel-Konfigurationsdatei hinzu und bauen/installieren Sie den Kernel neu.

Die DDB-Befehle ähneln grob einigen `gdb`-Befehlen. Das Erste, das Sie vermutlich tun müssen, ist einen Breakpoint zu setzen:

```
break function-name address
```

Zahlen werden standardmäßig hexadezimal angegeben, aber um sie von Symbolnamen zu unterscheiden, muss Zahlen, die mit den Buchstaben `a-f` beginnen, `0x` vorangehen (dies ist für andere Zahlen beliebig). Einfache Ausdrücke sind erlaubt, zum Beispiel: `function-name + 0x103`.

Um den Debugger zu verlassen und mit der Abarbeitung fortzufahren, geben Sie ein:

```
continue
```

Um eine Stack-Ablaufverfolgung zu erhalten, benutzen Sie:

```
trace
```

Beachten Sie, dass wenn Sie DDB mittels einer Schnelltaste betreten, der Kernel zurzeit einen Interrupt bereitstellt, sodass die Stack-Ablaufverfolgung Ihnen nicht viel nützen könnte.

Falls Sie einen Breakpoint entfernen möchten, benutzen Sie

```
del
del address-expression
```

Die erste Form wird direkt, nachdem ein Breakpoint anschlug, angenommen und entfernt den aktuellen Breakpoint. Die zweite kann jeden Breakpoint löschen, aber Sie müssen die genaue Adresse angeben; diese kann bezogen werden durch:

```
show b
```

oder:

```
show break
```

Um den Kernel in Einzelschritten auszuführen, probieren Sie:

```
s
```

Dies springt in Funktionen, aber Sie können DDB veranlassen, diese schrittweise zu verfolgen, bis die passende Rückkehranweisung (Return-Statement) erreicht ist. Nutzen Sie hierzu:

```
n
```

Dies ist nicht das gleiche wie die `next`-Anweisung von `gdb`; es ist wie `gdb`s `finish`. Mehrmaliges Drücken von `n` führt zu einer Fortsetzung.

Um Daten aus dem Speicher zu untersuchen, benutzen Sie (zum Beispiel):

```
x/wx 0xf0133fe0,40
x/hd db_symtab_space
x/bc termbuf,10
x/s stringbuf
```

für Word/Halfword/Byte-Zugriff und Hexadezimal/Dezimal/Character/String-Ausgabe. Die Zahl nach dem Komma ist der Objektzähler. Um die nächsten 0x10 Objekte anzuzeigen benutzen Sie einfach:

```
x ,10
```

Gleichermaßen benutzen Sie

```
x/ia foofunc,10
```

um die ersten 0x10 Anweisungen aus `foofunc` zu zerlegen (disassemble) und Sie zusammen mit ihrem Adressabstand (Offset) vom Anfang von `foofunc` auszugeben.

Um Speicher zu verändern benutzen Sie den Schreibbefehl:

```
w/b termbuf 0xa 0xb 0
w/w 0xf0010030 0 0
```

Die Befehlsoption (b/h/w) legt die Größe der Daten fest, die geschrieben werden sollen, der erste Ausdruck danach ist die Adresse, wohin geschrieben werden soll, und der Rest wird als Daten verarbeitet, die in aufeinander folgende Speicherstellen geschrieben werden.

Falls Sie die aktuellen Register wissen möchten, benutzen Sie:

```
show reg
```

Alternativ können Sie den Inhalt eines einzelnen Registers ausgeben mit z.B.

```
p $eax
```

und ihn bearbeiten mit:

```
set $eax new-value
```

Sollten Sie irgendeine Kernel-Funktion aus DDB heraus aufrufen wollen, geben Sie einfach ein:

```
call func(arg1, arg2, ...)
```

Der Rückgabewert wird ausgegeben.

Für eine Zusammenfassung aller laufenden Prozesse im Stil von ps(1) benutzen Sie:

```
ps
```

Nun haben Sie herausgefunden, warum Ihr Kernel fehlschlägt, und möchten neu starten. Denken Sie daran, dass, abhängig von der Schwere vorhergehender Störungen, nicht alle Teile des Kernels wie gewohnt funktionieren könnten. Führen Sie eine der folgenden Aktionen durch, um Ihr System herunterzufahren und neu zu starten:

```
panic
```

Dies wird Ihren Kernel dazu veranlassen abzustürzen, einen Speicherauszug abzulegen und neu zu starten, sodass Sie den Kernspeicherauszug später auf höherer Ebene mit `gdb` auswerten können. Diesem Befehl muss normalerweise eine weitere `continue`-Anweisung folgen.

```
call boot(0)
```

Dürfte ein guter Weg sein, um das laufende System sauber herunterzufahren, alle Festplatten mittels `sync()` zu schreiben und schließlich, in manchen Fällen, neu zu starten. Solange die Festplatten- und Dateisystemschnittstellen des Kernels nicht beschädigt sind, könnte dies ein guter Weg für ein beinahe sauberes Abschalten sein.

```
call cpu_reset()
```

Dies ist der letzte Ausweg aus der Katastrophe und kommt beinahe dem Drücken des Ausschaltknopfes gleich.

Falls Sie eine kurze Zusammenfassung aller Befehle benötigen, geben Sie einfach ein:

```
help
```

Es ist strengstens empfohlen, eine ausgedruckte Version der ddb(4)-Manualpage während der Fehlersuche neben sich liegen zu haben. Denken Sie daran, dass es schwer ist, die Online-Hilfe zu lesen, während der Ausführung des Kernels in Einzelschritten.

10.5. Online-Kernel-Fehlersuche mit GDB auf einem entfernten System

Diese Funktion wird seit FreeBSD 2.2 unterstützt und ist wirklich sehr geschickt.

GDB unterstützt *die Fehlersuche von einem entfernten System aus* bereits einige Zeit. Dies geschieht unter Benutzung eines sehr einfachen Protokolls über eine serielle Verbindung. Anders als bei den anderen, oben beschriebenen, Vorgehensweisen werden hier zwei Systeme benötigt. Das eine ist das Hostsystem, welches die Umgebung zur Fehlersuche, einschließlich aller Quellen und einer Kopie der Kernel-Binärdatei mit allen Symbolen bereitstellt, und das andere das Zielsystem,

welches einfach nur eine Kopie desselben Kernels ausführt (ohne die Informationen zur Fehlersuche).

Sie sollten den Kernel im Zweifelsfall mit `config -g` konfigurieren, `DDB` in die Konfiguration aufnehmen und den Kernel, wie sonst auch, kompilieren. Dies ergibt, aufgrund der zusätzlichen Informationen zur Fehlersuche, eine umfangreiche Binärdatei. Kopieren Sie diesen Kernel auf das Zielsystem, entfernen Sie die Symbole zur Fehlersuche mit `strip -x` und starten Sie ihn mit der `-d` -Boot-Option. Stellen Sie die serielle Verbindung zwischen dem Zielsystem, welches "flags 80" für dessen sio-Gerät gesetzt hat, und dem Hostsystem, welches die Fehlersuche übernimmt, her. Nun wechseln Sie auf dem Hostsystem in das Bauverzeichnis des Ziel-Kernels und starten `gdb`:

```
% kgdb 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)
```

Stellen Sie die entfernte Sitzung zur Fehlersuche ein mit (angenommen, der erste serielle Port ist in Verwendung):

```
(kgdb) target remote /dev/cuaa0
```

Jetzt geben Sie auf dem Zielsystem, welches noch vor Beginn der Gerätesuche in DDB gelangt ist, ein:

```
Debugger("Boot flags requested debugger")
Stopped at Debugger+0x35: movb  $0, edata+0x51bc
db> gdb
```

DDB antwortet dann mit:

```
Next trap will enter GDB remote protocol mode
```

Jedesmal wenn Sie `gdb` eingeben, wird zwischen dem lokalen DDB und entfernten GDB umgeschaltet. Um einen nächsten Trap sofort zu erzwingen, geben Sie einfach `s` (step) ein. Ihr GDB auf dem Hostsystem erhält nun die Kontrolle über den Ziel-Kernel:

```
Remote debugging using /dev/cuaa0
Debugger (msg=0xf01b0383 "Boot flags requested debugger")
    at ../../i386/i386/db_interface.c:257
(kgdb)
```

Sie können mit dieser Sitzung wie mit jeder anderen GDB-Sitzung umgehen, einschließlich vollem Zugriff auf die Quellen, Starten im gud-Modus innerhalb eines Emacs-Fensters (was Ihnen automatische Quelltext-Ausgabe in einem weiteren Emacs-Fenster bietet), usw.

10.6. Fehlersuche bei einem Konsolen-Treiber

Da Sie nunmal einen Konsolen-Treiber benötigen, um DDB zu starten, ist alles ein wenig komplizierter, sobald der Konsolen-Treiber selbst versagt. Sie erinnern sich vielleicht an die Benutzung einer seriellen Konsole (entweder durch Verändern des Boot-Blocks oder Eingabe von `-h` an der `Boot:`-Eingabeaufforderung) und das Anschließen eines Standard-Terminals an Ihren ersten seriellen Port. DDB funktioniert auf jedem konfigurierten Konsolen-Treiber, auch auf einer seriellen Konsole.

10.7. Fehlersuche bei Deadlocks

Sie erleben vielleicht mal sogenannte Deadlocks, wobei ein System aufhört, nützliche Arbeit zu machen. Um in einer solchen Situation einen hilfreichen Fehlerbericht zu liefern, benutzen Sie ddb(4), wie oben beschrieben. Hängen Sie die Ausgabe von `ps` und `trace` für verdächtige Prozesse an den Bericht an.

Falls möglich, versuchen Sie, weitere Untersuchungen anzustellen. Der Empfang der Ausgaben unten ist besonders dann nützlich, wenn Sie den Auslöser für die Blockade des Systems auf VFS-Ebene vermuten. Fügen Sie die folgenden Optionen

```
makeoptions     DEBUG=-g
    options     INVARIANTS
    options     INVARIANT_SUPPORT
    options     WITNESS
    options     DEBUG_LOCKS
    options     DEBUG_VFS_LOCKS
    options     DIAGNOSTIC
```

der Kernel-Konfigurationsdatei hinzu. Wenn die Blockade ausgelöst wird, stellen Sie, zusätzlich der Ausgabe vom `ps`-Befehl, die Informationen aus `show pcpu`, `show allpcpu`, `show locks`, `show alllocks`, `show lockedvnods` und `alltrace` bereit.

Um aussagekräftige Zurückverfolgungen von in Threads aufgeteilten Prozesse zu erhalten, benutzen Sie `thread thread-id`, um zum Thread-Stack zu wechseln und eine Zurückverfolgung mit `where` anzustellen.

10.8. Glossar der Kernel-Optionen zur Fehlersuche

Dieser Abschnitt bietet ein kurzes Glossar der zur Kompilierzeit verfügbaren Kernel-Optionen, die die Fehlersuche unterstützen:

- `options KDB`: Kompiliert das Kernel-Debugger-Framework ein. Wird von `options DDB` und `options GDB` benötigt. Kein oder nur geringer Leistungs-Overhead. Standardmäßig wird bei einer Panic

der Debugger gestartet, anstatt automatisch neu zu starten.

- `options KDB_UNATTENDED`: Setzt den Standard des `debug.debugger_on_panic`-sysctl-Werts auf 0, welcher regelt, ob der Debugger bei einer Panic gestartet wird. Solange `options KDB` nicht in den Kernel einkompiliert ist, wird bei einer Panic automatisch neu gestartet; sobald es in den Kernel einkompiliert ist, wird standardmäßig der Debugger gestartet, solange `options KDB_UNATTENDED` nicht einkompiliert ist. Falls Sie den Kernel-Debugger in den Kernel einkompiliert lassen wollen, aber möchten, dass das System neu startet, wenn Sie nicht zur Stelle sind, um den Debugger zur Diagnose zu benutzen, wählen Sie diese Option.
- `options KDB_TRACE`: Setzt den Standard des `debug.trace_on_panic`-sysctl-Werts auf 1, welcher regelt, ob der Debugger bei einer Panic automatisch eine Stack-Ablaufverfolgung ausgibt. Besonders wenn der Kernel mit `KDB_UNATTENDED` läuft, kann dies hilfreich sein, um grundlegende Informationen zur Fehlersuche auf der seriellen oder Firewire-Konsole zu erhalten, während immer noch zur Wiederherstellung neu gestartet wird.
- `options DDB`: Kompiliert die Unterstützung für den Konsolen-Debugger DDB ein. Dieser interaktive Debugger läuft auf was auch immer die aktive Konsole des Systems auf niedrigster Ebene ist, dazu gehören die Video-, serielle und Firewire-Konsole. Er bietet grundlegende, eingebaute Möglichkeiten zur Fehlersuche wie zum Beispiel das Erstellen von Stack-Ablaufverfolgungen, das Auflisten von Prozessen und Threads, das Ablegen des Lock-, VM- und Dateisystemstatus und die Verwaltung des Kernel-Speichers. DDB benötigt keine Software, die auf einem zweiten System läuft, oder die Fähigkeit, einen Kernspeicherauszug oder Kernel-Symbole zur vollen Fehlersuche zu erzeugen und bietet detaillierte Fehlerdiagnose des Kernels zur Laufzeit. Viele Fehler können allein unter Benutzung der DDB-Ausgabe untersucht werden. Diese Option hängt von `options KDB` ab.
- `options GDB`: Kompiliert die Unterstützung für den Debugger GDB ein, welcher von einem entfernten System aus über ein serielles Kabel oder Firewire agieren kann. Wenn der Debugger gestartet ist, kann GDB dazu verwendet werden, um Struktur-Inhalte einzusehen, Stack-Ablaufverfolgungen zu erzeugen, usw. Bei manchem Kernel-Status ist der Zugriff ungeschickter als mit DDB, welcher dazu in der Lage ist, nützliche Zusammenfassungen des Kernel-Status automatisch zu erstellen wie zum Beispiel das automatische Abgehen der Lock-Fehlersuche oder der Strukturen zur Kernel-Speicher-Verwaltung, und es wird ein zweites System benötigt. Auf der anderen Seite verbindet GDB Informationen aus den Kernel-Quellen mit vollständigen Symbolen zur Fehlersuche, erkennt komplette Datenstrukturdefinitionen, lokale Variablen und ist in Skripten einsetzbar. Diese Option hängt von `options KDB` ab, ist aber nicht zur Benutzung von GDB auf einem Kernel-Kernspeicherauszug nötig.
- `options BREAK_TO_DEBUGGER`, `options ALT_BREAK_TO_DEBUGGER`: Erlaubt ein Abbruch- oder Alternativ-Signal auf der Konsole, um in den Debugger zu gelangen. Falls sich das System ohne eine Panic aufhängt, ist dies ein nützlicher Weg, um den Debugger zu erreichen. Aufgrund der aktuellen Verriegelung durch den Kernel ist ein Abbruch-Signal, das auf einer seriellen Konsole erzeugt wurde, deutlich vertrauenswürdiger beim Gelangen in den Debugger und wird allgemein empfohlen. Diese Option hat kaum oder keine Auswirkung auf den Durchsatz.
- `options INVARIANTS`: Kompiliert eine große Anzahl an Aussageprüfungen und -tests (Assertion-Checks und -Tests) ein, welche ständig die Intaktheit der Kernel-Datenstrukturen und die Invarianten der Kernel-Algorithmen prüfen. Diese Tests können aufwendig sein und sind deswegen nicht von Anfang an einkompiliert, aber helfen nützliches "fail stop"-Verhalten, wobei bestimmte Gruppen nicht erwünschten Verhaltens den Debugger öffnen, bevor

Beschädigungen an Kernel-Daten auftreten, bereitzustellen, welches es einfacher macht, diese auf Fehler hin zu untersuchen. Die Tests beinhalten Säubern von Speicher und use-after-free-Prüfungen, was eine der bedeutenderen Quellen von Overhead ist. Diese Option hängt von `options INVARIANT_SUPPORT` ab.

- `options INVARIANT_SUPPORT`: Viele der in `options INVARIANTS` vorhandenen Tests benötigen veränderte Datenstrukturen und zusätzliche Kernel-Symbole, die festgelegt werden müssen.
- `options WITNESS`: Diese Option aktiviert Verfolgung und Prüfung von Lock-Anforderungen zur Laufzeit und ist als Werkzeug für die Deadlock-Diagnose von unschätzbarem Wert. WITNESS pflegt ein Diagramm mit erworbenen Lock-Anträgen nach Typ geordnet und prüft bei jedem Erwerb nach Zyklen (implizit oder explizit). Falls ein Zyklus entdeckt wird, werden eine Warnung und eine Stack-Ablaufverfolgung erzeugt und als Hinweis, dass ein möglicher Deadlock gefunden wurde, auf der Konsole ausgegeben. WITNESS wird benötigt, um die DDB-Befehle `show locks`, `show witness` und `show alllocks` benutzen zu können. Diese Debug-Option hat einen bedeutenden Leistung-Overhead, welcher ein ein wenig durch Benutzung von `options WITNESS_SKIPSPIN` gemildert werden kann. Detaillierte Dokumentation kann in witness(4) gefunden werden.
- `options WITNESS_SKIPSPIN`: Deaktiviert die Prüfung von Spinlock-Lock-Anforderungen mit WITNESS zur Laufzeit. Da Spinlocks am häufigsten im Scheduler angefordert werden und Scheduler-Ereignisse oft auftreten, kann diese Option Systeme, die mit WITNESS laufen, merklich beschleunigen. Diese Option hängt von `options WITNESS` ab.
- `options WITNESS_KDB`: Setzt den Standard des `debug.witness.kdb`-sysctl-Werts auf 1, was bewirkt, dass WITNESS den Debugger aufruft, sobald eine Lock-Anforderungsverletzung vorliegt, anstatt einfach nur eine Warnung auszugeben. Diese Option hängt von `options WITNESS` ab.
- `options SOCKBUF_DEBUG`: Führt umfassende Beschaffenheitsprüfungen in Socket-Puffern durch, was nützlich zur Fehlersuche bei Socket-Fehlern und Anzeichen für Ressourceblockaden (Race) in Protokollen und Gerätetreibern, die mit Sockets arbeiten, sein kann. Diese Option hat bedeutende Auswirkung auf die Netzwerkleistung und kann die Zeitverhältnisse bei gegenseitiger Ressourceblockade in Gerätetreibern ändern.
- `options DEBUG_VFS_LOCKS`: Verfolgt Lock-Anforderungs-Einzelheiten bei lockmgr/vnode-Locks, was die Menge der Informationen, die von `show lockdevnods` in DDB angezeigt werden, vergrößert. Diese Option hat messbare Auswirkung auf die Leistung.
- `options DEBUG_MEMGUARD`: Ein Ersatz für die Kernel-Speicher-Zuweisung durch malloc(9), die das VM-System benutzt, um Lese- und Schreibzugriffe auf zugewiesenen Speicher nach der Freigabe zu entdecken. Details können in memguard(9) gefunden werden. Diese Option hat bedeutende Auswirkung auf die Leistung, aber kann sehr nützlich bei der Fehlersuche sein, wenn Kernel-Speicher-Beschädigungen durch Fehler verursacht werden.
- `options DIAGNOSTIC`: Aktiviert zusätzliche, aufwendigere Diagnosetests analog zu `options INVARIANTS`.

