Руководство разработчика FreeBSD

Аннотация

Добро пожаловать в Руководство разработчика. Этот документ находится в *процессе разработки* и создаётся усилиями многих людей. Многие разделы пока отсутствуют, а существующие нуждаются в обновлении. Если вы хотите помочь с этим проектом, отправьте письмо на Список рассылки Проекта Документации FreeBSD.

Последняя версия этого документа всегда доступна по ссылке веб-сервер FreeBSD. Его также можно загрузить в различных форматах и с разными вариантами сжатия с сервера загрузки FreeBSD или одного из многочисленных зеркальных сайтов.

Содержание

Часть I: Основы

Глава 1. Введение

1.1. Разработка на FreeBSD

Вот мы и здесь. Система установлена, и вы готовы начать программировать. Но с чего начать? Что предоставляет FreeBSD? Что она может сделать для меня как для программиста?

Вот некоторые вопросы, на которые эта глава пытается ответить. Конечно, программирование, как и любое другое ремесло, имеет разные уровни мастерства. Для кого-то это хобби, для других — профессия. Информация в этой главе может быть ориентирована на начинающего программиста; действительно, она может быть полезна программисту, не знакомому с платформой FreeBSD.

1.2. Видение BSD

Создать наилучший пакет операционной системы, подобной UNIX®, с должным уважением к оригинальной идеологии программных инструментов, а также к удобству использования, производительности и стабильности.

1.3. Архитектурные рекомендации

Наша идеология может быть описана следующими принципами

- Не добавляйте новую функциональность, если разработчик не может завершить реальное приложение без неё.
- Важно не только определить, чем является система, но и чем она не является. Не стоит пытаться удовлетворить все возможные потребности; вместо этого сделайте систему расширяемой, чтобы дополнительные требования могли быть реализованы с сохранением совместимости.
- Худшее, чем обобщение на основе одного примера — это обобщение без примеров вообще.
- Если проблема не до конца понятна, вероятно, лучше вообще не предоставлять решения.
- Если вы можете получить 90% желаемого эффекта за 10% работы, используйте более простое решение.
- Изолируйте сложность настолько, насколько это возможно.
- Предоставлять механизмы, а не политики. В частности, передайте политику пользовательского интерфейса в руки клиента.

Из Шейфлера и Геттиса: «X Window System»

1.4. Структура каталога /usr/src

Полный исходный код FreeBSD доступен в нашем публичном Git-репозитории. Исходный код обычно устанавливается в /usr/src. Структура дерева каталогов исходного кода описана в файле README.md на верхнем уровне дерева.

Глава 2. Инструменты разработки

2.1. Обзор

В этой главе представлено введение в использование некоторых инструментов для программирования, поставляемых с FreeBSD, хотя многое из описанного применимо и к другим версиям UNIX®. Она *не* претендует на детальное описание процесса написания кода. Большая часть главы предполагает наличие минимальных или отсутствие знаний в программировании, хотя предполагается, что даже опытные программисты найдут в ней что-то полезное.

2.2. Введение

FreeBSD предоставляет отличную среду разработки. Компиляторы для C и C++, а также ассемблер входят в базовую систему, не говоря уже о классических инструментах UNIX®, таких как `sed` и `awk`. Если этого недостаточно, в коллекции Ports доступно множество других компиляторов и интерпретаторов. В следующем разделе, Введение в программирование, перечислены некоторые из доступных вариантов. FreeBSD обладает высокой совместимостью со стандартами, такими как POSIX® и ANSI C, а также с собственным наследием BSD, что позволяет создавать приложения, которые будут компилироваться и запускаться с минимальными изменениями или без них на широком спектре платформ.

Однако вся эта мощь может поначалу ошеломить, если вы никогда раньше не писали программы на платформе UNIX®. Этот документ призван помочь вам начать работу, не углубляясь слишком сильно в более сложные темы. Цель заключается в том, чтобы дать вам достаточно базовых знаний для понимания документации.

Большая часть документа не требует или почти не требует знаний программирования, хотя предполагает базовые навыки работы с UNIX® и готовность учиться!

2.3. Введение в программирование

Программа — это набор инструкций, которые указывают компьютеру выполнять различные действия; иногда выполняемая инструкция зависит от результата предыдущей. В этом разделе представлен обзор двух основных способов передачи таких инструкций, или, как их обычно называют, «команд». Один способ использует *интерпретатор*, другой — *компилятор*. Поскольку человеческие языки слишком сложны для однозначного понимания компьютером, команды обычно записываются на одном из специально разработанных для этого языков.

2.3.1. Интерпретаторы

С интерпретатором язык поставляется как среда, в которой вы вводите команды в приглашении, и среда выполняет их для вас. Для более сложных программ вы можете ввести команды в файл и заставить интерпретатор загрузить файл и выполнить команды в нём. Если что-то пойдёт не так, многие интерпретаторы переведут вас в отладчик, чтобы

помочь найти проблему.

Преимущество этого подхода в том, что вы сразу видите результаты выполнения команд, а ошибки можно легко исправить. Самый большой недостаток проявляется, когда вы хотите поделиться своими программами с кем-то. У них должен быть такой же интерпретатор, или у вас должен быть способ предоставить его, и они должны понимать, как им пользоваться. Кроме того, пользователям может не понравиться, если они попадут в отладчик при нажатии не той клавиши! С точки зрения производительности интерпретаторы могут потреблять много памяти и обычно генерируют код менее эффективно, чем компиляторы.

По моему мнению, интерпретируемые языки — это лучший способ начать, если вы раньше не занимались программированием. Такая среда обычно встречается в языках вроде Lisp, Smalltalk, Perl и Basic. Можно также утверждать, что UNIX® shell (`sh`, `csh`) сам по себе является интерпретатором, и многие часто пишут shell-«скрипты» для помощи в различных «хозяйственных» задачах на своих машинах. Действительно, часть оригинальной философии UNIX® заключалась в предоставлении множества небольших утилит, которые можно было связывать вместе в shell-скриптах для выполнения полезных задач.

2.3.2. Доступные интерпретаторы в FreeBSD

Вот список интерпретаторов, доступных в Коллекции портов FreeBSD, с кратким обзором некоторых наиболее популярных интерпретируемых языков.

Инструкции по получению и установке приложений из Коллекции портов можно найти в разделе Порты руководства.

BASIC

Сокращение от Beginner's All-purpose Symbolic Instruction Code. Разработан в 1950-х годах для обучения студентов университетов программированию и поставлялся с каждым уважающим себя персональным компьютером в 1980-х. BASIC — первый язык программирования для многих программистов. Он также является основой для Visual Basic.

Интерпретатор Bywater Basic можно найти в Коллекции портов как lang/bwbasic, а интерпретатор Phil Cockroft's Basic (ранее известный как Rabbit Basic) доступен как lang/pbasic.

Lisp

Язык, разработанный в конце 1950-х годов как альтернатива популярным в то время языкам для «численных расчётов». В отличие от них, Lisp основан на списках; фактически, название является сокращением от «List Processing» (обработка списков). Он очень популярен в кругах, связанных с ИИ (искусственным интеллектом).

Lisp — это чрезвычайно мощный и сложный язык, но он может показаться довольно большим и громоздким.

В Коллекции портов FreeBSD доступны различные реализации Lisp, которые могут работать в системах UNIX®. CLISP от Bruno Haible и Michael Stoll доступен как lang/clisp. Более простая реализация Lisp, SLisp, доступна как lang/slisp.

Perl

Очень популярен среди системных администраторов для написания скриптов; также часто используется на веб-серверах для написания CGI-скриптов.

Perl доступен в Коллекции портов как lang/perl5.36 для всех выпусков FreeBSD.

Scheme

Диалект Lisp, который более компактен и чист по сравнению с Common Lisp. Популярен в университетах, так как достаточно прост для обучения студентов в качестве первого языка, и при этом обладает достаточным уровнем абстракции для использования в исследовательской работе.

Схема доступна из Коллекции Портов как lang/elk для Интерпретатора Elk Scheme. Интерпретатор MIT Scheme можно найти в lang/mit-scheme, а Интерпретатор SCM Scheme — в lang/scm.

Lua

Lua — это легковесный встраиваемый язык сценариев. Он обладает высокой переносимостью и относительно прост. Lua доступен в коллекции портов в пакете lang/lua54. Он также включен в базовую систему как /usr/libexec/flua для использования компонентами базовой системы. Стороннее программное обеспечение не должно зависеть от flua.

Python

Python — это объектно-ориентированный интерпретируемый язык. Его сторонники утверждают, что это один из лучших языков для начала программирования, поскольку он относительно прост в освоении, но не уступает другим популярным интерпретируемым языкам, используемым для разработки крупных и сложных приложений (Perl и Tcl — два других языка, популярных для таких задач).

Последняя версия Python доступна в Коллекции портов в пакете lang/python.

Ruby

Ruby — это интерпретируемый, чисто объектно-ориентированный язык программирования. Он получил широкую популярность благодаря простому для понимания синтаксису, гибкости при написании кода и возможности легко разрабатывать и поддерживать большие, сложные программы.

Ruby доступен в Коллекции портов как lang/ruby32.

Tcl и Tk

Tcl — это встраиваемый интерпретируемый язык, который получил широкое распространение и популярность в основном благодаря своей переносимости на множество платформ. Он может использоваться как для быстрого написания небольших прототипов приложений, так и (в сочетании с Tk, набором инструментов для графического интерфейса) полноценных программ с богатым функционалом.

Различные версии Tcl доступны в качестве портов для FreeBSD. Последняя версия, Tcl 8.7, находится в пакете lang/tcl87.

2.3.3. Компиляторы

Компиляторы довольно сильно различаются. Прежде всего, вы пишете свой код в файле (или файлах) с помощью редактора. Затем вы запускаете компилятор и проверяете, принимает ли он вашу программу. Если программа не скомпилировалась, стисните зубы и вернитесь к редактору; если же компиляция прошла успешно и программа была создана, вы можете запустить её либо в командной строке оболочки, либо в отладчике, чтобы проверить её работу.[1]

Очевидно, это требует больше усилий по сравнению с использованием интерпретатора. Однако это позволяет делать множество вещей, которые очень сложны или даже невозможны с интерпретатором, например, писать код, тесно взаимодействующий с операционной системой — или даже создавать собственную операционную систему! Это также полезно, если требуется написать очень эффективный код, так как компилятор может не спешить и оптимизировать код, что было бы неприемлемо в интерпретаторе. Более того, распространение программы, написанной для компилятора, обычно проще, чем для интерпретатора — можно просто предоставить копию исполняемого файла, предполагая, что у пользователя та же операционная система, что и у вас.

Поскольку цикл редактирования-компиляции-запуска-отладки довольно утомителен при использовании отдельных программ, многие производители коммерческих компиляторов создали интегрированные среды разработки (сокращённо IDE). FreeBSD не включает IDE в базовую систему, но в Коллекции портов доступен devel/kdevelop, и многие используют для этой цели Emacs. Использование Emacs в качестве IDE обсуждается в Использование Emacs как среды разработки.

2.4. Компиляция с помощью cc

Этот раздел посвящён компилятору clang для языков C и C++, так как он устанавливается вместе с базовой системой FreeBSD. Clang устанавливается как `cc`; пакет GNU-компилятора gcc доступен в Коллекции портов. Детали создания программы с интерпретатором значительно различаются в зависимости от интерпретатора и обычно хорошо описаны в документации и онлайн-справке интерпретатора.

Как только вы напишете свой шедевр, следующий шаг — преобразовать его во что-то, что (надеюсь!) будет работать на FreeBSD. Обычно это включает несколько шагов, каждый из которых выполняется отдельной программой.

1. Обработать исходный код, чтобы удалить комментарии и выполнить другие действия, такие как раскрытие макросов в C.
2. Проверить синтаксис вашего кода, чтобы убедиться, что вы соблюдаете правила языка. Если нет, он пожалуется!
3. Преобразовать исходный код в ассемблерный язык — это очень близко к машинному коду, но всё ещё понятно человеку. Как утверждается.
4. Преобразовать язык ассемблера в машинный код — да, здесь речь идёт о битах и байтах, единицах и нулях.
5. Проверить, что вы использовали такие элементы, как функции и глобальные

переменные, правильно и последовательно. Например, если вы вызвали несуществующую функцию, это будет отмечено.

6. Если вы пытаетесь создать исполняемый файл из нескольких исходных файлов, определить, как объединить их все вместе.
7. Определить, как создать что-то, что загрузчик времени выполнения системы сможет загрузить в память и запустить.
8. Наконец, записать исполняемый файл в файловую систему.

Слово *компиляция* часто относится только к шагам с 1 по 4, а остальные шаги называются *линковкой*. Иногда шаг 1 называют *препроцессированием*, а шаги 3-4 — *ассемблированием*.

К счастью, почти все эти детали скрыты от вас, так как `cc` — это интерфейс, который управляет вызовом всех этих программ с правильными аргументами за вас; достаточно просто набрать

```
% cc foobar.c
```

и это вызовет компиляцию файла foobar.c всеми перечисленными выше шагами. Если у вас несколько файлов для компиляции, просто сделайте что-то вроде

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

Обратите внимание, что проверка синтаксиса — это всего лишь проверка синтаксиса. Она не выявит логических ошибок, которые вы могли допустить, например, создание бесконечного цикла или использование пузырьковой сортировки вместо бинарной.[2]

Существует множество опций для `cc`, все они описаны в руководстве. Вот несколько наиболее важных из них с примерами использования.

`-o filename`

Имя выходного файла. Если вы не используете эту опцию, `cc` создаст исполняемый файл с именем a.out.[3]

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

`-c`

Просто скомпилирует файл, не связывая его. Полезно для небольших программ, где нужно только проверить синтаксис, или если вы используете Makefile.

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

Это создаст *объектный файл* (не исполняемый) с именем foobar.o. Его можно скомпоновать с другими объектными файлами в исполняемый файл.

`-g`

Создать отладочную версию исполняемого файла. Это заставляет компилятор записывать в исполняемый файл информацию о том, какая строка какого исходного файла соответствует какому вызову функции. Отладчик может использовать эту информацию для отображения исходного кода при пошаговом выполнении программы, что *очень* полезно; недостатком является то, что вся эта дополнительная информация значительно увеличивает размер программы. Обычно вы компилируете с `-g` во время разработки программы, а затем компилируете "релизную версию" без `-g`, когда убедитесь, что она работает правильно.

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

Это создаст отладочную версию программы. [4]

`-O`

Создаёт оптимизированную версию исполняемого файла. Компилятор применяет различные хитрые приёмы, чтобы попытаться создать исполняемый файл, который работает быстрее обычного. Вы можете добавить число после `-O`, чтобы указать более высокий уровень оптимизации, но это часто выявляет ошибки в оптимизаторе компилятора.

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

Это создаст оптимизированную версию foobar.

Следующие три флага заставят `cc` проверять, что ваш код соответствует соответствующему международному стандарту, часто называемому стандартом ANSI, хотя строго говоря, это стандарт ISO.

`-Wall`

Включить все предупреждения, которые разработчики `cc` считают полезными. Несмотря на название, это не включит все предупреждения, которые `cc` способен выдавать.

`-ansi`

Отключит большинство, но не все, не-ANSI C функции, предоставляемые `cc`. Несмотря на название, это не гарантирует строгого соответствия вашего кода стандарту.

`-pedantic`

Отключит *все* не-ANSI C возможности `cc`.

Без этих флагов `cc` позволит вам использовать некоторые из своих нестандартных расширений стандарта. Некоторые из них очень полезны, но не будут работать с другими компиляторами — фактически, одна из основных целей стандарта заключается в том, чтобы позволить людям писать код, который будет работать с любым компилятором на любой системе. Это известно как *переносимый код*.

Обычно следует стремиться к тому, чтобы ваш код был как можно более переносимым,

иначе позже вам, возможно, придётся полностью переписать программу для её работы в другом месте — а кто знает, что вы будете использовать через несколько лет?

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

В результате будет создан исполняемый файл foobar после проверки foobar.c на соответствие стандартам.

`-llibrary`

Укажите библиотеку функций, которая будет использоваться во время компоновки.

Наиболее распространённый пример этого — компиляция программы, использующей некоторые математические функции в C. В отличие от большинства других платформ, они находятся в отдельной библиотеке, отличной от стандартной библиотеки C, и необходимо указать компилятору добавить её.

Правило заключается в том, что если библиотека называется libsomething.a, то вы передаёте `cc` аргумент `-lsomething`. Например, математическая библиотека называется libm.a, поэтому вы передаёте `cc` аргумент `-lm`. Типичный подводный камень с математической библиотекой заключается в том, что она должна быть последней библиотекой в командной строке.

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

Это приведёт к подключению функций математической библиотеки в foobar.

Если вы компилируете код на C++, используйте c++. c++ также может быть вызван как clang++ в FreeBSD.

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

Это создаст исполняемый файл foobar из исходного файла на C++ foobar.cc.

2.4.1. Распространённые вопросы и проблемы `cc`

2.4.1.1. Я скомпилировал файл с именем foobar.c и не могу найти исполняемый файл с именем foobar. Куда он пропал?

Помните, что `cc` вызовет исполняемый файл a.out, если вы не укажете иное. Используйте опцию `-o имя_файла`:

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

2.4.1.2. Хорошо, у меня есть исполняемый файл с именем foobar, я вижу его при выполнении команды ls, но когда я ввожу foobar в командной строке, система сообщает, что такого файла нет. Почему он не может его найти?

В отличие от MS-DOS®, UNIX® не ищет в текущем каталоге, когда пытается определить, какую программу нужно запустить, если вы явно не укажете это. Введите `./foobar`, что означает "запустить файл с именем foobar в текущем каталоге."

2.4.2. Я назвал свой исполняемый файл test, но при запуске ничего не происходит. В чем дело?

Большинство UNIX® систем имеют программу под названием `test` в /usr/bin, и оболочка выбирает её, прежде чем проверить текущий каталог. Введите следующее:

```
% ./test
```

или выберите более подходящее название для вашей программы!

2.4.2.1. Я скомпилировал свою программу, и сначала она работала нормально, но потом произошла ошибка, и было сообщение о core dumped. Что это значит?

Название *core dump* восходит к самым ранним дням UNIX®, когда машины использовали ферритовую память для хранения данных. По сути, если программа завершалась сбоем при определённых условиях, система записывала содержимое ферритовой памяти на диск в файл с именем core, который программист затем мог изучить, чтобы выяснить причину ошибки.

2.4.2.2. Увлекательный материал, но что мне теперь делать?

Используйте отладчик для анализа образа памяти (см. Отладка).

2.4.2.3. Когда моя программа сбросила core, она сообщила что-то о segmentation fault. Что это?

Это означает, что ваша программа попыталась выполнить какую-то недопустимую операцию с памятью; UNIX® разработана для защиты операционной системы и других программ от некорректно работающих программ.

Распространенные причины этого:

- Попытка записи в NULL-указатель, например:

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

- Использование неинициализированного указателя, например:

```
char *foo;
```

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

Указатель будет иметь случайное значение, которое, возможно, укажет на область памяти, недоступную вашей программе, и ядро завершит вашу программу до того, как она сможет нанести какой-либо ущерб. Если вам не повезет, он укажет внутрь вашей собственной программы и повредит одну из структур данных, что приведет к загадочному сбою программы.

- Попытка доступа за пределы массива, например

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

- Попытка сохранить что-то в память только для чтения, например

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

Версии UNIX® компиляторы часто помещают строковые литералы, такие как `"Моя строка"`, в области памяти только для чтения.

- Выполнение нежелательных действий с `malloc()` и `free()`, например

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

или

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

Совершение одной из этих ошибок не всегда приведет к сбою, но это всегда плохая практика. Некоторые системы и компиляторы более терпимы, чем другие, поэтому программы, которые хорошо работают на одной системе, могут аварийно завершаться при попытке запустить их на другой.

2.4.2.4. Иногда при получении дампа памяти я вижу сообщение ошибки шины (bus error). В моей книге по UNIX® сказано, что это означает аппаратную проблему, но компьютер продолжает работать. Это правда?

Нет, к счастью, нет (если, конечно, у вас действительно нет аппаратной проблемы...). Обычно это означает, что вы обратились к памяти способом, который не следует использовать.

2.4.2.5. Этот процесс создания дампа памяти звучит довольно полезно, если я могу запускать его по своему желанию. Могу ли я это сделать, или нужно ждать возникновения ошибки?

Можете. Просто перейдите на другую консоль или xterm, выполните

```
% ps
```

чтобы узнать идентификатор процесса вашей программы и выполните

```
% kill -ABRT pid
```

где `pid` — идентификатор процесса, который вы нашли.

Это полезно, если ваша программа зависла в бесконечном цикле, например. Если ваша программа перехватывает SIGABRT, есть несколько других сигналов, которые оказывают аналогичный эффект.

В качестве альтернативы, вы можете создать дамп памяти изнутри вашей программы, вызвав функцию `abort()`. Дополнительную информацию можно найти на abort(3).

Если вы хотите создать дамп памяти извне вашей программы, но не хотите завершать процесс, вы можете использовать программу `gcore`. Подробнее см. на странице руководства gcore(1).

2.5. Make

2.5.1. Что такое `make`?

Когда вы работаете над простой программой с одним или двумя исходными файлами, вводя

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

это не слишком плохо, но быстро становится очень утомительным, когда есть несколько файлов — и компиляция тоже может занять время.

Один из способов обойти это — использовать объектные файлы и перекомпилировать исходный файл только в случае изменения исходного кода. Таким образом, у нас может получиться что-то вроде:

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

если бы мы изменили файл file37.c, но не трогали остальные с момента последней компиляции. Это может значительно ускорить компиляцию, но не решает проблему с вводом.

Или мы могли бы написать shell-скрипт для решения проблемы с вводом, но тогда пришлось бы перекомпилировать всё, что сделало бы его очень неэффективным для крупного проекта.

Что произойдет, если у нас есть сотни исходных файлов? Что, если мы работаем в команде с другими людьми, которые забывают сообщить нам, когда они изменили один из своих исходных файлов, которые мы используем?

Возможно, мы могли бы объединить два решения и написать что-то вроде shell-скрипта, который содержал бы какое-то волшебное правило, указывающее, когда исходный файл нужно компилировать. Теперь нам осталось только найти программу, которая сможет понимать эти правила, так как для shell это немного слишком сложно.

Эта программа называется `make`. Она читает файл, называемый *makefile*, который указывает, как различные файлы зависят друг от друга, и определяет, какие файлы нужно перекомпилировать, а какие нет. Например, правило может звучать так: «если fromboz.o старше, чем fromboz.c, значит, кто-то изменил fromboz.c, и его нужно перекомпилировать». В makefile также содержатся правила, указывающие make, *как* именно перекомпилировать исходный файл, что делает эту программу гораздо более мощным инструментом.

Файлы Makefile обычно хранятся в том же каталоге, что и исходный код, к которому они применяются, и могут называться makefile, Makefile или MAKEFILE. Большинство программистов используют имя Makefile, так как это помещает его в начало списка файлов в каталоге, где его легко заметить.[5]

2.5.2. Пример использования `make`

Вот очень простой файл для make:

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

Он состоит из двух строк: строки зависимости и строки создания.

Строка зависимости здесь состоит из имени программы (известного как *цель*), за которым следует двоеточие, пробел и имя исходного файла. Когда `make` читает эту строку, он проверяет, существует ли файл foo; если он существует, программа сравнивает время последнего изменения файла foo с временем последнего изменения файла foo.c. Если файл foo не существует или старше файла foo.c, программа смотрит на строку создания, чтобы выяснить, что делать. Другими словами, это правило для определения, когда файл foo.c нужно перекомпилировать.

Строка создания начинается с табуляции (нажмите `tab`), а затем следует команда, которую вы бы ввели для создания foo, если бы делали это в командной строке. Если foo устарел или не существует, `make` выполняет эту команду для его создания. Другими словами, это правило, которое сообщает make, как перекомпилировать foo.c.

Таким образом, при вводе команды `make` система обеспечит актуальность файла foo относительно последних изменений в foo.c. Этот принцип можно распространить на

Makefile, содержащие сотни целей — фактически, в FreeBSD можно собрать всю операционную систему, просто введя `make buildworld buildkernel` в корневом каталоге дерева исходных кодов (src).