Teil IV: Architekturen

Kapitel 11. x86-Assembler-Programmierung

Dieses Kapitel wurde geschrieben von G. Adam Stanislav <adam@redprince.net>.

11.1. Synopsis

Assembler-Programmierung unter UNIX® ist höchst undokumentiert. Es wird allgemein angenommen, dass niemand sie jemals benutzen will, da UNIX®-Systeme auf verschiedenen Mikroprozessoren laufen, und man deshalb aus Gründen der Portabilität alles in C schreiben sollte.

In Wirklichkeit ist die Portabilität von C größtenteils ein Mythos. Auch C-Programme müssen angepasst werden, wenn man sie von einem UNIX® auf ein anderes portiert, egal auf welchem Prozessor jedes davon läuft. Typischerweise ist ein solches Programm voller Bedingungen, die unterscheiden für welches System es kompiliert wird.

Sogar wenn wir glauben, dass jede UNIX®-Software in C, oder einer anderen High-Level-Sprache geschrieben werden sollte, brauchen wir dennoch Assembler-Programmierer: Wer sonst sollte den Abschnitt der C-Bibliothek schreiben, die auf den Kernel zugreift?

In diesem Kapitel möchte ich versuchen zu zeigen, wie man Assembler-Sprache verwenden kann, um UNIX®-Programme, besonders unter FreeBSD, zu schreiben.

Dieses Kapitel erklärt nicht die Grundlagen der Assembler-Sprache. Zu diesem Thema gibt es bereits genug Quellen (einen vollständigen Online-Kurs finden Sie in Randall Hydes Art of Assembly Language; oder falls Sie ein gedrucktes Buch bevorzugen, können Sie einen Blick auf Jeff Duntemanns Assembly Language Step-by-Step werfen). Jedenfalls sollte jeder Assembler-Programmierer nach diesem Kapitel schnell und effizient Programme für FreeBSD schreiben können.

Copyright ® 2000-2001 G. Adam Stanislav. All rights reserved.

11.2. Die Werkzeuge

11.2.1. Der Assembler

Das wichtigste Werkzeug der Assembler-Programmierung ist der Assembler, diese Software übersetzt Assembler-Sprache in Maschinencode.

Für FreeBSD stehen zwei verschiedene Assembler zur Verfügung. Der erste ist as(1), der die traditionelle UNIX®-Assembler-Sprache verwendet. Dieser ist Teil des Systems.

Der andere ist /usr/ports/devel/nasm. Dieser benutzt die Intel-Syntax und sein Vorteil ist, dass es Code fü viele Vetriebssysteme übersetzen kann. Er muss gesondert installiert werden, aber ist völlig frei.

In diesem Kapitel wird die nasm-Syntax verwendet. Einerseits weil es die meisten Assembler-Programmierer, die von anderen Systemen zu FreeBSD kommen, leichter verstehen werden. Und offen gesagt, weil es das ist, was ich gewohnt bin.

11.2.2. Der Linker

Die Ausgabe des Assemblers muss, genau wie der Code jedes Compilers, gebunden werden, um eine ausführbare Datei zu bilden.

Der Linker ld(1) ist der Standard und Teil von FreeBSD. Er funktioniert mit dem Code beider Assembler.

11.3. Systemaufrufe

11.3.1. Standard-Aufrufkonvention

Standardmäßig benutzt der FreeBSD-Kernel die C-Aufrufkonvention. Weiterhin wird, obwohl auf den Kernel durch `int 80h` zugegriffen wird, angenommen, dass das Programm eine Funktion aufruft, die `int 80h` verwendet, anstatt `int 80h` direkt aufzurufen.

Diese Konvention ist sehr praktisch und der Microsoft®-Konvention von MS-DOS® überlegen. Warum? Weil es die UNIX®-Konvention jedem Programm, egal in welcher Sprache es geschrieben ist, erlaubt auf den Kernel zuzugreifen.

Ein Assembler-Programm kann das ebenfalls. Beispielsweise könnten wir eine Datei öffnen:

```
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
```

Das ist ein sehr sauberer und portabler Programmierstil. Wenn Sie das Programm auf ein anderes UNIX® portieren, das einen anderen Interrupt oder eie andere Art der Parameterübergabe verwendet, müssen sie nur die Prozedur kernel ändern.

Aber Assembler-Programmierer lieben es Taktzyklen zu schinden. Das obige Beispiel benötigt eine `call/ret`-Kombination. Das können wir entfernen, indem wir einen weiteren Parameter mit `push` übergeben:

```
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
```

Die Konstante 5, die wir in `EAX` ablegen, identifiziert die Kernel-Funktion, die wir aufrufen. In diesem Fall ist das `open`.

11.3.2. Alternative Aufruf-Konvention

FreeBSD ist ein extrem flexibles System. Es bietet noch andere Wege, um den Kernel aufzurufen. Damit diese funktionieren muss allerdings die Linux-Emulation installiert sein.

Linux ist ein UNIX®-artiges System. Allerdings verwendet dessen Kernel die gleiche Systemaufruf-Konvention, bei der Parameter in Registern abgelegt werden, wie MS-DOS®. Genau wie bei der UNIX®-Konvention wird die Nummer der Funktion in `EAX` abgelegt. Allerdings werden die Parameter nicht auf den Stack gelegt, sondern in die Register `EBX, ECX, EDX, ESI, EDI, EBP`:

```
open:
    mov eax, 5
    mov ebx, path
    mov ecx, flags
    mov edx, mode
    int 80h
```

Diese Konvention hat einen großen Nachteil gegenüber der von UNIX®, was die Assembler-Programmierung angeht: Jedesmal, wenn Sie einen Kernel-Aufruf machen, müssen Sie die Register `push`en und sie später `pop`en. Das macht Ihren Code unförmiger und langsamer. Dennoch lässt FreeBSD ihnen die Wahl.

Wenn Sie sich für die Linux-Konvention entscheiden, müssen Sie es das System wissen lassen. Nachdem ihr Programm übersetzt und gebunden wurde, müssen Sie die ausführbare Datei kennzeichnen:

```
%
    brandelf -t Linux
    filename
```

11.3.3. Welche Konvention Sie verwenden sollten

Wenn Sie speziell für FreeBSD programmieren, sollten Sie die UNIX®-Konvention verwenden: Diese ist schneller, Sie können globale Variablen in Registern ablegen, Sie müssen die ausführbare Datei nicht kennzeichnen und Sie erzwingen nicht die Installation der Linux-Emulation auf dem Zielsystem.

Wenn Sie portablen Programmcode erzeugen wollen, der auch unter Linux funktioniert, wollen Sie den FreeBSD-Nutzern vielleicht dennoch den effizientesten Programmcode bieten, der möglich ist. Ich werde Ihnen zeigen, wie Sie das erreichen können, nachdem ich die Grundlagen erklärt habe.

11.3.4. Aufruf-Nummern

Um dem Kernel mitzuteilen welchen Dienst Sie aufrufen, legen Sie dessen Nummer in `EAX` ab. Natürlich müssen Sie dazu wissen welche Nummer die Richtige ist.

11.3.4.1. Die Datei syscalls

Die Nummer der Funktionen sind in der Datei syscalls aufgeführt. Mittels `locate syscalls` finden Sie diese in verschiedenen Formaten, die alle auf die gleiche Weise aus syscalls.master erzeugt werden.

Die Master-Datei für die UNIX®-Standard-Aufrufkonvention finden sie unter /usr/src/sys/kern/syscalls.master. Falls Sie die andere Konvention, die im Linux-Emulations-Modus implementiert ist, verwenden möchten, lesen Sie bitte /usr/src/sys/i386/linux/syscalls.master.

FreeBSD und Linux unterscheiden sich nicht nur in den Aufrufkonventionen, sie haben teilweise auch verschiedene Nummern für die gleiche Funktion.

syscalls.master beschreibt, wie der Aufruf gemacht werden muss:

```
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...
```

In der ersten Spalte steht die Nummer, die in `EAX` abgelegt werden muss.

Die Spalte ganz rechts sagt uns welche Parameter wir `push`en müssen. Die Reihenfolge ist dabei *von rechts nach links*.

Um beispielsweise eine Datei mittels `open` zu öffnen, müssen wir zuerst den `mode` auf den Stack `push`en, danach die `flags`, dann die Adresse an der der `path` gespeichert ist.

11.4. Rückgabewerte

Ein Systemaufruf wäre meistens nicht sehr nützlich, wenn er nicht irgendeinen Wert zurückgibt: Beispielsweise den Dateideskriptor einer geöffneten Datei, die Anzahl an Bytes die in einen Puffer gelesen wurde, die Systemzeit, etc.

Außerdem muss Sie das System informieren, falls ein Fehler auftritt: Wenn eine Datei nicht existiert, die Systemressourcen erschöpft sind, wir ein ungültiges Argument übergeben haben, etc.

11.4.1. Manualpages

Der herkömmliche Ort, um nach Informationen über verschiedene Systemaufrufe unter UNIX®-Systemen zu suchen, sind die Manualpages. FreeBSD beschreibt seine Systemaufrufe in Abschnitt 2, manchmal auch Abschnitt 3.

In open(2) steht beispielsweise:

Falls erfolgreich, gibt `open()` einen nicht negativen Integerwert, als Dateideskriptor bezeichnet, zurück. Es gibt `-1` im Fehlerfall zurück und setzt `errno` um den Fehler anzuzeigen.

Ein Assembler-Programmierer, der neu bei UNIX® und FreeBSD ist, wird sich sofort fragen: Wo finde ich `errno` und wie erreiche ich es?

Die Information der Manualpage bezieht sich auf C-Programme. Der Assembler-Programmierer benötigt zusätzliche Informationen.

11.4.2. Wo sind die Rückgabewerde?

Leider gilt: Es kommt darauf an... Für die meisten Systemaufrufe liegt er in `EAX`, aber nicht für alle. Eine gute Daumenregel, wenn man zum ersten Mal mit einem Systemaufruf arbeitet, ist in `EAX` nach dem Rückgabewert zu suchen. Wenn er nicht dort ist, sind weitere Untersuchungen nötig.

Mir ist ein Systemaufruf bekannt, der den Rückgabewert in `EDX` ablegt: `SYS_fork` Alle anderen mit denen ich bisher gearbeitet habe verwenden `EAX`. Allerdings habe ich noch nicht mit allen gearbeitet.

Wenn Sie die Antwort weder hier, noch irgendwo anders finden, studieren Sie den Quelltext von libc und sehen sich an, wie es mit dem Kernel zusammenarbeitet.

11.4.3. Wo ist `errno`?

Tatsächlich, nirgendwo...

`errno` ist ein Teil der Sprache C, nicht des UNIX®-Kernels. Wenn man direkt auf Kernel-Dienste zugreift, wird der Fehlercode in `EAX` zurückgegeben, das selbe Register in dem der Rückgabewert, bei einem erfolgreichen Aufruf landet.

Das macht auch Sinn. Wenn kein Fehler auftritt, gibt es keinen Fehlercode. Wenn ein Fehler auftritt, gibt es keinen Rückgabewert. Ein einziges Register kann also beides enthalten.

11.4.4. Feststellen, dass ein Fehler aufgetreten ist

Wenn Sie die Standard FreeBSD-Aufrufkonvention verwenden wird das `carry flag` gelöscht wenn der Aufruf erfolgreich ist und gesetzt wenn ein Fehler auftritt.

Wenn Sie den Linux-Emulationsmodus verwenden ist der vorzeichenbehaftete Wert in `EAX` nicht negativ, bei einem erfolgreichen Aufruf. Wenn ein Fehler auftritt ist der Wert negativ, also `-errno`.

11.5. Portablen Code erzeugen

Portabilität ist im Allgemeinen keine Stärke der Assembler-Programmierung. Dennoch ist es, besonders mit nasm, möglich Assembler-Programme für verschiedene Plattformen zu schreiben. Ich selbst habe bereits Assembler-Bibliotheken geschrieben die auf so unterschiedlichen Systemen wie Windows® und FreeBSD übersetzt werden können.

Das ist um so besser möglich, wenn Ihr Code auf zwei Plattformen laufen soll , die, obwohl sie verschieden sind, auf ähnlichen Architekturen basieren.

Beispielsweise ist FreeBSD ein UNIX®, während Linux UNIX®-artig ist. Ich habe bisher nur drei Unterschiede zwischen beiden (aus Sicht eines Assembler-Programmierers) erwähnt: Die Aufruf-Konvention, die Funktionsnummern und die Art der Übergabe von Rückgabewerten.

11.5.1. Mit Funktionsnummern umgehen

In vielen Fällen sind die Funktionsnummern die selben. Allerdings kann man auch wenn sie es nicht sind leicht mit diesem Problem umgehen: Anstatt die Nummern in Ihrem Code zu verwenden, benutzen Sie Konstanten, die Sie abhängig von der Zielarchitektur unterschiedlich definieren:

```
%ifdef  LINUX
%define SYS_execve  11
%else
%define SYS_execve  59
%endif
```

11.5.2. Umgang mit Konventionen

Sowohl die Aufrufkonvention, als auch die Rückgabewerte (das `errno` Problem) kann man mit Hilfe von Makros lösen:

```
%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. Umgang mit anderen Portabilitätsangelegenheiten

Die oben genannte Lösung funktioniert in den meisten Fällen, wenn man Code schreibt, der zwischen FreeBSD und Linux portierbar sein soll. Allerdings sind die Unterschiede bei einigen Kernel-Diensten tiefgreifender.

In diesem Fällen müssen Sie zwei verschiedene Handler für diese Systemaufrufe schreiben und bedingte Assemblierung benutzen, um diese zu übersetzen. Glücklicherweise wird der größte Teil Ihres Codes nicht den Kernel aufrufen und Sie werden deshalb nur wenige solcher bedingten Abschnitte benötigen.

11.5.4. Eine Bibliothek benutzen

Sie können Portabilitätsprobleme im Hauptteil ihres Codes komplett vermeiden, indem Sie eine Bibliothek für Systemaufrufe schreiben. Erstellen Sie eine Bibliothek für FreeBSD, eine für Linux und weitere für andere Betriebssysteme.

Schreiben Sie in ihrer Bibliothek eine gesonderte Funktion (oder Prozedur, falls Sie die traditionelle Assembler-Terminologie bevorzugen) für jeden Systemaufruf. Verwenden Sie dabei die C-Aufrufkonvention um Parameter zu übergeben, aber verwenden Sie weiterhin `EAX`, für die

Aufrufnummer. In diesem Fall kann ihre FreeBSD-Bibliothek sehr einfach sein, da viele scheinbar unterschiedliche Funktionen als Label für denselben Code implementiert sein können:

```
sys.open:
sys.close:
[etc...]
    int 80h
    ret
```

Ihre Linux-Bibliothek wird mehr verschiedene Funktionen benötigen, aber auch hier können Sie Systemaufrufe, welche die Anzahl an Parametern akzeptieren zusammenfassen:

```
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
```

Der Bibliotheks-Ansatz mag auf den ersten Blick unbequem aussehen, weil Sie eine weitere Datei erzeugen müssen von der Ihr Code abhängt. Aber er hat viele Vorteile: Zum einen müssen Sie die Bibliothek nur einmal schreiben und können sie dann in allen Ihren Programmen verwenden. Sie können sie sogar von anderen Assembler-Programmierern verwenden lassen, oder eine die von jemand anderem geschrieben wurde verwenden. Aber der vielleicht größte Vorteil ist, dass Ihr Code sogar von anderen Programmierer auf andere Systeme portiert werden kann, einfach indem man eine neue Bibliothek schreibt, völlig ohne Änderungen an Ihrem Code.

Falls Ihnen der Gedanke eine Bibliothek zu nutzen nicht gefällt, können Sie zumindest all ihre Systemaufrufe in einer gesonderten Assembler-Datei ablegen und diese mit Ihrem Hauptprogramm zusammen binden. Auch hier müssen alle, die ihr Programm portieren, nur eine neue Objekt-Datei erzeugen und an Ihr Hauptprogramm binden.

11.5.5. Eine Include-Datei verwenden

Wenn Sie ihre Software als (oder mit dem) Quelltext ausliefern, können Sie Makros definieren und in einer getrennten Datei ablegen, die Sie ihrem Code beilegen.

Porter Ihrer Software schreiben dann einfach eine neue Include-Datei. Es ist keine Bibliothek oder eine externe Objekt-Datei nötig und Ihr Code ist portabel, ohne dass man ihn editieren muss.

Das ist der Ansatz den wir in diesem Kapitel verwenden werden. Wir werden unsere Include-Datei system.inc nennen und jedesmal, wenn wir einen neuen Systemaufruf verwenden, den entsprechenden Code dort einfügen.

Wir können unsere system.inc beginnen indem wir die Standard-Dateideskriptoren deklarieren:

```
%define stdin   0
%define stdout  1
%define stderr  2
```

Als Nächstes erzeugen wir einen symbolischen Namen für jeden Systemaufruf:

```
%define SYS_nosys   0
%define SYS_exit    1
%define SYS_fork    2
%define SYS_read    3
%define SYS_write   4
; [etc...]
```

Wir fügen eine kleine, nicht globale Prozedur mit langem Namen ein, damit wir den Namen nicht aus Versehen in unserem Code wiederverwenden:

```
section .text
align 4
access.the.bsd.kernel:
    int 80h
    ret
```

Wir erzeugen ein Makro, das ein Argument erwartet, die Systemaufruf-Nummer:

```
%macro  system  1
    mov eax, %1
    call    access.the.bsd.kernel
%endmacro
```

Letztlich erzeugen wir Makros für jeden Systemaufruf. Diese Argumente erwarten keine Argumente.

```
%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...]
```

Fahren Sie fort, geben das in Ihren Editor ein und speichern es als system.inc. Wenn wir Systemaufrufe besprechen, werden wir noch Ergänzungen in dieser Datei vornehmen.

11.6. Unser erstes Programm

Jetzt sind wir bereit für unser erstes Programm, das übliche 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
```

Hier folgt die Erklärung des Programms: Zeile 1 fügt die Definitionen ein, die Makros und den Code aus system.inc.

Die Zeilen 3 bis 5 enthalten die Daten: Zeile 3 beginnt den Datenabschnitt/das Datensegment. Zeile 4 enthält die Zeichenkette "Hello, World!", gefolgt von einem Zeilenumbruch (0Ah). Zeile 5 erstellt eine Konstante, die die Länge der Zeichenkette aus Zeile 4 in Bytes enthält.

Die Zeilen 7 bis 16 enthalten den Code. Beachten Sie bitte, dass FreeBSD das Dateiformat *elf* für diese ausführbare Datei verwendet, bei dem jedes Programm mit dem Label `_start` beginnt (oder, um genau zu sein, wird dies vom Linker erwartet). Diese Label muss global sein.

Die Zeilen 10 bis 13 weisen das System an `hbytes` Bytes der Zeichenkette `hello` nach `stdout` zu schreiben.

Die Zeilen 15 und 16 weisen das System an das Programm mit dem Rückgabewert 0 zu beenden. Der Systemaufruf `SYS_exit` kehrt niemals zurück, somit endet das Programm hier.

Wenn Sie von MS-DOS®-Assembler zu UNIX® gekommen sind, sind Sie es vielleicht gewohnt direktauf die Video-Hardware zu schreiben. Unter FreeBSD müssen Sie sich darum keine Gedanken machen, ebenso bei jeder anderen Art von UNIX®. Soweit es Sie betrifft schreiben Sie in eine Datei namens stdout. Das kann der Bildschirm, oder ein telnet-Terminal, eine wirkliche Datei, oder die Eingabe eines anderen Programms sein. Es liegt beim System herauszufinden, welches davon es tatsächlich ist.

11.6.1. Den Code assemblieren

Geben Sie den Code (außer den Zeilennummern) in einen Editor ein und speichern Sie ihn in einer Datei namens hello.asm. Um es zu assemblieren benötigen Sie nasm.

11.6.1.1. nasm installieren

Wenn Sie nasm noch nicht installiert haben geben Sie folgendes ein:

```
% su
Password:your root password
# cd /usr/ports/devel/nasm
# make install
# exit
%
```

Sie können auch `make install clean` anstatt `make install` eingeben, wenn Sie den Quelltext von nasm nicht behalten möchten.

Auf jeden Fall wird FreeBSD nasm automatisch aus dem Internet herunterladen, es kompilieren und auf Ihrem System installieren.

Wenn es sich bei Ihrem System nicht um FreeBSD handelt, müssen Sie nasm von dessen Homepage herunterladen. Sie können es aber dennoch verwenden um FreeBSD code zu assemblieren.

Nun können Sie den Code assemblieren, binden und ausführen:

```
% nasm -f elf hello.asm
% ld -s -o hello hello.o
```

```
% ./hello
Hello, World!
%
```

11.7. UNIX®-Filter schreiben

Ein häufiger Typ von UNIX®-Anwendungen ist ein Filter - ein Programm, das Eingaben von stdin liest, sie verarbeitet und das Ergebnis nach stdout schreibt.

In diesem Kapitel möchten wir einen einfachen Filter entwickeln und lernen, wie wir von stdin lesen und nach stdout schreiben. Dieser Filter soll jedes Byte seiner Eingabe in eine hexadezimale Zahl gefolgt von einem Leerzeichen umwandeln.

```
%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
```

Im Datenabschnitt erzeugen wir ein Array mit Namen `hex`. Es enthält die 16 hexadezimalen Ziffern in aufsteigender Reihenfolge. Diesem Array folgt ein Puffer, den wir sowohl für die Ein- als auch für die Ausgabe verwenden. Die ersten beiden Bytes dieses Puffers werden am Anfang auf 0 gesetzt. Dorthin schreiben wir die beiden hexadezimalen Ziffern (das erste Byte ist auch die Stelle an die wir die Eingabe lesen). Das dritte Byte ist ein Leerzeichen.

Der Code-Abschnitt besteht aus vier Teilen: Das Byte lesen, es in eine hexadezimale Zahl umwandeln, das Ergebnis schreiben und letztendlich das Programm verlassen.

Um das Byte zu lesen, bitten wir das System ein Byte von stdin zu lesen und speichern es im ersten Byte von `buffer`. Das System gibt die Anzahl an Bytes, die gelesen wurden, in `EAX` zurück. Diese wird 1 sein, wenn eine Eingabe empfangen wird und 0, wenn keine Eingabedaten mehr verfügbar sind. Deshalb überprüfen wir den Wert von `EAX`. Wenn dieser 0 ist, springen wir zu `.done`, ansonsten fahren wir fort.

Zu Gunsten der Einfachheit ignorieren wir hier die Möglichkeit eines Fehlers.

Die Umwandlungsroutine in eine Hexadezimalzahl liest das Byte aus `buffer` in `EAX`, oder genaugenommen nur in `AL`, wobei die übrigen Bits von `EAX` auf null gesetzt werden. Außerdem kopieren wir das Byte nach `EDX`, da wir die oberen vier Bits (Nibble) getrennt von den unteren vier Bits umwandeln müssen. Das Ergebnis speichern wir in den ersten beiden Bytes des Puffers.

Als Nächstes bitten wir das System die drei Bytes in den Puffer zu schreiben, also die zwei hexadezimalen Ziffern und das Leerzeichen nach stdout. Danach springen wir wieder an den Anfang des Programms und verarbeiten das nächste Byte.

Wenn die gesamte Eingabe verarbeitet ist, bitten wie das System unser Programm zu beenden und null zurückzuliefern, welches traditionell die Bedeutung hat, dass unser Programm erfolgreich war.

Fahren Sie fort und speichern Sie den Code in eine Datei namens hex.asm. Geben Sie danach folgendes ein (`^D` bedeutet, dass Sie die Steuerungstaste drücken und dann `D` eingeben, während Sie Steuerung gedrückt halten):

```
% 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 %
```

Wenn Sie von MS-DOS® zu UNIX® wechseln, wundern Sie sich vielleicht, warum jede Zeile mit 0A an Stelle von 0D 0A endet. Das liegt daran, dass UNIX® nicht die CR/LF-Konvention, sondern die "new line"-Konvention verwendet, welches

> hexadezimal als 0A dargestellt wird.

Können wir das Programm verbessern? Nun, einerseits ist es etwas verwirrend, dass die Eingabe, nachdem wir eine Zeile verarbeitet haben, nicht wieder am Anfang der Zeile beginnt. Deshalb können wir unser Programm anpassen um einen Zeilenumbruch an Stelle eines Leerzeichens nach jedem 0A auszugeben:

```
%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
```

Wir haben das Leerzeichen im Register `CL` abgelegt. Das können wir bedenkenlos tun, da UNIX®-Systemaufrufe im Gegensatz zu denen von Microsoft® Windows® keine Werte von Registern ändern in denen sie keine Werte zurückliefern.

Das bedeutet, dass wir `CL` nur einmal setzen müssen. Dafür haben wir ein neues Label `.loop` eingefügt, zu dem wir an Stelle von `_start` springen, um das nächste Byte einzulesen. Außerdem haben wir das Label `.hex` eingefügt, somit können wir wahlweise ein Leerzeichen oder einen Zeilenumbruch im dritten Byte von `buffer` ablegen.

Nachdem Sie hex.asm entsprechend der Neuerungen geändert haben, geben Sie Folgendes ein:

```
% 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 %
```

Das sieht doch schon besser aus. Aber der Code ist ziemlich ineffizient! Wir führen für jeden einzelne Byte zweimal einen Systemaufruf aus (einen zum Lesen und einen um es in die Ausgabe zu schreiben).

11.8. Gepufferte Eingabe und Ausgabe

Wir können die Effizienz unseres Codes erhöhen, indem wir die Ein- und Ausgabe puffern. Wir erzeugen einen Eingabepuffer und lesen dann eine Folge von Bytes auf einmal. Danach holen wir sie Byte für Byte aus dem Puffer.

Wir erzeugen ebenfalls einen Ausgabepuffer. Darin speichern wir unsere Ausgabe bis er voll ist. Dann bitten wir den Kernel den Inhalt des Puffers nach stdout zu schreiben.

Diese Programm endet, wenn es keine weitere Eingaben gibt. Aber wir müssen den Kernel immernoch bitten den Inhalt des Ausgabepuffers ein letztes Mal nach stdout zu schreiben, denn sonst würde ein Teil der Ausgabe zwar im Ausgabepuffer landen, aber niemals ausgegeben werden. Bitte vergessen Sie das nicht, sonst fragen Sie sich später warum ein Teil Ihrer Ausgabe verschwunden ist.

```
%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
```

Als dritten Abschnitt im Quelltext haben wir `.bss`. Dieser Abschnitt wird nicht in unsere ausführbare Datei eingebunden und kann daher nicht initialisiert werden. Wir verwenden `resb` anstelle von `db`. Dieses reserviert einfach die angeforderte Menge an uninitialisiertem Speicher zu unserer Verwendung.

Wir nutzen, die Tatsache, dass das System die Register nicht verändert: Wir benutzen Register, wo wir anderenfalls globale Variablen im Abschnitt `.data` verwenden müssten. Das ist auch der Grund, warum die UNIX®-Konvention, Parameter auf dem Stack zu übergeben, der von Microsoft, hierfür Register zu verwenden, überlegen ist: Wir können Register für unsere eigenen Zwecke verwenden.

Wir verwenden EDI und ESI als Zeiger auf das nächste zu lesende oder schreibende Byte. Wir verwenden EBX und ECX, um die Anzahl der Bytes in den beiden Puffern zu zählen, damit wir wissen, wann wir die Ausgabe an das System übergeben, oder neue Eingabe vom System entgegen nehmen müssen.

Lassen Sie uns sehen, wie es funktioniert:

```
% 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 %
```

Nicht was Sie erwartet haben? Das Programm hat die Ausgabe nicht auf dem Bildschirm ausgegeben bis sie ^D gedrückt haben. Das kann man leicht zu beheben indem man drei Zeilen Code einfügt, welche die Ausgabe jedesmal schreiben, wenn wir einen Zeilenumbruch in 0A umgewandelt haben. Ich habe die betreffenden Zeilen mit > markiert (kopieren Sie die > bitte nicht mit in Ihre 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
```

Lassen Sie uns jetzt einen Blick darauf werfen, wie es funktioniert.

```
% 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 %
```

Nicht schlecht für eine 644 Byte große Binärdatei, oder?

Dieser Ansatz für gepufferte Ein- und Ausgabe enthält eine Gefahr, auf die ich im Abschnitt Die dunkle Seite des Buffering eingehen werde.

11.8.1. Ein Zeichen ungelesen machen

Das ist vielleicht ein etwas fortgeschrittenes Thema, das vor allem für Programmierer interessant ist, die mit der Theorie von Compilern vertraut sind. Wenn Sie wollen, können Sie zum nächsten Abschnitt springen und das hier vielleicht später lesen.

Unser Beispielprogramm benötigt es zwar nicht, aber etwas anspruchsvollere Filter müssen häufig vorausschauen. Mit anderen Worten, sie müssen wissen was das nächste Zeichen ist (oder sogar mehrere Zeichen). Wenn das nächste Zeichen einen bestimmten Wert hat, ist es Teil des aktuellen Tokens, ansonsten nicht.

Zum Beispiel könnten Sie den Eingabestrom für eine Text-Zeichenfolge parsen (z.B. wenn Sie einen

Compiler einer Sprache implementieren): Wenn einem Buchstaben ein anderer Buchstabe oder vielleicht eine Ziffer folgt, ist er ein Teil des Tokens, das Sie verarbeiten. Wenn ihm ein Leerzeichen folgt, oder ein anderer Wert, ist er nicht Teil des aktuellen Tokens.

Das führt uns zu einem interessanten Problem: Wie kann man ein Zeichen zurück in den Eingabestrom geben, damit es später noch einmal gelesen werden kann?

Eine mögliche Lösung ist, das Zeichen in einer Variable zu speichern und ein Flag zu setzen. Wir können `getchar` so anpassen, dass es das Flag überprüft und, wenn es gesetzt ist, das Byte aus der Variable anstatt dem Eingabepuffer liest und das Flag zurück setzt. Aber natürlich macht uns das langsamer.

Die Sprache C hat eine Funktion `ungetc()` für genau diesen Zweck. Gibt es einen schnellen Weg, diese in unserem Code zu implementieren? Ich möchte Sie bitten nach oben zu scrollen und sich die Prozedur `getchar` anzusehen und zu versuchen eine schöne und schnelle Lösung zu finden, bevor Sie den nächsten Absatz lesen. Kommen Sie danach hierher zurück und schauen sich meine Lösung an.

Der Schlüssel dazu ein Zeichen an den Eingabestrom zurückzugeben, liegt darin, wie wir das Zeichen bekommen:

Als erstes überprüfen wir, ob der Puffer leer ist, indem wir den Wert von `EBX` testen. Wenn er null ist, rufen wir die Prozedur `read` auf.

Wenn ein Zeichen bereit ist verwenden wir `lodsb`, dann verringern wir den Wert von `EBX`. Die Anweisung `lodsb` ist letztendlich identisch mit:

```
    mov al, [esi]
      inc   esi
```

Das Byte, welches wir abgerufen haben, verbleibt im Puffer bis `read` zum nächsten Mal aufgerufen wird. Wir wissen nicht wann das passiert, aber wir wissen, dass es nicht vor dem nächsten Aufruf von `getchar` passiert. Daher ist alles was wir tun müssen um das Byte in den Strom "zurückzugeben" ist den Wert von `ESI` zu verringern und den von `EBX` zu erhöhen:

```
ungetc:
      dec   esi
      inc   ebx
      ret
```

Aber seien Sie vorsichtig! Wir sind auf der sicheren Seite, solange wir immer nur ein Zeichen im Voraus lesen. Wenn wir mehrere kommende Zeichen betrachten und `ungetc` mehrmals hintereinander aufrufen, wird es meistens funktionieren, aber nicht immer (und es wird ein schwieriger Debug). Warum?

Solange `getcharread nicht aufrufen muss, befinden sich alle im Voraus gelesenen Bytes noch im Puffer und ungetc arbeitet fehlerfrei. Aber sobald getcharread` aufruft verändert sich der Inhalt des Puffers.

Wir können uns immer darauf verlassen, dass `ungetc` auf dem zuletzt mit `getchar` gelesenen Zeichen korrekt arbeitet, aber nicht auf irgendetwas, das davor gelesen wurde.

Wenn Ihr Programm mehr als ein Byte im Voraus lesen soll, haben Sie mindestens zwei Möglichkeiten:

Die einfachste Lösung ist, Ihr Programm so zu ändern, dass es immer nur ein Byte im Voraus liest, wenn das möglich ist.

Wenn Sie diese Möglichkeit nicht haben, bestimmen Sie zuerst die maximale Anzahl an Zeichen, die Ihr Programm auf einmal an den Eingabestrom zurückgeben muss. Erhöhen Sie diesen Wert leicht, nur um sicherzugehen, vorzugsweise auf ein Vielfaches von 16-damit er sich schön ausrichtet. Dann passen Sie den `.bss` Abschnitt Ihres Codes an und erzeugen einen kleinen Reserver-Puffer, direkt vor ihrem Eingabepuffer, in etwa so:

```
section .bss
      resb  16  ; or whatever the value you came up with
  ibuffer   resb    BUFSIZE
  obuffer   resb    BUFSIZE
```

Außerdem müssen Sie `ungetc` anpassen, sodass es den Wert des Bytes, das zurückgegeben werden soll, in `AL` übergibt:

```
ungetc:
      dec   esi
      inc   ebx
      mov   [esi], al
      ret
```

Mit dieser Änderung können Sie sicher `ungetc` bis zu 17 Mal hintereinander gqapaufrufen (der erste Aufruf erfolgt noch im Puffer, die anderen 16 entweder im Puffer oder in der Reserve).

11.9. Kommandozeilenparameter

Unser hex-Programm wird nützlicher, wenn es die Dateinamen der Ein- und Ausgabedatei über die Kommandozeile einlesen kann, d.h., wenn es Kommandozeilenparameter verarbeiten kann. Aber… Wo sind die?

Bevor ein UNIX®-System ein Programm ausführt, legt es einige Daten auf dem Stack ab (`push`) und springt dann an das `_start`-Label des Programms. Ja, ich sagte springen, nicht aufrufen. Das bedeutet, dass auf die Daten zugegriffen werden kann, indem `[esp+offset]` ausgelesen wird oder die Daten einfach vom Stack genommen werden (`pop`).

Der Wert ganz oben auf dem Stack enthält die Zahl der Kommandozeilenparameter. Er wird traditionell `argc` wie "argument count" genannt.

Die Kommandozeilenparameter folgen einander, alle `argc`. Von diesen wird üblicherweise als `argv`

wie "argument value(s)" gesprochen. So erhalten wir `argv[0]`, `argv[1]`, ⋯ und `argv[argc-1]`. Dies sind nicht die eigentlichen Parameter, sondern Zeiger (Pointer) auf diese, d.h., Speicheradressen der tatsächlichen Parameter. Die Parameter selbst sind durch NULL beendete Zeichenketten.

Der `argv`-Liste folgt ein NULL-Zeiger, was einfach eine 0 ist. Es gibt noch mehr, aber dies ist erst einmal genug für unsere Zwecke.

Falls Sie von der MS-DOS®-Programmierumgebung kommen, ist der größte Unterschied die Tatsache, dass jeder Parameter eine separate Zeichenkette ist. Der zweite Unterschied ist, dass es praktisch keine Grenze gibt, wie viele Parameter vorhanden sein können.

Ausgerüstet mit diesen Kenntnissen, sind wir beinahe bereit für eine weitere Version von hex.asm. Zuerst müssen wir jedoch noch ein paar Zeilen zu system.inc hinzufügen:

Erstens benötigen wir zwei neue Einträge in unserer Liste mit den Systemaufrufnummern:

```
%define SYS_open    5
%define SYS_close   6
```

Zweitens fügen wir zwei neue Makros am Ende der Datei ein:

```
%macro  sys.open    0
    system  SYS_open
%endmacro

%macro  sys.close   0
    system  SYS_close
%endmacro
```

Und hier ist schließlich unser veränderter Quelltext:

```
%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 unserem `.data`-Abschnitt befinden sich nun die zwei neuen Variablen `fd.in` und `fd.out`. Hier legen wir die Dateideskriptoren der Ein- und Ausgabedatei ab.

Im `.text`-Abschnitt haben wir die Verweise auf `stdin` und `stdout` durch `[fd.in]` und `[fd.out]` ersetzt.

Der `.text`-Abschnitt beginnt nun mit einer einfachen Fehlerbehandlung, welche nur das Programm mit einem Rückgabewert von 1 beendet. Die Fehlerbehandlung befindet sich vor `_start`, sodass wir in geringer Entfernung von der Stelle sind, an der der Fehler auftritt.

Selbstverständlich beginnt die Programmausführung immer noch bei `_start`. Zuerst entfernen wir `argc` und `argv[0]` vom Stack: Sie sind für uns nicht von Interesse (sprich, in diesem Programm).