Еще одно полезное свойство makefile заключается в том, что цели не обязательно должны быть программами. Например, у нас может быть makefile, который выглядит так:

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

install:
    cp foo /home/me
```

Мы можем указать make, какую цель мы хотим собрать, набрав:

```
% make target
```

`make` будет рассматривать только указанную цель и игнорировать все остальные. Например, если мы введём `make foo` с указанным выше makefile, make проигнорирует цель `install`.

Если мы просто введем `make` без параметров, make всегда будет обращаться к первой цели и затем остановится, не рассматривая остальные. Поэтому если мы введем `make` здесь, он просто перейдет к цели `foo`, перекомпилирует foo при необходимости и затем остановится, не переходя к цели `install`.

Обратите внимание, что цель `install` не зависит ни от чего! Это означает, что команда в следующей строке всегда выполняется при попытке создать эту цель с помощью команды `make install`. В данном случае она скопирует foo в домашний каталог пользователя. Это часто используется в makefile приложений, чтобы приложение можно было установить в правильный каталог после успешной компиляции.

Это немного запутанная тема для объяснения. Если вы не до конца понимаете, как работает `make`, лучше всего написать простую программу, например, "hello world", и make-файл, как указано выше, и поэкспериментировать. Затем можно перейти к использованию нескольких исходных файлов или добавлению заголовочного файла в исходный код. В этом случае очень полезен `touch` — он изменяет дату файла без необходимости его редактирования.

2.5.3. make и include-файлы

Код на C часто начинается со списка подключаемых файлов, например stdio.h. Некоторые из этих файлов являются системными, а некоторые принадлежат текущему проекту:

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

```
int main(....
```

Чтобы убедиться, что этот файл перекомпилируется при изменении foo.h, необходимо добавить его в Makefile:

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

В момент, когда ваш проект становится больше и у вас появляется все больше собственных включаемых файлов для поддержки, отслеживание всех включаемых файлов и файлов, которые от них зависят, становится проблемой. Если вы измените включаемый файл, но забудете перекомпилировать все файлы, которые от него зависят, последствия будут катастрофическими. У `clang` есть опция для анализа ваших файлов и создания списка включаемых файлов и их зависимостей: `-MM`.

Если вы добавите это в ваш Makefile:

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

и выполните `make depend`, появится файл .depend со списком объектных файлов, C-файлов и включаемых файлов:

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

Если вы измените файл foo.h, при следующем запуске `make` все файлы, зависящие от foo.h, будут перекомпилированы.

Не забудьте выполнить `make depend` каждый раз, когда вы добавляете include-файл в один из своих файлов.

2.5.4. Файлы Makefile системы FreeBSD

Makefile-ы могут быть довольно сложными для написания. К счастью, в BSD-системах, таких как FreeBSD, есть очень мощные Makefile-ы, поставляемые в составе системы. Отличным примером этого является система портов FreeBSD. Вот основная часть типичного Makefile для портов:

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

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

Теперь, если мы перейдем в каталог этого порта и наберем `make`, произойдет следующее:

1. Проверяется, есть ли исходный код этого порта уже в системе.

2. Если это не так, устанавливается FTP-соединение с URL в MASTER_SITES для загрузки исходного кода.
3. Контрольная сумма исходного кода вычисляется и сравнивается с контрольной суммой известной и хорошей копии исходного кода. Это делается для того, чтобы убедиться, что исходный код не был поврежден во время передачи.
4. Все необходимые изменения для адаптации исходного кода к работе в FreeBSD применяются — это называется применением *патча*.
5. Любая необходимая специальная настройка для исходного кода выполнена. (Многие дистрибутивы программ UNIX® пытаются определить, на какой версии UNIX® они компилируются и какие дополнительные функции UNIX® доступны — именно здесь они получают эту информацию в сценарии портов FreeBSD).
6. Компилируется исходный код программы. По сути, мы переходим в каталог, куда были распакованы исходные файлы, и выполняем `make` — собственный make-файл программы содержит необходимую информацию для сборки программы.
7. Теперь у нас есть скомпилированная версия программы. При желании мы можем протестировать её сейчас; когда мы уверены в программе, можно ввести `make install`. Это приведёт к копированию программы и всех необходимых вспомогательных файлов в нужные места, а также к добавлению записи в `базу данных пакетов`, чтобы позже можно было легко удалить порт, если мы передумаем.

Вот теперь, я думаю, вы согласитесь, что это довольно впечатляюще для скрипта из четырёх строк!

Секрет кроется в последней строке, которая указывает `make` обратиться к системному makefile под названием bsd.port.mk. Эту строку легко пропустить, но именно здесь начинается вся магия — кто-то написал makefile, который предписывает `make` выполнить все вышеперечисленные действия (плюс несколько других, которые я не упомянул, включая обработку возможных ошибок), и любой может получить доступ к этому функционалу, просто добавив одну строку в свой собственный makefile!

Если вы хотите взглянуть на эти системные makefile-ы, они находятся в /usr/share/mk, но, вероятно, лучше подождать, пока у вас не появится немного практики с makefile, так как они очень сложные (и если вы всё же решите их посмотреть, убедитесь, что у вас под рукой есть фляга крепкого кофе!)

2.5.5. Более сложные способы использования `make`

`Make` — это очень мощный инструмент, способный на гораздо большее, чем показано в простом примере выше. К сожалению, существует несколько различных версий `make`, и все они значительно отличаются друг от друга. Лучший способ узнать, на что они способны, — вероятно, прочитать документацию. Надеюсь, это введение дало вам основу, с которой вы сможете это сделать. В make(1) подробно обсуждаются переменные, аргументы и то, как использовать `make`.

Многие приложения в портах используют GNU make, который имеет очень хороший набор страниц "info". Если вы установили любой из этих портов, GNU make будет автоматически установлен как `gmake`. Он также доступен как отдельный порт и пакет.

Для просмотра справочных страниц (info) GNU make вам потребуется отредактировать файл dir в каталоге /usr/local/info, добавив соответствующую запись. Добавьте строку

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

в файл. После этого вы можете ввести `info` и затем выбрать make из меню (или в Emacs выполнить `C-h i`).

2.6. Отладка

2.6.1. Обзор отладчиков, поставляемых в системе

Использование отладчика позволяет запускать программу в более контролируемых условиях. Обычно можно выполнять программу построчно, проверять значения переменных, изменять их, указывать отладчику выполнение до определённой точки и затем останавливаться и так далее. Также можно подключиться к уже работающей программе или загрузить core-файл, чтобы исследовать причину аварийного завершения программы.

Этот раздел представляет собой краткое введение в использование отладчиков и не затрагивает специализированные темы, такие как отладка ядра. Для получения дополнительной информации по этой теме обратитесь к главе Отладка ядра.

Стандартный отладчик, поставляемый с FreeBSD, называется `lldb` (LLVM debugger). Поскольку он является частью стандартной установки для данного выпуска, нет необходимости выполнять какие-либо дополнительные действия для его использования. Он обладает хорошей справкой по командам, доступной через команду `help`, а также руководством и документацией в интернете.

Команда `lldb` также доступна из портов или пакетов как пакет devel/llvm.

Другой отладчик, доступный в FreeBSD, называется `gdb` (GNU debugger). В отличие от lldb, он не устанавливается по умолчанию в FreeBSD; для его использования необходимо установить пакет devel/gdb из портов или пакетов. Он обладает отличной встроенной справкой, а также набором info-страниц.

Два отладчика обладают схожим набором функций, поэтому выбор между ними в основном зависит от личных предпочтений. Если вы знакомы только с одним из них, используйте его. Тем, кто не знаком ни с одним или знаком с обоими, но хочет использовать отладчик внутри Emacs, придётся выбрать `gdb`, так как `lldb` не поддерживается Emacs. В остальных случаях попробуйте оба и решите, какой вам больше нравится.

2.6.2. Использование lldb

2.6.2.1. Запуск lldb

Запустите lldb, набрав

```
% lldb -- progname
```

2.6.2.2. Запуск программы с lldb

Скомпилируйте программу с `-g`, чтобы максимально использовать возможности `lldb`. Без этого флаг она будет работать, но отображать только имя текущей выполняемой функции вместо исходного кода. Если отображается строка вида:

```
Breakpoint 1: where = temp`main, address = ⋯
```

(без указания имени файла исходного кода и номера строки) при установке точки останова это означает, что программа не была скомпилирована с параметром `-g`.

Большинство команд `lldb` имеют более короткие формы, которые можно использовать вместо полных. Здесь используются полные формы для ясности.

На строке `lldb` введите `breakpoint set -n main`. Это укажет отладчику не показывать предварительный код настройки в запускаемой программе и остановить выполнение в начале кода программы. Теперь введите `process launch`, чтобы фактически запустить программу — она начнётся с кода настройки, а затем будет остановлена отладчиком при вызове `main()`.

Для пошагового выполнения программы строка за строкой введите `thread step-over`. Когда программа дойдёт до вызова функции, войдите в неё, набрав `thread step-in`. Оказавшись внутри вызова функции, вернитесь из него с помощью команды `thread step-out` или используйте `up` и `down`, чтобы быстро посмотреть на вызывающий код.

Вот простой пример того, как найти ошибку в программе с помощью `lldb`. Это наша программа (с умышленной ошибкой):

```
#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;
```

```
}
```

Эта программа устанавливает значение `i` равным `5` и передаёт его в функцию `bazz()`, которая выводит переданное число.

Компиляция и запуск программы отображают

```
% cc -g -o temp temp.c
% ./temp
This is my program
anint = -5360
```

Это не то, что ожидалось! Пора разобраться, что происходит!

```
% lldb -- temp
(lldb) target create "temp"
Current executable set to 'temp' (x86_64).
(lldb) breakpoint set -n main               Skip the set-up code
Breakpoint 1: where = temp`main + 15 at temp.c:8:2, address = 0x00000000002012ef
lldb puts breakpoint at main()
(lldb) process launch                   Run as far as main()
Process 9992 launching
Process 9992 launched: '/home/pauamma/tmp/temp' (x86_64)    Program starts running

Process 9992 stopped
* thread #1, name = 'temp', stop reason = breakpoint 1.1    lldb stops at main()
    frame #0: 0x00000000002012ef temp`main at temp.c:8:2
   5    main() {
   6        int i;
   7
-> 8        printf("This is my program\n");         Indicates the line where it
stopped
   9        bazz(i);
   10       return 0;
   11   }
(lldb) thread step-over         Go to next line
This is my program                      Program prints out
Process 9992 stopped
* thread #1, name = 'temp', stop reason = step over
    frame #0: 0x0000000000201300 temp`main at temp.c:9:7
   6        int i;
   7
   8        printf("This is my program\n");
-> 9        bazz(i);
   10       return 0;
   11   }
   12
(lldb) thread step-in           step into bazz()
Process 9992 stopped
```

```
* thread #1, name = 'temp', stop reason = step in
    frame #0: 0x000000000020132b temp`bazz(anint=-5360) at temp.c:14:29 lldb displays
stack frame
   11   }
   12
   13   int bazz(int anint) {
-> 14       printf("You gave me %d\n", anint);
   15       return anint;
   16   }
(lldb)
```

Подождите минуту! Как переменная `int` стала равна `-5360`? Разве она не была установлена в `5` в `main()`? Давайте поднимемся к `main()` и посмотрим.

```
(lldb) up       Move up call stack
frame #1: 0x000000000020130b temp`main at temp.c:9:2        lldb displays stack frame
   6        int i;
   7
   8        printf("This is my program\n");
-> 9        bazz(i);
   10       return 0;
   11   }
   12
(lldb) frame variable i         Show us the value of i
(int) i = -5360                         lldb displays -5360
```

О боже! Глядя на код, мы забыли инициализировать i. Мы хотели добавить

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

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

но мы пропустили строку `i=5;`. Поскольку мы не инициализировали `i`, она содержала любое число, которое оказалось в той области памяти при запуске программы, и в данном случае это оказалось `-5360`.

Команда `lldb` отображает стек вызовов каждый раз, когда мы входим в функцию или выходим из неё, даже если мы используем `up` и `down` для перемещения по стеку вызовов. Это показывает имя функции и значения её аргументов, что помогает отслеживать текущее положение и происходящее. (Стек — это область хранения, где программа сохраняет информацию об аргументах, переданных в функции, и о том, куда возвращаться после вызова функции.)

2.6.2.3. Изучение файла Core с помощью lldb

Файл core — это, по сути, файл, содержащий полное состояние процесса на момент его аварийного завершения. В «старые добрые времена» программистам приходилось распечатывать шестнадцатеричные дампы файлов core и корпеть над руководствами по машинному коду, но сейчас жизнь стала немного проще. Кстати, в FreeBSD и других системах на базе 4.4BSD файл core называется progname.core, а не просто core, чтобы было понятнее, какой программе он принадлежит.

Для анализа файла core укажите имя файла core в дополнение к самой программе. Вместо обычного запуска `lldb` введите `lldb -c имя_программы.core -- имя_программы`.

Отладчик отобразит что-то вроде этого:

```
% lldb -c progname.core -- progname
(lldb) target create "progname" --core "progname.core"
Core file '/home/pauamma/tmp/progname.core' (x86_64) was loaded.
(lldb)
```

В этом случае программа называлась progname, поэтому файл дампа имеет имя progname.core. Отладчик не показывает, почему программа завершилась аварийно или где это произошло. Для этого используйте команду `thread backtrace all`. Она также покажет, как была вызвана функция, в которой программа завершилась дампом ядра.

```
(lldb) thread backtrace all
* thread #1, name = 'progname', stop reason = signal SIGSEGV
  * frame #0: 0x0000000000201347 progname`bazz(anint=5) at temp2.c:17:10
    frame #1: 0x0000000000201312 progname`main at temp2.c:10:2
    frame #2: 0x000000000020110f progname`_start(ap=<unavailable>,
cleanup=<unavailable>) at crt1.c:76:7
(lldb)
```

`SIGSEGV` указывает, что программа пыталась получить доступ к памяти (обычно выполнить код или прочитать/записать данные) по адресу, который ей не принадлежит, но не предоставляет конкретных деталей. Для этого обратитесь к исходному коду на строке 10 файла temp2.c, в функции `bazz()`. Трассировка также показывает, что в данном случае `bazz()` была вызвана из `main()`.

2.6.2.4. Подключение к работающей программе с помощью lldb

Одной из самых замечательных особенностей `lldb` является возможность подключения к уже работающей программе. Конечно, для этого требуются соответствующие разрешения. Распространённая проблема — пошаговое выполнение программы, которая создаёт ответвления, с необходимостью отслеживать дочерний процесс, но отладчик отслеживает только родительский.

Для этого запустите другой `lldb`, используйте `ps` для поиска идентификатора процесса дочернего процесса и выполните

```
(lldb) process attach -p pid
```

в `lldb`, а затем отлаживайте как обычно.

Для того чтобы это работало правильно, код, который вызывает `fork` для создания дочернего процесса, должен делать что-то вроде следующего (предоставлено из документации `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 */
    ...
```

Вот все, что нужно сделать: подключиться к дочернему процессу, установить `PauseMode` в `0` с помощью `expr PauseMode = 0` и дождаться возврата из вызова `sleep()`.

2.6.3. Удалённая отладка с использованием LLDB

Описанная функциональность доступна начиная с версии LLDB 12.0.0. Пользователи релизов FreeBSD, содержащих более раннюю версию LLDB, могут воспользоваться снимком из портов или пакетов, как devel/llvm-devel.

Начиная с LLDB 12.0.0, удалённая отладка поддерживается в FreeBSD. Это означает, что `lldb-server` может быть запущен для отладки программы на одном узле, в то время как интерактивный клиент `lldb` подключается к нему с другого.

Чтобы запустить новый процесс для удалённой отладки, выполните `lldb-server` на удалённом сервере, набрав

```
% lldb-server g host:port -- progname
```

Процесс будет остановлен сразу после запуска, и `lldb-server` будет ожидать подключения клиента.

Запустите `lldb` локально и введите следующую команду для подключения к удалённому серверу:

```
(lldb) gdb-remote host:port
```

`lldb-server` также может присоединиться к работающему процессу. Для этого введите следующее на удалённом сервере:

```
% lldb-server g host:port --attach pid-or-name
```

2.6.4. Использование gdb

2.6.4.1. Запуск gdb

Запустите gdb, набрав

```
% gdb progname
```

хотя многие предпочитают запускать его внутри Emacs. Для этого введите:

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

Наконец, для тех, кого отпугивает текстовый интерфейс командной строки, существует графический интерфейс (devel/xxgdb) в Коллекции портов.

2.6.4.2. Запуск программы под отладчиком gdb

Скомпилируйте программу с `-g`, чтобы максимально использовать возможности `gdb`. Она будет работать и без этого, но отобразит только имя текущей выполняемой функции вместо исходного кода. Строка вида:

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

когда `gdb` запускается, это означает, что программа не была скомпилирована с опцией `-g`.

На приглашении `gdb` введите `break main`. Это укажет отладчику пропустить предварительный код настройки в выполняемой программе и остановить выполнение в начале кода программы. Теперь введите `run`, чтобы запустить программу — она начнётся с начала кода настройки, а затем будет остановлена отладчиком при вызове `main()`.

Для пошагового выполнения программы нажимайте `n`. При вызове функции войдите в неё, нажав `s`. Оказавшись внутри функции, вернитесь из неё, нажав `f`, или используйте `up` и `down` для быстрого просмотра вызывающего кода.

Вот простой пример того, как найти ошибку в программе с помощью `gdb`. Это наша программа (с умышленной ошибкой):

```
#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;
}
```

Эта программа устанавливает значение `i` равным `5` и передаёт его в функцию `bazz()`, которая выводит переданное число.

Компиляция и запуск программы отображают

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

Это было не то, что мы ожидали! Пора разобраться, что происходит!

```
% 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)
```

Подождите минуту! Как `int` стал равен `4231`? Разве он не был установлен в `5` в `main()`? Давайте поднимемся к `main()` и посмотрим.

```
(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
```

О боже! Глядя на код, мы забыли инициализировать i. Мы хотели добавить

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

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

но мы пропустили строку `i=5;`. Поскольку мы не инициализировали `i`, она содержала любое число, которое оказалось в той области памяти при запуске программы, и в данном случае это оказалось `4231`.

Команда `gdb` отображает стек вызовов каждый раз при входе в функцию или выходе из неё, даже при использовании `up` и `down` для перемещения по стеку вызовов. Это показывает имя функции и значения её аргументов, что помогает отслеживать текущее положение и происходящее. (Стек — это область хранения, где программа сохраняет информацию об аргументах, переданных в функции, и о месте, куда нужно вернуться после вызова функции.)

2.6.4.3. Изучение файла core с помощью gdb

Файл core — это, по сути, файл, содержащий полное состояние процесса на момент его аварийного завершения. В «старые добрые времена» программистам приходилось распечатывать шестнадцатеричные дампы файлов core и корпеть над руководствами по машинному коду, но сейчас жизнь стала немного проще. Кстати, в FreeBSD и других системах на базе 4.4BSD файл core называется progname.core, а не просто core, чтобы было понятнее, какой программе он принадлежит.

Для анализа файла core запустите `gdb` обычным способом. Вместо ввода команд `break` или `run` введите

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

Если файл core отсутствует в текущем каталоге, сначала введите `dir /путь/к/core/файлу`.

Отладчик должен отобразить что-то вроде этого:

```
% gdb progname
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 progname.core
Core was generated by `progname'.
Program terminated with signal 11, Segmentation fault.
Cannot access memory at address 0x7020796d.
#0  0x164a in bazz (anint=0x5) at temp.c:17
(gdb)
```

В этом случае программа называлась progname, поэтому файл дампа памяти называется progname.core. Мы видим, что программа завершилась аварийно из-за попытки доступа к области памяти, которая ей не доступна, в функции bazz.

Иногда полезно увидеть, как была вызвана функция, поскольку проблема могла возникнуть гораздо выше по стеку вызовов в сложной программе. bt заставляет gdb вывести трассировку стека вызовов:

```
(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)
```

Функция end() вызывается при аварийном завершении программы; в данном случае функция bazz() была вызвана из main().

2.6.4.4. Подключение к работающей программе с помощью gdb

Одной из самых удобных функций gdb является возможность подключения к уже запущенной программе. Конечно, для этого требуются соответствующие разрешения. Частой проблемой является пошаговое выполнение программы, которая создаёт дочерний процесс, когда нужно отслеживать дочерний процесс, но отладчик продолжает отслеживать только родительский.

Для этого запустите другой gdb, используйте ps для поиска идентификатора процесса дочернего элемента и выполните

```
(gdb) attach pid
```

в gdb, а затем отлаживайте как обычно.

Для того чтобы это работало правильно, код, который вызывает fork для создания дочернего процесса, должен делать что-то вроде следующего (предоставлено из документации 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 */
    ...
```

Теперь осталось только подключиться к дочернему процессу, установить PauseMode в `0` и дождаться возврата из вызова `sleep()`!

2.7. Использование Emacs в качестве среды разработки

2.7.1. Emacs

Emacs — это высоконастраиваемый редактор — настолько, что его можно скорее назвать операционной системой, чем редактором! Многие разработчики и системные администраторы действительно проводят практически всё своё время, работая внутри Emacs, выходя из него только для завершения сеанса.

Невозможно даже кратко описать все, что может делать Emacs, но вот некоторые особенности, которые могут быть интересны разработчикам:

- Очень мощный редактор, позволяющий выполнять поиск и замену как строк, так и регулярных выражений (шаблонов), переход к началу/концу блока выражения и многое другое.
- Выпадающие меню и встроенная справка.
- Подсветка синтаксиса и форматирование отступов в зависимости от языка.
- Полностью настраиваемый.
- Вы можете компилировать и отлаживать программы из Emacs.
- При ошибке компиляции можно перейти к проблемной строке исходного кода.
- Дружелюбный интерфейс для программы `info`, используемой для чтения гипертекстовой документации GNU, включая документацию по самому Emacs.
- Дружелюбный интерфейс для `gdb`, позволяющий просматривать исходный код во время пошагового выполнения программы.

И, несомненно, множество других, которые были упущены из виду.

Emacs можно установить на FreeBSD с помощью пакета editors/emacs.

После установки запустите его и выполните `C-h t`, чтобы прочитать руководство по Emacs — это означает, что нужно удерживать `control`, нажать `h`, отпустить `control`, а затем нажать `t`.

(Также можно использовать мышь для выбора Руководство по Emacs в меню **Help**.)

Хотя в Emacs и есть меню, стоит изучить сочетания клавиш, так как редактировать что-либо с их помощью гораздо быстрее, чем искать мышку и кликать в нужное место. Кроме того, общаясь с опытными пользователями Emacs, вы часто услышите выражения вроде «`M-x replace-s RET foo RET bar RET`» — полезно понимать, что они значат. Да и вообще, в Emacs столько полезных функций, что все они просто не поместятся на панелях меню.

К счастью, освоить сочетания клавиш довольно легко, так как они отображаются рядом с пунктами меню. Мой совет — использовать пункты меню для, скажем, открытия файла, пока вы не разберётесь, как это работает, и не почувствуете себя уверенно, а затем попробуйте выполнить `C-x C-f`. Когда освоитесь с этим, переходите к следующей команде меню.

Если вы не можете вспомнить, что делает определённая комбинация клавиш, выберите Описание Клавиши в меню **Help** и введите её — Emacs сообщит, что она делает. Вы также можете использовать пункт меню Command Apropos, чтобы найти все команды, содержащие определённое слово, с указанием соответствующих клавишных сочетаний.

Между прочим, выражение выше означает: удерживайте клавишу `Meta`, нажмите `x`, отпустите клавишу `Meta`, введите `replace-s` (сокращение от `replace-string` — ещё одна особенность Emacs в том, что команды можно сокращать), нажмите клавишу `return`, введите `foo` (строка, которую нужно заменить), нажмите клавишу `return`, введите `bar` (строка, на которую нужно заменить `foo`) и снова нажмите `return`. Emacs выполнит операцию поиска и замены, которую вы только что запросили.

Если вам интересно, что такое `Meta`, то это специальная клавиша, которая есть на многих рабочих станциях UNIX®. К сожалению, на PC её нет, поэтому обычно используется `alt` (или, если вам не повезло, `escape`).

Ах да, чтобы выйти из Emacs, нажмите `C-x C-c` (это значит зажмите клавишу `control`, нажмите `x`, затем `c` и отпустите `control`). Если у вас есть несохранённые файлы, Emacs спросит, хотите ли вы их сохранить. (Игнорируйте часть документации, где говорится, что `C-z` — это обычный способ выхода из Emacs — это оставляет Emacs работающим в фоне и полезно только на системах без виртуальных терминалов).

2.7.2. Настройка Emacs

Emacs делает много замечательных вещей; некоторые из них встроены, некоторые требуют настройки.

Вместо использования проприетарного языка макросов для конфигурации, Emacs применяет версию Lisp, специально адаптированную для редакторов, известную как Emacs Lisp. Работа с Emacs Lisp может быть весьма полезной, если вы хотите продолжить и изучить что-то вроде Common Lisp. Emacs Lisp обладает многими возможностями Common Lisp, хотя и значительно меньше (и, следовательно, проще для освоения).

Лучший способ изучить Emacs Lisp — это прочитать онлайн-руководство Emacs Reference.

Однако для начала настройки Emacs не обязательно знать Lisp, так как я включил пример

файла .emacs, которого должно быть достаточно для старта. Просто скопируйте его в свой домашний каталог и перезапустите Emacs, если он уже запущен; он прочитает команды из файла и (надеюсь) предоставит вам полезную базовую конфигурацию.

2.7.3. Пример файла .emacs

К сожалению, здесь слишком много информации, чтобы объяснять всё подробно; однако есть один или два момента, которые стоит упомянуть.

- Всё, что начинается с `;`, является комментарием и игнорируется Emacs.
- В первой строке `-- Emacs-Lisp --` нужен для того, чтобы мы могли редактировать сам файл .emacs в Emacs и использовать все удобные функции для редактирования Emacs Lisp. Обычно Emacs пытается угадать это по имени файла, но может не сделать это правильно для .emacs.
- Клавиша `tab` связана с функцией отступа в некоторых режимах, поэтому при нажатии клавиши tab текущая строка кода будет с отступом. Если вы хотите вставить символ табуляции в текст, удерживайте клавишу `control` во время нажатия `tab`.
- Этот файл поддерживает подсветку синтаксиса для C, C++, Perl, Lisp и Scheme, определяя язык по имени файла.
- В Emacs уже есть предопределённая функция `next-error`. В окне вывода компиляции это позволяет переходить от одной ошибки компиляции к следующей с помощью `M-n`; мы определяем дополнительную функцию `previous-error`, которая позволяет вернуться к предыдущей ошибке с помощью `M-p`. Самое приятное — сочетание `C-c C-c` откроет исходный файл, в котором произошла ошибка, и перейдёт на соответствующую строку.
- Включаем возможность Emacs работать как сервер, так что если вы заняты чем-то вне Emacs и хотите отредактировать файл, можно просто ввести

  ```
  % emacsclient filename
  ```

  и затем вы можете редактировать файл в вашем Emacs![6]

Пример 1. Пример файла .emacs

```
;; -*-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. Расширение списка языков, понимаемых Emacs

Вот, это все хорошо, если вы хотите программировать только на языках, уже предусмотренных в .emacs (C, C++, Perl, Lisp и Scheme), но что произойдет, если появится новый язык под названием "whizbang", полный захватывающих возможностей?