Wir nehmen `argv[1]` vom Stack und legen es in `ECX` ab. Dieses Register ist besonders für Zeiger geeignet, da wir mit `jecxz` NULL-Zeiger verarbeiten können. Falls `argv[1]` nicht NULL ist, versuchen wir, die Datei zu öffnen, die der erste Parameter festlegt. Andernfalls fahren wir mit dem Programm fort wie vorher: Lesen von `stdin` und Schreiben nach `stdout`. Falls wir die Eingabedatei nicht öffnen können (z.B. sie ist nicht vorhanden), springen wir zur Fehlerbehandlung und beenden das Programm.

Falls es keine Probleme gibt, sehen wir nun nach dem zweiten Parameter. Falls er vorhanden ist, öffnen wir die Ausgabedatei. Andernfalls schreiben wir die Ausgabe nach `stdout`. Falls wir die Ausgabedatei nicht öffnen können (z.B. sie ist zwar vorhanden, aber wir haben keine Schreibberechtigung), springen wir auch wieder in die Fehlerbehandlung.

Der Rest des Codes ist derselbe wie vorher, außer dem Schließen der Ein- und Ausgabedatei vor dem Verlassen des Programms und, wie bereits erwähnt, die Benutzung von `[fd.in]` und `[fd.out]`.

Unsere Binärdatei ist nun kolossale 768 Bytes groß.

Können wir das Programm immer noch verbessern? Natürlich! Jedes Programm kann verbessert werden. Hier finden sich einige Ideen, was wir tun könnten:

- Die Fehlerbehandlung eine Warnung auf `stderr` ausgeben lassen.
- Den `Lese`- und `Schreib`funkionen eine Fehlerbehandlung hinzufügen.
- Schließen von `stdin`, sobald wir eine Eingabedatei öffnen, von `stdout`, sobald wir eine Ausgabedatei öffnen.
- Hinzufügen von Kommandozeilenschaltern wie zum Beispiel -i und -o, sodass wir die Ein- und Ausgabedatei in irgendeiner Reihenfolge angeben oder vielleicht von `stdin` lesen und in eine Datei schreiben können.
- Ausgeben einer Gebrauchsanweisung, falls die Kommandozeilenparameter fehlerhaft sind.

Ich beabsichtige, diese Verbesserungen dem Leser als Übung zu hinterlassen: Sie wissen bereits alles, das Sie wissen müssen, um die Verbesserungen durchzuführen.

11.10. Die UNIX®-Umgebung

Ein entscheidendes Konzept hinter UNIX® ist die Umgebung, die durch *Umgebungsvariablen* festgelegt wird. Manche werden vom System gesetzt, andere von Ihnen und wieder andere von der shell oder irgendeinem Programm, das ein anderes lädt.

11.10.1. Umgebungsvariablen herausfinden

Ich sagte vorher, dass wenn ein Programm mit der Ausführung beginnt, der Stack `argc` gefolgt vom durch NULL beendeten `argv`-Array und etwas Anderem enthält. Das "etwas Andere" ist die *Umgebung* oder, um genauer zu sein, ein durch NULL beendetes Array von Zeigern auf *Umgebungsvariablen*. Davon wird oft als `env` gesprochen.

Der Aufbau von `env` entspricht dem von `argv`, eine Liste von Speicheradressen gefolgt von NULL (0). In diesem Fall gibt es kein `"envc"`-wir finden das Ende heraus, indem wir nach dem letzten NULL suchen.

Die Variablen liegen normalerweise in der Form `name=value` vor, aber manchmal kann der `=value` -Teil fehlen. Wir müssen diese Möglichkeit in Betracht ziehen.

11.10.2. webvars

Ich könnte Ihnen einfach etwas Code zeigen, der die Umgebung in der Art vom UNIX®-Befehl env ausgibt. Aber ich dachte, dass es interessanter sei, ein einfaches CGI-Werkzeug in Assembler zu schreiben.

11.10.2.1. CGI: Ein kurzer Überblick

Ich habe eine detaillierte CGI-Anleitung auf meiner Webseite, aber hier ist ein sehr kurzer Überblick über CGI:

- Der Webserver kommuniziert mit dem CGI-Programm, indem er *Umgebungsvariablen* setzt.
- Das CGI-Programm schreibt seine Ausgabe auf stdout. Der Webserver liest von da.
- Die Ausgabe muss mit einem HTTP-Kopfteil gefolgt von zwei Leerzeilen beginnen.
- Das Programm gibt dann den HTML-Code oder was für einen Datentyp es auch immer verarbeitet aus. *

> Während bestimmte *Umgebungsvariablen* Standardnamen benutzen, unterscheiden sich andere, abhängig vom Webserver. Dies macht webvars zu einem recht nützlichen Werkzeug.

11.10.2.2. Der Code

Unser webvars-Programm muss also den HTTP-Kopfteil gefolgt von etwas HTML-Auszeichnung versenden. Dann muss es die *Umgebungsvariablen* eine nach der anderen auslesen und sie als Teil der HTML-Seite versenden.

Nun der Code. Ich habe Kommentare und Erklärungen direkt in den Code eingefügt:

```
;;;;;;; webvars.asm ;;;;;;
```

```
;
; 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
```

Dieser Code erzeugt eine 1.396-Byte große Binärdatei. Das meiste davon sind Daten, d.h., die HTML-Auszeichnung, die wir versenden müssen.

Assemblieren Sie es wie immer:

```
% nasm -f elf webvars.asm
% ld -s -o webvars webvars.o
```

Um es zu benutzen, müssen Sie webvars auf Ihren Webserver hochladen. Abhängig von Ihrer Webserver-Konfiguration, müssen Sie es vielleicht in einem speziellen cgi-bin-Verzeichnis ablegen oder es mit einer .cgi-Dateierweiterung versehen.

Schließlich benötigen Sie Ihren Webbrowser, um sich die Ausgabe anzusehen. Um die Ausgabe auf meinem Webserver zu sehen, gehen Sie bitte auf http://www.int80h.org/webvars/. Falls Sie neugierig sind, welche zusätzlichen Variablen in einem passwortgeschützten Webverzeichnis vorhanden sind, gehen Sie auf http://www.int80h.org/private/ unter Benutzung des Benutzernamens `asm` und des Passworts `programmer`.

11.11. Arbeiten mit Dateien

Wir haben bereits einfache Arbeiten mit Dateien gemacht: Wir wissen wie wir sie öffnen und schliessen, oder wie man sie mit Hilfe von Buffern liest und schreibt. Aber UNIX® bietet viel mehr Funktionalität wenn es um Dateien geht. Wir werden einige von ihnen in dieser Sektion untersuchen und dann mit einem netten Datei Konvertierungs Werkzeug abschliessen.

In der Tat, Lasst uns am Ende beginnen, also mit dem Datei Konvertierungs Werkzeug. Es macht Programmieren immer einfacher, wenn wir bereits am Anfang wissen was das End Produkt bezwecken soll.

Eines der ersten Programme die ich für UNIX® schrieb war tuc, ein Text-Zu-UNIX® Datei Konvertierer. Es konvertiert eine Text Datei von einem anderen Betriebssystem zu einer UNIX® Text Datei. Mit anderen Worten, es ändert die verschiedenen Arten von Zeilen Begrenzungen zu der Zeilen Begrenzungs Konvention von UNIX®. Es speichert die Ausgabe in einer anderen Datei. Optional konvertiert es eine UNIX® Text Datei zu einer DOS Text Datei.

Ich habe tuc sehr oft benutzt, aber nur von irgendeinem anderen OS nach UNIX® zu konvertieren, niemals anders herum. Ich habe mir immer gewünscht das die Datei einfach überschrieben wird anstatt das ich die Ausgabe in eine andere Datei senden muss. Meistens, habe ich diesen Befehl verwendet:

```
% tuc myfile tempfile
% mv tempfile myfile
```

Es wäre schö ein ftuc zu haben, also, *fast tuc*, und es so zu benutzen:

```
% ftuc myfile
```

In diesem Kapitel werden wir dann, ftuc in Assembler schreiben (das Original tuc ist in C), und verschiedene Datei-Orientierte Kernel Dienste in dem Prozess studieren.

Auf erste Sicht, ist so eine Datei Konvertierung sehr simpel: Alles was du zu tun hast, ist die Wagenrückläufe zu entfernen, richtig?

Wenn du mit ja geantwortet hast, denk nochmal darüber nach: Dieses Vorgehen wird die meiste Zeit funktionieren (zumindest mit MSDOS Text Dateien), aber gelegentlich fehlschlagen.

Das Problem ist das nicht alle UNIX® Text Dateien ihre Zeilen mit einer Wagen Rücklauf / Zeilenvorschub Sequenz beenden. Manche benutzen Wagenrücklauf ohne Zeilenvorschub. Andere kombinieren mehrere leere Zeilen in einen einzigen Wagenrücklauf gefolgt von mehreren Zeilenvorschüben. Und so weiter.

Ein Text Datei Konvertierer muss dann also in der Lage sein mit allen möglichen Zeilenenden umzugehen:

- Wagenrücklauf / Zeilenvorschub
- Wagenrücklauf
- Zeilenvorschub / Wagenrücklauf
- Zeilenvorschub

Es sollte außerdem in der Lage sein mit Dateien umzugehen die irgendeine Art von Kombination der oben stehenden Möglichkeiten verwendet. (z.B., Wagenrücklauf gefolgt von mehreren Zeilenvorschüben).

11.11.1. Endlicher Zustandsautomat

Das Problem wird einfach gelöst in dem man eine Technik benutzt die sich *Endlicher Zustandsautomat* nennt, ursprünglich wurde sie von den Designern digitaler elektronischer Schaltkreise entwickelt. Eine *Endlicher Zustandsautomat* ist ein digitaler Schaltkreis dessen Ausgabe nicht nur von der Eingabe abhängig ist sondern auch von der vorherigen Eingabe, d.h., von seinem Status. Der Mikroprozessor ist ein Beispiel für einen *Endlichen Zustandsautomaten*: Unser Assembler Sprach Code wird zu Maschinensprache übersetzt in der manche Assembler Sprach Codes ein einzelnes Byte produzieren, während andere mehrere Bytes produzieren. Da der Microprozessor die Bytes einzeln aus dem Speicher liest, ändern manche nur seinen Status anstatt eine Ausgabe zu produzieren. Wenn alle Bytes eines OP Codes gelesen wurden, produziert der Mikroprozessor eine Ausgabe, oder ändert den Wert eines Registers, etc.

Aus diesem Grund, ist jede Software eigentlich nur eine Sequenz von Status Anweisungen für den Mikroprozessor. Dennoch, ist das Konzept eines *Endlichen Zustandsautomaten* auch im Software Design sehr hilfreich.

Unser Text Datei Konvertierer kann als *Endlicher Zustandsautomat* mit 3 möglichen Stati desgined werden. Wir könnten diese von 0-2 benennen, aber es wird uns das Leben leichter machen wenn

wir ihnen symbolische Namen geben:

- ordinary
- cr
- lf

Unser Programm wird in dem ordinary Status starten. Während dieses Status, hängt die Aktion des Programms von seiner Eingabe wie folgt ab:

- Wenn die Eingabe etwas anderes als ein Wagenrücklauf oder einem Zeilenvorschub ist, wird die Eingabe einfach nur an die Ausgabe geschickt. Der Status bleibt unverändert.
- Wenn die Eingabe ein Wagenrücklauf ist, wird der Status auf cr gesetzt. Die Eingabe wird dann verworfen, d.h., es entsteht keine Ausgabe.
- Wenn die Eingabe ein Zeilenvorschub ist, wird der Status auf lf gesetzt. Die Eingabe wird dann verworfen.

Wann immer wir in dem cr Status sind, ist das weil die letzte Eingabe ein Wagenrücklauf war, welcher nicht verarbeitet wurde. Was unsere Software in diesem Status macht hängt von der aktuellen Eingabe ab:

- Wenn die Eingabe irgendetwas anderes als ein Wagenrücklauf oder ein Zeilenvorschub ist, dann gib einen Zeilenvorschub aus, dann gib die Eingabe aus und dann ändere den Status zu ordinary.
- Wenn die Eingabe ein Wagenrücklauf ist, haben wir zwei (oder mehr) Wagenrückläufe in einer Reihe. Wir verwerfen die Eingabe, wir geben einen Zeilenvorschub aus und lassen den Status unverändert.
- Wenn die Eingabe ein Zeilenvorschub ist, geben wir den Zeilenvorschub aus und ändern den Status zu ordinary. Achte darauf, dass das nicht das gleiche wie in dem Fall oben drüber ist - würden wir versuchen beide zu kombinieren, würden wir zwei Zeilenvorschübe anstatt einen ausgeben.

Letztendlich, sind wir in dem lf Status nachdem wir einen Zeilenvorschub empfangen haben der nicht nach einem Wagenrücklauf kam. Das wird passieren wenn unsere Datei bereits im UNIX® Format ist, oder jedesmal wenn mehrere Zeilen in einer Reihe durch einen einzigen Wagenrücklauf gefolgt von mehreren Zeilenvorschüben ausgedrückt wird, oder wenn die Zeile mit einer Zeilenvorschub / Wagenrücklauf Sequenz endet. Wir sollten mit unserer Eingabe in diesem Status folgendermaßen umgehen:

- Wenn die Eingabe irgendetwas anderes als ein Wagenrücklauf oder ein Zeilenvorschub ist, geben wir einen Zeilenvorschub aus, geben dann die Eingabe aus und ändern dann den Status zu ordinary. Das ist exakt die gleiche Aktion wie in dem cr Status nach dem Empfangen der selben Eingabe.
- Wenn die Eingabe ein Wagenrücklauf ist, verwerfen wir die Eingabe, geben einen Zeilenvorschub aus und ändern dann den Status zu ordinary.
- Wenn die Eingabe ein Zeilenvorschub ist, geben wir den Zeilenvorschub aus und lassen den Status unverändert.

11.11.1.1. Der Endgültige Status

Der obige *Endliche Zustandsautomat* funktioniert für die gesamte Datei, aber lässt die Möglichkeit das die letzte Zeile ignoriert wird. Das wird jedesmal passieren wenn die Datei mit einem einzigen Wagenrücklauf oder einem einzigen Zeilenvorschub endet. Daran habe ich nicht gedacht als ich tuc schrieb, nur um festzustellen, daß das letzte Zeilenende gelegentlich weggelassen wird.

Das Problem wird einfach dadurch gelöst, indem man den Status überprüft nachdem die gesamte Datei verarbeitet wurde. Wenn der Status nicht ordinary ist, müssen wir nur den letzten Zeilenvorschub ausgeben.

Nachdem wir unseren Algorithmus nun als einen *Endlichen Zustandsautomaten* formuliert haben, könnten wir einfach einen festgeschalteten digitalen elektronischen Schaltkreis (einen "Chip") designen, der die Umwandlung für uns übernimmt. Natürlich wäre das sehr viel teurer, als ein Assembler Programm zu schreiben.

11.11.1.2. Der Ausgabe Zähler

Weil unser Datei Konvertierungs Programm möglicherweise zwei Zeichen zu einem kombiniert, müssen wir einen Ausgabe Zähler verwenden. Wir initialisieren den Zähler zu 0 und erhöhen ihn jedes mal wenn wir ein Zeichen an die Ausgabe schicken. Am Ende des Programms, wird der Zähler uns sagen auf welche Grösse wir die Datei setzen müssen.

11.11.2. Implementieren von EZ als Software

Der schwerste Teil beim arbeiten mit einer *Endlichen Zustandsmaschine* ist das analysieren des Problems und dem ausdrücken als eine *Endliche Zustandsmaschine*. That geschafft, schreibt sich die Software fast wie von selbst.

In eine höheren Sprache, wie etwa C, gibt es mehrere Hauptansätze. Einer wäre ein `switch` Angabe zu verwenden die auswählt welche Funktion genutzt werden soll. Zum Beispiel,

```
    switch (state) {
    default:
    case REGULAR:
        regular(inputchar);
        break;
    case CR:
        cr(inputchar);
        break;
    case LF:
        lf(inputchar);
        break;
    }
```

Ein anderer Ansatz ist es ein Array von Funktions Zeigern zu benutzen, etwa wie folgt:

```
    (output[state])(inputchar);
```

Noch ein anderer ist es aus `state` einen Funktions Zeiger zu machen und ihn zu der entsprechenden Funktion zeigen zu lassen:

```
    (*state)(inputchar);
```

Das ist der Ansatz den wir in unserem Programm verwenden werden, weil es in Assembler sehr einfach und schnell geht. Wir werden einfach die Adresse der Prozedur in `EBX` speichern und dann einfach das ausgeben:

```
    call    ebx
```

Das ist wahrscheinlich schneller als die Adresse im Code zu hardcoden weil der Mikroprozessor die Adresse nicht aus dem Speicher lesen muss-es ist bereits in einer der Register gespeichert. Ich sagte *wahrscheinlich* weil durch das Cachen neuerer Mikroprozessoren beide Varianten in etwa gleich schnell sind.

11.11.3. Speicher abgebildete Dateien

Weil unser Programm nur mit einzelnen Dateien funktioniert, können wir nicht den Ansatz verwedenden der zuvor funktioniert hat, d.h., von einer Eingabe Datei zu lesen und in eine Ausgabe Datei zu schreiben.

UNIX® erlaubt es uns eine Datei, oder einen Bereich einer Datei, in den Speicher abzubilden. Um das zu tun, müssen wir zuerst eine Datei mit den entsprechenden Lese/Schreib Flags öffnen. Dann benutzen wir den `mmap` system call um sie in den Speicher abzubilden. Ein Vorteil von `mmap` ist, das es automatisch mit virtuellem Speicher arbeitet: Wir können mehr von der Datei im Speicher abbilden als wir überhaupt physikalischen Speicher zur Verfügung haben, noch immer haben wir aber durch normale OP Codes wie `mov`, `lods`, und `stos` Zugriff darauf. Egal welche Änderungen wir an dem Speicherabbild der Datei vornehmen, sie werden vom System in die Datei geschrieben. Wir müssen die Datei nicht offen lassen: So lange sie abgebildet bleibt, können wir von ihr lesen und in sie schreiben.

Ein 32-bit Intel Mikroprozessor kann auf bis zu vier Gigabyte Speicher zugreifen - physisch oder virtuell. Das FreeBSD System erlaubt es uns bis zu der Hälfte für die Datei Abbildung zu verwenden.

Zur Vereinfachung, werden wir in diesem Tutorial nur Dateien konvertieren die in ihrere Gesamtheit im Speicher abgebildet werden können. Es gibt wahrscheinlich nicht all zu viele Text Dateien die eine Grösse von zwei Gigabyte überschreiben. Falls unser Programm doch auf eine trifft, wird es einfach eine Meldung anzeigen mit dem Vorschlag das originale tuc statt dessen zu verwenden.

Wenn du deine Kopie von syscalls.master überprüfst, wirst du zwei verschiedene Systemaufrufe finden die sich `mmap` nennen. Das kommt von der Entwicklung von UNIX®: Es gab das traditionelle

BSD`mmap`, Systemaufruf 71. Dieses wurde durch das POSIX® `mmap` ersetzt, Systemaufruf 197. Das FreeBSD System unterstützt beide, weil ältere Programme mit der originalen BSD Version geschrieben wurden. Da neue Software die POSIX® Version nutzt, werden wir diese auch verwenden.

Die syscalls.master Datei zeigt die POSIX® Version wie folgt:

```
197 STD BSD { caddr_t mmap(caddr_t addr, size_t len, int prot, \
                int flags, int fd, long pad, off_t pos); }
```

Das weicht etwas von dem ab was mmap(2) sagt. Das ist weil mmap(2) die C Version beschreibt.

Der Unterschiede liegt in dem `long pad` Argument, welches in der C Version nicht vorhanden ist. Wie auch immer, der FreeBSD Systemaufruf fügt einen 32-bit Block ein nachdem es ein 64-Bit Argument auf den Stack gepusht hat. In diesem Fall, ist `off_t` ein 64-Bit Wert.

Wenn wir fertig sind mit dem Arbeiten einer im Speicher abgebildeten Datei, entfernen wir das Speicherabbild mit dem `munmap` Systemaufruf:

Für eine detailliert Behandlung von `mmap`, sieh in W. Richard Stevens' Unix Network Programming, Volume 2, Chapter 12 nach.

11.11.4. Feststellen der Datei Grösse

Weil wir `mmap` sagen müssen wie viele Bytes von Datei wir im Speicher abbilden wollen und wir außerdem die gesamte Datei abbilden wollen, müssen wir die Grösse der Datei feststellen.

Wir können den `fstat` Systemaufruf verwenden um alle Informationen über eine geöffnete Datei zu erhalten die uns das System geben kann. Das beinhaltet die Datei Grösse.

Und wieder, zeigt uns syscalls.master zwei Versionen von `fstat`, eine traditionelle (Systemaufruf 62), und eine POSIX® (Systemaufruf 189) Variante. Natürlich, verwenden wir die POSIX® Version:

```
189 STD POSIX   { int fstat(int fd, struct stat *sb); }
```

Das ist ein sehr unkomplizierter Aufruf: Wir übergeben ihm die Adresse einer `stat` Structure und den Deskriptor einer geöffneten Datei. Es wird den Inhalt der `stat` Struktur ausfüllen.

Ich muss allerdings sagen, das ich versucht habe die `stat` Struktur in dem `.bss` Bereich zu deklarieren, und `fstat` mochte es nicht: Es setzte das Carry Flag welches einen Fehler anzeigt. Nachdem ich den Code veränderte so dass er die Struktur auf dem Stack anlegt, hat alles gut funktioniert.

11.11.5. Ändern der Dateigrösse

Dadurch das unser Programm Wagenrücklauf/Zeilenvorschub-Sequenzen in einfache Zeilenvorschübe zusammenfassen könnte, könnte unsere Ausgabe kleiner sein als unsere Eingabe.

Und da wir die Ausgabe in dieselbe Datei um, aus der wir unsere Eingabe erhalten, müssen wir eventuell die Dateigrösse anpassen.

Der Systemaufruf `ftruncate` erlaubt uns, dies zu tun. Abgesehen von dem etwas unglücklich gewählten Namen `ftruncate` können wir mit dieser Funktion eine Datei vergrössern, oder verkleinern.

Und ja, wir werden zwei Versionen von `ftruncate` in syscalls.master finden, eine ältere (130) und eine neuere (201). Wir werden die neuere Version verwenden:

```
201 STD BSD { int ftruncate(int fd, int pad, off_t length); }
```

Beachten Sie bitte, dass hier wieder `int pad` verwendet wird.

11.11.6. ftuc

Wir wissen jetzt alles nötige, um ftuc zu schreiben. Wir beginnen, indem wir ein paar neue Zeilen der Datei system.inc hinzufügen. Als erstes definieren wir irgendwo am Anfang der Datei einige Konstanten und Strukturen:

```
;;;;;;; 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
```

Wir definieren die neuen Systemaufrufe:

```
%define SYS_mmap    197
%define SYS_munmap  73
%define SYS_fstat   189
%define SYS_ftruncate   201
```

Wir fügen die Makros hinzu:

```
%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
```

Und hier ist unser 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
```

Verwenden Sie dieses Programm nicht mit Dateien, die sich auf Datenträgern befinden, welche mit MS-DOS® oder Windows® formatiert wurden. Anscheinend gibt es im Code von FreeBSD einen subtilen Bug, wenn `mmap` auf solchen Datenträgern verwendet wird: Wenn die Datei eine bestimmte Grösse überschreitet, füllt `mmap` den Speicher mit lauter Nullen, und überschreibt damit anschliessend den Dateiinhalt.

11.12. One-Pointed Mind

Als ein Zen-Schüler liebe ich die Idee eines fokussierten Bewußtseins: Tu nur ein Ding zur gleichen Zeit, aber mache es richtig.

Das ist ziemlich genau die gleiche Idee, welche UNIX® richtig funktionieren lässt. Während eine typische Windows®-Applikation versucht alles Vorstellbare zu tun (und daher mit Fehler durchsetzt ist), versucht eine UNIX®-Applikation nur eine Funktion zu erfüllen und das gut.

Der typische UNIX®-Nutzer stellt sich sein eigenes System durch Shell-Skripte zusammen, die er selbst schreibt, und welche die Vorteile bestehender Applikationen dadurch kombinieren, indem sie die Ausgabe eines Programmes als Eingabe in ein anderes Programm durch eine Pipe übergeben.

Wenn Sie ihre eigene UNIX®-Software schreiben, ist es generell eine gute Idee zu betrachten, welcher Teil der Problemlösung durch bestehende Programme bewerkstelligt werden kann. Man schreibt nur die Programme selbst, für die keine vorhandene Lösung existiert.

11.12.1. CSV

Ich will dieses Prinzip an einem besonderen Beispiel aus der realen Welt demonstrieren, mit dem ich kürzlich konfrontiert wurde:

Ich mußte jeweils das elfte Feld von jedem Datensatz aus einer Datenbank extrahieren, die ich von einer Webseite heruntergeladen hatte. Die Datenbank war eine CSV-Datei, d.h. eine Liste von *Komma-getrennten Werten*. Dies ist ein ziemlich gewöhnliches Format für den Code-Austausch zwischen Menschen, die eine unterschiedliche Datenbank-Software nutzen.

Die erste Zeile der Datei enthält eine Liste der Felder durch Kommata getrennt. Der Rest der Datei enthält die einzelnen Datensätze mit durch Kommata getrennten Werten in jeder Zeile.

Ich versuchte awk unter Nutzung des Kommas als Trenner. Da aber einige Zeilen durch in Bindestriche gesetzte Kommata getrennt waren, extrahierte awk das falsche Feld aus diesen Zeilen.

Daher mußte ich meine eigene Software schreiben, um das elfte Feld aus der CSV-Datei auszulesen. Aber durch Anwendung der UNIX®-Philosophie mußte ich nur einen einfachen Filter schreiben, das Folgende tat:

- Entferne die erste Zeile aus der Datei.
- Ändere alle Kommata ohne Anführungszeichen in einen anderen Buchstaben.
- Entferne alle Anführungszeichen.

Streng genommen könnte ich sed benutzen, um die erste Zeile der Datei zu entfernen, aber das zu Bewerkstelligen war in meinem Programm sehr einfach, also entschloss ich mich dazu und reduzierte dadurch die Größe der Pipeline.

Unter Berücksichtigung aller Faktoren kostete mich das Schreiben dieses Programmes ca. 20 Minuten. Das Schreiben eines Programmes, welches jeweils das elfte Feld aus einer CSV-Datei extrahiert hätte wesentlich länger gedauert und ich hätte es nicht wiederverwenden können, um ein anderes Feld aus irgendeiner anderen Datenbank zu extrahieren.

Diesmal entschied ich mich dazu, etwas mehr Arbeit zu investieren, als man normalerweise für ein typisches Tutorial verwenden würde:

- Es parst die Kommandozeilen nach Optionen.
- Es zeigt die richtige Nutzung an, falls es ein falsches Argument findet.
- Es gibt vernünftige Fehlermeldungen aus.

Hier ist ein Beispiel für seine Nutzung:

```
Usage: csv [-t<delim>] [-c<comma>] [-p] [-o <outfile>] [-i <infile>]
```

Alle Parameter sind optional und können in beliebiger Reihenfolge auftauchen.

Der -t-Parameter legt fest, was zu die Kommata zu ersetzen sind. Der tab ist die Vorgabe hierfür. Zum Beispiel wird -t; alle unquotierten Kommata mit Semikolon ersetzen.

Ich brauche die -c-Option nicht, aber sie könnte zukünftig nützlich sein. Sie ermöglicht mir festzulegen, daß ich einen anderen Buchstaben als das Kommata mit etwas anderem ersetzen möchte. Zum Beispiel wird der Parameter -c@ alle @-Zeichen ersetzen (nützlich, falls man eine Liste von Email-Adressen in Nutzername und Domain aufsplitten will).

Die -p-Option erhält die erste Zeile, d.h. die erste Zeile der Datei wird nicht gelöscht. Als Vorgabe löschen wir die erste Zeile, weil die CSV-Datei in der ersten Zeile keine Daten, sondern Feldbeschreibungen enthält.

Die Parameter -i- und -o-Optionen erlauben es mir, die Ausgabe- und Eingabedateien festzulegen. Vorgabe sind stdin und stdout, also ist es ein regulärer UNIX®-Filter.

Ich habe sichergestellt, daß sowohl -i filename und -ifilename akzeptiert werden. Genauso habe ich dafür Sorge getragen, daß sowohl Eingabe- als auch Ausgabedateien festgelegt werden können.

Um das elfte Feld jeden Datensatzes zu erhalten kann ich nun folgendes eingeben:

```
% csv '-t;' data.csv | awk '-F;' '{print $11}'
```

Der Code speichert die Optionen (bis auf die Dateideskriptoren) in EDX: Das Kommata in DH, den neuen Feldtrenner in DL und das Flag für die -p-Option in dem höchsten Bit von EDX. Ein kurzer Abgleich des Zeichens wird uns also eine schnelle Entscheidung darüber erlauben, was zu tun ist.

Hier ist der 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
```

Vieles daraus ist aus hex.asm entnommen worden. Aber es gibt einen wichtigen Unterschied: Ich rufe nicht länger `write` auf, wann immer ich eine Zeilenvorschub ausgebe. Nun kann der Code sogar interaktiv genutzt werden.

Ich habe eine bessere Lösung gefunden für das Interaktivitätsproblem seit ich mit dem Schreiben dieses Kapitels begonnen habe. Ich wollte sichergehen, daß jede Zeile einzeln ausgegeben werden

kann, falls erforderlich. Aber schlussendlich gibt es keinen Bedarf jede Zeile einzeln auszugeben, falls nicht-interaktiv genutzt.

Die neue Lösung besteht darin, die Funktion `write` jedesmal aufzurufen, wenn ich den Eingabepuffer leer vorfinde. Auf diesem Wege liest das Programm im interaktiven Modus eine Zeile aus der Tastatur des Nutzers, verarbeitet sie und stellt fest, ob deren Eingabepuffer leer ist, dann leert es seine Ausgabe und liest die nächste Zeile.

11.12.1.1. Die dunkle Seite des Buffering

Diese Änderung verhindert einen mysteriösen Aufhänger in einem speziellen Fall. Ich bezeichne dies als die *dunkle Seite des Buffering*, hauptsächlich, weil es eine nicht offensichtliche Gefahr darstellt.

Es ist unwahrscheinlich, daß dies mit dem csv-Programm oben geschieht aber lassen Sie uns einen weiteren Filter betrachten: Nehmen wir an ihre Eingabe sind rohe Daten, die Farbwerte darstellen, wie z.B. die Intensität eines Pixel mit den Farben *rot*, *grün* und *blau*. Unsere Ausgabe wird der negative Wert unserer Eingabe sein.

Solch ein Filter würde sehr einfach zu schreiben sein. Der größte Teil davon würde so aussehen wie all die anderen Filter, die wir bisher geschrieben haben, daher beziehe ich mich nur auf den Kern der Prozedur:

```
.loop:
    call    getchar
    not al      ; Create a negative
    call    putchar
    jmp short .loop
```

Da dieser Filter mit rohen Daten arbeitet ist es unwahrscheinlich, daß er interaktiv genutzt werden wird.

Aber das Programm könnte als Bildbearbeitungssoftware tituliert werden. Wenn es nicht `write` vor jedem Aufruf von `read` durchführt, ist die Möglichkeit gegeben, das es sich aufhängt.

Dies könnte passieren:

1. Der Bildeditor wird unseren Filter laden mittels der C-Funktion `popen()`.
2. Er wird die erste Zeile von Pixeln laden aus einer Bitmap oder Pixmap.
3. Er wird die erste Zeile von Pixeln geschrieben in die *Pipe*, welche zur Variable `fd.in` unseres Filters führt.
4. Unser Filter wird jeden Pixel auslesen von der Eingabe, in in seinen negativen Wert umkehren und ihn in den Ausgabepuffer schreiben.
5. Unser Filter wird die Funktion `getchar` aufrufen, um das nächste Pixel abzurufen.
6. Die Funktion `getchar` wird einen leeren Eingabepuffer vorfinden und daher die Funktion `read` aufrufen.

7. `read` wird den Systemaufruf `SYS_read` starten.
8. Der *Kernel* wird unseren Filter unterbrechen, bis der Bildeditor mehr Daten zur Pipe sendet.
9. Der Bildedior wird aus der anderen Pipe lesen, welche verbunden ist mit `fd.out` unseres Filters, damit er die erste Zeile des auszugebenden Bildes setzen kann *bevor* er uns die zweite Zeile der Eingabe einliest.
10. Der *Kernel* unterbricht den Bildeditor, bis er eine Ausgabe unseres Filters erhält, um ihn an den Bildeditor weiterzureichen.

An diesem Punkt wartet unser Filter auf den Bildeditor, daß er ihm mehr Daten zur Verarbeitung schicken möge. Gleichzeitig wartet der Bildeditor darauf, daß unser Filter das Resultat der Berechnung ersten Zeile sendet. Aber das Ergebnis sitzt in unserem Ausgabepuffer.

Der Filter und der Bildeditor werden fortfahren bis in die Ewigkeit aufeinander zu warten (oder zumindest bis sie per kill entsorgt werden). Unsere Software hat den eine Race Condition erreicht.

Das Problem tritt nicht auf, wenn unser Filter seinen Ausgabepuffer leert *bevor* er vom *Kernel* mehr Eingabedaten anfordert.

11.13. Die FPU verwenden

Seltsamerweise erwähnt die meiste Literatur zu Assemblersprachen nicht einmal die Existenz der FPU, oder *floating point unit* (Fließkomma-Recheneinheit), geschweige denn, daß auf die Programmierung mit dieser eingegangen wird.

Dabei kann die Assemblerprogrammierung gerade bei hoch optimiertem FPU-Code, der *nur* mit einer Assemblersprache realisiert werden kann, ihre große Stärke ausspielen.

11.13.1. Organisation der FPU

Die FPU besteht aus 8 80-bit Fließkomma-Registern. Diese sind in Form eines Stacks organisiert-Sie können einen Wert durch den Befehl `push` auf dem TOS (*top of stack*) ablegen, oder durch `pop` von diesem holen.

Da also die Befehle `push` und `pop` schon verwendet werden, kann es keine op-Codes in Assemblersprache mit diesen Namen geben.

Sie können mit einen Wert auf dem TOS ablegen, indem Sie `fld`, `fild`, und `fbld` verwenden. Mit weiteren op-Codes lassen sich *Konstanten*-wie z.B. *Pi*-auf dem TOS ablegen.

Analog dazu können Sie einen Wert holen, indem Sie `fst`, `fstp`, `fist`, `fistp`, und `fbstp` verwenden. Eigentlich holen (`pop`) nur die op-Codes, die auf *p* enden, einen Wert, während die anderen den Wert irgendwo speichern (`store`) ohne ihn vom TOS zu entfernen.

Daten können zwischen dem TOS und dem Hauptspeicher als 32-bit, 64-bit oder 80-bit *real*, oder als 16-bit, 32-bit oder 64-bit *Integer*, oder als 80-bit *packed decimal* übertragen werden.

Das 80-bit *packed decimal*-Format ist ein Spezialfall des *binary coded decimal*-Formates, welches

üblicherweise bei der Konvertierung zwischen der ASCII- und FPU-Darstellung von Daten verwendet wird. Dieses erlaubt die Verwendung von 18 signifikanten Stellen.

Unabhängig davon, wie Daten im Speicher dargestellt werden, speichert die FPU ihre Daten immer im 80-bit *real*-Format in den Registern.

Ihre interne Genauigkeit beträgt mindestens 19 Dezimalstellen. Selbst wenn wir also Ergebnisse im ASCII-Format mit voller 18-stelliger Genauigkeit darstellen lassen, werden immer noch korrekte Werte angezeigt.

Des weiteren können mathematische Operationen auf dem TOS ausgeführt werden: Wir können dessen *Sinus* berechnen, wir können ihn *skalieren* (z.B. können wir ihn mit dem Faktor 2 Multiplizieren oder Dividieren), wir können dessen *Logarithmus* zur Basis 2 nehmen, und viele weitere Dinge.

Wir können auch FPU-Register *multiplizieren*, *dividieren*, *addieren* und *subtrahieren*, sogar einzelne Register mit sich selbst.

Der offizielle Intel op-Code für den TOS ist `st` und für die *Register* `st(0)`- `st(7)`. `st` und `st(0)` beziehen sich dabei auf das gleiche Register.

Aus welchen Gründen auch immer hat sich der Originalautor von nasm dafür entschieden, andere op-Codes zu verwenden, nämlich `st0`- `st7`. Mit anderen Worten, es gibt keine Klammern, und der TOS ist immer `st0`, niemals einfach nur `st`.

11.13.1.1. Das Packed Decimal-Format

Das *packed decimal*-Format verwendet 10 Bytes (80 Bits) zur Darstellung von 18 Ziffern. Die so dargestellte Zahl ist immer ein *Integer*.

Sie können durch Multiplikation des TOS mit Potenzen von 10 die einzelnen Dezimalstellen verschieben.

Das höchste Bit des höchsten Bytes (Byte 9) ist das *Vorzeichenbit*: Wenn es gesetzt ist, ist die Zahl *negativ*, ansonsten *positiv*. Die restlichen Bits dieses Bytes werden nicht verwendet bzw. ignoriert.

Die restlichen 9 Bytes enthalten die 18 Ziffern der gespeicherten Zahl: 2 Ziffern pro Byte.

Die *signifikantere Ziffer* wird in der *oberen Hälfte* (4 Bits) eines Bytes gespeichert, die andere in der *unteren Hälfte*.