Первое, что нужно сделать, — это выяснить, поставляются ли с whizbang какие-либо файлы, сообщающие Emacs о языке. Обычно они заканчиваются на .el, что означает "Emacs Lisp". Например, если whizbang является портом FreeBSD, мы можем найти эти файлы, выполнив

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

и установите их, скопировав в каталог Emacs, где находятся файлы Lisp (site Lisp). В FreeBSD это /usr/local/share/emacs/site-lisp.

Вот пример, если вывод команды find был

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

мы бы сделали

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

Далее нам нужно решить, какое расширение имеют исходные файлы whizbang. Допустим, для примера, что все они заканчиваются на .wiz. Нам необходимо добавить запись в наш .emacs, чтобы убедиться, что Emacs сможет использовать информацию из whizbang.el.

Найдите запись auto-mode-alist в файле .emacs и добавьте строку для whizbang, например:

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

Это означает, что Emacs автоматически перейдёт в режим `whizbang-mode` при редактировании файла с расширением .wiz.

Непосредственно ниже вы найдете запись font-lock-auto-mode-list. Добавьте `whizbang-mode` в неё следующим образом:

```
;; 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")
```

Это означает, что Emacs всегда будет включать `font-lock-mode` (т.е. подсветку синтаксиса) при редактировании файла .wiz.

И это всё, что требуется. Если вам нужно, чтобы что-то ещё выполнялось автоматически при открытии .wiz, вы можете добавить `whizbang-mode hook` (см. `my-scheme-mode-hook` для простого примера, который добавляет `auto-indent`).

2.8. Для дальнейшего ознакомления

Для получения информации о настройке среды разработки для внесения исправлений в саму FreeBSD см. development(7).

- Brian Harvey and Matthew Wright *Simply Scheme* MIT 1994. ISBN 0-262-08226-8
- Randall Schwartz *Learning Perl* O'Reilly 1993 ISBN 1-56592-042-2
- Patrick Henry Winston and Berthold Klaus Paul Horn *Lisp (3rd Edition)* Addison-Wesley 1989 ISBN 0-201-08319-1
- Brian W. Kernighan and Rob Pike *The Unix Programming Environment* Prentice-Hall 1984 ISBN 0-13-937681-X
- Brian W. Kernighan and Dennis M. Ritchie *The C Programming Language (2nd Edition)* Prentice-Hall 1988 ISBN 0-13-110362-8
- Bjarne Stroustrup *The C++ Programming Language* Addison-Wesley 1991 ISBN 0-201-53992-6
- W. Richard Stevens *Advanced Programming in the Unix Environment* Addison-Wesley 1992 ISBN 0-201-56317-7
- W. Richard Stevens *Unix Network Programming* Prentice-Hall 1990 ISBN 0-13-949876-1

[1] Если вы запустите её в оболочке, может произойти дамп памяти.

[2] На случай, если вы не знали: бинарная сортировка — это эффективный способ упорядочивания элементов, в отличие от пузырьковой.

[3] Причины этого кроются в глубинах истории.

[4] Примечание: мы не использовали флаг -o для указания имени исполняемого файла, поэтому получим исполняемый файл с именем a.out. Создание отладочной версии с именем foobar остаётся упражнением для читателя!

[5] Они не используют форму MAKEFILE, так как заглавные буквы часто применяются для файлов документации, таких как README.

[6] Многие пользователи Emacs устанавливают переменную окружения EDITOR в emacsclient, так что это происходит каждый раз, когда им нужно отредактировать файл.

Глава 3. Безопасное программирование

3.1. Обзор

В этой главе описываются некоторые проблемы безопасности, которые преследуют программистов UNIX® на протяжении десятилетий, а также новые инструменты, помогающие избежать написания уязвимого кода.

3.2. Методология безопасного проектирования

Написание безопасных приложений требует очень внимательного и пессимистичного взгляда на жизнь. Приложения должны работать по принципу "наименьших привилегий", чтобы ни один процесс не выполнялся с доступом, превышающим необходимый минимум для выполнения его функций. По возможности следует повторно использовать уже проверенный код, чтобы избежать распространённых ошибок, которые, возможно, уже исправили другие.

Одной из ловушек среды UNIX® является то, насколько легко делать предположения о разумности окружения. Приложения никогда не должны доверять пользовательскому вводу (во всех его формах), системным ресурсам, межпроцессному взаимодействию или времени событий. Процессы UNIX® выполняются не синхронно, поэтому логические операции редко бывают атомарными.

3.3. Переполнение буфера

Переполнение буфера существовало с самых истоков архитектуры фон Неймана 1. Впервые оно получило широкую известность в 1988 году благодаря червю Морриса. К сожалению, эта базовая атака остаётся эффективной и по сей день. Наиболее распространённый тип атаки с переполнением буфера основан на повреждении стека.

Большинство современных компьютерных систем используют стек для передачи аргументов процедурам и хранения локальных переменных. Стек — это буфер типа "последним пришёл — первым ушёл" (LIFO) в верхней области памяти процесса. Когда программа вызывает функцию, создаётся новый "стековый кадр". Этот стековый кадр состоит из аргументов, переданных функции, а также динамического количества места для локальных переменных. "Указатель стека" — это регистр, который содержит текущее местоположение вершины стека. Поскольку это значение постоянно меняется по мере добавления новых значений на вершину стека, многие реализации также предоставляют "указатель кадра", который располагается вблизи начала стекового кадра, чтобы локальные переменные могли легче адресоваться относительно этого значения. 1 Адрес возврата для вызовов функций также хранится в стеке, и это является причиной эксплойтов переполнения стека, поскольку переполнение локальной переменной в функции может перезаписать адрес возврата этой функции, потенциально позволяя злоумышленнику выполнить любой код по своему усмотрению.

Хотя атаки на стек являются наиболее распространёнными, также возможно переполнение стека с помощью атаки на кучу (malloc/free).

Язык программирования C не выполняет автоматическую проверку границ массивов или указателей, как это делают многие другие языки. Кроме того, стандартная библиотека C содержит множество очень опасных функций.

`strcpy`(char *dest, const char *src)	Может переполнить буфер назначения
`strcat`(char *dest, const char *src)	Может переполнить буфер назначения
`getwd`(char *buf)	Может переполнить буфер buf
`gets`(char *s)	Может переполнить буфер s
`[vf]scanf`(const char *format, ...)	Может переполнить свои аргументы.
`realpath`(char *path, char resolved_path[])	Может переполнить буфер пути
`[v]sprintf`(char *str, const char *format, ...)	Может переполнить буфер str.

3.3.1. Пример переполнения буфера

Следующий пример кода содержит переполнение буфера, предназначенное для перезаписи адреса возврата и пропуска инструкции, следующей сразу после вызова функции. (Вдохновлено 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;
}
```

Давайте рассмотрим, как будет выглядеть образ памяти этого процесса, если мы введем 160 пробелов в нашу небольшую программу перед нажатием Enter.

Очевидно, что можно разработать более вредоносные входные данные для выполнения реальных скомпилированных инструкций (например, exec(/bin/sh)).

3.3.2. Избегание переполнения буфера

Наиболее простое решение проблемы переполнения стека — всегда использовать функции копирования памяти и строк с ограничением длины. `strncpy` и `strncat` являются частью стандартной библиотеки C. Эти функции принимают параметр длины, который не должен превышать размер целевого буфера. Затем эти функции копируют до 'length' байтов из источника в назначение. Однако у этих функций есть ряд проблем. Ни одна из них не гарантирует завершающий NUL, если размер входного буфера равен размеру целевого. Параметр длины также используется неодинаково между `strncpy` и `strncat`, что может сбивать программистов с толку относительно их правильного использования. Также наблюдается значительное снижение производительности по сравнению с `strcpy` при копировании короткой строки в большой буфер, поскольку `strncpy` заполняет оставшееся пространство до указанного размера символами NUL.

Существует другая реализация копирования памяти для решения этих проблем. Функции `strlcpy` и `strlcat` гарантируют, что они всегда завершат строку назначения нулевым символом при передаче аргумента ненулевой длины.

3.3.2.1. Скомпилированная проверка границ во время выполнения

К сожалению, до сих пор существует очень большое количество кода в открытом доступе, который бездумно копирует память, не используя ни одну из ограниченных функций копирования, которые мы только что обсудили. К счастью, есть способ помочь предотвратить такие атаки — проверка границ во время выполнения, которая реализована в нескольких компиляторах C/C++.

ProPolice — это одна из таких функций компилятора, интегрированная в gcc(1) версий 4.1 и выше. Она заменяет и расширяет более раннее расширение StackGuard для gcc(1).

ProPolice помогает защититься от переполнений буфера на стеке и других атак, размещая псевдослучайные числа в ключевых областях стека перед вызовом любой функции. Когда функция завершается, эти "канарейки" проверяются, и если обнаруживается, что они были изменены, выполнение программы немедленно прекращается. Таким образом, любая попытка изменить адрес возврата или другие переменные, хранящиеся на стеке, с целью запуска вредоносного кода, вряд ли увенчается успехом, так как злоумышленнику также необходимо оставить псевдослучайные канарейки нетронутыми.

Перекомпиляция вашего приложения с использованием ProPolice является эффективным способом предотвращения большинства атак, связанных с переполнением буфера, но оно всё ещё может быть скомпрометировано.

3.3.2.2. Библиотечная проверка границ во время выполнения

Механизмы на основе компилятора совершенно бесполезны для проприетарного программного обеспечения, которое невозможно перекомпилировать. Для таких ситуаций существует ряд библиотек, которые переопределяют небезопасные функции стандартной библиотеки C (`strcpy`, `fscanf`, `getwd` и т.д.) и гарантируют, что эти функции никогда не смогут записать данные за указатель стека.

- libsafe

- libverify
- libparanoia

К сожалению, эти защиты на основе библиотек имеют ряд недостатков. Они защищают лишь от очень небольшого набора проблем, связанных с безопасностью, и не устраняют основную причину. Эти защиты могут не сработать, если приложение было скомпилировано с флагом -fomit-frame-pointer. Кроме того, переменные окружения LD_PRELOAD и LD_LIBRARY_PATH могут быть перезаписаны или сброшены пользователем.

3.4. Проблемы с SetUID

Существует как минимум 6 различных идентификаторов, связанных с каждым процессом, поэтому необходимо очень внимательно следить за уровнем доступа вашего процесса в любой момент времени. В частности, все приложения с seteuid должны отказываться от своих привилегий, как только в них больше нет необходимости.

Действительный идентификатор пользователя может быть изменён только процессом с правами суперпользователя. Программа login устанавливает его при первоначальном входе пользователя в систему, и он редко изменяется.

Эффективный идентификатор пользователя устанавливается функциями `exec()`, если у программы установлен бит seteuid. Приложение может вызывать `seteuid()` в любое время, чтобы установить эффективный идентификатор пользователя либо в реальный идентификатор пользователя, либо в сохранённый set-user-ID. Когда эффективный идентификатор пользователя устанавливается функциями `exec()`, предыдущее значение сохраняется в сохранённом set-user-ID.

3.5. Ограничение окружения вашей программы

Традиционный метод ограничения процесса — это системный вызов `chroot()`. Этот системный вызов изменяет корневой каталог, от которого ссылаются все остальные пути для процесса и любых дочерних процессов. Для успешного выполнения этого вызова процесс должен иметь право на выполнение (поиск) в указанном каталоге. Новая среда фактически не вступает в силу, пока вы не выполните `chdir()` в новой среде. Также следует отметить, что процесс может легко выйти из окружения chroot, если он имеет привилегии root. Это может быть достигнуто путем создания узлов устройств для чтения памяти ядра, подключения отладчика к процессу вне окружения chroot(8) или многими другими творческими способами.

Поведение системного вызова `chroot()` можно частично контролировать с помощью переменной `sysctl` kern.chroot_allow_open_directories. Если этому параметру присвоено значение 0, `chroot()` завершится с ошибкой EPERM, если есть какие-либо открытые каталоги. Если установлено значение по умолчанию 1, то `chroot()` завершится с ошибкой EPERM, если есть открытые каталоги и процесс уже находится внутри вызова `chroot()`. Для любого другого значения проверка на открытые каталоги будет полностью пропущена.

3.5.1. Функциональность клеток FreeBSD

Концепция клетки расширяет возможности chroot(), ограничивая права суперпользователя для создания настоящего виртуального сервера. После настройки клетки все сетевые взаимодействия должны осуществляться через указанный IP-адрес, а привилегии root внутри этой клетки сильно ограничены.

Находясь в клетке, любые проверки прав суперпользователя в ядре с использованием вызова suser() завершатся неудачей. Однако некоторые вызовы suser() были заменены на новый интерфейс suser_xxx(). Эта функция отвечает за распознавание или запрет доступа к правам суперпользователя для процессов в клетке.

Суперпользователь в среде клетки имеет возможность:

- Управлять учётными данными с помощью setuid, seteuid, setgid, setegid, setgroups, setreuid, setregid, setlogin
- Устанавливать ограничений ресурсов с помощью setrlimit
- Изменять некоторые узлы sysctl (kern.hostname)
- chroot()
- Устанавливать флаги на vnode: chflags, fchflags
- Устанавливать атрибуты vnode, такие как права доступа к файлу, владелец, группа, размер, время доступа и время изменения.
- Привязываться к привилегированным портам в домене Интернета (порты < 1024)

Клетка — это очень полезный инструмент для запуска приложений в безопасной среде, но у него есть некоторые недостатки. В настоящее время механизмы IPC не были преобразованы в suser_xxx, поэтому такие приложения, как MySQL, не могут быть запущены внутри клетки. Доступ суперпользователя может иметь очень ограниченное значение внутри клетки, но нет возможности точно указать, что означает «очень ограниченный».

3.5.2. Возможности процесса в POSIX®.1e

POSIX® выпустил рабочий проект, который добавляет аудит событий, списки контроля доступа, детализированные привилегии, маркировку информации и обязательный контроль доступа.

Это работа в процессе, и она является основным направлением проекта TrustedBSD. Некоторые первоначальные наработки были добавлены в FreeBSD-CURRENT (cap_set_proc(3)).

3.6. Доверие

Приложение никогда не должно предполагать, что окружение пользователя является предсказуемым. Это включает (но не ограничивается): пользовательский ввод, сигналы, переменные окружения, ресурсы, IPC, mmaps, текущий рабочий каталог файловой системы, файловые дескрипторы, количество открытых файлов и т.д.

Никогда не следует предполагать, что можно отловить все виды некорректных входных данных, которые может предоставить пользователь. Вместо этого ваше приложение должно использовать позитивную фильтрацию, разрешая только определённое подмножество входных данных, которые вы считаете безопасными. Некорректная проверка данных стала причиной многих уязвимостей, особенно в CGI-скриптах во всемирной паутине. Для имён файлов необходимо быть особенно осторожными с путями ("../", "/"), символическими ссылками и escape-символами оболочки.

В Perl есть замечательная функция под названием "Режим Taint", которая может использоваться для предотвращения небезопасного использования данных, полученных извне программы. Этот режим проверяет аргументы командной строки, переменные окружения, информацию о локали, результаты определённых системных вызовов (`readdir()`, `readlink()`, `getpwxxx()`) и все вводимые данные из файлов.

3.7. Состояние гонки

Состояние гонки — это аномальное поведение, вызванное непредвиденной зависимостью от относительного времени событий. Другими словами, программист ошибочно предположил, что определённое событие всегда произойдет раньше другого.

Некоторые из распространённых причин состояний гонки — это сигналы, проверки доступа и открытие файлов. Сигналы по своей природе являются асинхронными событиями, поэтому при работе с ними необходимо проявлять особую осторожность. Проверка доступа с помощью `access(2)`, а затем `open(2)` явно неатомарна. Пользователи могут перемещать файлы между этими двумя вызовами. Вместо этого привилегированные приложения должны использовать `seteuid()`, а затем вызывать `open()` напрямую. По аналогии, приложение всегда должно устанавливать правильную маску (`umask`) перед вызовом `open()`, чтобы избежать необходимости в лишних вызовах `chmod()`.

Глава 4. Локализация и интернационализация - L10N и I18N

4.1. Программирование приложений с поддержкой I18N

Чтобы сделать ваше приложение более полезным для пользователей, говорящих на других языках, мы надеемся, что вы будете разрабатывать его с поддержкой интернационализации (I18N). Компилятор GNU gcc и библиотеки для графического интерфейса, такие как QT и GTK, поддерживают I18N за счёт специальной обработки строк. Создание программы с поддержкой I18N очень просто. Это позволяет участникам быстро адаптировать ваше приложение для других языков. Для получения более подробной информации обратитесь к документации по I18N для конкретной библиотеки.

Вопреки распространённому мнению, написание кода, соответствующего стандарту I18N, является простой задачей. Обычно оно заключается лишь в оборачивании строк в специфичные для библиотеки функции. Кроме того, убедитесь, что поддерживаются широкие или многобайтовые символы.

4.1.1. Призыв к объединению усилий по интернационализации

Нам стало известно, что индивидуальные усилия по интернационализации (I18N) и локализации (L10N) в каждой стране дублируют работу друг друга. Многие из нас снова и снова неэффективно изобретают велосипед. Мы надеемся, что различные крупные группы разработчиков в области I18N смогут объединится для совместной работы и ответственности, подобно Основной команде в FreeBSD.

В настоящее время мы надеемся, что при написании или портировании I18N-программ вы будете отправлять их в соответствующие списки рассылки FreeBSD каждой страны для тестирования. В будущем мы надеемся создать приложения, которые будут работать на всех языках "из коробки" без грязных хаков.

Создана группа Список рассылки, посвящённый интернационализации FreeBSD. Если вы разработчик в области I18N/L10N, присылайте свои комментарии, идеи, вопросы и всё, что относится к этой теме по вашему мнению.

4.1.2. Perl и Python

В Perl и Python есть библиотеки для I18N и работы с широкими символами. Пожалуйста, используйте их для соответствия требованиям I18N.

4.2. Локализованные сообщения с поддержкой родного языка POSIX.1 (NLS — Native Language Support)

Помимо основных функций интернационализации (I18N), таких как поддержка различных кодировок ввода или национальных соглашений, например, различных разделителей десятичных знаков, на более высоком уровне I18N можно локализовать сообщения, выводимые различными программами. Распространённый способ сделать это — использовать функции NLS POSIX.1, которые предоставляются как часть базовой системы FreeBSD.

4.2.1. Организация локализованных сообщений в файлы каталогов

POSIX.1 NLS использует файлы каталогов. Эти файлы содержат локализованные сообщения в желаемой кодировке. Сообщения организованы в наборы, и каждое сообщение идентифицируется целым числом в соответствующем наборе. Файлы каталогов традиционно называются по имени локали, для которой они содержат локализованные сообщения, с добавлением расширения `.msg`. Например, венгерские сообщения для кодировки ISO8859-2 должны храниться в файле с именем hu_HU.ISO8859-2.

Эти файлы каталогов представляют собой обычные текстовые файлы, содержащие нумерованные сообщения. Можно добавлять комментарии, начиная строку со знака `$`. Границы наборов также разделяются специальными комментариями, где ключевое слово `set` должно следовать непосредственно за знаком `$`. После ключевого слова `set` указывается номер набора. Например:

```
$set 1
```

Фактические записи сообщений начинаются с номера сообщения, за которым следует локализованное сообщение. Допускаются известные модификаторы из printf(3):

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

Файлы языкового каталога должны быть скомпилированы в бинарный формат перед тем, как они могут быть открыты программой. Это преобразование выполняется с помощью утилиты gencat(1). Её первый аргумент — это имя файла скомпилированного каталога, а последующие аргументы — входные каталоги. Локализованные сообщения также могут быть организованы в несколько файлов каталогов, и затем все они могут быть обработаны с помощью gencat(1).

4.2.2. Использование файлов каталога из исходного кода

Использование файлов каталогов простое. Чтобы вызвать функции, работающие с ними, необходимо включить файл nl_types.h. Перед использованием каталога его нужно открыть с помощью catopen(3). Функция принимает два аргумента. Первый параметр — это имя

установленного и скомпилированного каталога. Обычно используется имя программы, например, grep. Это имя будет использоваться при поиске скомпилированного файла каталога. Вызов catopen(3) ищет этот файл в /usr/share/nls/locale/catname и в /usr/local/share/nls/locale/catname, где `locale` — установленная локализация, а `catname` — имя обсуждаемого каталога. Второй параметр — это константа, которая может принимать два значения:

- `NL_CAT_LOCALE`, что означает, что используемый файл каталога будет основан на `LC_MESSAGES`.
- `0`, что означает, что для открытия соответствующего каталога необходимо использовать `LANG`.

Вызов catopen(3) возвращает идентификатор каталога типа `nl_catd`. Обратитесь к справочной странице для получения списка возможных кодов ошибок.

После открытия каталога catgets(3) может быть использована для извлечения сообщения. Первый параметр — это идентификатор каталога, возвращаемый catopen(3), второй — номер набора, третий — номер сообщения, а четвертый — резервное сообщение, которое будет возвращено, если запрошенное сообщение не может быть извлечено из файла каталога.

После использования файла каталога его необходимо закрыть, вызвав catclose(3), которая принимает один аргумент — идентификатор каталога.

4.2.3. Практический пример

Следующий пример демонстрирует простое решение по гибкому использованию каталогов NLS.

Нижеследующие строки необходимо поместить в общий заголовочный файл программы, который включается во все исходные файлы, где необходимы локализованные сообщения:

```
#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[];
```

Далее, добавьте эти строки в глобальную секцию объявлений основного исходного файла:

```
#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"
};
```

Далее следуют реальные фрагменты кода, которые открывают, читают и закрывают каталог:

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

...

printf(getstr(1));

...

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

4.2.3.1. Уменьшение количества строк для локализации

Хороший способ уменьшить количество строк, требующих локализации, — использовать сообщения об ошибках из libc. Это также полезно для избежания дублирования и обеспечения единообразия сообщений об ошибках, с которыми может столкнуться множество программ.

Вот пример, который не использует сообщения об ошибках из libc:

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

Это можно преобразовать для вывода сообщения об ошибке, считывая `errno` и выводя соответствующее сообщение об ошибке:

```
#include <err.h>
```

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

В этом примере пользовательская строка исключена, что упростит работу переводчиков при локализации программы, а пользователи увидят стандартное сообщение об ошибке "Not a directory" при возникновении данной ошибки. Это сообщение, вероятно, будет выглядеть более привычным для них. Обратите внимание, что для прямого доступа к `errno` потребовалось включить errno.h.

Стоит отметить, что бывают случаи, когда `errno` устанавливается автоматически предыдущим вызовом, поэтому нет необходимости устанавливать его явно:

```
#include <err.h>
...
if ((p = malloc(size)) == NULL)
    err(1, NULL);
```

4.2.4. Использование bsd.nls.mk

Использование файлов каталогов требует нескольких повторяемых шагов, таких как компиляция каталогов и их установка в нужное место. Чтобы ещё больше упростить этот процесс, bsd.nls.mk вводит некоторые макросы. Нет необходимости явно включать bsd.nls.mk, он подключается автоматически из общих Makefiles, таких как bsd.prog.mk или bsd.lib.mk.

Обычно достаточно определить `NLSNAME`, которое должно содержать имя каталога, указанное в качестве первого аргумента catopen(3), и перечислить файлы каталогов в `NLS` без расширения `.msg`. Вот пример, который позволяет отключить NLS при использовании с предыдущими примерами кода. Для сборки программы без поддержки NLS необходимо определить переменную `WITHOUT_NLS` make(1).

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

Обычно файлы каталогов размещаются в подкаталоге nls, и это поведение по умолчанию для bsd.nls.mk. Однако можно переопределить расположение каталогов с помощью переменной `NLSSRCDIR` make(1). Имя по умолчанию для предварительно скомпилированных файлов каталогов также следует упомянутому ранее соглашению об именовании. Его

можно переопределить, установив переменную `NLSNAME`. Существуют и другие параметры для точной настройки обработки файлов каталогов, но обычно в этом нет необходимости, поэтому они здесь не описаны. Для получения дополнительной информации о bsd.nls.mk обратитесь к самому файлу — он короткий и легко понятен.

Глава 5. Руководство и политика работы с деревом исходного кода

Эта глава документирует различные руководства и политики, действующие для дерева исходных кодов FreeBSD.

5.1. Рекомендации по стилю

Соблюдение единого стиля написания кода чрезвычайно важно, особенно в крупных проектах, таких как FreeBSD. Код должен соответствовать стилям программирования FreeBSD, описанным в style(9) и style.Makefile(5).

5.2. `MAINTAINER` в Makefile-ах

Если определённая часть дистрибутива FreeBSD src/ поддерживается человеком или группой лиц, это указывается в файле src/MAINTAINERS. Сопровождающие портов в Коллекции портов указывают свою ответственность, добавляя строку `MAINTAINER` в Makefile соответствующего порта:

```
MAINTAINER= email-addresses
```

Для других частей репозитория или для разделов, в которых не указан сопровождающий, или если вы не уверены, кто является активным сопровождающим, попробуйте посмотреть историю последних коммитов соответствующих частей дерева исходного кода. Довольно часто сопровождающий явно не указан, но люди, которые активно работали с частью дерева исходного кода, скажем, последние пару лет, заинтересованы в проверке изменений. Даже если это не указано явно в документации или в самом исходном коде, запросить проверку из вежливости — вполне разумное действие.

Роль сопровождающего заключается в следующем:

- Сопровождающий является владельцем и ответственным за этот код. Это означает, что он или она отвечает за исправление ошибок и решение проблем, связанных с этой частью кода, а в случае с предоставленным программным обеспечением — за отслеживание новых версий, если это необходимо.
- Изменения в каталогах, для которых определен сопровождающий, должны быть отправлены сопровождающему на проверку и рецензирование перед коммитом. Только если сопровождающий не отвечает в течение недопустимо долгого времени на несколько писем, допустимо закоммитить изменения без его проверки. Тем не менее, рекомендуется по возможности попытаться получить рецензирование изменений у кого-нибудь ещё.
- Конечно, недопустимо добавлять человека или группу в качестве сопровождающего,

если они не согласны взять на себя эти обязанности. С другой стороны, это не обязательно должен быть один коммиттер, и это может быть и группа людей.

5.3. Стороннее программное обеспечение

Некоторые части дистрибутива FreeBSD состоят из программного обеспечения, которое активно поддерживается за пределами проекта FreeBSD. По историческим причинам мы называем это *сторонним* программным обеспечением. Некоторые примеры: LLVM, zlib(3) и awk(1).

Принятая процедура управления вносимым программным обеспечением включает создание *ветки поставщика* (*vendor branch*), где программное обеспечение может быть импортировано в чистом виде (без изменений), а обновления могут отслеживаться с учётом версий. Затем содержимое ветки поставщика применяется к дереву исходного кода, возможно, с локальными изменениями. Специфичные для FreeBSD элементы сборки поддерживаются в дереве исходного кода, а не в ветке поставщика.

В зависимости от потребностей и сложности, отдельные программные проекты могут отклоняться от этой процедуры по усмотрению сопровождающего. Точные шаги, необходимые для обновления конкретного программного обеспечения, должны быть записаны в файле с именем `FREEBSD-upgrade`; например, файл FREEBSD-upgrade libarchive.

Стороннее программное обеспечение обычно размещается в подкаталоге contrib/ дерева исходных кодов, за некоторыми исключениями. Стороннее программное обеспечение, используемое только ядром, находится в sys/contrib/.

Поскольку это затрудняет импорт будущих версий, незначительные, тривиальные и/или косметические изменения *настоятельно не рекомендуются* для файлов, которые всё ещё отслеживают ветку поставщика.

5.3.1. Импорт веток поставщика

Стандартный процесс управления сторонним программным обеспечением и ветками поставщиков подробно описан в Руководстве коммиттера.

5.4. Файлы с правовыми ограничениями

Время от времени может возникнуть необходимость включить файл с правовыми ограничениями (обремененными лицензиями, патентами) в дерево исходного кода FreeBSD. Например, если устройство требует загрузки небольшого бинарного кода перед началом работы, а у нас нет исходного кода для него, то такой бинарный файл считается обремененным. Следующие политики применяются к включению обремененных файлов в дерево исходного кода FreeBSD.

1. Любой файл, который интерпретируется или выполняется процессором(-ами) системы и не представлен в исходном формате, является обременённым.

2. Любой файл с лицензией более ограничительной, чем BSD или GNU, является обременённым.
3. Файл, содержащий загружаемые двоичные данные для использования оборудованием, не является обремененным, если к нему не применяется пункт (1) или (2).
4. Любой файл с правовыми ограничениями требует специального одобрения от Основной команды (Core Team) перед добавлением в репозиторий.
5. Обремененные файлы помещаются в src/contrib или src/sys/contrib.
6. Весь модуль должен храниться вместе. Нет смысла разделять его, если только нет совместного использования кода с необременённой частью кода.
7. В прошлом бинарные файлы обычно кодировались с помощью uuencode и назывались arch/filename.o.uu. Теперь в этом нет необходимости, и бинарные файлы могут добавляться в репозиторий без изменений.
8. Файлы ядра системы:
    a. Всегда должны быть указана в conf/files.* (для упрощения сборки).
    b. Всегда должны быть в LINT, но Основная команда (Core Team) решает в каждом конкретном случае, следует ли их закомментировать или нет. Основная команда может, конечно, позже изменить свое решение.
    c. *Инженер по выпуску* решает, будет ли это включено в выпуск.
9. Пользовательские файлы:
    a. Команда Основная команда (Core team) принимает решение о включении кода в базовую устанавливаемую систему.
    b. Команда подготовки релизов решает, будет ли это включено в релиз.

5.5. Динамические библиотеки

Если вы добавляете поддержку динамических библиотек в порт или другое программное обеспечение, у которого её нет, номера версий библиотек должны следовать этим правилам. Обычно итоговые номера не будут иметь ничего общего с версией выпуска программного обеспечения.

Для портов:

- Предпочитайте использовать номер, уже выбранный вышестоящим проектом
- Если вышестоящий источник предоставляет управление версиями символов, убедитесь, что мы используем их скрипт

Для базовой системы:

- Начните версии библиотеки с 1
- Настоятельно рекомендуется добавить контроль версий символов в новую библиотеку
- Если есть несовместимое изменение, обработайте его с помощью версионирования символов, сохраняя обратную совместимость ABI

- Если это невозможно или библиотека не использует версионирование символов, увеличьте версию библиотеки
- Прежде чем даже рассматривать увеличение версии библиотеки для библиотеки с версионированием символов, проконсультируйтесь с Командой подготовки релизов, предоставив причины, почему изменение настолько важно, что его следует разрешить, несмотря на нарушение ABI

Например, добавленные функции и исправления ошибок, не изменяющие интерфейсы, допустимы, тогда как удалённые функции, изменённый синтаксис вызовов и т.д. должны либо предоставлять обратно-совместимые символы, либо приведут к изменению старшего номера версии.

Обязанность коммиттера, вносящего изменения, — управлять версионированием библиотек.

Динамический загрузчик ELF сопоставляет имена библиотек буквально. Существует популярное соглашение, согласно которому версия библиотеки записывается в виде `libexample.so.x.y`, где x — это мажорная версия, а y — минорная. Общепринятой практикой является установка поля soname у библиотеки (тег ELF `DT_SONAME`) в `libexample.so.x`, а также создание символических ссылок `libexample.so.x→libexample.so.x.y`, `libexample.so→libexample.so.x` при установке библиотеки для последней минорной версии y. Таким образом, поскольку статический компоновщик ищет `libexample.so`, когда указана опция командной строки `-lexample`, объекты, скомпонованные с libexample, получают информацию о зависимости от правильной библиотеки. Почти все популярные системы сборки автоматически используют эту схему.

Глава 6. Регрессионное и нагрузочное тестирование

Регрессионные тесты используются для проверки определённой части системы, чтобы убедиться, что она работает как ожидается, и для предотвращения повторного появления старых ошибок.

Инструменты для регрессионного тестирования FreeBSD можно найти в дереве исходных кодов FreeBSD в каталоге src/tools/regression.

6.1. Контрольный список для бенчмарка низкоуровневых операций

Этот раздел содержит рекомендации по проведению корректного бенчмарка низкоуровненых операций на FreeBSD или самой FreeBSD.

Невозможно использовать все приведённые ниже рекомендации каждый раз, но чем больше их применяется, тем лучше способность теста выявлять небольшие различия.

- Отключить APM и любые другие манипуляции с часами (ACPI ?).
- Запускайте тесты в однопользовательском режиме. Например, cron(8) и другие демоны только добавляют шум. Демон sshd(8) также может вызвать проблемы. Если требуется доступ по SSH во время тестирования, либо отключите перегенерацию ключа SSHv1, либо завершите родительский демон `sshd` во время тестов.
- Не запускайте ntpd(8).
- Если события syslog(3) генерируются, запустите syslogd(8) с пустым /etc/syslogd.conf, в противном случае не запускайте его.
- Минимизируйте дисковые операции ввода-вывода, по возможности избегайте их полностью.
- Не монтируйте файловые системы, которые не требуются.
- Смонтируйте /, /usr и любые другие файловые системы в режиме только для чтения, если это возможно. Это исключает обновления atime на диске (и т.д.) из общей картины ввода-вывода.
- Переинициализируйте тестовую файловую систему с возможностью чтения/записи с помощью newfs(8) и заполните её из файла tar(1) или dump(8) перед каждым запуском. Размонтируйте и смонтируйте её перед началом теста. Это обеспечит согласованную структуру файловой системы. Для теста worldstone это применимо к /usr/obj (просто переинициализируйте с помощью `newfs` и смонтируйте). Для достижения 100% воспроизводимости заполните файловую систему из файла dd(1) (например: `dd if=myimage of=/dev/ad0s1h bs=1m`)
- Используйте разделы md(4) с поддержкой malloc или предзагруженные.
- Перезагружайтесь между отдельными итерациями теста, это обеспечивает более согласованное состояние.

- Удалите все необязательные драйверы устройств из ядра. Например, если USB не нужен для теста, не включайте поддержку USB в ядре. Драйверы, которые подключаются, часто имеют работающие таймауты.
- Отключите неиспользуемое оборудование. Отсоедините диски с помощью atacontrol(8) и camcontrol(8), если диски не используются для тестирования.
- Не настраивайте сеть, если она не тестируется, или дождитесь завершения тестирования, чтобы отправить результаты на другой компьютер.
- Отключите "турбо-режимы", так как они делают тактовую частоту явно зависимой от окружающей среды. Это означает, что результаты тестирования на 100% идентичном коде могут зависеть от времени суток, употребления кофе или газировки или даже от количества людей в офисе.

Если система должна быть подключена к общедоступной сети, следите за всплесками широковещательного трафика. Даже если они почти незаметны, они будут занимать циклы процессора. Многоадресная рассылка имеет аналогичные предостережения. * Размещайте каждую файловую систему на отдельном диске. Это минимизирует задержки, вызванные оптимизацией перемещения головок диска. * Минимизируйте вывод на последовательные или VGA-консоли. Запись вывода в файлы снижает дрожание. (Консоли на последовательном порту легко становятся узким местом.) Не касайтесь клавиатуры во время выполнения теста, даже нажатия `пробел` или `back-space` отражаются в числах. * Убедитесь, что тест достаточно длинный, но не слишком. Если тест слишком короткий, возникают проблемы с временными метками. Если он слишком длинный, изменения температуры и дрейф повлияют на частоту кварцевых кристаллов в компьютере. Эмпирическое правило: больше минуты, меньше часа. * Попытайтесь поддерживать температуру вокруг машины как можно более стабильной. Это влияет как на кварцевые резонаторы, так и на алгоритмы работы дисковых накопителей. Для получения действительно стабильных часов рассмотрите возможность использования стабилизированного тактового сигнала. Например, используйте OCXO + PLL и подавайте выходной сигнал в тактовые схемы вместо кварцевого резонатора на материнской плате. Для получения дополнительной информации по этому вопросу свяжитесь с Poul-Henning Kamp <phk@FreeBSD.org>. * Выполните тест как минимум 3 раза, но лучше запустить более 20 раз как для кода "до", так и для кода "после". По возможности чередуйте запуски (т.е. не следует запускать 20 раз "до", а затем 20 раз "после"), это поможет выявить влияние окружения. Не чередуйте строго 1:1, а лучше 3:3, чтобы можно было обнаружить эффекты взаимодействия.

+ Хороший шаблон: `bababa{bbbaaa}*`. Это даёт подсказку после первых 1+1 прогонов (так что можно остановить тест, если всё идёт совсем не так), стандартное отклонение после первых 3+3 (дает хорошее представление, стоит ли проводить длительный прогон), а также тренды и показатели взаимодействия позже. * Используйте ministat(1), чтобы определить, являются ли числа значимыми. Рекомендуется приобрести книгу "Cartoon guide to statistics" ISBN: 0062731025, особенно если вы забыли или никогда не изучали стандартное отклонение и t-критерий Стьюдента. * Не используйте фоновый fsck(8), если тест не является бенчмарком фонового `fsck`. Также отключите `background_fsck` в /etc/rc.conf, если бенчмарк не запускается как минимум через 60+«время работы `fsck`» секунд после загрузки, так как rc(8) пробуждается и проверяет, нужно ли запускать `fsck` для каких-либо файловых систем, когда включен фоновый `fsck`. Аналогично, убедитесь, что нет оставшихся снимков, если только

бенчмарк не является тестом со снимками. * Если тесты производительности показывают неожиданно низкие результаты, проверьте такие факторы, как высокий объём прерываний из неожиданного источника. Сообщалось, что некоторые версии ACPI могут "вести себя неправильно" и генерировать избыточные прерывания. Для диагностики необычных результатов тестов сделайте несколько снимков `vmstat -i` и поищите что-то необычное. * Будьте внимательны к параметрам оптимизации для ядра и пользовательского пространства, а также отладки. Легко упустить что-то и позже понять, что тест сравнивал не одно и то же. * Никогда не проводите тестирование производительности с включёнными параметрами ядра `WITNESS` и `INVARIANTS`, если тест не направлен на оценку производительности именно этих функций. `WITNESS` может привести к снижению производительности на 400% и более. Аналогично, параметры userspace malloc(3) по умолчанию отличаются в -CURRENT от тех, что поставляются в релизах.

6.2. Tinderbox для исходного текста FreeBSD

Исходный Tinderbox состоит из:

- Скрипта сборки tinderbox, который автоматизирует выгрузку определённой версии исходного кода FreeBSD и её сборку.
- Скрипта-супервизора tbmaster, который отслеживает отдельные экземпляры Tinderbox, записывает их вывод и отправляет уведомления о сбоях по электронной почте.
- Скрипта CGI с именем index.cgi, который читает набор журналов tbmaster и представляет их в виде удобочитаемой HTML-сводки.
- Набора серверов сборки, которые постоянно тестируют последние изменения наиболее важных веток кода FreeBSD.
- Веб-сервера, хранящего полный набор журналов Tinderbox и отображающий актуальную сводку.

Скрипты поддерживаются и были разработаны Dag-Erling Smørgrav <des@FreeBSD.org>, и сейчас написаны на Perl, что стало шагом вперёд по сравнению с их первоначальной версией в виде shell-скриптов. Все скрипты и конфигурационные файлы хранятся в /projects/tinderbox/.

Для получения дополнительной информации о скриптах tinderbox и tbmaster на данном этапе обратитесь к соответствующим руководствам: tinderbox(1) и tbmaster(1).

6.3. Скрипт index.cgi

Скрипт index.cgi генерирует HTML-сводку журналов tinderbox и tbmaster. Хотя изначально он предназначался для использования в качестве CGI-скрипта, как следует из его названия, этот скрипт также может быть запущен из командной строки или из задачи cron(8), в таком случае он будет искать логи в каталоге, где расположен сам скрипт. Он автоматически определяет контекст, генерируя HTTP-заголовки при запуске в качестве CGI-скрипта. Он соответствует стандартам XHTML и использует CSS для стилизации.

Скрипт начинает работу в блоке `main()`, пытаясь проверить, что он выполняется на

официальном сайте Tinderbox. Если это не так, создаётся страница с указанием, что это не официальный сайт, и предоставляется URL официального сайта.

Далее выполняется сканирование каталога журналов для получения перечня конфигураций, веток и архитектур, для которых существуют файлы журналов, чтобы избежать жёсткого задания списка в скрипте и потенциального появления пустых строк или столбцов. Эта информация извлекается из имён файлов журналов, соответствующих следующему шаблону:

```
tinderbox-$config-$branch-$arch-$machine.{brief,full}
```

Конфигурации, используемые на официальных серверах сборки Tinderbox, названы в соответствии с ветками, которые они собирают. Например, конфигурация `releng_8` используется для сборки `RELENG_8`, а также всех поддерживаемых веток выпусков.

После успешного завершения всей процедуры запуска для каждой конфигурации вызывается `do_config()`.

Функция `do_config()` генерирует HTML для отдельной конфигурации Tinderbox.

Он работает, сначала создавая строку заголовка, затем перебирая каждую сборку ветки с указанной конфигурацией, формируя одну строку результатов для каждой следующим образом:

- Для каждого элемента:
  - Для каждой машины в рамках этой архитектуры:
    - Если существует краткий файл журнала, то:
      - Вызвать `success()`, чтобы определить результат сборки.
      - Вывести размер изменения.
      - Вывести размер краткого файла журнала со ссылкой на сам файл журнала.
      - Если также существует полный файл журнала, то:
        - Вывести размер полного файла журнала со ссылкой на сам файл журнала.
    - В противном случае:
      - Нечего не выводить.

Упомянутая выше функция `success()` проверяет краткий лог-файл на наличие строки "tinderbox run completed", чтобы определить, был ли сборка успешной.

Конфигурации и ветви сортируются в соответствии с их рангом. Это вычисляется следующим образом:

- `HEAD` и `CURRENT` имеют ранг 9999.
- `RELENG_x` имеет ранг `xx`99.
- `RELENG_x_y` имеет ранг *xxyy*.

Это означает, что HEAD всегда имеет наивысший приоритет, а ветви RELENG ранжируются в числовом порядке, причём каждая ветвь STABLE имеет более высокий приоритет, чем ветви выпусков, ответвлённые от неё. Например, для FreeBSD 8 порядок от наивысшего к низшему будет следующим:

- RELENG_8 (ранг ветки 899).
- RELENG_8_3 (ранг ветки 803).
- RELENG_8_2 (ранг ветки 802).
- RELENG_8_1 (ранг ветки 801).
- RELENG_8_0 (ранг ветки 800).

Цвета, которые Tinderbox использует для каждой ячейки в таблице, определяются CSS. Успешные сборки отображаются зелёным текстом; неудачные сборки отображаются красным текстом. Цвет блекнет со временем с момента соответствующей сборки, приближаясь к серому каждые полчаса.

6.4. Официальные серверы сборки

Официальные серверы сборки Tinderbox размещены на площадке Sentex Data Communications, которая также предоставляет хостинг для кластера Netperf FreeBSD.

В настоящее время работают три сервера сборки:

freebsd-current.sentex.ca собирает:

- HEAD для amd64, arm, i386, i386/pc98, ia64, mips, powerpc, powerpc64 и sparc64.
- RELENG_9 и поддерживаемые ветки 9.*X* для amd64, arm, i386, i386/pc98, ia64, mips, powerpc, powerpc64 и sparc64.

freebsd-stable.sentex.ca собирает:

- RELENG_8 и поддерживаемые ветки 8.*X* для amd64, i386, i386/pc98, ia64, mips, powerpc и sparc64.

freebsd-legacy.sentex.ca собирает:

- RELENG_7 и поддерживаемые ветки 7.*X* для amd64, i386, i386/pc98, ia64, powerpc и sparc64.

6.5. Официальный сайт со сводками

Сводки и журналы с официальных серверов сборки доступны в сети по адресу http://tinderbox.FreeBSD.org, размещены на Dag-Erling Smørgrav <des@FreeBSD.org> и настроены следующим образом:

- Задание cron(8) проверяет серверы сборки через регулярные интервалы и загружает все новые файлы журналов с помощью rsync(1).
- Apache настроен на использование index.cgi в качестве DirectoryIndex.

Часть II: Межпроцессное взаимодействие

Глава 7. Сокеты

7.1. Обзор

Сокеты BSD выводят межпроцессное взаимодействие на новый уровень. Теперь взаимодействующие процессы не обязательно должны выполняться на одной машине. Они всё ещё *могут*, но не обязаны.

Не только эти процессы не обязаны выполняться на одной машине, они также могут работать под разными операционными системами. Благодаря BSD-сокетам, ваше ПО на FreeBSD может легко взаимодействовать с программой, работающей на Macintosh®, другой — на рабочей станции Sun™, и ещё одной — под Windows® 2000, при этом все они подключены к локальной сети на основе Ethernet.

Но ваше программное обеспечение может так же эффективно взаимодействовать с процессами, работающими в другом здании, на другом континенте, внутри подводной лодки или космического челнока.

Он также может взаимодействовать с процессами, которые не являются частью компьютера (по крайней мере, не в строгом смысле этого слова), а таких устройств, как принтеры, цифровые камеры, медицинское оборудование. Практически со всем, что способно к цифровой коммуникации.

7.2. Сетевое взаимодействие и разнообразие

Мы уже упоминали о *разнообразии* сетевых технологий. Множество различных систем должны взаимодействовать друг с другом. И они должны говорить на одном языке. Также они должны *понимать* этот язык одинаковым образом.

Часто думают, что *язык тела* универсален. Но это не так. В ранней юности отец взял меня с собой в Болгарию. Мы сидели за столиком в парке Софии, когда к нам подошел продавец, предлагая купить жареный миндаль.

Я тогда ещё не знал болгарского, поэтому вместо словесного отказа я покачал головой из стороны в сторону — это «универсальный» язык тела для обозначения *нет*. Продавец тут же начал угощать нас миндалем.

Затем я вспомнил, что мне говорили, будто в Болгарии покачивание головой из стороны в сторону означает *да*. Быстро я начал кивать головой вверх-вниз. Продавец заметил, забрал свои орехи и ушёл. Для непосвящённого наблюдателя я не изменил язык тела: я продолжал использовать движения головой — покачивание и кивание. Изменился *смысл* языка тела. Сначала продавец и я интерпретировали одни и те же жесты как имеющие совершенно разный смысл. Мне пришлось скорректировать свою собственную интерпретацию этих жестов, чтобы продавец меня понял.

То же самое и с компьютерами: одни и те же символы могут иметь разное, даже полностью противоположное значение. Поэтому, чтобы два компьютера понимали друг друга, они должны договориться не только об одном *языке*, но и об одном *толковании* языка.

7.3. Протоколы

В то время как различные языки программирования обычно имеют сложный синтаксис и используют множество многосимвольных зарезервированных слов (что облегчает их понимание для человека-программиста), языки передачи данных, как правило, очень лаконичны. Вместо многосимвольных слов они часто используют отдельные *биты*. Для этого есть очень убедительная причина: хотя данные внутри вашего компьютера передаются со скоростью, близкой к скорости света, между двумя компьютерами они часто передаются значительно медленнее.

Поскольку языки, используемые в передаче данных, настолько лаконичны, мы обычно называем их *протоколами*, а не языками.

При передаче данных от одного компьютера к другому всегда используется более одного протокола. Эти протоколы *располагаются слоями*. Данные можно сравнить с внутренней частью лука: необходимо снять несколько слоёв «кожи», чтобы добраться до данных. Это лучше всего проиллюстрировано на рисунке:

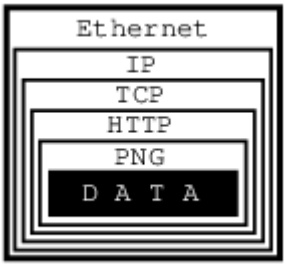

Рисунок 1. Уровни протоколов

В этом примере мы пытаемся получить изображение с веб-страницы, к которой подключены через Ethernet.

Изображение состоит из необработанных данных, которые представляют собой просто последовательность значений RGB, которые наше программное обеспечение может обработать, т.е. преобразовать в изображение и отобразить на нашем мониторе.

Увы, наше программное обеспечение не может определить, как организованы сырые данные: это последовательность значений RGB, последовательность интенсивностей в градациях серого или, возможно, цвета в кодировке CMYK? Представлены ли данные 8-битными квантами, или они имеют размер 16 бит, а может быть, 4 бита? Из скольких строк и столбцов состоит изображение? Должны ли определённые пиксели быть прозрачными?

Я думаю, вы поняли…

Чтобы наше программное обеспечение понимало, как обрабатывать сырые данные, они кодируются в формате PNG. Это мог бы быть GIF или JPEG, но выбран PNG.

И PNG — это протокол.

В этот момент я слышу, как некоторые из вас кричат: *"Нет, это не так! Это формат файла!"*

Ну что ж, конечно, это формат файла. Но с точки зрения передачи данных, формат файла — это протокол: структура файла — это *язык*, причем лаконичный, сообщающий нашему *процессу*, как организованы данные. Следовательно, это *протокол*.

Увы, если бы мы получили только PNG-файл, наше программное обеспечение столкнулось бы с серьёзной проблемой: как ему узнать, что данные представляют изображение, а не текст, звук или что-то ещё? Во-вторых, как ему определить, что изображение сохранено в формате PNG, а не GIF, JPEG или каком-либо другом формате изображений?

Для получения этой информации мы используем другой протокол: HTTP. Этот протокол может точно сообщить нам, что данные представляют изображение и используют протокол PNG. Он также может сообщить некоторые другие сведения, но давайте сосредоточимся на уровнях протоколов здесь.

Итак, теперь у нас есть некоторые данные, упакованные в протокол PNG, который в свою очередь упакован в протокол HTTP. Как мы получили их с сервера?

Используя TCP/IP поверх Ethernet, вот как. Действительно, это ещё три протокола. Вместо того чтобы продолжать изнутри наружу, я теперь расскажу про Ethernet, просто потому что так проще объяснить остальное.

Ethernet — это интересная система соединения компьютеров в *локальной сети* (LAN). У каждого компьютера есть *карта сетевого интерфейса* (NIC — Network Interface Card) с уникальным 48-битным идентификатором, называемым *адресом*. Не существует двух сетевых интерфейсов Ethernet в мире с одинаковым адресом.

Эти сетевые карты соединены между собой. Когда один компьютер хочет связаться с другим в той же локальной сети Ethernet, он отправляет сообщение по сети. Каждая сетевая карта видит это сообщение. Однако, согласно *протоколу* Ethernet, данные содержат адрес сетевой карты назначения (среди прочего). Таким образом, только одна из всех сетевых карт обратит на него внимание, остальные проигнорируют его.

Но не все компьютеры подключены к одной сети. Тот факт, что мы получили данные через наш Ethernet, не означает, что они возникли в нашей локальной сети. Они могли попасть к нам из другой сети (которая может быть даже не на основе Ethernet), соединённой с нашей сетью через Интернет.

Все данные передаются через Интернет с использованием IP, что означает *Internet Protocol*. Его основная роль — сообщать нам, откуда в мире пришли данные и куда они должны быть направлены. Он не *гарантирует*, что мы получим данные, только что мы узнаем, откуда они пришли, *если* мы их получим.

Даже если мы получим данные, IP не гарантирует, что различные фрагменты данных придут в том же порядке, в котором их отправил другой компьютер. Например, мы можем получить центр нашего изображения до того, как получим его верхний левый угол, а после — нижний правый.

Это TCP (*Transmission Control Protocol*), который запрашивает у отправителя повторную отправку потерянных данных и располагает их в правильном порядке.

В итоге потребовалось *пять* различных протоколов, чтобы один компьютер мог сообщить другому, как выглядит изображение. Мы получили данные, упакованные в протокол PNG, который был упакован в протокол HTTP, который был упакован в протокол TCP, который был упакован в протокол IP, который был упакован в протокол Ethernet.

О, и кстати, вероятно, на пути были задействованы и несколько других протоколов. Например, если наша локальная сеть была подключена к Интернету через дозвон, то использовался протокол PPP над модемом, который, в свою очередь, использовал один (или несколько) из различных модемных протоколов, и так далее, и так далее, и так далее…

Как разработчик, вы уже должны задаваться вопросом: *"Как я должен со всем этим справляться?"*

К счастью для вас, вам *не* нужно разбираться во всём этом. Вам *придётся* разобраться в некоторой части, но не во всей. В частности, вам не нужно беспокоиться о физическом подключении (в нашем случае Ethernet и, возможно, PPP и т.д.). Также вам не нужно разбираться с протоколом IP или протоколом TCP.

Другими словами, вам не нужно ничего делать, чтобы получить данные с другого компьютера. Ну, разве что *попросить* их, но это почти так же просто, как открыть файл.

Получив данные, вам предстоит решить, что с ними делать. В нашем случае потребуется понимание протокола HTTP и структуры файла PNG.

Используя аналогию, все межсетевые протоколы становятся серой зоной: не столько потому, что мы не понимаем, как они работают, а потому, что нас это больше не беспокоит. Интерфейс сокетов берёт на себя заботу об этой серой зоне:

Рисунок 2. Уровни протоколов, покрываемые сокетами

Нам нужно понимать только те протоколы, которые говорят нам, как *интерпретировать данные*, а не как *получать* их от другого процесса или как *передавать* их другому процессу.

7.4. Модель сокетов

Сокеты BSD построены по базовой модели UNIX®: *Все является файлом.* Таким образом, в нашем примере сокеты позволят нам получить, образно говоря, *HTTP-файл*. Затем нам предстоит извлечь из него *PNG-файл*.

Из-за сложности межсетевого взаимодействия мы не можем просто использовать системный вызов `open` или функцию `open()` в языке C. Вместо этого необходимо выполнить несколько шагов для "открытия" сокета.

Однако, как только мы это сделаем, мы можем начать обращаться с *сокетом* так же, как и с любым *файловым дескриптором*: мы можем `читать` из него, `писать` в него, передавать его через `канал` и, в конечном итоге, `закрывать` его.

7.5. Основные функции сокетов

В то время как FreeBSD предлагает различные функции для работы с сокетами, нам *требуется* только четыре, чтобы "открыть" сокет. А в некоторых случаях достаточно двух.

7.5.1. Разница между клиентом и сервером

Обычно одним из концов связи на основе сокетов является *сервер*, а другой — *клиент*.

7.5.1.1. Общие элементы

7.5.1.1.1. `socket`

Функция, используемая как клиентами, так и серверами, это socket(2). Она объявляется следующим образом:

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

Возвращаемое значение имеет тот же тип, что и у `open`, целое число. FreeBSD выделяет его значение из того же пула, что и дескрипторы файлов. Это позволяет обрабатывать сокеты так же, как файлы.

Аргумент `domain` указывает системе, какое *семейство протоколов* следует использовать. Существует множество семейств, некоторые из них специфичны для определённых поставщиков, другие широко распространены. Они объявлены в sys/socket.h.

Используйте `PF_INET` для UDP, TCP и других интернет-протоколов (IPv4).

Для аргумента `type` определено пять значений, также указанных в sys/socket.h. Все они начинаются с “SOCK_”. Наиболее распространённое — `SOCK_STREAM`, которое указывает системе, что запрашивается *надёжный сервис потоковой доставки* (это TCP при использовании с `PF_INET`).

Если бы вы запросили `SOCK_DGRAM`, вы бы запросили *сервис доставки датаграмм без установления соединения* (в нашем случае, UDP).

Если вы хотите управлять низкоуровневыми протоколами (такими как IP) или даже сетевыми интерфейсами (например, Ethernet), вам потребуется указать `SOCK_RAW`.

Наконец, аргумент `protocol` зависит от двух предыдущих аргументов и не всегда имеет смысл. В таком случае используйте значение `0`.

Неподключенный сокет

Нигде в функции `socket` мы не указали, к какой другой системе должны быть

подключены. Наш только что созданный сокет остаётся *неподключённым*.

Это сделано намеренно: если проводить аналогию с телефоном, мы только что подключили модем к телефонной линии. Мы не сказали модему совершить звонок или ответить, если телефон зазвонит.

7.5.1.1.2. sockaddr

Различные функции семейства сокетов ожидают адрес (или указатель, если использовать терминологию языка C) небольшой области памяти. Различные объявления на языке C в файле sys/socket.h ссылаются на неё как на `struct sockaddr`. Эта структура объявлена в том же файле:

```
/*
 * Structure used by kernel to store most
 * addresses.
 */
struct sockaddr {
    unsigned char   sa_len;     /* total length */
    sa_family_t sa_family;  /* address family */
    char        sa_data[14];    /* actually longer; address value */
};
#define SOCK_MAXADDRLEN 255     /* longest possible addresses */
```

Обратите внимание на *неопределённость*, с которой объявлено поле `sa_data` — просто как массив из 14 байт, с комментарием, намекающим, что их может быть больше 14.

Эта неопределённость вполне преднамеренна. Сокеты — это очень мощный интерфейс. Хотя большинство людей, возможно, считают их не более чем интерфейсом для Интернета — и большинство приложений, вероятно, используют их именно для этого в наши дни — сокеты могут быть использованы практически для *любого* вида межпроцессного взаимодействия, из которых Интернет (или, точнее, IP) — лишь один из них.

sys/socket.h ссылается на различные типы протоколов, с которыми работают сокеты, как на *семейства адресов*, и перечисляет их непосредственно перед определением `sockaddr`:

```
/*
 * Address families.
 */
#define AF_UNSPEC   0       /* unspecified */
#define AF_LOCAL    1       /* local to host (pipes, portals) */
#define AF_UNIX     AF_LOCAL    /* backward compatibility */
#define AF_INET     2       /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK  3       /* arpanet imp addresses */
#define AF_PUP      4       /* pup protocols: e.g. BSP */
#define AF_CHAOS    5       /* mit CHAOS protocols */
#define AF_NS       6       /* XEROX NS protocols */
#define AF_ISO      7       /* ISO protocols */
#define AF_OSI      AF_ISO
```

```
#define AF_ECMA     8       /* European computer manufacturers */
#define AF_DATAKIT  9       /* datakit protocols */
#define AF_CCITT    10      /* CCITT protocols, X.25 etc */
#define AF_SNA      11      /* IBM SNA */
#define AF_DECnet   12      /* DECnet */
#define AF_DLI      13      /* DEC Direct data link interface */
#define AF_LAT      14      /* LAT */
#define AF_HYLINK   15      /* NSC Hyperchannel */
#define AF_APPLETALK    16      /* Apple Talk */
#define AF_ROUTE    17      /* Internal Routing Protocol */
#define AF_LINK     18      /* Link layer interface */
#define pseudo_AF_XTP   19      /* eXpress Transfer Protocol (no AF) */
#define AF_COIP     20      /* connection-oriented IP, aka ST II */
#define AF_CNT      21      /* Computer Network Technology */
#define pseudo_AF_RTIP  22      /* Help Identify RTIP packets */
#define AF_IPX      23      /* Novell Internet Protocol */
#define AF_SIP      24      /* Simple Internet Protocol */
#define pseudo_AF_PIP   25      /* Help Identify PIP packets */
#define AF_ISDN     26      /* Integrated Services Digital Network*/
#define AF_E164     AF_ISDN     /* CCITT E.164 recommendation */
#define pseudo_AF_KEY   27      /* Internal key-management function */
#define AF_INET6    28      /* IPv6 */
#define AF_NATM     29      /* native ATM access */
#define AF_ATM      30      /* ATM */
#define pseudo_AF_HDRCMPLT 31       /* Used by BPF to not rewrite headers
                     * in interface output routine
                     */
#define AF_NETGRAPH 32      /* Netgraph sockets */
#define AF_SLOW     33      /* 802.3ad slow protocol */
#define AF_SCLUSTER 34      /* Sitara cluster protocol */
#define AF_ARP      35
#define AF_BLUETOOTH    36      /* Bluetooth sockets */
#define AF_MAX      37
```

Используемый для IP — это AF_INET. Это символ для константы 2.

Это *семейство адресов*, указанное в поле `sa_family` структуры `sockaddr`, определяет, как именно будут использоваться нечетко названные байты `sa_data`.

В частности, когда *семейство адресов* — AF_INET, можно использовать `struct sockaddr_in` из netinet/in.h везде, где ожидается `sockaddr`:

```
/*
 * Socket address, internet style.
 */
struct sockaddr_in {
    uint8_t     sin_len;
    sa_family_t sin_family;
    in_port_t   sin_port;
    struct  in_addr sin_addr;
```

```
    char    sin_zero[8];
};
```

Мы можем визуализировать его организацию следующим образом:

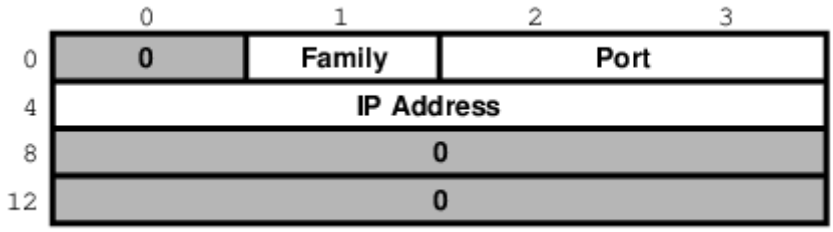

Рисунок 3. Структура `sockaddr_in`

Три важных поля — это `sin_family`, которое находится в байте 1 структуры, `sin_port`, 16-битное значение, расположенное в байтах 2 и 3, и `sin_addr`, 32-битное целочисленное представление IP-адреса, хранящееся в байтах 4–7.

Теперь попробуем заполнить его. Предположим, мы пытаемся написать клиент для протокола *daytime*, который просто указывает, что его сервер записывает текстовую строку с текущей датой и временем в порт 13. Мы хотим использовать TCP/IP, поэтому нам нужно указать `AF_INET` в поле семейства адресов. `AF_INET` определен как `2`. Давайте используем IP-адрес `132.163.96.1`, который является сервером времени федерального правительства США (`time.nist.gov`).

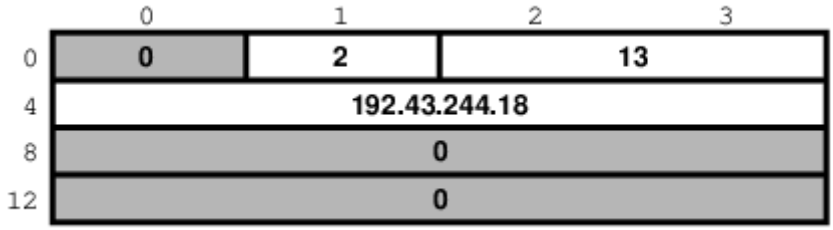

Рисунок 4. Конкретный пример sockaddr_in

Кстати, поле `sin_addr` объявлено как имеющее тип `struct in_addr`, который определён в netinet/in.h:

```
/*
 * Internet address (a structure for historical reasons)
 */
struct in_addr {
    in_addr_t s_addr;
};
```

В дополнение, `in_addr_t` является 32-битным целым числом.

`132.163.96.1` — это просто удобная форма записи 32-битного целого числа, в которой перечисляются все его 8-битные байты, начиная с *старшего*.

До сих пор мы рассматривали `sockaddr` как абстракцию. Наш компьютер не хранит `short` целые числа как единую 16-битную сущность, а как последовательность 2 байт. Аналогично, он хранит 32-битные целые числа как последовательность 4 байт.

Предположим, мы написали что-то вроде этого:

```
sa.sin_family      = AF_INET;
sa.sin_port        = 13;
sa.sin_addr.s_addr = (((((132 << 8) | 163) << 8) | 96) << 8) | 1;
```

Как будет выглядеть результат?

Ну, это, конечно, зависит от многого. На компьютере с процессором Pentium® или другим на базе x86 это будет выглядеть так:

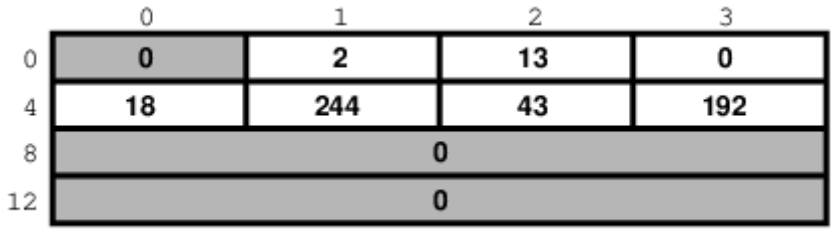

Рисунок 5. `sockaddr_in` *в системе с архитектурой Intel*

На другой системе это может выглядеть так:

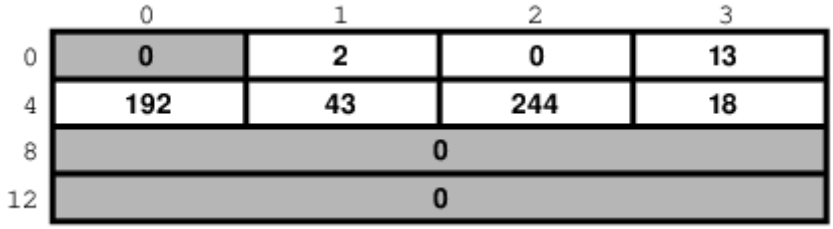

Рисунок 6. `sockaddr_in` *в системе с порядком байтов от старшего к младшему*

И на PDP это может выглядеть иначе. Однако два приведённых выше варианта являются наиболее распространёнными на сегодняшний день.

Обычно, стремясь писать переносимый код, программисты делают вид, что этих различий не существует. И им это сходит с рук (за исключением случаев, когда они пишут на ассемблере). Увы, при программировании сокетов так легко отделаться не получится.

Почему?

Потому что при обмене данными с другим компьютером вы обычно не знаете, хранит ли он данные, начиная со *старшего байта* (MSB) или с *младшего байта* (LSB).

Вы можете задаться вопросом: *"Значит, сокеты не будут это делать за меня?"*

Не будут.

Хотя этот ответ может сначала вас удивить, помните, что общий интерфейс сокетов понимает только поля `sa_len` и `sa_family` структуры `sockaddr`. Вам не нужно беспокоиться о порядке байтов (конечно, в FreeBSD `sa_family` занимает всего 1 байт, но многие другие UNIX®-системы не имеют `sa_len` и используют 2 байта для `sa_family`, ожидая данные в том порядке, который является родным для компьютера).

Но остальные данные — это просто `sa_data[14]` с точки зрения сокетов. В зависимости от *семейства адресов* сокеты просто передают эти данные по назначению.

Действительно, когда мы указываем номер порта, это делается для того, чтобы другая компьютерная система знала, какую службу мы запрашиваем. И, когда мы выступаем в роли сервера, мы считываем номер порта, чтобы понять, какую службу ожидает от нас другая система. В любом случае, сокетам нужно лишь передать номер порта в качестве данных. Они никак его не интерпретируют.

Аналогично, мы указываем IP-адрес, чтобы сообщить всем на пути, куда отправлять наши данные. Сокеты, опять же, просто пересылают их как данные.

Вот почему мы (программисты, а не сокеты) должны различать порядок байтов, используемый нашим компьютером, и условный порядок байтов для отправки данных на другой компьютер.

Мы будем называть порядок байтов, который использует наш компьютер, *порядком байтов хоста* или просто *хост-порядком*.

Существует соглашение о передаче многобайтовых данных по IP *старшим байтом вперёд*. Это мы будем называть *порядком байтов сети* или просто *сетевым порядком*.

Вот, если бы мы скомпилировали приведённый выше код для компьютера на базе Intel, наш *порядок байтов хоста* выдал бы:

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

Рисунок 7. Порядок байтов на хосте в системе Intel

Но порядок байтов в *сетевом формате* требует, чтобы данные хранились начиная со старшего байта (MSB):

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

Рисунок 8. Порядок байтов в сети

К сожалению, наш *порядок хоста* полностью противоположен *порядку сети*.

У нас есть несколько способов решения этой проблемы. Один из них — *инвертировать* значения в нашем коде:

```
sa.sin_family      = AF_INET;
sa.sin_port        = 13 << 8;
sa.sin_addr.s_addr = (((((1 << 8) | 96) << 8) | 163) << 8) | 132;
```

Это *обманет* наш компилятор, заставив его сохранить данные в *порядке байтов сети*. В некоторых случаях это именно тот способ, который нужен (например, при программировании на ассемблере). Однако в большинстве случаев это может вызвать проблему.

Предположим, вы написали программу на C, использующую сокеты. Вы знаете, что она будет работать на Pentium®, поэтому вводите все константы в обратном порядке и приводите их к *порядку байтов сети*. Она работает хорошо.

Затем, однажды, ваш надёжный старый Pentium® превращается в ржавый старый Pentium®. Вы заменяете его системой, у которой *порядок байтов хоста* совпадает с *сетевым порядком байтов*. Вам нужно перекомпилировать все ваше программное обеспечение. Все ваши программы продолжают работать хорошо, кроме той одной программы, которую вы написали.

Вы уже забыли, что принудительно задали все свои константы противоположными *порядку хоста*. Вы проводите некоторое время, яростно рвя на себе волосы, взывая ко всем известным вам богам (и к некоторым, которых вы придумали), стуча нерф-битой по монитору и выполняя прочие традиционные ритуалы в попытке понять, почему то, что работало так хорошо, внезапно перестало работать вообще.

В конце концов, вы разбираетесь в проблеме, произносите пару крепких словечек и начинаете переписывать свой код.

К счастью, вы не первый, кто столкнулся с этой проблемой. Кто-то уже создал функции htons(3) и htonl(3) на языке C для преобразования `short` и `long` соответственно из *порядка байтов хоста* в *порядок байтов сети*, а также функции ntohs(3) и ntohl(3) на языке C для обратного преобразования.

На системах с порядком *старший байт первый* эти функции не выполняют никаких действий. На системах с порядком *младший байт первый* они преобразуют значения в правильный порядок.

Итак, независимо от того, на какой системе компилируется ваше программное обеспечение, ваши данные будут в правильном порядке, если вы используете эти функции.

7.5.1.2. Функции клиента

Обычно клиент инициирует подключение к серверу. Клиент знает, к какому серверу он собирается обратиться: он знает его IP-адрес и *порт*, на котором работает сервер. Это

похоже на то, как вы поднимаете трубку и набираете номер (*адрес*), а затем, когда кто-то отвечает, просите соединить со специалистом по непонятным символам (*порт*).

7.5.1.2.1. `connect`

Как только клиент создал сокет, ему нужно подключить его к определённому порту на удалённой системе. Для этого используется connect(2):

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

Аргумент `s` — это сокет, то есть значение, возвращаемое функцией `socket`. Аргумент `name` — это указатель на структуру `sockaddr`, которую мы подробно обсуждали. Наконец, `namelen` сообщает системе, сколько байт находится в нашей структуре `sockaddr`.

Если `connect` завершается успешно, он возвращает `0`. В противном случае возвращается `-1`, а код ошибки сохраняется в `errno`.

Существует множество причин, по которым `connect` может завершиться неудачей. Например, при попытке подключения к интернету, IP-адрес может не существовать, быть недоступен, перегружен или на указанном порту может не быть сервера. Или же подключение может быть явно *отклонено* по определённым причинам.

7.5.1.2.2. Наш первый клиент

Теперь мы знаем достаточно, чтобы написать очень простого клиента, который получит текущее время от `132.163.96.1` и выведет его в stdout.

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

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

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

```
  memset(&sa, '\0', sizeof(sa));

  sa.sin_family = AF_INET;
  sa.sin_port = htons(13);
  sa.sin_addr.s_addr = htonl((((((132 << 8) | 163) << 8) | 96) << 8) | 1);
  if (connect(s, (struct sockaddr *)&sa, sizeof sa) < 0) {
    perror("connect");
    close(s);
    return 2;
  }

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

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

Вперед! Введите это в вашем редакторе, сохраните как daytime.c, затем скомпилируйте и запустите:

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

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

В данном случае дата была 19 июня 2001 года, время — 02:29:25 UTC. Естественно, ваши результаты могут отличаться.

7.5.1.3. Функции сервера

Типичный сервер не инициирует соединение. Вместо этого он ожидает, когда клиент обратится к нему и запросит услуги. Он не знает, когда клиент обратится, ни сколько клиентов обратится. В один момент он может просто спокойно ожидать, а в следующий момент он может оказаться перегруженным запросами от множества клиентов, обращающихся одновременно.

Интерфейс сокетов предоставляет три основные функции для обработки этого.

7.5.1.3.1. `bind`

Порты подобны внутренним номерам телефонной линии: после набора основного номера вы набираете внутренний номер, чтобы связаться с конкретным человеком или отделом.

Существует 65535 IP-портов, но сервер обычно обрабатывает запросы, поступающие только на один из них. Это как сказать оператору телефонной комнаты, что мы сейчас на месте и готовы отвечать на звонки по определённому внутреннему номеру. Мы используем bind(2), чтобы указать сокетам, на каком порту мы хотим обслуживать запросы.

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

Помимо указания порта в `addr`, сервер может включать свой IP-адрес. Однако он может просто использовать символическую константу INADDR_ANY, чтобы указать, что будет обслуживать все запросы на указанный порт, независимо от его IP-адреса. Этот символ, наряду с несколькими аналогичными, объявлен в netinet/in.h

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

Предположим, мы пишем сервер для протокола *daytime* поверх TCP/IP. Напомним, что он использует порт 13. Наша структура `sockaddr_in` будет выглядеть так:

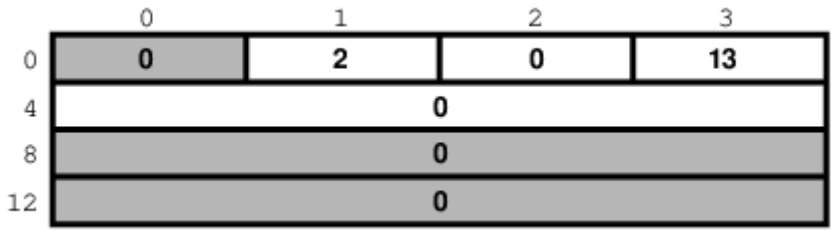

Рисунок 9. Пример sockaddr_in сервера

7.5.1.3.2. `listen`

Продолжая аналогию с офисным телефоном, после того как вы сообщили оператору АТС, на каком внутреннем номере вы будете находиться, вы заходите в свой офис и убеждаетесь, что ваш телефон подключен и звонок включен. Кроме того, вы активируете функцию ожидания вызова, чтобы слышать звонок даже во время разговора с кем-то.

Сервер обеспечивает все это с помощью функции listen(2).

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

Здесь переменная `backlog` указывает сокетам, сколько входящих запросов принимать, пока вы заняты обработкой последнего запроса. Другими словами, она определяет максимальный размер очереди ожидающих соединений.

7.5.1.3.3. `accept`

После того как вы услышите телефонный звонок, вы принимаете вызов, отвечая на звонок. Теперь вы установили соединение с вашим клиентом. Это соединение остаётся активным, пока вы или ваш клиент не повесите трубку.

Сервер принимает соединение, используя функцию accept(2).

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

Обратите внимание, что в этот раз `addrlen` является указателем. Это необходимо, потому что в данном случае именно сокет заполняет структуру `addr` — `sockaddr_in`.

Возвращаемое значение является целым числом. Действительно, `accept` возвращает *новый сокет*. Этот новый сокет будет использоваться для обмена данными с клиентом.

Что происходит со старым сокетом? Он продолжает ожидать новые запросы (помните переменную `backlog`, которую мы передали в `listen`?), пока мы не закроем его (`close`).

Теперь новый сокет предназначен только для обмена данными. Он полностью подключен. Мы не можем снова передать его в `listen`, чтобы принимать дополнительные соединения.

7.5.1.3.4. Наш первый сервер

Наш первый сервер будет несколько сложнее, чем первый клиент: нам нужно не только использовать больше функций сокетов, но и написать его как демон.

Это лучше всего достигается созданием *дочернего процесса* после привязки порта. Затем основной процесс завершается и возвращает управление оболочке (или любой другой программе, которая его вызвала).

Дочерний процесс вызывает `listen`, затем запускает бесконечный цикл, который принимает соединение, обслуживает его и в конечном итоге закрывает свой сокет.