Vielleicht würden Sie jetzt annehmen, das -1234567 auf die folgende Art im Speicher abgelegt wird (in hexadezimaler Notation):

```
80 00 00 00 00 00 01 23 45 67
```

Dem ist aber nicht so! Bei Intel werden alle Daten im *little-endian*-Format gespeichert, auch das *packed decimal*-Format.

Dies bedeutet, daß -1234567 wie folgt gespeichert wird:

```
67 45 23 01 00 00 00 00 00 80
```

Erinnern Sie sich an diesen Umstand, bevor Sie sich aus lauter Verzweiflung die Haare ausreißen.

Das lesenswerte Buch-falls Sie es finden können-ist Richard Startz' 8087/80287/80387 for the IBM PC & Compatibles. Obwohl es anscheinend die Speicherung der *packed decimal* im little-endian-Format für gegeben annimmt. Ich mache keine Witze über meine Verzweiflung, als ich den Fehler im unten stehenden Filter gesucht habe, *bevor* mir einfiel, daß ich einfach mal versuchen sollte, das little-endian-Format, selbst für diesen Typ von Daten, anzuwenden.

11.13.2. Ausflug in die Lochblendenphotographie

Um sinnvolle Programme zu schreiben, müssen wir nicht nur unsere Programmierwerkzeuge beherrschen, sondern auch das Umfeld, für das die Programme gedacht sind.

Unser nächster Filter wird uns dabei helfen, wann immer wir wollen, eine *Lochkamera* zu bauen. Wir brauchen also etwas Hintergrundwissen über die *Lochblendenphotographie*, bevor wir weiter machen können.

11.13.2.1. Die Kamera

Die einfachste Form, eine Kamera zu beschreiben, ist die eines abgeschlossenen, lichtundurchlässigen Raumes, in dessen Abdeckung sich ein kleines Loch befindet.

Die Abdeckung ist normalerweise fest (z.B. eine Schachtel), manchmal jedoch auch flexibel (z.B. ein Balgen). Innerhalb der Kamera ist es sehr dunkel. Nur durch ein kleines Loch kann Licht von einem einzigen Punkt aus in den Raum eindringen (in manchen Fällen sind es mehrere Löcher). Diese Lichtstrahlen kommen von einem Bild, einer Darstellung von dem was sich außerhalb der Kamera, vor dem kleinen Loch, befindet.

Wenn ein lichtempfindliches Material (wie z.B. ein Film) in der Kamera angebracht wird, so kann dieses das Bild einfangen.

Das Loch enthält häufig eine *Linse*, oder etwas linsenartiges, häufig auch einfach *Objektiv* genannt.

11.13.2.2. Die Lochblende

Streng genommen ist die Linse nicht notwendig: Die ursprünglichen Kameras verwendeten keine Linse, sondern eine *Lochblende*. Selbst heutzutage werden noch *Lochblenden* verwendet, zum einen, um die Funktionsweise einer Kamera zu erlernen, und zum anderen, um eine spezielle Art von Bildern zu erzeugen.

Das Bild, das von einer *Lochblende* erzeugt wird, ist überall scharf. Oder unscharf. Es gibt eine ideale Größe für eine Lochblende: Wenn sie größer oder kleiner ist, verliert das Bild seine Schärfe.

11.13.2.3. Brennweite

Dieser ideale Lochblendendurchmesser ist eine Funktion der Quadratwurzel der *Brennweite*, welche dem Abstand der Lochblende von dem Film entspricht.

```
D = PC * sqrt(FL)
```

Hier ist `D` der ideale Durchmesser der Lochblende, `FL` die Brennweite und PC eine Konstante der Brennweite. Nach Jay Bender hat die Konstante den Wert 0.04, nach Kenneth Connors 0.037. Andere Leute haben andere Werte vorgeschlagen. Des weiteren gelten diese Werte nur für Tageslicht: Andere Arten von Licht benötigen andere konstante Werte, welche nur durch Experimente bestimmt werden können.

11.13.2.4. Der f-Wert

Der f-Wert ist eine sehr nützliche Größe, die angibt, wieviel Licht den Film erreicht. Ein Belichtungsmesser kann dies messen, um z.B. für einen Film mit einer Empfindlichkeit von f5.6 eine Belichtungsdauer von 1/1000 Sekunden auszurechnen.

Es spielt keine Rolle, ob es eine 35-mm- oder eine 6x9cm-Kamera ist, usw. Solange wir den f-Wert kennen, können wir die benötigte Belichtungszeit berechnen.

Der f-Wert läßt sich einfach wie folgt berechnen:

```
F = FL / D
```

Mit anderen Worten, der f-Wert ergibt sich aus der Brennweite (FL), dividiert durch den Durchmesser (D) der Lochblende. Ein großer f-Wert impliziert also entweder eine kleine Lochblende, oder eine große Brennweite, oder beides. Je größer also der f-Wert ist, um so länger muß die Belichtungszeit sein.

Des weiteren sind der Lochblendendurchmesser und die Brennweite eindimensionale Meßgrößen, während der Film und die Lochblende an sich zweidimensionale Objekte darstellen. Das bedeutet, wenn man für einen f-Wert `A` eine Belichtungsdauer `t` bestimmt hat, dann ergibt sich daraus für einen f-Wert `B` eine Belichtungszeit von:

```
t * (B / A)²
```

11.13.2.5. Normalisierte f-Werte

Während heutige moderne Kameras den Durchmesser der Lochblende, und damit deren f-Wert, weich und schrittweise verändern können, war dies früher nicht der Fall.

Um unterschiedliche f-Werte einstellen zu können, besaßen Kameras typischerweise eine Metallplatte mit Löchern unterschiedlichen Durchmessers als Lochblende.

Die Durchmesser wurden entsprechend obiger Formel gewählt, daß der resultierende f-Wert ein

fester Standardwert war, der für alle Kameras verwendet wurde. Z.B. hat eine sehr alte Kodak Duaflex IV Kamera in meinem Besitz drei solche Löcher für die f-Werte 8, 11 und 16.

Eine neuere Kamera könnte f-Werte wie 2.8, 4, 5.6, 8, 11, 16, 22, und 32 (und weitere) besitzen. Diese Werte wurden nicht zufällig ausgewählt: Sie sind alle vielfache der Quadratwurzel aus 2, wobei manche Werte gerundet wurden.

11.13.2.6. Der f-Stopp

Eine typische Kamera ist so konzipiert, daß die Nummernscheibe bei den normalisierten f-Werten einrastet. Die Nummernscheibe *stoppt* an diesen Positionen. Daher werden diese Positionen auch f-Stopps genannt.

Da die f-Werte bei jedem Stopp vielfache der Quadratwurzel aus 2 sind, verdoppelt die Drehung der Nummernscheibe um einen Stopp die für die gleiche Belichtung benötigte Lichtmenge. Eine Drehung um 2 Stopps vervierfacht die benötigte Belichtungszeit. Eine Drehung um 3 Stopps verachtfacht sie, etc.

11.13.3. Entwurf der Lochblenden-Software

Wir können jetzt festlegen, was genau unsere Lochblenden-Software tun soll.

11.13.3.1. Verarbeitung der Programmeingaben

Da der Hauptzweck des Programms darin besteht, uns beim Entwurf einer funktionierenden Lochkamera zu helfen, wird die *Brennweite* die Programmeingabe sein. Dies ist etwas, das wir ohne zusätzliche Programme feststellen können: Die geeignete Brennweite ergibt sich aus der Größe des Films und der Art des Fotos, ob dieses ein "normales" Bild, ein Weitwinkelbild oder ein Telebild sein soll.

Die meisten bisher geschriebenen Programme arbeiteten mit einzelnen Zeichen, oder Bytes, als Eingabe: Das hex-Programm konvertierte einzelne Bytes in hexadezimale Werte, das csv-Programm ließ entweder einzelne Zeichen unverändert, löschte oder veränderte sie, etc.

Das Programm ftuc verwendete einen Zustandsautomaten, um höchstens zwei gleichzeitig eingegebene Bytes zu verarbeiten.

Das pinhole-Programm dagegen kann nicht nur mit einzelnen Zeichen arbeiten, sondern muß mit größeren syntaktischen Einheiten zurrecht kommen.

Wenn wir z.B. möchten, daß unser Programm den Lochblendendurchmesser (und weitere Werte, die wir später noch diskutieren werden) für die Brennweiten 100 mm, 150 mm und 210 mm berechnet, wollen wir etwa folgendes eingeben:

```
100, 150, 210
```

Unser Programm muß mit der gleichzeitigen Eingabe von mehr als nur einem einzelnen Byte zurecht kommen. Wenn es eine 1 erkennt, muß es wissen, daß dies die erste Stelle einer dezimalen Zahl ist. Wenn es eine 0, gefolgt von einer weiteren 0 sieht, muß es wissen, daß dies zwei

unterschiedliche Stellen mit der gleichen Zahl sind.

Wenn es auf das erste Komma trifft, muß es wissen, daß die folgenden Stellen nicht mehr zur ersten Zahl gehören. Es muß die Stellen der ersten Zahl in den Wert 100 konvertieren können. Und die Stellen der zweiten Zahl müssen in den Wert 150 konvertiert werden. Und die Stellen der dritten Zahl müssen in den Wert 210 konvertiert werden.

Wir müssen festlegen, welche Trennsymbole zulässig sind: Sollen die Eingabewerte durch Kommas voneinander getrennt werden? Wenn ja, wie sollen zwei Zahlen behandelt werden, die durch ein anderes Zeichen getrennt sind?

Ich persönlich mag es einfach. Entweder etwas ist eine Zahl, dann wird es verarbeitet, oder es ist keine Zahl, dann wird es verworfen. Ich mag es nicht, wenn sich der Computer bei der *offensichtlichen* Eingabe eines zusätzlichen Zeichens beschwert. Duh!

Zusätzlich erlaubt es mir, die Monotonie des Tippens zu durchbrechen, und eine Anfrage anstelle einer simplen Zahl zu stellen:

```
Was ist der beste Lochblendendurchmesser
     bei einer Brennweite von 150?
```

Es gibt keinen Grund dafür, die Ausgabe mehrerer Fehlermeldungen aufzuteilen:

```
Syntax error: Was
Syntax error: ist
Syntax error: der
Syntax error: beste
```

Et cetera, et cetera, et cetera.

Zweitens mag ich das #-Zeichen, um Kommentare zu markieren, die ab dem Zeichen bis zum Ende der jeweiligen Zeile gehen. Dies verlangt nicht viel Programmieraufwand, und ermöglicht es mir, Eingabedateien für meine Programme als ausführbare Skripte zu handhaben.

In unserem Fall müssen wir auch entscheiden, in welchen Einheiten die Dateneingabe erfolgen soll: Wir wählen *Millimeter*, da die meisten Photographen die Brennweite in dieser Einheit messen.

Letztendlich müssen wir noch entscheiden, ob wir die Verwendung des dezimalen Punktes erlauben (in diesem Fall müssen wir berücksichtigen, daß in vielen Ländern der Welt das dezimale *Komma* verwendet wird).

In unserem Fall würde das Zulassen eines dezimalen Punktes/Kommas zu einer fälschlicherweise angenommenen, höheren Genauigkeit führen: Der Unterschied zwischen den Brennweiten 50 und 51 ist fast nicht wahrnehmbar. Die Zulassung von Eingaben wie 50.5 ist also keine gute Idee. Beachten Sie bitte, das dies meine Meinung ist. In diesem Fall bin ich der Autor des Programmes. Bei Ihren eigenen Programmen müssen Sie selbst solche Entscheidungen treffen.

11.13.3.2. Optionen anbieten

Das wichtigste, was wir zum Bau einer Lochkamera wissen müssen, ist der Durchmesser der Lochblende. Da wir scharfe Bilder schießen wollen, werden wir obige Formel für die Berechnung des korrekten Durchmessers zu gegebener Brennweite verwenden. Da Experten mehrere Werte für die PC-Konstante anbieten, müssen wir uns hier für einen Wert entscheiden.

In der Programmierung unter UNIX® ist es üblich, zwei Hauptvarianten anzubieten, um Parameter an Programme zu übergeben, und des weiteren eine Standardeinstellung für den Fall zu haben, das der Benutzer gar keine Parameter angibt.

Warum zwei Varianten, Parameter anzugeben?

Ein Grund ist, eine (relativ) *feste* Einstellung anzubieten, die automatisch bei jedem Programmaufruf verwendet wird, ohne das wir diese Einstellung immer und immer wieder mit angeben müssen.

Die feste Einstellung kann in einer Konfigurationsdatei gespeichert sein, typischerweise im Heimatverzeichnis des Benutzers. Die Datei hat üblicherweise denselben Namen wie das zugehörige Programm, beginnt jedoch mit einem Punkt. Häufig wird *"rc"* dem Dateinamen hinzugefügt. Unsere Konfigurationsdatei könnte also ~/.pinhole oder ~/.pinholerc heißen. (Die Zeichenfolge ~/ steht für das Heimatverzeichnis des aktuellen Benutzers.)

Konfigurationsdateien werden häufig von Programmen verwendet, die viele konfigurierbare Parameter besitzen. Programme, die nur eine (oder wenige) Parameter anbieten, verwenden häufig eine andere Methode: Sie erwarten die Parameter in einer *Umgebungsvariablen*. In unserem Fall könnten wir eine Umgebungsvariable mit dem Namen `PINHOLE` benutzen.

Normalerweise verwendet ein Programm entweder die eine, oder die andere der beiden obigen Methoden. Ansonsten könnte ein Programm verwirrt werden, wenn eine Konfigurationsdatei das eine sagt, die Umgebungsvariable jedoch etwas anderes.

Da wir nur *einen* Parameter unterstützen müssen, verwenden wir die zweite Methode, und benutzen eine Umgebungsvariable mit dem Namen `PINHOLE`.

Der andere Weg erlaubt uns, *ad hoc* Entscheidungen zu treffen: *"Obwohl ich normalerweise einen Wert von 0.039 verwende, will ich dieses eine Mal einen Wert von 0.03872 anwenden."* Mit anderen Worten, dies erlaubt uns, die Standardeinstellung außer Kraft zu setzen.

Diese Art der Auswahl wird häufig über Kommandozeilenparameter gemacht.

Schließlich braucht ein Programm *immer* eine *Standardeinstellung*. Der Benutzer könnte keine Parameter angeben. Vielleicht weiß er auch gar nicht, was er einstellen sollte. Vielleicht will er es "einfach nur ausprobieren". Vorzugsweise wird die Standardeinstellung eine sein, die die meisten Benutzer sowieso wählen würden. Somit müssen diese keine zusätzlichen Parameter angeben, bzw. können die Standardeinstellung ohne zusätzlichen Aufwand benutzen.

Bei diesem System könnte das Programm widersprüchliche Optionen vorfinden, und auf die folgende Weise reagieren:

1. Wenn es eine *ad hoc*-Einstellung vorfindet (z.B. ein Kommandozeilenparameter), dann sollte es diese Einstellung annehmen. Es muß alle vorher festgelegten sowie die standardmäßige Einstellung ignorieren.
2. *Andererseits*, wenn es eine festgelegte Option (z.B. eine Umgebungsvariable) vorfindet, dann sollte es diese akzeptieren und die Standardeinstellung ignorieren.
3. *Ansonsten* sollte es die Standardeinstellung verwenden.

Wir müssen auch entscheiden, welches *Format* unsere PC-Option haben soll.

Auf den ersten Blick scheint es einleuchtend, das Format `PINHOLE=0.04` für die Umgebungsvariable, und -p0.04 für die Kommandozeile zu verwenden.

Dies zuzulassen wäre eigentlich eine Sicherheitslücke. Die PC-Konstante ist eine sehr kleine Zahl. Daher würden wir unsere Anwendung mit verschiedenen, kleinen Werten für PC testen. Aber was würde passieren, wenn jemand das Programm mit einem sehr großen Wert aufrufen würde?

Es könnte abstürzen, weil wir das Programm nicht für den Umgang mit großen Werten entworfen haben.

Oder wir investieren noch weiter Zeit in das Programm, so daß dieses dann auch mit großen Zahlen umgehen kann. Wir könnten dies machen, wenn wir kommerzielle Software für computertechnisch unerfahrene Benutzer schreiben würden.

Oder wir könnten auch sagen *"Pech gehabt! Der Benutzer sollte es besser wissen."*

Oder wir könnten es für den Benutzer unmöglich machen, große Zahlen einzugeben. Dies ist die Variante, die wir verwenden werden: Wir nehmen einen *impliziten 0.*-Präfix an.

Mit anderen Worten, wenn der Benutzer den Wert 0.04 angeben will, so muß er entweder -p04 als Parameter angeben, oder `PINHOLE=04` als Variable in seiner Umgebung definieren. Falls der Benutzer -p9999999 angibt, so wird dies als 0.9999999 interpretiert-zwar immer noch sinnlos, aber zumindest sicher.

Zweitens werden viele Benutzer einfach die Konstanten von Bender oder Connors benutzen wollen. Um es diesen Benutzern einfacher zu machen, werden wir -b als -p04, und -c als -p037 interpretieren.

11.13.3.3. Die Ausgabe

Wir müssen festlegen, was und in welchem Format unsere Anwendung Daten ausgeben soll.

Da wir als Eingabe beliebig viele Brennweiten erlauben, macht es Sinn, die Ergebnisse in Form einer traditionellen Datenbank-Ausgabe darzustellen, bei der zeilenweise zu jeder Brennweite der zugehörige berechnete Wert, getrennt durch ein tab-Zeichen, ausgegeben wird.

Optional sollten wir dem Benutzer die Möglichkeit geben, die Ausgabe in dem schon beschriebenen CSV-Format festzulegen. In diesem Fall werden wir zu Beginn der Ausgabe eine Zeile einfügen, in der die Beschreibungen der einzelnen Felder, durch Kommas getrennt, aufgelistet werden, gefolgt

von der Ausgabe der Daten wie schon beschrieben, wobei das tab-Zeichen durch ein Komma ersetzt wird.

Wir brauchen eine Kommandozeilenoption für das CSV-Format. Wir können nicht -c verwenden, da diese Option bereits für *verwende Connors Konstante* steht. Aus irgendeinem seltsamen Grund bezeichnen viele Webseiten CSV-Dateien als *"Excel Kalkulationstabelle"* (obwohl das CSV-Format älter ist als Excel). Wir werden daher -e als Schalter für die Ausgabe im CSV-Format verwenden.

Jede Zeile der Ausgabe wird mit einer Brennweite beginnen. Dies mag auf den ersten Blick überflüssig erscheinen, besonders im interaktiven Modus: Der Benutzer gibt einen Wert für die Brennweite ein, und das Programm wiederholt diesen.

Der Benutzer kann jedoch auch mehrere Brennweiten in einer Zeile angeben. Die Eingabe kann auch aus einer Datei, oder aus der Ausgabe eines anderen Programmes, kommen. In diesen Fällen sieht der Benutzer die Eingabewerte überhaupt nicht.