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

#define BACKLOG 4

int main() {
    int s, c;
    socklen_t b;
    struct sockaddr_in sa;
    time_t t;
    struct tm *tm;
    FILE *client;

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

```
    }

    memset(&sa, '\0', sizeof(sa));

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

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

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

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

    listen(s, BACKLOG);

    for (;;) {
        b = sizeof sa;

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

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

        if ((t = time(NULL)) < 0) {
            perror("daytimed time");
            return 6;
        }

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

```
            tm->tm_hour,
            tm->tm_min,
            tm->tm_sec);

        fclose(client);
    }
}
```

Начинаем с создания сокета. Затем заполняем структуру `sockaddr_in` в `sa`. Обратите внимание на условное использование INADDR_ANY:

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

Его значение равно `0`. Поскольку мы только что использовали `memset`, чтобы установить в 0 все байты структуры, будет избыточным снова устанавливать его в `0`. Но если мы перенесём наш код на другую систему, где INADDR_ANY, возможно, не равен нулю, нам нужно будет присвоить его `sa.sin_addr.s_addr`. Большинство современных компиляторов C достаточно умны, чтобы заметить, что INADDR_ANY — это константа. Пока она равна нулю, они оптимизируют все условное выражение из кода.

После успешного вызова `bind` мы готовы стать *демоном*: используем `fork` для создания дочернего процесса. В обоих процессах, родительском и дочернем, переменная `s` является нашим сокетом. Родительскому процессу он больше не нужен, поэтому он вызывает `close`, затем возвращает `0`, чтобы сообщить своему родителю об успешном завершении.

Между тем, дочерний процесс продолжает работать в фоновом режиме. Он вызывает `listen` и устанавливает размер очереди ожидания (`backlog`) равным `4`. Здесь не требуется большое значение, так как *daytime* — это не протокол, который часто запрашивают клиенты, и, кроме того, он может мгновенно обрабатывать каждый запрос.

Наконец, демон запускает бесконечный цикл, который выполняет следующие шаги:

1. Вызовите `accept`. Он ожидает здесь, пока клиент не свяжется с ним. В этот момент он получает новый сокет, `c`, который можно использовать для обмена данными с этим конкретным клиентом.
2. Он использует функцию C `fdopen` для преобразования сокета из низкоуровневого *дескриптора файла* в указатель типа `FILE` в стиле C. Это позволит в дальнейшем использовать `fprintf`.
3. Он проверяет время и выводит его в формате *ISO 8601* в «файл» `client`. Затем он использует `fclose` для закрытия файла. Это также автоматически закроет сокет.

Мы можем *обобщить* это и использовать в качестве модели для многих других серверов:

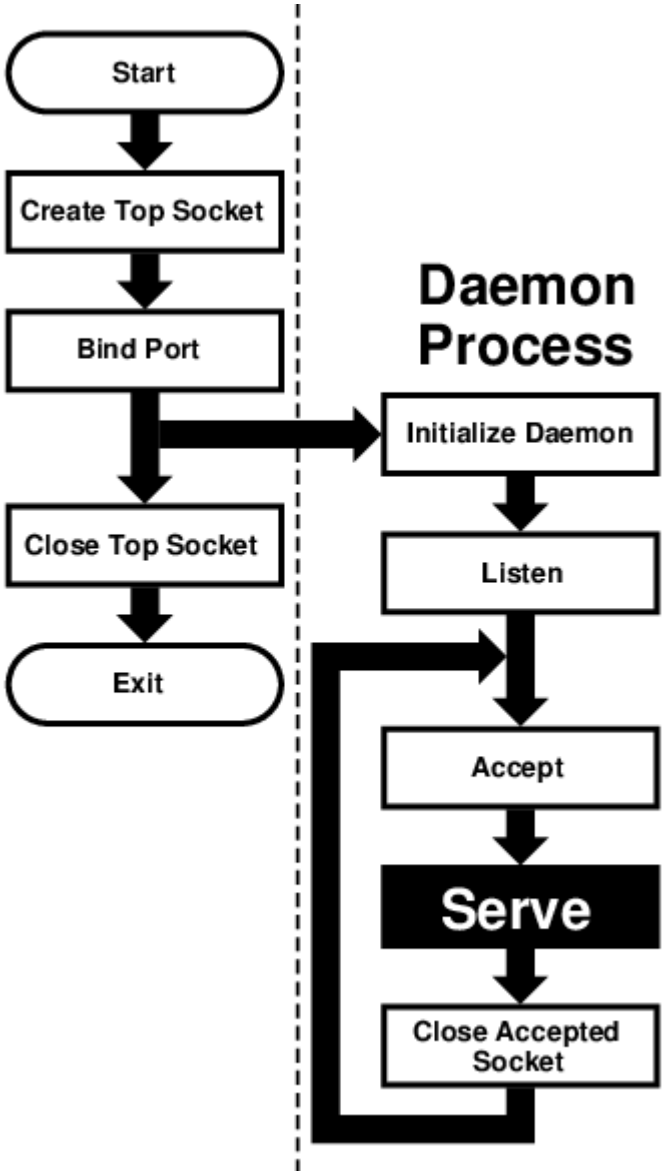

Рисунок 10. Последовательный Сервер

Эта блок-схема подходит для *последовательных серверов*, то есть серверов, которые могут обслуживать одного клиента за раз, как это было возможно с нашим *daytime* сервером. Это возможно только в тех случаях, когда между клиентом и сервером не происходит реального "диалога": как только сервер обнаруживает подключение клиента, он отправляет некоторые данные и закрывает соединение. Вся операция может занять наносекунды, и она завершена.

Преимущество этой блок-схемы в том, что, за исключением короткого момента после того, как родительский процесс выполняет `fork` и до его завершения, всегда активен только один *процесс*: Наш сервер не занимает много памяти и других системных ресурсов.

Обратите внимание, что мы добавили *инициализацию демона* в нашу блок-схему. Нам не нужно было инициализировать собственный демон, но это подходящее место в потоке выполнения программы для настройки обработчиков `signal`, открытия необходимых файлов и т. д.

Почти все элементы блок-схемы могут быть использованы буквально на множестве различных серверов. Элемент *serve* является исключением. Мы рассматриваем его как *"чёрный ящик"*, то есть нечто, что вы проектируете специально для своего сервера и просто

"подключаете к остальной системе."

Не все протоколы настолько просты. Многие получают запрос от клиента, отвечают на него, а затем получают ещё один запрос от того же клиента. В результате, они не знают заранее, как долго будут обслуживать клиента. Такие серверы обычно запускают новый процесс для каждого клиента. Пока новый процесс обслуживает своего клиента, демон может продолжать прослушивать новые подключения.

Теперь сохраните приведённый исходный код в файл daytimed.c (обычно имена демонов оканчиваются буквой `d`). После компиляции попробуйте запустить его:

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

Что произошло? Как вы помните, протокол *daytime* использует порт 13. Однако все порты ниже 1024 зарезервированы для суперпользователя (в противном случае любой мог бы запустить демон, притворяясь, что обслуживает часто используемый порт, создавая угрозу безопасности).

Попробуйте снова, на этот раз как суперпользователь:

```
# ./daytimed
#
```

Что… Ничего? Давайте попробуем ещё раз:

```
# ./daytimed

bind: Address already in use
#
```

Каждый порт может быть связан только одной программой одновременно. Наша первая попытка действительно была успешной: она запустила дочерний демон и завершилась без ошибок. Он продолжает работать и будет работать до тех пор, пока вы его не завершите командой kill, пока какой-либо из его системных вызовов не завершится с ошибкой или пока вы не перезагрузите систему.

Хорошо, мы знаем, что он работает в фоновом режиме. Но работает ли он? Как мы можем убедиться, что это настоящий сервер *daytime*? Просто:

```
% telnet localhost 13

Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
```

```
Connected to localhost.
Escape character is '^]'.
2001-06-19T21:04:42Z
Connection closed by foreign host.
%
```

telnet попробовал использовать новый IPv6, но не смог. Затем он повторил попытку с IPv4, и это удалось. Демон работает.

Если у вас есть доступ к другой UNIX®-системе через telnet, вы можете использовать её для проверки удалённого доступа к серверу. Мой компьютер не имеет статического IP-адреса, поэтому я сделал следующее:

```
% who

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

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

Снова, это сработало. Сработает ли это с использованием доменного имени?

```
% telnet r47.bfm.org 13

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

Кстати, telnet выводит сообщение *Connection closed by foreign host* после того, как наш демон закрыл сокет. Это показывает, что использование `fclose(client);` в нашем коде действительно работает, как заявлено.

7.6. Вспомогательные функции

Библиотека C в FreeBSD содержит множество вспомогательных функций для программирования сокетов. Например, в нашем примере клиента мы жёстко прописали IP-адрес `time.nist.gov`. Но мы не всегда знаем IP-адрес. Даже если знаем, наше программное обеспечение будет более гибким, если позволит пользователю ввести IP-адрес или даже

доменное имя.

7.6.1. gethostbyname

Хотя нет возможности передать имя домена напрямую в какие-либо функции сокетов, стандартная библиотека C в FreeBSD предоставляет функции gethostbyname(3) и gethostbyname2(3), объявленные в netdb.h.

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

Оба возвращают указатель на структуру `hostent`, содержащую много информации о домене. Для наших целей поле `h_addr_list[0]` структуры указывает на `h_length` байтов правильного адреса, уже сохранённого в *порядке байтов сети*.

Это позволяет нам создать гораздо более гибкую — и гораздо более полезную — версию нашей программы daytime:

```
/*
 * daytime.c
 *
 * Programmed by G. Adam Stanislav
 * 19 June 2001
 */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

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

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

  memset(&sa, '\0', sizeof(sa));

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

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

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

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

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

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

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

Теперь мы можем ввести доменное имя (или IP-адрес, это работает в обоих направлениях) в командной строке, и программа попытается подключиться к его серверу *daytime*. В противном случае по умолчанию будет использоваться `time.nist.gov`. Однако даже в этом случае мы будем использовать `gethostbyname` вместо жёсткого указания `132.163.96.1`. Таким образом, даже если его IP-адрес изменится в будущем, мы всё равно сможем его найти.

Поскольку получение времени от локального сервера занимает практически нулевое время, вы можете запустить daytime дважды подряд: сначала для получения времени от `time.nist.gov`, а затем от вашей собственной системы. После этого вы можете сравнить результаты и увидеть, насколько точны часы вашей системы:

```
% daytime ; daytime localhost

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

Как видно, моя система опережала время NIST на две секунды.

7.6.2. `getservbyname`

Иногда вы можете быть не уверены, какой порт использует определённая служба. В таких случаях очень полезна функция getservbyname(3), также объявленная в netdb.h:

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

Структура `servent` содержит `s_port`, в котором находится соответствующий порт, уже в *порядке байтов сети*.

Если бы мы не знали правильный порт для службы *daytime*, мы могли бы найти его следующим образом:

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

Обычно порт известен. Но если вы разрабатываете новый протокол, вы можете тестировать его на неофициальном порту. Когда-нибудь вы зарегистрируете протокол и его порт (если не где-то ещё, то хотя бы в вашем /etc/services, где `getservbyname` ищет). Вместо возврата ошибки в приведённом выше коде вы просто используете временный номер порта. Как только вы добавите протокол в /etc/services, ваше программное обеспечение найдёт его порт без необходимости переписывать код.

7.7. Многозадачные серверы

В отличие от последовательного сервера, *многозадачный сервер* должен иметь возможность обслуживать более одного клиента одновременно. Например, *сервер чата* может обслуживать конкретного клиента часами — он не может ждать, пока закончит обслуживать текущего клиента, прежде чем перейти к следующему.

Это требует значительных изменений в нашей блок-схеме:

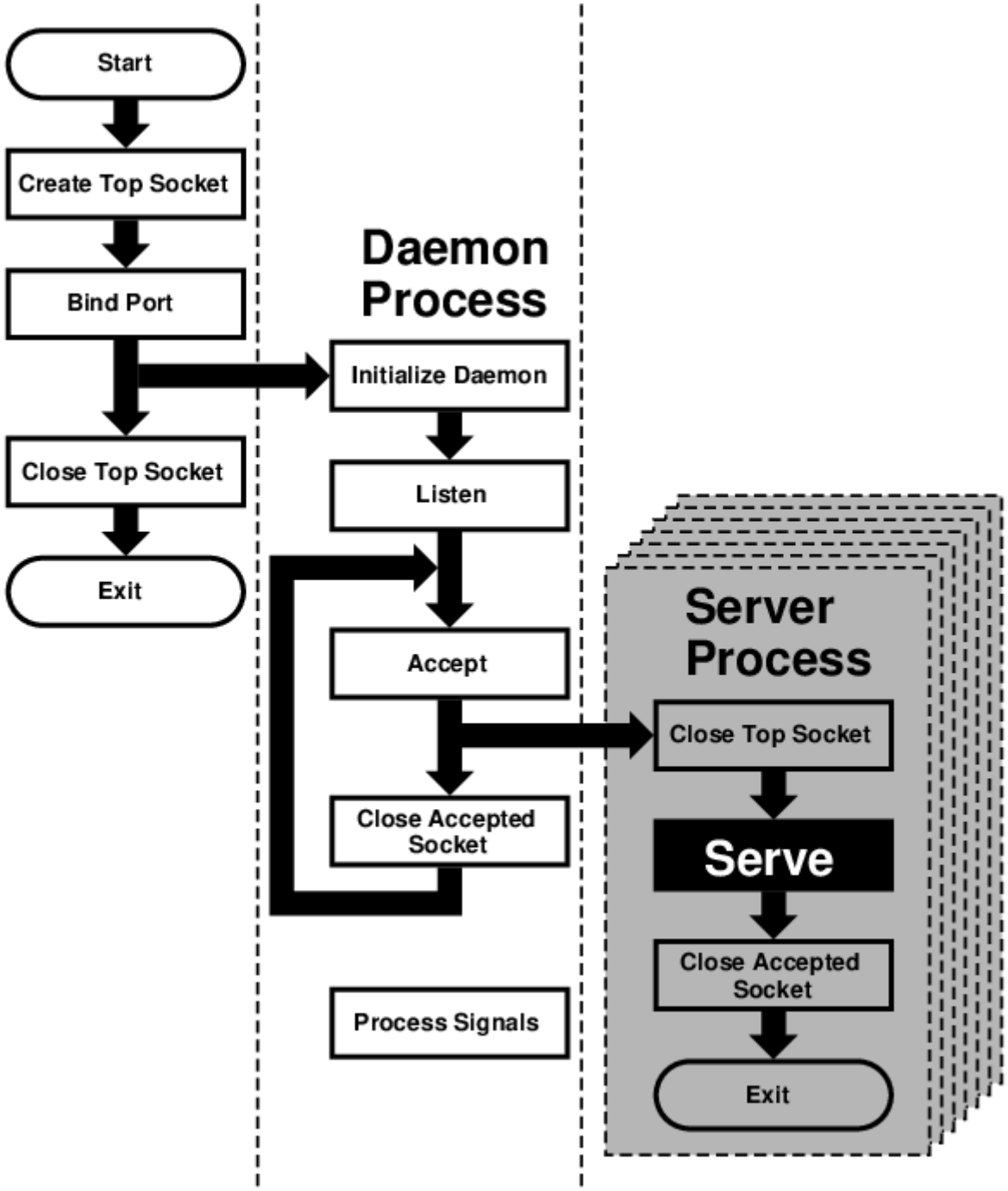

Рисунок 11. Многозадачный сервер

Мы переместили *службу* из *демона* в её собственный *серверный процесс*. Однако, поскольку каждый дочерний процесс наследует все открытые файлы (а сокет обрабатывается так же, как файл), новый процесс наследует не только *"принятый дескриптор"*, т.е. сокет, возвращённый вызовом `accept`, но и *главный сокет*, т.е. тот, который был открыт главным процессом в самом начале.

Однако *серверному процессу* этот сокет не нужен, и он должен немедленно вызвать ему `close`. Аналогично, *демону* больше не нужен *сокет, принятый вызовом accept*, и он не только должен, но и *обязан* вызвать ему `close` — в противном случае рано или поздно закончатся доступные *файловые дескрипторы*.

После завершения обслуживания *серверного процесса* он должен закрыть *принятый сокет*. Вместо возврата к `accept`, процесс теперь завершается.

В UNIX® процесс на самом деле не *завершается*. Вместо этого он *возвращается* к своему родителю. Обычно родительский процесс `ждёт` (wait) завершения своего дочернего процесса и получает возвращаемое значение. Однако наш *демон-процесс* не может просто остановиться и ждать. Это бы свело на нет всю цель создания дополнительных процессов. Но если он никогда не выполняет `wait`, его дочерние процессы станут *зомби* — более не функционирующими, но всё ещё бродящими вокруг.

По этой причине *демону* необходимо установить *обработчики сигналов* на этапе *инициализации демона*. Как минимум, должен обрабатываться сигнал SIGCHLD, чтобы демон мог удалять зомби-процессы из системы и освобождать занимаемые ими системные ресурсы.

Вот почему наша блок-схема теперь содержит блок *обработки сигналов*, который не соединен с другими блоками. Кстати, многие серверы также обрабатывают SIGHUP и обычно интерпретируют его как сигнал от суперпользователя, указывающий на необходимость перечитать конфигурационные файлы. Это позволяет нам изменять настройки без необходимости завершать и перезапускать эти серверы.

Глава 8. Внутреннее устройство IPv6

8.1. Реализация IPv6/IPsec

В этом разделе мы объясним внутреннюю реализацию, связанную с IPv6 и IPsec. Данная функциональность заимствована из проекта KAME

8.1.1. IPv6

8.1.1.1. Соответствие

Функции, связанные с IPv6, соответствуют или пытаются соответствовать последнему набору спецификаций IPv6. Для дальнейшего использования мы приводим некоторые из соответствующих документов ниже (*ПРИМЕЧАНИЕ*: это не полный список — его слишком сложно поддерживать...).

Для подробностей обратитесь к соответствующей главе документа, RFC, страницам Справочника или комментариям в исходном коде.

Тесты на соответствие стандартам были проведены для KAME STABLE в проекте TAHI. Результаты можно посмотреть по ссылке http://www.tahi.org/report/KAME/. Мы также участвовали в тестах IOL Университета Нью-Гэмпшира (http://www.iol.unh.edu/) в прошлом, используя наши предыдущие версии.

- RFC1639: FTP Operation Over Big Address Records (FOOBAR)
  - RFC2428 предпочтительнее RFC1639. FTP-клиенты сначала попробуют RFC2428, затем RFC1639 в случае неудачи.
- RFC1886: DNS Extensions to support IPv6
- RFC1933: Transition Mechanisms for IPv6 Hosts and Routers
  - IPv4-совместимый адрес не поддерживается.
  - автоматическое туннелирование (описано в разделе 4.3 данного RFC) не поддерживается.
  - Интерфейс gif(4) реализует IPv[46]-поверх-IPv[46] туннель в общем виде и включает "настроенный туннель", описанный в спецификации. За подробностями обратитесь к разделу Универсальный туннельный интерфейс этого документа.
- RFC1981: Path MTU Discovery for IPv6
- RFC2080: RIPng for IPv6
  - usr.sbin/route6d это поддерживает.
- RFC2292: Advanced Sockets API for IPv6
  - Для поддерживаемых функций библиотек/API ядра см. sys/netinet6/ADVAPI.
- RFC2362: Protocol Independent Multicast-Sparse Mode (PIM-SM)
  - RFC2362 определяет форматы пакетов для PIM-SM. draft-ietf-pim-ipv6-01.txt написан на

основе этого.

- RFC2373: IPv6 Addressing Architecture
  - поддерживает обязательные адреса узлов и соответствует требованиям области видимости.
- RFC2374: An IPv6 Aggregatable Global Unicast Address Format
  - поддерживает 64-битную длину Идентификатора Интерфейса.
- RFC2375: IPv6 Multicast Address Assignments
  - Пользовательские приложения используют общеизвестные адреса, назначенные в RFC.
- RFC2428: FTP Extensions for IPv6 and NATs
  - RFC2428 предпочтительнее RFC1639. FTP-клиенты сначала попробуют RFC2428, затем RFC1639 в случае неудачи.
- RFC2460: IPv6 specification
- RFC2461: Neighbor discovery for IPv6
  - См. раздел Функция "Обнаружение соседей" в этом документе для получения подробностей.
- RFC2462: IPv6 Stateless Address Autoconfiguration
  - См. раздел Plug and Play (подключи и работай) в этом документе для получения подробностей.
- RFC2463: ICMPv6 for IPv6 specification
  - См. ICMPv6 в этом документе для получения подробностей.
- RFC2464: Transmission of IPv6 Packets over Ethernet Networks
- RFC2465: MIB for IPv6: Textual Conventions and General Group
  - Необходимая статистика собирается ядром. Фактическая поддержка MIB для IPv6 предоставляется в виде набора патчей для ucd-snmp.
- RFC2466: MIB for IPv6: ICMPv6 group
  - Необходимая статистика собирается ядром. Фактическая поддержка MIB IPv6 предоставляется в виде патча для ucd-snmp.
- RFC2467: Transmission of IPv6 Packets over FDDI Networks
- RFC2497: Transmission of IPv6 packet over ARCnet Networks
- RFC2553: Basic Socket Interface Extensions for IPv6
  - Отображаемый адрес IPv4 (3.7) и особое поведение сокета с привязкой по шаблону IPv6 (3.8) поддерживаются. Подробности см. в разделе этого документа IPv4-отображённые адреса и IPv6-сокет с подстановочным адресом.
- RFC2675: IPv6 Jumbograms
  - См. раздел Джамбо-пакет (Jumbo Payload) в этом документе для получения подробностей.

- RFC2710: Multicast Listener Discovery for IPv6
- RFC2711: IPv6 router alert option
- draft-ietf-ipngwg-router-renum-08: Перенумерация маршрутизаторов для IPv6
- draft-ietf-ipngwg-icmp-namelookups-02: Поиск имён через ICMP в IPv6
- draft-ietf-ipngwg-icmp-name-lookups-03: Поиск имён IPv6 через ICMP
- draft-ietf-pim-ipv6-01.txt: PIM for IPv6
  - pim6dd(8) реализует плотный режим. pim6sd(8) реализует разреженный режим.
- draft-itojun-ipv6-tcp-to-anycast-00: Разрыв TCP-соединения с anycast-адресом IPv6
- draft-yamamoto-wideipv6-comm-model-00
  - См. раздел Выбор исходящего адреса в этом документе для более подробной информации.
- draft-ietf-ipngwg-scopedaddr-format-00.txt: Расширение формата для адресов с областью действия IPv6

8.1.1.2. Функция "Обнаружение соседей"

Обнаружение соседей достаточно стабильно. В настоящее время поддерживаются следующие функции: определение адреса (Address Resolution), обнаружение дублирования адресов (DAD — Duplicated Address Detection) и обнаружение недоступности соседей (Neighbor Unreachability Detection). В ближайшем будущем мы добавим поддержку прокси-объявлений соседей (Proxy Neighbor Advertisement) в ядро и команду передачи непрошенных объявлений соседей (Unsolicited Neighbor Advertisement) в качестве инструмента администратора.

Если DAD завершается неудачно, адрес будет помечен как "дублированный" (duplicated), и сообщение будет записано в syslog (а также обычно выведено на консоль). Метку "дублированный" можно проверить с помощью ifconfig(8). Обязанность администратора — проверять и устранять сбои DAD. В ближайшем будущем поведение должно быть улучшено.

Некоторые сетевые драйверы закольцовывают multicast-пакеты обратно на себя, даже если им указано так не делать (особенно в promiscuous mode). В таких случаях DAD может завершиться неудачей, так как механизм DAD видит входящий NS-пакет (на самом деле от самого узла) и считает его признаком дубликата. В качестве обходного решения можно рассмотреть условие #if с пометкой "heuristics" в sys/netinet6/nd6_nbr.c:nd6_dad_timer() (обратите внимание, что фрагмент кода в разделе "heuristics" не соответствует спецификации).

Спецификация обнаружения соседей (RFC2461) не рассматривает обработку кэша соседей в следующих случаях:

1. когда отсутствовала запись в кэше соседей, узел получал нежелательный пакет RS/NS/NA/редирект без адреса канального уровня
2. обработка кэша соседей в среде без адреса канального уровня (нам нужна запись в кэше соседей для бита IsRouter)

Для первого случая мы реализовали временное решение на основе обсуждений в рассылке IETF ipngwg. Подробности можно найти в комментариях исходного кода и ветке электронной почты, начавшейся с (IPng 7155) от 6 февраля 1999 года.

Правило определения локальной IPv6 сети (RFC2461) значительно отличается от предположений в сетевом коде BSD. На данный момент не поддерживается правило определения локальной сети при пустом списке маршрутизаторов по умолчанию (RFC2461, раздел 5.2, последнее предложение во 2-м абзаце - обратите внимание, что в спецификации некорректно используются слова "host" и "node" в нескольких местах раздела).

Во избежание возможных атак типа DoS и бесконечных циклов, сейчас принимается только 10 опций в ND-пакете. Таким образом, если к RA прикреплено 20 опций префиксов, будут распознаны только первые 10 префиксов. Если это вызывает проблемы, пожалуйста, задайте вопрос в рассылке FREEBSD-CURRENT и/или измените nd6_maxndopt в sys/netinet6/nd6.c. При высоком спросе мы можем предоставить sysctl-параметр для этой переменной.

8.1.1.3. Индекс зоны

В IPv6 используются адреса с областями видимости — зонами. Поэтому очень важно указывать индекс зоны (индекс интерфейса для линк-локального адреса или индекс сайта для сайт-локального адреса) вместе с IPv6 адресом. Без индекса зоны адрес IPv6 с ограниченной областью действия является неоднозначным для ядра, и ядро не сможет определить исходящий интерфейс для пакета.

Обычные пользовательские приложения должны использовать расширенный API (RFC2292) для указания индекса зоны или индекса интерфейса. Для аналогичных целей член sin6_scope_id в структуре sockaddr_in6 определён в RFC2553. Однако семантика sin6_scope_id довольно расплывчата. Если важна переносимость вашего приложения, мы рекомендуем использовать расширенный API вместо sin6_scope_id.

В ядре индекс интерфейса для адреса с областью действия link-local встраивается во второе 16-битное слово (3-й и 4-й байт) в IPv6-адресе. Например, вы можете увидеть что-то вроде:

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

в таблице маршрутизации и структуре адреса интерфейса (struct in6_ifaddr). Указанный выше адрес является линк-локальным уникастным адресом, который принадлежит сетевому интерфейсу с идентификатором интерфейса 1. Встроенный индекс позволяет эффективно идентифицировать локальные адреса IPv6 на нескольких интерфейсах с минимальными изменениями кода.

Демоны маршрутизации и программы настройки, такие как route6d(8) и ifconfig(8), должны управлять "встроенным" индексом зоны. Эти программы используют сокеты маршрутизации и ioctl (например, SIOCGIFADDR_IN6), и API ядра будет возвращать IPv6-адреса с заполненным вторым 16-битным словом. API предназначены для управления внутренними структурами ядра. Программы, использующие эти API, в любом случае должны быть готовы к различиям в ядрах.

При указании адреса с ограниченной областью действия в командной строке НИКОГДА не используйте встроенную форму (например, ff02:1::1 или fe80:2::fedc). Это не должно работать. Всегда используйте стандартную форму, такую как ff02::1 или fe80::fedc, с параметром командной строки для указания интерфейса (например, `ping -6 -I ne0 ff02::1`). В общем, если команда не имеет параметра командной строки для указания исходящего интерфейса, эта команда не готова принимать адрес с областью действия. Это кажется противоречащим принципу IPv6 поддерживать сценарий "кабинета стоматолога". Мы считаем, что спецификации нуждаются в некоторых улучшениях для этого.

Некоторые пользовательские утилиты поддерживают расширенный числовой синтаксис IPv6, как описано в draft-ietf-ipngwg-scopedaddr-format-00.txt. Можно указать исходящее соединение, используя имя исходящего интерфейса, например "fe80::1%ne0". Таким образом можно легко указать линк-локальный адрес с ограниченной областью действия.

Для использования этого расширения в вашей программе потребуется использовать getaddrinfo(3) и getnameinfo(3) с NI_WITHSCOPEID. В текущей реализации предполагается однозначное соответствие между каналом и интерфейсом, что является более строгим условием, чем указано в спецификациях.

8.1.1.4. Plug and Play (подключи и работай)

Большая часть автонастройки адресов IPv6 без сохранения состояния реализована в ядре. Функции обнаружения соседей (Neighbor Discovery) реализованы в ядре целиком. Ввод рекламы маршрутизатора (RA) для хостов реализован в ядре. Вывод запроса маршрутизатора (RS) для конечных хостов, ввод RS для маршрутизаторов и вывод RA для маршрутизаторов реализованы в пользовательском пространстве.

8.1.1.4.1. Назначение линк-локальных и специальных адресов

Линк-локальный адрес IPv6 генерируется из IEEE802 адреса (Ethernet MAC адреса). Каждому интерфейсу автоматически присваивается IPv6 линк-локальный адрес, когда интерфейс поднимается (IFF_UP). Также в таблицу маршрутизации добавляется прямой маршрут для линк-локального адреса.

Вот вывод команды netstat:

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

Интерфейсы, не имеющие адреса IEEE802 (псевдоинтерфейсы, такие как туннельные интерфейсы или интерфейсы ppp), будут заимствовать адрес IEEE802 у других интерфейсов, например, Ethernet-интерфейсов, когда это возможно. Если нет подключенного оборудования IEEE802, в качестве последнего средства будет использовано псевдослучайное значение MD5(hostname) для формирования линк-локального адреса. Если это не подходит для вашего использования, вам потребуется настроить линк-локальный адрес вручную.

Если интерфейс не поддерживает IPv6 (например, отсутствует поддержка multicast), на этот

интерфейс не будет назначен линк-локальный адрес. Подробности см. в разделе 2.

Каждый интерфейс присоединяется к запрашиваемому широковещательному адресу и линк-локальным широковещательным адресам всех узлов (например, fe80::1:ff01:6317 и ff02::1 соответственно на соединении, к которому подключен интерфейс). В дополнение к линк-локальному адресу, адрес обратной петли (::1 — loopback) будет назначен интерфейсу обратной петли. Также, ::1/128 и ff01::/32 автоматически добавляются в таблицу маршрутизации, а loopback-интерфейс (интерфейс обратной петли) присоединяется к групповому адресу в пределах узла ff01::1.

8.1.1.4.2. Автоматическая настройка адресов без состояния на узлах

В спецификации IPv6 узлы разделены на две категории: *маршрутизаторы* и *хосты*. Маршрутизаторы пересылают пакеты, адресованные другим, хосты не пересылают пакеты. Параметр net.inet6.ip6.forwarding определяет, является ли данный узел маршрутизатором или хостом (маршрутизатор, если значение равно 1, хост, если 0).

Когда хост получает Объявление Маршрутизатора (Router Advertisement) от маршрутизатора, он может автоматически настроить себя с помощью автонастройки адреса без сохранения состояния. Это поведение можно контролировать с помощью параметра net.inet6.ip6.accept_rtadv (хост автонастраивается, если значение равно 1). При автонастройке добавляется префикс сетевого адреса для принимающего интерфейса (обычно префикс глобального адреса). Также настраивается маршрут по умолчанию. Маршрутизаторы периодически генерируют пакеты Router Advertisement. Чтобы запросить соседний маршрутизатор сгенерировать RA-пакет, хост может отправить Router Solicitation. Для генерации RS-пакета в любое время используйте команду *rtsol*. Также доступен демон rtsold(8). rtsold(8) генерирует Router Solicitation по мере необходимости и отлично подходит для мобильного использования (ноутбуки/лэптопы). Если необходимо игнорировать Router Advertisements, используйте sysctl для установки net.inet6.ip6.accept_rtadv в 0.

Для генерации Router Advertisement от маршрутизатора используйте демон rtadvd(8).

Обратите внимание, что спецификация IPv6 предполагает следующие пункты, а случаи несоответствия остаются неуточнёнными:

- Только хосты будут принимать объявления от маршрутизаторов
- Узлы имеют один сетевой интерфейс (за исключением loopback)

Поэтому не рекомендуется включать net.inet6.ip6.accept_rtadv на маршрутизаторах или многопортовых хостах. Неправильно настроенный узел может вести себя странно (нестандартная конфигурация разрешена для тех, кто хочет провести эксперименты).

Резюмируя настройку sysctl:

```
accept_rtadv    forwarding  role of the node
---     ---     ---
0       0       host (to be manually configured)
0       1       router
1       0       autoconfigured host
```

```
                (spec assumes that host has single
                interface only, autoconfigured host
                with multiple interface is
                out-of-scope)
  1       1     invalid, or experimental
                (out-of-scope of spec)
```

В RFC2462 есть правило проверки для входящей информации о префиксе в RA, в разделе 5.5.3 (e). Это защищает хосты от злонамеренных (или неправильно настроенных) маршрутизаторов, которые анонсируют очень короткое время жизни префикса. Было обновление от Джима Баунда в рассылке ipngwg (ищите "(ipng 6712)" в архиве), и это обновление Джима реализовано.

См. раздел Функция "Обнаружение соседей" в этом документе для информации о взаимосвязи между DAD и автонастройкой.

8.1.1.5. Универсальный туннельный интерфейс

GIF (Generic InterFace) — это псевдоинтерфейс для настроенного туннеля. Подробности описаны в gif(4). В настоящее время

- v6 в v6
- v6 в v4
- v4 в v6
- v4 в v4

доступны. Используйте gifconfig(8) для назначения физических (внешних) исходных и конечных адресов интерфейсам gif. Конфигурация, использующая одно семейство адресов для внутреннего и внешнего IP-заголовка (v4 в v4 или v6 в v6), является опасной. Очень легко настроить интерфейсы и таблицы маршрутизации для выполнения бесконечного уровня туннелирования. *Пожалуйста, будьте осторожны.*

gif можно настроить так, чтобы он был дружественным к ECN. Подробнее о дружелюбности к ECN для туннелей см. Учёт ECN в IPsec-туннелях, а о настройке — в gif(4).

Если вы хотите настроить туннель IPv4-в-IPv6 с интерфейсом gif, внимательно прочитайте gif(4). Вам потребуется удалить линк-локальный адрес IPv6, автоматически назначенный интерфейсу gif.

8.1.1.6. Выбор исходящего адреса

Текущее правило выбора источника ориентировано на зону (есть несколько исключений — см. ниже). Для заданного адреса назначения исходящий IPv6-адрес выбирается по следующему правилу:

1. Если исходящий адрес явно указан пользователем (например, через расширенный API), используется указанный адрес.
2. Если на исходящем интерфейсе назначен адрес (который обычно определяется путем

просмотра таблицы маршрутизации) с той же зоной действия, что и адрес назначения, используется этот адрес.

Это наиболее типичный случай.

3. Если нет адреса, удовлетворяющего указанному выше условию, выберите глобальный адрес, назначенный одному из интерфейсов на отправляющем узле.
4. Если нет адреса, удовлетворяющего указанному выше условию, и адрес назначения имеет сайт-локальную зону, выберите сайт-локальный адрес, назначенный одному из интерфейсов на отправляющем узле.
5. Если нет адреса, удовлетворяющего указанному условию, выберите адрес, связанный с записью таблицы маршрутизации для назначения. Это крайняя мера, которая может нарушить границы зоны действия.

Например, ::1 выбирается для ff01::1, fe80:1::200:f8ff:fe01:6317 для fe80:1::2a0:24ff:feab:839b (обратите внимание, что встроенный индекс интерфейса — описанный в Индекс зоны — помогает нам выбрать правильный исходный адрес. Эти встроенные индексы не будут передаваться по сети). Если исходящий интерфейс имеет несколько адресов для данной зоны, исходный адрес выбирается на основе наибольшего соответствия (правило 3). Предположим, что 2001:0DB8:808:1:200:f8ff:fe01:6317 и 2001:0DB8:9:124:200:f8ff:fe01:6317 назначены исходящему интерфейсу. 2001:0DB8:808:1:200:f8ff:fe01:6317 выбирается в качестве исходящего адреса для адреса назначения 2001:0DB8:800::1.

Обратите внимание, что приведённое выше правило не документировано в спецификации IPv6. Оно считается элементом, оставленным "на усмотрение реализации". Существуют случаи, когда мы не используем это правило. Один из примеров — установленное TCP-соединение, где мы используем адрес, сохранённый в tcb, в качестве источника. Другой пример — исходящий адрес для Объявления Соседа (Neighbor Advertisement). Согласно спецификации (RFC2461 7.2.2) источник NA должен быть целевым адресом соответствующего NS. В этом случае мы следуем спецификации, а не приведённому выше правилу наибольшего совпадения.

Для новых соединений (когда правило 1 не применяется), устаревшие адреса (адреса с предпочтительным временем жизни = 0) не будут выбираться в качестве исходящего адреса, если доступны другие варианты. Если других вариантов нет, устаревший адрес будет использован в качестве последнего средства. Если есть несколько устаревших адресов, для выбора между ними будет применено указанное выше правило области видимости. Если вы хотите запретить использование устаревших адресов по какой-либо причине, установите параметр net.inet6.ip6.use_deprecated в значение 0. Проблема, связанная с устаревшими адресами, описана в RFC2462 5.5.4 (ПРИМЕЧАНИЕ: в IETF ipngwg ведутся дебаты о том, как использовать "устаревшие" адреса).

8.1.1.7. Джамбо-пакет (Jumbo Payload)

Опция джамбо-пакет типа "от прыжка к прыжку" реализована и может использоваться для отправки IPv6-пакетов с полезной нагрузкой длиной более 65 535 октетов. Однако в настоящее время не поддерживаются физические интерфейсы с MTU более 65 535, поэтому такие нагрузки могут быть только на интерфейсе loopback (т.е. lo0).

Если вы хотите попробовать джамбо-пакеты, сначала необходимо переконфигурировать ядро, чтобы MTU интерфейса loopback превышал 65 535 байт; добавьте следующее в конфигурационный файл ядра:

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

и пересоберите новое ядро.

Затем вы можете проверить работу с большими пакетами с помощью команды ping(8) с опциями -6, -b и -s. Опция -b необходима для увеличения размера буфера сокета, а опция -s задает длину пакета, которая должна быть больше 65 535. Например, введите следующее:

```
% ping -6 -b 70000 -s 68000 ::1
```

Спецификация IPv6 требует, чтобы опция Джамбо-пакет не использовалась в пакете, содержащем заголовок фрагмента. Если это условие нарушено, должно быть отправлено ICMPv6 сообщение Parameter Problem отправителю. Спецификация соблюдается, но обычно вы не можете увидеть ICMPv6 ошибку, вызванную этим требованием.

При получении IPv6-пакета проверяется длина кадра и сравнивается с длиной, указанной в поле длины полезной нагрузки заголовка IPv6 или в значении опции Джамбо-пакета, если она присутствует. Если первое значение меньше второго, пакет отбрасывается, и статистика увеличивается. Статистику можно увидеть в выводе команды netstat(8) с опцией `-s -p ip6`:

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

Итак, ядро не отправляет ICMPv6 ошибку, если ошибочный пакет не является фактически Джамбо-пакетом, то есть его размер пакета превышает 65 535 байт. Как описано выше, в настоящее время не поддерживаются физические интерфейсы с таким огромным MTU, поэтому ICMPv6 ошибка возвращается редко.

В настоящее время поддержка TCP/UDP через jumbogram не реализована. Это связано с отсутствием среды (кроме loopback) для тестирования данной функциональности. Свяжитесь с нами, если вам это необходимо.

IPsec не работает с jumbogram. Это связано с особенностями спецификации, касающимися поддержки AH для джамбограмм (размер заголовка AH влияет на длину полезной нагрузки, что делает крайне сложной аутентификацию входящего пакета с опцией Джамбо-пакет, а также AH).

Существуют фундаментальные проблемы в поддержке *BSD для jumbogram. Мы хотели бы решить их, но нам нужно больше времени для завершения работы. Вот некоторые из них:

- Поле `mbuf pkthdr.len` имеет тип `int` в 4.4BSD, поэтому оно не сможет содержать

джамбограмму с длиной > 2G на 32-битных архитектурах CPU. Если мы хотим правильно поддерживать джамбограммы, это поле необходимо расширить, чтобы оно могло содержать 4G + заголовок IPv6 + заголовок канального уровня. Следовательно, его необходимо расширить как минимум до `int64_t` (`u_int32_t` НЕ достаточно).

- Мы ошибочно используем "int" для хранения длины пакета во многих местах. Нам необходимо преобразовать их в более крупный целочисленный тип. Это требует большой осторожности, так как мы можем столкнуться с переполнением во время вычисления длины пакета.
- Мы ошибочно проверяем поле ip6_plen заголовка IPv6 для определения длины полезной нагрузки пакета в различных местах. Вместо этого следует проверять mbuf pkthdr.len. Функция ip6_input() выполняет проверку корректности опции Джамбо-пакет при вводе, и после этого можно безопасно использовать mbuf pkthdr.len.
- Код TCP требует тщательного обновления в ряде мест, разумеется.

8.1.1.8. Предотвращение петель при обработке заголовков

Спецификация IPv6 допускает размещение произвольного количества расширений в заголовках пакетов. Если реализовать код обработки пакетов IPv6 так, как реализован код IPv4 в BSD, может произойти переполнение стека ядра из-за длинной цепочки вызовов функций. Код в sys/netinet6 тщательно спроектирован, чтобы избежать переполнения стека ядра, поэтому он определяет собственную структуру переключения протоколов — "struct ip6protosw" (см. netinet6/ip6protosw.h). Для IPv4 части (sys/netinet) подобных обновлений не было сделано для сохранения совместимости, но в прототип pr_input() внесено небольшое изменение. Поэтому также определена "struct ipprotosw". В результате, если получен пакет IPsec-over-IPv4 с большим количеством заголовков IPsec, стек ядра может переполниться. С IPsec-over-IPv6 такой проблемы нет. (Разумеется, чтобы все эти заголовки IPsec были обработаны, каждый такой заголовок должен пройти все проверки IPsec. Поэтому анонимный злоумышленник не сможет осуществить подобную атаку.)

8.1.1.9. ICMPv6

После публикации RFC2463 IETF ipngwg решил запретить ICMPv6 пакеты ошибок для ICMPv6 перенаправлений, чтобы предотвратить ICMPv6 шторм в сетевой среде. Это уже реализовано в ядре.

8.1.1.10. Приложения (Applications)

Для программирования в пользовательском пространстве мы поддерживаем API сокетов IPv6, как указано в RFC2553, RFC2292 и готовящихся интернет-черновиках.

TCP/UDP поверх IPv6 доступны и достаточно стабильны. Вы можете использовать telnet(1), ftp(1), rlogin(1), rsh(1), ssh(1) и т.д. Эти приложения не зависят от протокола. То есть они автоматически выбирают IPv4 или IPv6 в соответствии с DNS.

8.1.1.11. Внутреннее устройство ядра

В то время как ip_forward() вызывает ip_output(), ip6_forward() напрямую вызывает if_output(), поскольку маршрутизаторы не должны разделять пакеты IPv6 на фрагменты.

ICMPv6 должен содержать исходный пакет по возможности вплоть до 1280 байт. Например, сообщение "Ошибка недоступности порта UDP6/IP6" должно содержать все расширенные заголовки и **неизменённые** заголовки UDP6 и IP6. Таким образом, все функции IP6, кроме TCP, никогда не преобразуют порядок байтов сети в порядок байтов хоста, чтобы сохранить исходный пакет.

Функции tcp_input(), udp6_input() и icmp6_input() не могут предполагать, что заголовок IP6 предшествует транспортным заголовкам из-за наличия расширенных заголовков. Поэтому была реализована in6_cksum() для обработки пакетов, у которых заголовок IP6 и транспортный заголовок не являются непрерывными. Но ни для TCP/IP6, ни для UDP6/IP6 в заголовке нет структуры для расчёта контрольной суммы.

Для удобной обработки заголовка IP6, дополнительных заголовков и транспортных заголовков, от сетевых драйверов теперь требуется хранить пакеты в одном внутреннем mbuf или одном или нескольких внешних mbuf. Типичный старый драйвер подготавливает два внутренних mbuf для данных размером 96–204 байт, однако теперь такие данные пакета хранятся в одном внешнем mbuf.

`netstat -s -p ip6` показывает, соответствует ли ваш драйвер этому требованию. В следующем примере "cce0" нарушает это требование. (Для получения дополнительной информации обратитесь к разделу 2.)

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

Каждая входная функция вызывает IP6_EXTHDR_CHECK в начале, чтобы проверить, является ли область между IP6 и его заголовком непрерывной. IP6_EXTHDR_CHECK вызывает m_pullup() только если mbuf имеет флаг M_LOOP, то есть пакет пришел с интерфейса loopback. m_pullup() никогда не вызывается для пакетов, приходящих с физических сетевых интерфейсов.

Как функции повторной сборки IP, так и IP6 никогда не вызывают m_pullup().

8.1.1.12. IPv4-отображённые адреса и IPv6-сокет с подстановочным адресом

RFC2553 описывает IPv4 отображённые адреса (3.7) и особое поведение IPv6 сокета с привязкой к любому адресу (3.8). Спецификация позволяет вам:

- Принимать IPv4-подключения через сокет с привязкой к подстановочному адресу AF_INET6.
- Передача IPv4-пакета через сокет AF_INET6 с использованием специальной формы адреса, например ::ffff:10.1.1.1.

но сама спецификация очень сложна и не определяет, как должен вести себя сокетный

уровень. Здесь мы называем первую сторону «слушающей», а вторую — «инициирующей» для удобства ссылок.

Вы можете выполнить привязку к подстановочному адресу для обоих семейств адресов на одном и том же порту.

Следующая таблица показывает поведение FreeBSD 4.x.

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

Следующие разделы предоставят вам более подробную информацию и объяснят, как можно настроить поведение.

Комментарии о принимающей стороне:

Похоже, что в RFC2553 слишком мало сказано о проблеме привязки к подстановочному адресу, особенно о вопросе пространства портов, режиме отказа и взаимосвязи между AF_INET/INET6 wildcard bind. Может быть несколько различных интерпретаций этого RFC, которые соответствуют ему, но ведут себя по-разному. Поэтому для создания переносимых приложений не следует делать никаких предположений о поведении в ядре. Использование getaddrinfo(3) является наиболее безопасным способом. Вопросы пространства номеров портов и привязки к подстановочному адресу подробно обсуждались в рассылке ipv6imp в середине марта 1999 года, и похоже, что конкретного консенсуса нет (то есть, остаётся на усмотрение реализаторов). Возможно, вам стоит проверить архивы рассылки.

Если серверное приложение хочет принимать IPv4 и IPv6 соединения, есть два варианта.

Один из способов — использование сокетов AF_INET и AF_INET6 (вам понадобятся два сокета). Используйте getaddrinfo(3) с AI_PASSIVE в ai_flags, а также socket(2) и bind(2) для всех возвращённых адресов. Открыв несколько сокетов, вы можете принимать соединения сокетом соответствующей адресной семьи. IPv4-соединения будут приниматься сокетом AF_INET, а IPv6-соединения — сокетом AF_INET6.

Еще один способ — использование одного сокета с универсальной привязкой AF_INET6. Используйте getaddrinfo(3) с AI_PASSIVE в ai_flags и AF_INET6 в ai_family, установив первый аргумент hostname в NULL. Затем используйте socket(2) и bind(2) для адреса, который был возвращен. (должен быть неспецифицированный адрес IPv6). Через этот один сокет можно принимать пакеты как IPv4, так и IPv6.

Для поддержки только IPv6-трафика на AF_INET6-сокете с привязкой к любому адресу переносимым способом всегда проверяйте адрес узла при установке соединения с AF_INET6-сокетом в режиме прослушивания. Если адрес является IPv4-отображённым, возможно, стоит отклонить соединение. Это условие можно проверить с помощью макроса

IN6_IS_ADDR_V4MAPPED().

Для более простого решения этой задачи существует зависящий от системы параметр setsockopt(2) под названием IPV6_BINDV6ONLY, используемый следующим образом.

```
    int on;

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

При успешном вызове этот сокет будет принимать только IPv6-пакеты.

Комментарии о стороне инициатора:

Совет разработчикам приложений: для создания переносимого IPv6-приложения (которое работает на различных IPv6-ядрах), мы считаем, что следующие моменты являются ключом к успеху:

- НИКОГДА не используйте жёстко заданные AF_INET или AF_INET6.
- Используйте getaddrinfo(3) и getnameinfo(3) во всей системе. Никогда не используйте gethostby*(), getaddrby*(), inet_*() или getipnodeby*(). (Для облегчения обновления существующих приложений для поддержки IPv6 иногда может быть полезен getipnodeby*(). Но по возможности старайтесь переписать код для использования getaddrinfo(3) и getnameinfo(3).)
- Если вы хотите подключиться к назначению, используйте getaddrinfo(3) и попробуйте все возвращённые назначения, как это делает telnet(1).
- Некоторые реализации стека IPv6 поставляются с некорректной getaddrinfo(3). Включите минимально рабочую версию в ваше приложение и используйте её в крайнем случае.

Если вы хотите использовать сокет AF_INET6 для исходящих подключений как IPv4, так и IPv6, вам потребуется использовать getipnodebyname(3). Если вы хотите обновить существующее приложение для поддержки IPv6 с минимальными усилиями, можно выбрать этот подход. Однако учтите, что это временное решение, поскольку getipnodebyname(3) сам по себе не рекомендуется, так как он вообще не обрабатывает IPv6-адреса с зоной. Для разрешения IPv6-имён предпочтительным API является getaddrinfo(3). Поэтому вам следует переписать ваше приложение для использования getaddrinfo(3), когда у вас будет время это сделать.

При написании приложений, которые устанавливают исходящие соединения, история становится намного проще, если рассматривать AF_INET и AF_INET6 как совершенно отдельные семейства адресов. Проблемы с {set,get}sockopt упрощаются, проблемы с DNS также станут проще. Мы не рекомендуем полагаться на IPv4-отображённые адреса.

8.1.1.12.1. унифицированный код tcp и inpcb

FreeBSD 4.x использует общий код tcp для IPv4 и IPv6 (из sys/netinet/tcp*) и раздельный код udp4/6. В нём используется унифицированная структура inpcb.

Платформа может быть настроена для поддержки IPv4-отображённых адресов. Конфигурация ядра кратко описана ниже:

- По умолчанию сокет AF_INET6 может принимать IPv4-соединения при определённых условиях и инициировать соединение с IPv4-адресами, встроенными в IPv4-отображённые IPv6-адреса.
- Вы можете отключить это во всей системе с помощью sysctl, как показано ниже.