Ebenso kann die Ausgabe in eine Datei umgelenkt werden, was wir später noch untersuchen werden, oder sie könnte an einen Drucker geschickt werden, oder auch als Eingabe für ein weiteres Programm dienen.

Es macht also wohl Sinn, jede Zeile mit einer durch den Benutzer eingegebenen Brennweite beginnen zu lassen.

Halt! Nicht, wie der Benutzer die Daten eingegeben hat. Was passiert, wenn der Benutzer etwas wie folgt eingibt:

```
00000000150
```

Offensichtlich müssen wir die führenden Nullen vorher abschneiden.

Wir müssen also die Eingabe des Benutzers sorgfältig prüfen, diese dann in der FPU in die binäre Form konvertieren, und dann von dort aus ausgeben.

Aber…

Was ist, wenn der Benutzer etwas wie folgt eingibt:

```
1745976572345235345353453535353053053456350730967676442 3
```

Ha! Das packed decimal-Format der FPU erlaubt uns die Eingabe einer 18-stelligen Zahl. Aber der Benutzer hat mehr als 18 Stellen eingegeben. Wie gehen wir damit um?

Wir *könnten* unser Programm so modifizieren, daß es die ersten 18 Stellen liest, der FPU übergibt, dann weitere 18 Stellen liest, den Inhalt des TOS mit einem Vielfachen von 10, entsprechend der Anzahl der zusätzlichen Stellen multipliziert, und dann beide Werte mittels `add` zusammen addiert.

Ja, wir könnten das machen. Aber in *diesem* Programm wäre es unnötig (in einem anderen wäre es vielleicht der richtige Weg): Selbst der Erdumfang in Millimetern ergibt nur eine Zahl mit 11

Stellen. Offensichtlich können wir keine Kamera dieser Größe bauen (jedenfalls jetzt noch nicht).

Wenn der Benutzer also eine so große Zahl eingibt, ist er entweder gelangweilt, oder er testet uns, oder er versucht, in das System einzudringen, oder er spielt- indem er irgendetwas anderes macht als eine Lochkamera zu entwerfen.

Was werden wir tun?

Wir werden ihn ohrfeigen, gewissermaßen:

```
174597657234523534535345355353053053456350730967676442 3    ??? ??? ??? ??? ???
```

Um dies zu erreichen, werden wir einfach alle führenden Nullen ignorieren. Sobald wir eine Ziffer gefunden haben, die nicht Null ist, initialisieren wir einen Zähler mit 0 und beginnen mit drei Schritten:

1. Sende die Ziffer an die Ausgabe.
2. Füge die Ziffer einem Puffer hinzu, welchen wir später benutzen werden, um den packed decimal-Wert zu erzeugen, den wir an die FPU schicken können.
3. Erhöhe den Zähler um eins.

Während wir diese drei Schritte wiederholen, müssen wir auf zwei Bedingungen achten:

- Wenn der Zähler den Wert 18 übersteigt, hören wir auf, Ziffern dem Puffer hinzuzufügen. Wir lesen weiterhin Ziffern und senden sie an die Ausgabe.
- Wenn, bzw. *falls*, das nächste Eingabezeichen keine Zahl ist, sind wir mit der Bearbeitung der Eingabe erst einmal fertig.

  Übrigends können wir einfach Zeichen, die keine Ziffern sind, verwerfen, solange sie kein # -Zeichen sind, welches wir an den Eingabestrom zurückgeben müssen. Dieses Zeichen markiert den Beginn eines Kommentars. An dieser Stelle muß die Erzeugung der Ausgabe fertig sein, und wir müssen mit der Suche nach weiteren Eingabedaten fortfahren.

Es bleibt immer noch eine Möglichkeit unberücksichtigt: Wenn der Benutzer eine Null (oder mehrere) eingibt, werden wir niemals eine von Null verschiedene Zahl vorfinden.

Wir können solch einen Fall immer anhand des Zählerstandes feststellen, welcher dann immer bei 0 bleibt. In diesem Fall müssen wir einfach eine 0 an die Ausgabe senden, und anschließend dem Benutzer erneut eine "Ohrfeige" verpassen:

```
0   ??? ??? ??? ??? ???
```

Sobald wir die Brennweite ausgegeben, und die Gültigkeit dieser Eingabe verifiziert haben, (größer als 0 und kleiner als 18 Zahlen) können wir den Durchmesser der Lochblende berechnen.

Es ist kein Zufall, daß *Lochblende* das Wort *Loch* enthält. In der Tat ist eine Lochblende buchstäblich

eine *Loch Blende*, also eine Blende, in die mit einer Nadel vorsichtig ein kleines Loch gestochen wird.

Daher ist eine typische Lochblende sehr klein. Unsere Formel liefert uns das Ergebnis in Millimetern. Wir werden dieses mit 1000 multiplizieren, so daß die Ausgabe in Mikrometern erfolgt.

An dieser Stelle müssen wir auf eine weitere Falle achten: *Zu hohe Genauigkeit.*

Ja, die FPU wurde für mathematische Berechnungen mit hoher Genauigkeit entworfen. Unsere Berechnungen hier erfordern jedoch keine solche mathematische Genauigkeit. Wir haben es hier mit Physik zu tun (Optik, um genau zu sein).

Angenommen, wir wollten aus eine Lastkraftwagen eine Lochkamera bauen (wir wären dabei nicht die ersten, die das versuchen würden!). Angenommen, die Länge des Laderaumes beträgt 12 Meter lang, so daß wir eine Brennweite von 12000 hätten. Verwenden wir Benders Konstante, so erhalten wir durch Multiplizieren von 0.04 mit der Quadratwurzel aus 12000 einen Wert von 4.381780460 Millimetern, oder 4381.780460 Micrometern.

So oder so ist das Rechenergebnis absurd präzise. Unser Lastkraftwagen ist nicht *genau*12000 Millimeter lang. Wir haben diese Länge nicht mit einer so hohen Genauigkeit gemessen, weswegen es falsch wäre zu behaupten, unser Lochblendendurchmesser müsse exakt 4.381780460 Millimeter sein. Es reicht vollkommen aus, wenn der Durchmesser 4.4 Millimeter beträgt.

Ich habe in obigem Beispiel das Rechenergebnis "nur" auf 10 Stellen genau angegeben. Stellen Sie sich vor, wie absurd es wäre, die vollen uns zur Verfügung stehenden, 18 Stellen anzugeben!

Wir müssen also die Anzahl der signifikanten Stellen beschränken. Eine Möglichkeit wäre, die Mikrometer durch eine ganze Zahl darzustellen. Unser Lastkraftwaren würde dann eine Lochblende mit einem Durchmesser von 4382 Mikrometern benötigen. Betrachten wir diesen Wert, dann stellen wir fest, das 4400 Mikrometer, oder 4.4 Millimeter, immer noch genau genug ist.

Zusätzlich können wir noch, unabhängig von der Größe eines Rechenergebnisses, festlegen, daß wir nur vier signifikante Stellen anzeigen wollen (oder weniger). Leider bietet uns die FPU nicht die Möglichkeit, das Ergebnis automatisch bis auf eine bestimmte Stelle zu runden (sie sieht die Daten ja nicht als Zahlen, sondern als binäre Daten an).

Wir müssen also selber einen Algorithmus entwerfen, um die Anzahl der signifikanten Stellen zu reduzieren.

Hier ist meiner (ich denke er ist peinlich-wenn Ihnen ein besserer Algorithmus einfällt, verraten sie ihn mir *bitte*):

1. Initialisiere einen Zähler mit 0.
2. Solange die Zahl größer oder gleich 10000 ist, dividiere die Zahl durch 10, und erhöhe den Zähler um eins.
3. Gebe das Ergebnis aus.

4. Solange der Zähler größer als 0 ist, gebe eine 0 aus, und reduziere den Zähler um eins.

Der Wert 10000 ist nur für den Fall, daß Sie *vier* signifikante Stellen haben wollen. Für eine andere Anzahl signifikanter Stellen müssen Sie den Wert 10000 mit 10, hoch der Anzahl der gewünschten signifikanten Stellen, ersetzen.

Wir können so den Lochblendendurchmesser, auf vier signifikante Stellen gerundet, ausgeben.

An dieser Stellen kennen wir nun die *Brennweite* und den *Lochblendendurchmesser*. Wir haben also jetzt genug Informationen, um den *f-Wert* zu bestimmen.

Wir werden den f-Wert, auf vier signifikante Stellen gerundet, ausgeben. Es könnte passieren, daß diese vier Stellen recht wenig aussagen. Um die Aussagekraft des f-Wertes zu erhöhen, könnten wir den nächstliegenden, *normalisierten f-Wert* bestimmen, also z.B. das nächstliegende Vielfache der Quadratwurzel aus 2.

Wir erreichen dies, indem wir den aktuellen f-Wert mit sich selbst multiplizieren, so daß wir dessen Quadrat (`square`) erhalten. Anschließend berechnen wir den Logarithmus zur Basis 2 von dieser Zahl. Dies ist sehr viel einfacher, als direkt den Logarithmus zur Basis der Quadratwurzel aus 2 zu berechnen! Wir runden dann das Ergebnis auf die nächstliegende ganze Zahl. Genau genommen können wir mit Hilfe der FPU diese Berechnung beschleunigen: Wir können den op-Code `fscale` verwenden, um eine Zahl um 1 zu "skalieren", was dasselbe ist, wie eine Zahl mittels `shift` um eine Stelle nach links zu verschieben. Am Ende berechnen wir noch die Quadratwurzel aus allem, und erhalten dann den nächstliegenden, normalisierten f-Wert.

Wenn das alles jetzt viel zu kompliziert wirkt-oder viel zu aufwendig-wird es vielleicht klarer, wenn man den Code selber betrachtet. Wir benötigen insgesamt 9 op-Codes:

```
fmul    st0, st0
    fld1
    fld     st1
    fyl2x
    frndint
    fld1
    fscale
    fsqrt
    fstp    st1
```

Die erste Zeile, `fmul st0, st0`, quadriert den Inhalt des TOS (Top Of Stack, was dasselbe ist wie `st`, von nasm auch `st0` genannt). Die Funktion `fld1` fügt eine 1 dem TOS hinzu.

Die nächste Zeile, `fld st1`, legt das Quadrat auf dem TOS ab. An diesem Punkt befindet sich das Quadrat sowohl in `st` als auch in `st(2)` (es wird sich gleich zeigen, warum wir eine zweite Kopie auf dem Stack lassen.) `st(1)` enthält die 1.

Im nächsten Schritt, `fyl2x`, wird der Logarithmus von `st` zur Basis 2 berechnet, und anschließend mit `st(1)` multipliziert. Deshalb haben wir vorher die 1 in `st(1)` abgelegt.

An dieser Stelle enthält `st` den gerade berechneten Logarithmus, und `st(1)` das Quadrat des aktuellen f-Wertes, den wir für später gespeichert haben.

`frndint` rundet den TOS zur nächstliegenden ganzen Zahl. `fld1` legt eine 1 auf dem Stack ab. `fscale` shiftet die 1 auf dem TOS um `st(1)` Stellen, wodurch im Endeffekt eine 2 in `st(1)` steht.

Schließlich berechnet `fsqrt` die Quadratwurzel des Rechenergebnisses, also des nächstliegenden, normalisierten f-Wertes.

Wir haben nun den nächstliegenden, normalisierten f-Wert auf dem TOS liegen, den auf den Logarithmus zur Basis 2 gerundeten, nächstliegenden ganzzahligen Wert in `st(1)`, und das Quadrat des aktuellen f-Wertes in `st(2)`. Wir speichern den Wert für eine spätere Verwendung in `st(2)`.

Aber wir brauchen den Inhalt von `st(1)` gar nicht mehr. Die letzte Zeile, `fstp st1`, platziert den Inhalt von `st` in `st(1)`, und erniedrigt den Stackpointer um eins. Dadurch ist der Inhalt von `st(1)` jetzt `st`, der Inhalt von `st(2)` jetzt `st(1)` usw. Der neue `st` speichert jetzt den normalisierten f-Wert. Der neue `st(1)` speichert das Quadrat des aktuellen f-Wertes für die Nachwelt.

Jetzt können wir den normalisierten f-Wert ausgeben. Da er normalisiert ist, werden wir ihn nicht auf vier signifikante Stellen runden, sondern stattdessen mit voller Genauigkeit ausgeben.

Der normalisierte f-Wert ist nützlich, solange er so klein ist, daß wir ihn auf einem Photometer wiederfinden können. Ansonsten brauchen wir eine andere Methode, um die benötigten Belichtungsdaten zu bestimmen.

Wir haben weiter oben eine Formel aufgestellt, über die wir einen f-Wert mit Hilfe eines anderen f-Wertes und den zugehörigen Belichtungsdaten bestimmen können.

Jedes Photometer, das ich jemals gesehen habe, konnte die benötigte Belichtungszeit für f5.6 berechnen. Wir werden daher einen *"f5.6 Multiplizierer"* berechnen, der uns den Faktor angibt, mit dem wir die bei f5.6 gemessene Belichtungszeit für unsere Lochkamera multiplizieren müssen.

Durch die Formel wissen wir, daß dieser Faktor durch Dividieren unseres f-Wertes (der aktuelle Wert, nicht der normalisierte) durch 5.6 und anschließendes Quadrieren, berechnen können.

Mathematisch äquivalent dazu wäre, wenn wir das Quadrat unseres f-Wertes durch das Quadrat von 5.6 dividieren würden.

Numerisch betrachtet wollen wir nicht zwei Zahlen quadrieren, wenn es möglich ist, nur eine Zahl zu quadrieren. Daher wirkt die erste Variante auf den ersten Blick besser.

Aber…

5.6 ist eine *Konstante*. Wir müssen nicht wertvolle Rechenzeit der FPU verschwenden. Es reicht aus, daß wir die Quadrate der einzelnen f-Werte durch den konstanten Wert 5.6^2 dividieren. Oder wir können den jeweiligen f-Wert durch 5.6 dividieren, und dann das Ergebnis quadrieren. Zwei Möglichkeiten, die gleich erscheinen.

Aber das sind sie nicht!

Erinnern wir uns an die Grundlagen der Photographie weiter oben, dann wissen wir, daß sich die

Konstante 5.6 aus dem 5-fachen der Quadratwurzel aus 2 ergibt. Eine *irrationale* Zahl. Das Quadrat dieser Zahl ist *exakt*32.

32 ist nicht nur eine ganze Zahl, sondern auch ein Vielfaches von 2. Wir brauchen also gar nicht das Quadrat eines f-Wertes durch 32 zu teilen. Wir müssen lediglich mittels `fscale` den f-Wert um fünf Stellen nach rechts shiften. Aus Sicht der FPU müssen wir also `fscale` mit `st(1)`, welcher gleich -5 ist, auf den f-Wert anwenden. Dies ist *sehr viel schneller* als die Division.

Jetzt wird es auch klar, warum wir das Quadrat des f-Wertes ganz oben auf dem Stack der FPU gespeichert haben. Die Berechnung des f5.6 Multiplizierers ist die einfachste Berechnung des gesamten Programmes! Wir werden das Ergebnis auf vier signifikante Stellen gerundet ausgeben.

Es gibt noch eine weitere nützliche Zahl, die wir berechnen können: Die Anzahl der Stopps, die unser f-Wert von f5.6 entfernt ist. Dies könnte hilfreich sein, wenn unser f-Wert außerhalb des Meßbereiches unseres Photometers liegt, wir aber eine Blende haben, bei der wir unterschiedliche Geschwindigkeiten einstellen können, und diese Blende Stopps benutzt.

Angenommen, unser f-Wert ist 5 Stopps von f5.6 entfernt, und unser Photometer sagt uns, daß wir eine Belichtungszeit von 1/1000 Sek. einstellen sollen. Dann können wir unsere Blende auf die Geschwindigkeit 1/1000 einstellen, und unsere Skala um 5 Stopps verschieben.

Diese Rechnung ist ebenfalls sehr einfach. Alles, was wir tun müssen, ist, den Logarithmus des f5.6 Multiplizierers, den wir schon berechnet haben (wobei wir dessen Wert vor der Rundung nehmen müssen) zur Basis 2 zu nehmen. Wir runden dann das Ergebnis zur nächsten ganzen Zahl hin, und geben dies aus. Wir müssen uns nicht darum kümmern, ob wir mehr als vier signifikante Stellen haben: Das Ergebnis besteht höchstwahrscheinlich nur aus einer oder zwei Stellen.

11.13.4. FPU Optimierungen

In Assemblersprache können wir den Code für die FPU besser optimieren, als in einer der Hochsprachen, inklusive C.

Sobald eine C-Funktion die Berechnung einer Fließkommazahl durchführen will, lädt sie erst einmal alle benötigten Variablen und Konstanten in die Register der FPU. Dann werden die Berechnungen durchgeführt, um das korrekte Ergebnis zu erhalten. Gute C-Compiler können diesen Teil des Codes sehr gut optimieren.

Das Ergebnis wird "zurückgegeben", indem dieses auf dem TOS abgelegt wird. Vorher wird aufgeräumt. Sämtliche Variablen und Konstanten, die während der Berechnung verwendet wurden, werden dabei aus der FPU entfernt.

Was wir im vorherigen Abschnitt selber getan haben, kann so nicht durchgeführt werden: Wir haben das Quadrat des f-Wertes berechnet, und das Ergebnis für eine weitere Berechnung mit einer anderen Funktion auf dem Stack behalten.

Wir *wußten*, daß wir diesen Wert später noch einmal brauchen würden. Wir wußten auch, daß auf dem Stack genügend Platz war (welcher nur Platz für 8 Zahlen bietet), um den Wert dort zu speichern.

Ein C-Compiler kann nicht wissen, ob ein Wert auf dem Stack in naher Zukunft noch einmal

gebraucht wird.

Natürlich könnte der C-Programmierer dies wissen. Aber die einzige Möglichkeit, die er hat, ist, den Wert im verfügbaren Speicher zu halten.

Das bedeutet zum einen, daß der Wert mit der FPU-internen, 80-stelligen Genauigkeit in einer normalen C-Variable vom Typ *double* (64 Bit) oder vom Typ *single* (32 Bit) gespeichert wird.

Dies bedeutet außerdem, daß der Wert aus dem TOS in den Speicher verschoben werden muß, und später wieder zurück. Von allen Operationen mit der FPU ist der Zugriff auf den Speicher die langsamste.

Wann immer also mit der FPU in einer Assemblersprache programmiert wird, sollte nach Möglichkeiten gesucht werden, Zwischenergebnisse auf dem Stack der FPU zu lassen.

Wir können mit dieser Idee noch einen Schritt weiter gehen! In unserem Programm verwenden wir eine *Konstante* (die wir PC genannt haben).

Es ist unwichtig, wieviele Lochblendendurchmesser wir berechnen: 1, 10, 20, 1000, wir verwenden immer dieselbe Konstante. Daher können wir unser Programm so optimieren, daß diese Konstante immer auf dem Stack belassen wird.

Am Anfang unseres Programmes berechnen wir die oben erwähnte Konstante. Wir müssen die Eingabe für jede Dezimalstelle der Konstanten durch 10 dividieren.

Multiplizieren geht sehr viel schneller als Dividieren. Wir teilen also zu Beginn unseres Programmes 1 durch 10, um 0.1 zu erhalten, was wir auf dem Stack speichern: Anstatt daß wir nun für jede einzelne Dezimalstelle die Eingabe wieder durch 10 teilen, multiplizieren wir sie stattdessen mit 0.1.

Auf diese Weise geben wir 0.1 nicht direkt ein, obwohl wir dies könnten. Dies hat einen Grund: Während 0.1 durch nur eine einzige Dezimalstelle dargestellt werden kann, wissen wir nicht, wieviele *binäre* Stellen benötigt werden. Wir überlassen die Berechnung des binären Wertes daher der FPU, mit dessen eigener, hoher Genauigkeit.

Wir verwenden noch weitere Konstanten: Wir multiplizieren den Lochblendendurchmesser mit 1000, um den Wert von Millimeter in Micrometer zu konvertieren. Wir vergleichen Werte mit 10000, wenn wir diese auf vier signifikante Stellen runden wollen. Wir behalten also beide Konstanten, 1000 und 10000, auf dem Stack. Und selbstverständlich verwenden wir erneut die gespeicherte 0.1, um Werte auf vier signifikante Stellen zu runden.

Zu guter letzt behalten wir -5 noch auf dem Stack. Wir brauchen diesen Wert, um das Quadrat des f-Wertes zu skalieren, anstatt diesen durch 32 zu teilen. Es ist kein Zufall, daß wir diese Konstante als letztes laden. Dadurch wird diese Zahl die oberste Konstante auf dem Stack. Wenn später das Quadrat des f-Wertes skaliert werden muß, befindet sich die -5 in `st(1)`, also genau da, wo die Funktion `fscale` diesen Wert erwartet.

Es ist üblich, einige Konstanten per Hand zu erzeugen, anstatt sie aus dem Speicher zu laden. Genau das machen wir mit der -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
```

Wir können all diese Optimierungen in einer Regel zusammenfassen: *Behalte wiederverwendbare Werte auf dem Stack!*

PostScript® ist eine Stack-orientierte Programmiersprache. Es gibt weit mehr Bücher über PostScript®, als über die Assemblersprache der FPU: Werden Sie in PostScript® besser, dann werden Sie auch automatisch in der Programmierung der FPU besser.

11.13.5. pinhole-Der 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
```

Der Code folgt demselben Aufbau wie alle anderen Filter, die wir bisher gesehen haben, bis auf eine Kleinigkeit:

Wir nehmen nun nicht mehr an, daß das Ende der Eingabe auch das Ende der nötigen Arbeit bedeutet, etwas, das wir für *zeichenbasierte* Filter automatisch angenommen haben.

Dieser Filter verarbeitet keine Zeichen. Er verarbeitet eine *Sprache* (obgleich eine sehr einfache, die nur aus Zahlen besteht).

Wenn keine weiteren Eingaben vorliegen, kann das zwei Ursachen haben:

- Wir sind fertig und können aufhören. Dies ist dasselbe wie vorher.
- Das Zeichen, das wir eingelesen haben, war eine Zahl. Wir haben diese am Ende unseres ASCII -zu-float Kovertierungspuffers gespeichert. Wir müssen nun den gesamten Pufferinhalt in eine Zahl konvertieren, und die letzte Zeile unserer Ausgabe ausgeben.

Aus diesem Grund haben wir unsere `getchar`- und `read`-Routinen so angepaßt, daß sie das `carry flag` *clear* immer dann zurückgeben, wenn wir ein weiteres Zeichen aus der Eingabe lesen, und das `carry flag` *set* immer dann zurückgeben, wenn es keine weiteren Eingabedaten gibt.

Selbstverständlich verwenden wir auch hier die Magie der Assemblersprache! Schauen Sie sich `getchar` näher an. Dieses gibt *immer* das `carry flag` *clear* zurück.

Dennoch basiert der Hauptteil unseres Programmes auf dem `carry flag`, um diesem eine Beendigung mitzuteilen-und es funktioniert.

Die Magie passiert in `read`. Wann immer weitere Eingaben durch das System zur Verfügung stehen, ruft diese Funktion `getchar` auf, welche ein weiteres Zeichen aus dem Eingabepuffer einliest, und anschließend das `carry flag` *clear*t.

Wenn aber `read` keine weiteren Eingaben von dem System bekommt, ruft dieses *nicht* `getchar` auf. Stattdessen addiert der op-Code `add esp, byte 4` 4 zu `ESP` hinzu, *setzt* das `carry flag`, und springt zurück.

Wo springt diese Funktion hin? Wann immer ein Programm den op-Code `call` verwendet, `push`t der Mikroprozessor die Rücksprungandresse, d.h. er speichert diese ganz oben auf dem Stack (nicht auf dem Stack der FPU, sondern auf dem Systemstack, der sich im Hauptspeicher befindet). Wenn ein Programm den op-Code `ret` verwendet, `pop`t der Mikroprozessor den Rückgabewert von dem Stack, und springt zu der Adresse, die dort gespeichert wurde.

Da wir aber 4 zu `ESP` hinzuaddiert haben (welches das Register der Stackzeiger ist), haben wir effektiv dem Mikroprzessor eine kleine *Amnesie* verpaßt: Dieser erinnert sich nun nicht mehr daran, daß `getchar` durch `read` aufgerufen wurde.

Und da `getchar` nichts vor dem Aufruf von `read` auf dem Stack abgelegt hat, enthält der Anfang des Stacks nun die Rücksprungadresse von der Funktion, die `getchar` aufgerufen hat. Soweit es den

Aufrufer betrifft, hat dieser `getchar` gecallt, welche mit einem gesetzten `carry flag` returned.

Des weiteren wird die Routine `bcdload` bei einem klitzekleinen Problem zwischen der Big-Endian- und Little-Endian-Codierung aufgerufen.

Diese konvertiert die Textrepräsentation einer Zahl in eine andere Textrepräsentation: Der Text wird in der Big-Endian-Codierung gespeichert, die *packed decimal*-Darstellung jedoch in der Little-Endian-Codierung.

Um dieses Problem zu lösen haben wir vorher den op-Code `std` verwendet. Wir machen diesen Aufruf später mittels `cld` wieder rückgängig: Es ist sehr wichtig, daß wir keine Funktion mittels `call` aufrufen, die von einer Standardeinstellung des *Richtungsflags* abhängig ist, während `std` ausgeführt wird.

Alles weitere in dem Programm sollte leicht zu verstehen sein, vorausgesetzt, daß Sie das gesamte vorherige Kapitel gelesen haben.

Es ist ein klassisches Beispiel für das Sprichwort, daß das Programmieren eine Menge Denkarbeit, und nur ein wenig Programmcode benötigt. Sobald wir uns über jedes Detail im klaren sind, steht der Code fast schon da.

11.13.6. Das Programm pinhole verwenden

Da wir uns bei dem Programm dafür entschieden haben, alle Eingaben, die keine Zahlen sind, zu ignorieren (selbst die in Kommentaren), können wir jegliche *textbasierten Eingaben* verarbeiten. Wir *müssen* dies nicht tun, wir *könnten* aber.

Meiner bescheidenen Meinung nach wird ein Programm durch die Möglichkeit, anstatt einer strikten Eingabesyntax textbasierte Anfragen stellen zu können, sehr viel benutzerfreundlicher.

Angenommen, wir wollten eine Lochkamera für einen 4x5 Zoll Film bauen. Die standardmäßige Brennweite für diesen Film ist ungefähr 150mm. Wir wollen diesen Wert *optimieren*, so daß der Lochblendendurchmesser eine möglichst runde Zahl ergibt. Lassen Sie uns weiter annehmen, daß wir zwar sehr gut mit Kameras umgehen können, dafür aber nicht so gut mit Computern. Anstatt das wir nun eine Reihe von Zahlen eingeben, wollen wir lieber ein paar *Fragen* stellen.

Unsere Sitzung könnte wie folgt aussehen:

```
% pinhole

Computer,

Wie groß müßte meine Lochblende bei einer Brennweite
von 150 sein?
150 490 306 362 2930    12
Hmmm... Und bei 160?
160 506 316 362 3125    12
Laß uns bitte 155 nehmen.
155 498 311 362 3027    12
Ah, laß uns 157 probieren...
```

```
157 501 313 362 3066    12
156?
156 500 312 362 3047    12
Das ist es! Perfekt! Vielen Dank!
^D
```

Wir haben herausgefunden, daß der Lochblendendurchmesser bei einer Brennweite von 150 mm 490 Mikrometer, oder 0.49 mm ergeben würde. Bei einer fast identischen Brennweite von 156 mm würden wir einen Durchmesser von genau einem halben Millimeter bekommen.

11.13.7. Skripte schreiben

Da wir uns dafür entschieden haben, das Zeichen # als den Anfang eines Kommentares zu interpretieren, können wir unser pinhole-Programm auch als *Skriptsprache* verwenden.

Sie haben vielleicht schon einmal shell -*Skripte* gesehen, die mit folgenden Zeichen begonnen haben:

```
#! /bin/sh
```

oder

```
#!/bin/sh
```

i. da das Leerzeichen hinter dem `#!` optional ist.

Wann immer UNIX® eine Datei ausführen soll, die mit einem `#!` beginnt, wird angenommen, das die Datei ein Skript ist. Es fügt den Befehl an das Ende der ersten Zeile an, und versucht dann, dieses auszuführen.

Angenommen, wir haben unser Programm pinhole unter /usr/local/bin/ installiert, dann können wir nun Skripte schreiben, um unterschiedliche Lochblendendurchmesser für mehrere Brennweiten zu berechnen, die normalerweise mit 120er Filmen verwendet werden.

Das Skript könnte wie folgt aussehen:

```
#! /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
```

Da ein 120er Film ein Film mittlerer Größe ist, könnten wir die Datei medium nennen.

Wir können die Datei ausführbar machen und dann aufrufen, als wäre es ein Programm:

```
% chmod 755 medium
% ./medium
```

UNIX® wird den letzten Befehl wie folgt interpretieren:

```
% /usr/local/bin/pinhole -b -i ./medium
```

Es wird den Befehl ausführen und folgendes ausgeben:

```
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
```

Lassen Sie uns nun das folgende eingeben:

```
% ./medium -c
```

UNIX® wird dieses wie folgt behandeln:

```
% /usr/local/bin/pinhole -b -i ./medium -c
```

Dadurch erhält das Programm zwei widersprüchliche Optionen: -b und -c (Verwende Benders Konstante und verwende Connors Konstante). Wir haben unser Programm so geschrieben, daß später eingelesene Optionen die vorheringen überschreiben-unser Programm wird also Connors Konstante für die Berechnungen verwenden:

```
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
```

Wir entscheiden uns am Ende doch für Benders Konstante. Wir wollen die Ergebnisse im CSV-Format in einer Datei speichern:

```
% ./medium -b -e > bender
% cat bender
focal length in millimeters,pinhole diameter in microns,F-number,normalized F-
number,F-5.6 multiplier,stops from F-5.6
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. Vorsichtsmassnahmen

Assembler-Programmierer, die aufwuchsen mit MS-DOS® und windows Windows® neigen oft dazu Shotcuts zu verwenden. Das Lesen der Tastatur-Scancodes und das direkte Schreiben in den Grafikspeicher sind zwei klassische Beispiele von Gewohnheiten, die unter MS-DOS® nicht verpönt sind, aber nicht als richtig angesehen werden.

Warum dies? Sowohl das PC-BIOS als auch MS-DOS® sind notorisch langsam bei der Ausführung dieser Operationen.

Sie mögen versucht sein ähnliche Angewohnheiten in der UNIX®-Umgebung fortzuführen. Zum Beispiel habe ich eine Webseite gesehen, welche erklärt, wie man auf einem beliebten UNIX®-Ableger die Tastatur-Scancodes verwendet.

Das ist generell eine *sehr schlechte Idee* in einer UNIX®-Umgebung! Lassen Sie mich erklären warum.

11.14.1. UNIX® ist geschützt

Zum Einen mag es schlicht nicht möglich sein. UNIX® läuft im Protected Mode. Nur der Kernel und Gerätetreiber dürfen direkt auf die Hardware zugreifen. Unter Umständen erlaubt es Ihnen ein bestimmter UNIX®-Ableger Tastatur-Scancodes auszulesen, aber ein wirkliches UNIX®-Betriebssystem wird dies zu verhindern wissen. Und falls eine Version es Ihnen erlaubt wird es eine andere nicht tun, daher kann eine sorgfältig erstellte Software über Nacht zu einem überkommenen Dinosaurier werden.

11.14.2. UNIX® ist eine Abstraktion

Aber es gibt einen viel wichtigeren Grund, weshalb Sie nicht versuchen sollten, die Hardware direkt anzusprechen (natürlich nicht, wenn Sie einen Gerätetreiber schreiben), selbst auf den UNIX®-ähnlichen Systemen, die es Ihnen erlauben:

UNIX® ist eine Abstraktion!

Es gibt einen wichtigen Unterschied in der Design-Philosophie zwischen MS-DOS® und UNIX®. MS-DOS® wurde entworfen als Einzelnutzer-System. Es läuft auf einem Rechner mit einer direkt angeschlossenen Tastatur und einem direkt angeschlossenem Bildschirm. Die Eingaben des Nutzers kommen nahezu immer von dieser Tastatur. Die Ausgabe Ihres Programmes erscheint fast immer auf diesem Bildschirm.

Dies ist NIEMALS garantiert unter UNIX®. Es ist sehr verbreitet für ein UNIX®, daß der Nutzer seine Aus- und Eingaben kanalisiert und umleitet:

```
% program1 | program2 | program3 > file1
```

Falls Sie eine Anwendung program2 geschrieben haben, kommt ihre Eingabe nicht von der Tastatur, sondern von der Ausgabe von program1. Gleichermassen geht Ihre Ausgabe nicht auf den Bildschirm, sondern wird zur Eingabe für program3, dessen Ausgabe wiederum in file1 endet.

Aber es gibt noch mehr! Selbst wenn Sie sichergestellt haben, daß Ihre Eingabe und Ausgabe zum Terminal kommt bzw. gelangt, dann ist immer noch nicht garantiert, daß ihr Terminal ein PC ist: Es mag seinen Grafikspeicher nicht dort haben, wo Sie ihn erwarten, oder die Tastatur könnte keine PC-ähnlichen Scancodes erzeugen können. Es mag ein Macintosh® oder irgendein anderer Rechner sein.

Sie mögen nun den Kopf schütteln: Mein Programm ist in PC-Assembler geschrieben, wie kann es auf einem Macintosh® laufen? Aber ich habe nicht gesagt, daß Ihr Programm auf Macintosh® läuft, nur sein Terminal mag ein Macintosh® sein.

Unter UNIX® muß der Terminal nicht direkt am Rechner angeschlossen sein, auf dem die Software läuft, er kann sogar auf einem anderen Kontinent sein oder sogar auf einem anderen Planeten. Es ist nicht ungewöhnlich, daß ein Macintosh®-Nutzer in Australien sich auf ein UNIX®-System in Nordamerika (oder sonstwo) mittels telnet verbindet. Die Software läuft auf einem Rechner während das Terminal sich auf einem anderen Rechner befindet: Falls Sie versuchen sollten die Scancodes auszulesen werden Sie die falschen Eingaben erhalten!

Das Gleiche gilt für jede andere Hardware: Eine Datei, welche Sie einlesen, mag auf einem Laufwerk sein, auf das Sie keinen direkten Zugriff haben. Eine Kamera, deren Bilder Sie auslesen, befindet sich möglicherweise in einem Space Shuttle, durch Satelliten mit Ihnen verbunden.

Das sind die Gründe, weshalb Sie niemals unter UNIX® Annahmen treffen dürfen, woher Ihre Daten kommen oder gehen. Lassen Sie immer das System den physischen Zugriff auf die Hardware regeln.

Das sind Vorsichtsmassnahmen, keine absoluten Regeln. Ausnahmen sind möglich.

> Wenn zum Beispiel ein Texteditor bestimmt hat, daß er auf einer lokalen Maschine läuft, dann mag er die Tastatur-Scancodes direkt auslesen, um eine bessere Kontrolle zu gewährleisten. Ich erwähne diese Vorsichtsmassnahmen nicht, um Ihnen zu sagen, was sie tun oder lassen sollen, ich will Ihnen nur bewusst machen, daß es bestimmte Fallstricke gibt, die Sie erwarten, wenn Sie soeben ihn UNIX® von MS-DOS® angelangt sind. Kreative Menschen brechen oft Regeln und das ist in Ordnung, solange sie wissen welche Regeln und warum.

11.15. Danksagungen

Dieses Handbuch wäre niemals möglich gewesen ohne die Hilfe vieler erfahrener FreeBSD-Programmierer aus FreeBSD technical discussions. Viele dieser Personen haben geduldig meine Fragen beantwortet und mich in die richtige Richtung gewiesen bei meinem Versuch, die tieferen liegenden Mechanismen der UNIX®-Systemprogrammierung zu erforschen im Allgemeinen und bei FreeBSD im Besonderen.

Thomas M. Sommers öffnete die Türen für mich. Seine Wie schreibe ich "Hallo Welt" in FreeBSD-Assembler? Webseite war mein erster Kontakt mit Assembler-Programmierung unter FreeBSD.

Jake Burkholder hat die Tür offen gehalten durch das bereitwillige Beantworten all meiner Fragen und das Zurverfügungstellen von Assembler-Codebeispielen.

Copyright ® 2000-2001 G. Adam Stanislav. Alle Rechte vorbehalten.

Teil V: Anhang

Literaturverzeichnis

[1] Dave A Patterson and John L Hennessy. Copyright® 1998 Morgan Kaufmann Publishers, Inc. 1-55860-428-6. Morgan Kaufmann Publishers, Inc. Computer Organization and Design. The Hardware / Software Interface. 1-2.

[2] W. Richard Stevens. Copyright® 1993 Addison Wesley Longman, Inc. 0-201-56317-7. Addison Wesley Longman, Inc. Advanced Programming in the Unix Environment. 1-2.

[3] Marshall Kirk McKusick and George Neville-Neil. Copyright® 2004 Addison-Wesley. 0-201-70245-2. Addison-Wesley. The Design and Implementation of the FreeBSD Operating System. 1-2.

[4] Aleph One. Phrack 49; "Smashing the Stack for Fun and Profit".

[5] Chrispin Cowan, Calton Pu, and Dave Maier. StackGuard; Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.

[6] Todd Miller and Theo de Raadt. strlcpy and strlcat — consistent, safe string copy and concatenation.