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

8.1.1.12.1.1. Сторона, принимающая соединения

Каждый сокет может быть настроен для поддержки специальной привязки к подстановочному адресу AF_INET6 (включено по умолчанию). Это можно отключить для каждого отдельного сокета с помощью setsockopt(2), как показано ниже.

```
    int on;

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

Сокет с универсальной привязкой AF_INET6 перехватывает IPv4-подключение тогда и только тогда, когда выполнены следующие условия:

- нет AF_INET сокета, соответствующего IPv4-подключению
- Сокет AF_INET6 настроен на прием IPv4-трафика, т.е., getsockopt(IPV6_BINDV6ONLY) возвращает 0.

Нет проблем с порядком открытия/закрытия.

8.1.1.12.1.2. Инициирующая сторона

FreeBSD 4.x поддерживает исходящее соединение с IPv4-отображённым адресом (::ffff:10.1.1.1), если узел настроен на поддержку IPv4-отображённых адресов.

8.1.1.13. sockaddr_storage

Когда RFC2553 был близок к завершению, велись дискуссии о том, как называть элементы структуры `sockaddr_storage`. Одно предложение заключалось в добавлении "*" перед именами элементов (например, *"ss_len"), так как к ним не следует обращаться напрямую. Другое предложение было не добавлять префикс (например, "ss_len"), поскольку необходимо прямое обращение к этим элементам. Четкого консенсуса по этому вопросу достигнуто не было.

В результате, RFC2553 определяет структуру sockaddr_storage следующим образом:

```
    struct sockaddr_storage {
        u_char  __ss_len;   /* address length */
```

```
        u_char  __ss_family;    /* address family */
        /* and bunch of padding */
    };
```

Напротив, черновик XNET определяет следующее:

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

В декабре 1999 года было согласовано, что RFC2553bis должен принять последнее (XNET) определение.

Текущая реализация соответствует определению XNET, основанному на обсуждении RFC2553bis.

Если вы рассмотрите несколько реализаций IPv6, то сможете увидеть оба определения. Для программиста в пользовательском пространстве наиболее переносимый способ работы с этим:

1. с помощью GNU autoconf сконфигурировать доступ к `ss_family` и/или `ss_len` на целевой платформе,
2. добавить -Dss_family=*ss_family для унификации всех использований (включая заголовочный файл)* ss_family, или
3. никогда не трогайте __ss_family. Приводите к sockaddr * и используйте sa_family, например:

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

8.1.2. Драйверы сетевых устройств

В настоящее время следующие два пункта должны поддерживаться стандартными драйверами:

1. Требование к кластеризации mbuf. В этом стабильном выпуске мы изменили MINCLSIZE на MHLEN+1 для всех операционных систем, чтобы все драйверы работали так, как мы ожидаем.
2. многоадресная рассылка (multicast). Если ifmcstat(8) не выводит ни одной многоадресной группы для интерфейса, этот интерфейс необходимо исправить.

Если какие-либо драйверы не поддерживают требования, то их нельзя использовать для IPv6 и/или IPsec-связи. Если вы обнаружили проблему с вашей картой при использовании IPv6/IPsec, пожалуйста, сообщите об этом в Список рассылки FreeBSD, посвящённый

сообщениям о проблемах.

(NOTE: Раньше мы требовали, чтобы все драйверы PCMCIA содержали вызов in6_ifattach(). Теперь такого требования нет)

8.1.3. Транслятор

Мы классифицируем трансляторы IPv4/IPv6 на 4 типа:

- *Транслятор А* --- Он используется на раннем этапе перехода, чтобы позволить установить соединение с IPv6-хоста на IPv6-острове к IPv4-хосту в IPv4-океане.
- *Транслятор Б* --- Он используется на раннем этапе перехода, чтобы обеспечить возможность установления соединения с IPv6-узлом на IPv6-острове от IPv4-узла в IPv4-океане.
- *Транслятор C* --- Он используется на позднем этапе перехода, чтобы сделать возможным установление соединения с IPv6-узлом в IPv6-океане от IPv4-узла на IPv4-острове.
- *Транслятор D* --- Он используется на позднем этапе перехода, чтобы сделать возможным установление соединения с IPv6-хоста в IPv6-океане на IPv4-хост на IPv4-острове.

8.1.4. IPsec

IPsec состоит в основном из трёх компонент.

1. Управление политиками
2. Управление ключами
3. Обработка AH и ESP

8.1.4.1. Управление политиками

Ядро реализует экспериментальный код управления политикой безопасности. Существует два способа управления политикой безопасности. Первый — настройка политики для каждого сокета с помощью setsockopt(2). В этом случае конфигурация политики описана в ipsec_set_policy(3). Второй способ — настройка политики на основе фильтра пакетов ядра с использованием интерфейса PF_KEY через setkey(8).

Запись политики не переупорядочивается вместе со своими индексами, поэтому порядок добавления записей очень важен.

8.1.4.2. Управление ключами

Код управления ключами, реализованный в этом наборе (sys/netkey), представляет собой собственную реализацию PFKEY v2. Это соответствует RFC2367.

В комплект включена "домашняя" реализация демона IKE — "racoon" (kame/kame/racoon). Обычно вам потребуется запустить racoon в качестве демона, затем настроить политику для требования ключей (например, `ping -P 'out ipsec esp/transport//use'`). Ядро будет связываться с демоном racoon по мере необходимости для обмена ключами.

8.1.4.3. Обработка AH и ESP

Модуль IPsec реализован в виде "хуков" к стандартной обработке IPv4/IPv6. При отправке пакета функция ip{,6}_output() проверяет, требуется ли обработка ESP/AH, путем поиска соответствующей базы данных политик безопасности (SPD — Security Policy Database). Если ESP/AH необходим, вызывается {esp,ah}{4,6}_output(), и mbuf соответствующим образом обновляется. При получении пакета функция {esp,ah}4_input() вызывается на основе номера протокола, т.е. (*inetsw[proto])(). {esp,ah}4_input() расшифровывает/проверяет подлинность пакета, а также удаляет цепочку заголовков и выравнивание для ESP/AH. Безопасно удалять заголовок ESP/AH при получении пакета, так как полученный пакет никогда не будет использоваться в "сыром" виде.

Использование ESP/AH влияет на эффективный размер сегмента данных TCP4/6 из-за дополнительных цепочечных заголовков, вставляемых ESP/AH. Наш код учитывает этот случай.

Основные криптографические функции можно найти в каталоге `sys/crypto`. Преобразования ESP/AH перечислены в `{esp,ah}_core.c` с обёрточными функциями. Если вы хотите добавить какой-либо алгоритм, добавьте обёрточную функцию в `{esp,ah}_core.c` и поместите код вашего криптографического алгоритма в `sys/crypto`.

Режим туннеля частично поддерживается в этом выпуске со следующими ограничениями:

- Туннель IPsec не объединён с универсальным туннельным интерфейсом GIF. Это требует особой осторожности, так как может возникнуть бесконечный цикл между `ip_output()` и `tunnelifp→if_output()`. Мнения расходятся относительно того, лучше ли их объединить или нет.
- MTU и бит Don’t Fragment (IPv4) требуют дополнительной проверки, но в основном работают нормально.
- Модель аутентификации для туннеля AH должна быть пересмотрена. Нам потребуется улучшить механизм управления политиками в конечном итоге.

8.1.4.4. Соответствие RFC и ID

Код IPsec в ядре соответствует (или пытается соответствовать) следующим стандартам:

Спецификация "старого IPsec", описанная в rfc182[5-9].txt

Спецификация "new IPsec" описана в rfc240[1-6].txt, rfc241[01].txt, rfc2451.txt и draft-mcdonald-simple-ipsec-api-01.txt (черновик устарел, но его можно взять по ссылке ftp://ftp.kame.net/pub/internet-drafts/). (ПРИМЕЧАНИЕ: Спецификации IKE, rfc241[7-9].txt, реализованы в пользовательском пространстве в виде демона IKE "racoon")

В настоящее время поддерживаются следующие алгоритмы:

- old IPsec AH
  - нулевая криптографическая контрольная сумма (нет документа, только для отладки)
  - MD5 с ключом и с 128-битной криптографической контрольной суммой (rfc1828.txt)

    - SHA1 с ключом и с 128-битной криптографической контрольной суммой (без документа)
    - HMAC MD5 с 128-битной криптографической контрольной суммой (rfc2085.txt)
    - HMAC SHA1 с 128-битной криптографической контрольной суммой (без документа)
- old IPsec ESP
    - нулевое шифрование (нет документа, аналогично rfc2410.txt)
    - Режим DES-CBC (rfc1829.txt)
- new IPsec AH
    - нулевая криптографическая контрольная сумма (нет документа, только для отладки)
    - MD5 с ключом и с 96-битной криптографической контрольной суммой (нет документа)
    - SHA1 с ключом и с 96-битной криптографической контрольной суммой (без документа)
    - HMAC MD5 с 96-битной криптографической контрольной суммой (rfc2403.txt)
    - HMAC SHA1 с 96-битной криптографической контрольной суммой (rfc2404.txt)
- new IPsec ESP
    - нулевое шифрование (rfc2410.txt)
    - DES-CBC с производным IV (draft-ietf-ipsec-ciph-des-derived-01.txt, черновик истек)
    - DES-CBC с явным вектором инициализации (rfc2405.txt)
    - 3DES-CBC с явным вектором инициализации (rfc2451.txt)
    - BLOWFISH CBC (rfc2451.txt)
    - CAST128 CBC (rfc2451.txt)
    - RC5 CBC (rfc2451.txt)
    - каждый из вышеперечисленных может быть объединён с:
        - Аутентификация ESP с HMAC-MD5 (96 бит)
        - Аутентификация ESP с HMAC-SHA1(96 бит)

Следующие алгоритмы НЕ поддерживаются:

- old IPsec AH
    - HMAC MD5 с 128-битной криптографической контрольной суммой + 64-битная защита от повторного воспроизведения (rfc2085.txt)
    - SHA1 с ключом и с 160-битной криптографической контрольной суммой 32-битное дополнение (rfc1852.txt)

IPsec (в ядре) и IKE (в пользовательском пространстве как "racoon") были протестированы на нескольких мероприятиях по тестированию взаимодействия и известно, что они хорошо работают со многими другими реализациями. Кроме того, текущая реализация IPsec поддерживает довольно широкий спектр криптографических алгоритмов IPsec, описанных

в RFC (мы поддерживаем только алгоритмы без проблем с интеллектуальной собственностью).

8.1.4.5. Учёт ECN в IPsec-туннелях

Поддерживается ECN-совместимый IPsec-туннель, как описано в draft-ipsec-ecn-00.txt.

Обычный IPsec-туннель описан в RFC2401. При инкапсуляции поле TOS IPv4 (или поле класса трафика IPv6) копируется из внутреннего IP-заголовка во внешний IP-заголовок. При декапсуляции внешний IP-заголовок просто отбрасывается. Правило декапсуляции несовместимо с ECN, так как бит ECN в поле TOS/класса трафика внешнего IP-заголовка будет потерян.

Чтобы сделать IPsec-туннель дружественным к ECN, следует изменить процедуры инкапсуляции и декапсуляции. Это описано в http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt, глава 3.

Реализация туннеля IPsec может обеспечить три варианта поведения, в зависимости от значения параметра `net.inet.ipsec.ecn` (или `net.inet6.ipsec6.ecn`):

- RFC2401: отсутствие учёта ECN (значение sysctl -1)
- ECN запрещён (значение sysctl 0)
- ECN разрешён (значение sysctl 1)

Обратите внимание, что поведение настраивается для каждого узла, а не для каждой SA (в draft-ipsec-ecn-00 предлагается настройка для каждой SA, но это кажется излишним).

Поведение можно обобщить следующим образом (подробности см. в исходном коде):

```
encapsulate                    decapsulate
               ---                             ---
RFC2401        copy all TOS bits               drop TOS bits on outer
               from inner to outer.            (use inner TOS bits as is)

ECN forbidden  copy TOS bits except for ECN    drop TOS bits on outer
               (masked with 0xfc) from inner   (use inner TOS bits as is)
               to outer.  set ECN bits to 0.

ECN allowed    copy TOS bits except for ECN    use inner TOS bits with some
               CE (masked with 0xfe) from      change.  if outer ECN CE bit
               inner to outer.                 is 1, enable ECN CE bit on
               set ECN CE bit to 0.            the inner.
```

Общая стратегия настройки выглядит следующим образом:

- если оба конечных пункта туннеля IPsec поддерживают поведение, дружественное к ECN, лучше настроить оба конца на "разрешено ECN" (значение sysctl 1).
- если другая сторона очень строга к биту TOS, используйте "RFC2401" (значение sysctl -1).

- в остальных случаях используйте "ECN запрещено" (значение sysctl 0).

Поведение по умолчанию — "ECN запрещён" (значение sysctl 0).

Для получения дополнительной информации обратитесь к:

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

(Благодарности Kenjiro Cho kjc@csl.sony.co.jp за детальный анализ)

8.1.4.6. Совместимость

Вот некоторые из платформ, на которых код KAME тестировал взаимодействие IPsec/IKE в прошлом. Обратите внимание, что обе стороны могли изменить свои реализации, поэтому используйте следующий список только в справочных целях.

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

Часть III: Ядро системы

Глава 9. Сборка и установка ядра FreeBSD

Чтобы быть разработчиком ядра, требуется понимания процесса сборки ядра. Для отладки ядра FreeBSD необходимо уметь его собирать. Существует два известных способа сделать это:

Поддерживаемая процедура сборки и установки ядра описана в главе Сборка и установка пользовательского ядра Руководства FreeBSD.

Предполагается, что читатель этой главы знаком с информацией, изложенной в главе Сборка и установка пользовательского ядра Руководства FreeBSD. Если это не так, пожалуйста, ознакомьтесь с упомянутой главой, чтобы понять, как работает процесс сборки.

9.1. Построение более быстрым, но менее надёжным способом

Сборка ядра таким способом может быть полезной при работе с кодом ядра и может оказаться быстрее, чем описанная процедура, если в конфигурационном файле ядра были изменены только одна или две опции. С другой стороны, это может привести к неожиданным сбоям при сборке ядра.

1. Выполните config(8) для генерации исходного кода ядра:

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

2. Перейдите в каталог сборки. config(8) выведет имя этого каталога после выполнения, как указано выше.

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

3. Скомпилируйте ядро:

```
# make depend
# make
```

4. Установите новое ядро:

```
# make install
```

Глава 10. Отладка ядра

10.1. Получение аварийного дампа ядра

При работе с разрабатываемым ядром (например, FreeBSD-CURRENT), особенно в экстремальных условиях (например, при очень высокой загрузке, десятках тысяч соединений, чрезвычайно большом количестве одновременных пользователей, сотнях jail(8) и т.д.), или при использовании новой функции или драйвера устройства в FreeBSD-STABLE (например, PAE), иногда может возникнуть паника ядра. В случае, если это произойдет, данная глава покажет, как извлечь полезную информацию из аварийного дампа.

Перезагрузка системы неизбежна после паники ядра. После перезагрузки системы содержимое физической памяти (RAM) теряется, как и любые данные на устройстве подкачки перед паникой. Чтобы сохранить данные в физической памяти, ядро использует устройство подкачки как временное хранилище для данных из RAM после сбоя и перезагрузки. Благодаря этому, когда FreeBSD загружается после сбоя, образ ядра может быть извлечен и проведена отладка.

Устройство подкачки, настроенное как устройство для дампа, продолжает функционировать как устройство подкачки. В настоящее время дампы на устройства, не являющиеся устройствами подкачки (например, на ленты или CDRW), не поддерживаются. Термин "устройство подкачки" является синонимом термина "раздел подкачки".

Есть несколько типов аварийных дампов ядра:

Полные дампы памяти

Содержат полное содержимое физической памяти.

Минидампы

Содержат только страницы памяти, используемые ядром.

Текстовые дампы

Содержать захваченные, записанные или интерактивные выходные данные отладчика.

Минидампы являются типом дампа по умолчанию и в большинстве случаев они сохраняют всю необходимую информацию, присутствующую в полном дампе памяти, так как большинство проблем можно изолировать, используя только состояние ядра.

10.1.1. Настройка устройства дампа

Прежде чем ядро запишет содержимое своей физической памяти на устройство дампа, необходимо настроить это устройство. Устройство дампа указывается с помощью команды dumpon(8), чтобы сообщить ядру, куда сохранять аварийные дампы. Программа dumpon(8) должна быть вызвана после настройки раздела подкачки с помощью swapon(8). Обычно это обрабатывается установкой переменной `dumpdev` в rc.conf(5) в путь к устройству подкачки

(рекомендуемый способ извлечения дампа ядра) или в значение AUTO для использования первого настроенного устройства подкачки. По умолчанию dumpdev имеет значение AUTO в HEAD и изменено на NO в ветках RELENG_* (за исключением RELENG_7, где оставлено значение AUTO). Начиная с FreeBSD 9.0-RELEASE и более поздних версий, bsdinstall будет спрашивать, следует ли включить аварийные дампы на целевой системе во время процесса установки.

Проверьте /etc/fstab или swapinfo(8) для получения списка устройств подкачки.

Убедитесь, что каталог dumpdir, указанный в rc.conf(5), существует перед аварией ядра!

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

Также помните, что содержимое /var/crash является конфиденциальным и, скорее всего, содержит секретную информацию, такую как пароли.

10.1.2. Извлечение дампа ядра

После записи дампа на устройство дампа, дамп должен быть извлечен до монтирования устройства подкачки. Для извлечения дампа с устройства дампа используйте программу savecore(8). Если в rc.conf(5) установлен параметр dumpdev, savecore(8) будет автоматически вызван при первой загрузке в многопользовательском режиме после сбоя и до монтирования устройства подкачки. Расположение извлеченного ядра указывается в параметре dumpdir файла rc.conf(5), по умолчанию это /var/crash, а имя файла будет vmcore.0.

В случае, если файл с именем vmcore.0 уже существует в /var/crash (или в каталоге, указанном в параметре dumpdir), ядро будет увеличивать завершающее число при каждом сбое, чтобы избежать перезаписи существующего файла vmcore (например, vmcore.1). savecore(8) всегда создаёт символическую ссылку с именем vmcore.last в /var/crash после сохранения дампа. Эта символическая ссылка может быть использована для определения имени последнего дампа.

Утилита crashinfo(8) создаёт текстовый файл, содержащий сводную информацию из полного дампа памяти или минидампа. Если параметр dumpdev установлен в rc.conf(5), crashinfo(8) будет автоматически вызван после savecore(8). Результат сохраняется в файл с именем core.txt.N в каталоге dumpdir.

Если вы тестируете новое ядро, но вам нужно загрузить другое, чтобы снова запустить систему, загрузите его только в однопользовательском режиме, используя флаг -s при загрузке, а затем выполните следующие шаги:

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

```
# exit                  # exit to multi-user
```

Это указывает savecore(8) извлечь дамп ядра из /dev/ad0s1b и поместить содержимое в /var/crash. Не забудьте убедиться, что целевой каталог /var/crash имеет достаточно места для дампа. Также не забудьте указать правильный путь к вашему swap-устройству, так как он, скорее всего, отличается от /dev/ad0s1b!

10.1.3. Тестирование конфигурации дампа ядра

Ядро включает узел sysctl(8), который вызывает панику ядра. Это можно использовать для проверки того, что ваша система правильно настроена для сохранения дампов аварийного завершения работы ядра. Возможно, вы захотите перемонтировать существующие файловые системы в режиме только для чтения в однопользовательском режиме перед тем, как вызвать панику, чтобы избежать потери данных.

```
# shutdown now
...
Enter full pathname of shell or RETURN for /bin/sh:
# mount -a -u -r
# sysctl debug.kdb.panic=1
debug.kdb.panic:panic: kdb_sysctl_panic
...
```

После перезагрузки система должна сохранить дамп в /var/crash вместе с соответствующим отчётом из crashinfo(8).

10.2. Отладка аварийного дампа ядра с помощью kgdb

Этот раздел посвящен kgdb(1). Последняя версия включена в пакет devel/gdb.

Чтобы войти в отладчик и начать получение информации из дампа, запустите kgdb:

```
# kgdb -n N
```

Где *N* — это суффикс файла vmcore.N, который нужно изучить. Чтобы открыть последний дамп, используйте:

```
# kgdb -n last
```

Обычно kgdb(1) должен быть способен найти ядро, работавшее в момент создания дампа. Если он не может найти нужное ядро, передайте путь к ядру и дампу в качестве двух аргументов для kgdb:

```
# kgdb /boot/kernel/kernel /var/crash/vmcore.0
```

Вы можете отлаживать дамп аварийного завершения, используя исходные коды ядра, так же, как и для любой другой программы.

Этот дамп получен из ядра версии 5.2-BETA, а крах произошел глубоко внутри ядра. Приведенный ниже вывод был изменён для добавления номеров строк слева. Первый трассировочный вывод проверяет указатель инструкции и получает обратную трассировку. Адрес, используемый в строке 41 для команды `list`, является указателем инструкции и может быть найден в строке 17. Большинство разработчиков запросят как минимум эту информацию, если вы не сможете отладить проблему самостоятельно. Однако, если вы решите проблему, убедитесь, что ваш патч попадет в дерево исходников через отчёт о проблеме, списки рассылки, или, может быть, у вас есть возможность его закоммитить!

```
 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
```

Если ваша система регулярно завершается аварийно и у вас заканчивается место на диске, удаление старых файлов vmcore в /var/crash может освободить значительное количество дискового пространства!

10.3. Онлайн-отладка ядра с использованием DDB

В то время как `kgdb` как автономный отладчик предоставляет очень высокий уровень пользовательского интерфейса, есть некоторые вещи, которые он не может выполнить. Наиболее важные из них — установка точек останова и пошаговое выполнение кода ядра.

Если вам требуется выполнить низкоуровневую отладку ядра, доступен отладчик DDB, работающий в режиме реального времени. Он позволяет устанавливать точки останова, выполнять пошаговое выполнение функций ядра, проверять и изменять переменные ядра и т.д. Однако он не имеет доступа к исходным файлам ядра и работает только с глобальными и статическими символами, без доступа к полной отладочной информации, как это делает `kgdb`.

Для настройки ядра с включенной поддержкой DDB добавьте параметры

```
options KDB
```

```
options DDB
```

в ваш конфигурационный файл, и пересоберите. (Подробности о настройке ядра FreeBSD см. в Руководстве FreeBSD).

После загрузки ядра DDB существует несколько способов войти в него. Первый и самый ранний способ — использовать флаг загрузки `-d`. Ядро запустится в режиме отладки и перейдет в DDB до начала обнаружения любого из устройств. Таким образом, можно отлаживать даже функции обнаружить (probe)/ присоединить (attach) устройств. Для использования этого метода выйдите из меню загрузки загрузчика и введите `boot -d` в командной строке загрузчика.

Второй сценарий — перейти в отладчик после загрузки системы. Есть два простых способа это сделать. Если вы хотите перейти в отладчик из командной строки, просто введите команду:

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

В качестве альтернативы, если вы находитесь за системной консолью, можно использовать горячую клавишу на клавиатуре. Стандартной комбинацией для перехода в отладчик является `Ctrl` + `Alt` + `ESC`. В syscons эта последовательность может быть переназначена, и некоторые распространённые раскладки клавиатуры делают это, поэтому убедитесь, что знаете правильную комбинацию. Для последовательных консолей доступна опция, позволяющая использовать сигнал BREAK на линии консоли для входа в DDB (`options BREAK_TO_DEBUGGER` в конфигурационном файле ядра). Это не установлено по умолчанию, так как существует множество последовательных адаптеров, которые излишне генерируют условие BREAK, например, при отключении кабеля.

Третий способ заключается в том, чтобы любое условие паники переходило в DDB, если ядро настроено на его использование. По этой причине не рекомендуется настраивать ядро с DDB для машины, работающей без присмотра.

Для получения неинтерактивной функциональности добавьте:

```
options KDB_UNATTENDED
```

в файл конфигурации ядра и пересоберите/переустановите ядро.

Команды DDB примерно напоминают некоторые команды `gdb`. Первое, что вам, вероятно, нужно сделать, это установить точку останова:

```
break function-name address
```

Числа по умолчанию интерпретируются как шестнадцатеричные, но чтобы отличить их от символьных имен, шестнадцатеричные числа, начинающиеся с букв `a-f`, должны предваряться префиксом `0x` (для остальных чисел это необязательно). Допускаются простые выражения, например: `function-name 0x103`.

Для выхода из отладчика и продолжения выполнения введите:

```
continue
```

Для получения трассировки стека текущего потока используйте:

```
trace
```

Для получения трассировки стека произвольного потока укажите идентификатор процесса или идентификатор потока в качестве второго аргумента команды `trace`.

Если вы хотите удалить точку останова, используйте

```
del
del address-expression
```

Первая форма будет принята сразу после срабатывания точки останова и удаляет текущую точку останова. Вторая форма может удалить любую точку останова, но необходимо указать точный адрес; его можно получить из:

```
show b
```

или:

```
show break
```

Для пошагового выполнения ядра попробуйте:

```
s
```

Это позволит войти в функции, но вы можете заставить DDB отслеживать их до достижения соответствующего оператора return с помощью:

```
n
```

Это отличается от оператора `next` в `gdb`; это похоже на `finish` в `gdb`. Нажатие `n` более одного раза приведёт к продолжению.

Для просмотра данных в памяти используйте (например):

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

для доступа к словам/полусловам/байтам и отображения в шестнадцатеричном/десятичном/символьном/строковом формате. Число после запятой указывает количество объектов. Для отображения следующих 0x10 элементов просто введите:

```
x ,10
```

Аналогично, используйте

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

для дизассемблирования первых 0x10 инструкций функции `foofunc` и их отображения вместе с их смещением от начала `foofunc`.

Для записи в память используйте команду write:

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

Модификатор команды (`b`/`h`/`w`) определяет размер данных для записи, первое следующее выражение — это адрес для записи, а остальное интерпретируется как данные для записи в последующие ячейки памяти.

Если вам необходимо узнать текущее содержимое регистров, введите:

```
show reg
```

Также можно отобразить значение одного регистра, например:

```
p $eax
```

и изменить его с помощью:

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

Если вам потребуется вызвать некоторые функции ядра из DDB, просто напишите:

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

Будет выведено возвращаемое значение.

Для вывода информации о всех запущенных процессах в стиле ps(1) используйте:

```
ps
```

Теперь вы выяснили причину сбоя ядра и хотите выполнить перезагрузку. Помните, что в

зависимости от серьезности предыдущего сбоя не все части ядра могут работать корректно. Выполните одно из следующих действий для завершения работы и перезагрузки системы:

```
panic
```

Это приведёт к дампу ядра и перезагрузке, чтобы позже можно было проанализировать дамп на более высоком уровне с помощью kgdb(1).

```
call boot(0)
```

Может быть хорошим способом чисто завершить работу работающей системы, `sync()` все диски и, наконец, в некоторых случаях перезагрузиться. Пока интерфейсы дисков и файловых систем ядра не повреждены, это может быть хорошим способом для почти чистого завершения работы.

```
reset
```

Это последний способ избежать катастрофы, и он почти такой же, как нажатие на Большую Красную Кнопку.

Если вам нужна краткая сводка команд, просто введите:

```
help
```

Настоятельно рекомендуется иметь распечатанную копию страницы руководства ddb(4) для сеанса отладки. Помните, что читать онлайн-руководство во время пошагового выполнения ядра сложно.

10.4. Онлайн-отладка ядра с использованием удалённого GDB

Ядро FreeBSD предоставляет второй бэкенд KDB для отладки в реальном времени: gdb(4).

GDB давно поддерживает *удалённую отладку*. Это осуществляется с помощью очень простого протокола через последовательное соединение. В отличие от других методов отладки, описанных выше, для этого потребуются две машины. Одна — это хост, предоставляющий среду отладки, включая все исходные тексты и копию бинарного файла ядра со всеми символами. Другая — целевая машина, на которой запущена копия того же самого ядра (возможно, без отладочной информации).

Чтобы использовать удалённый GDB, убедитесь, что следующие параметры присутствуют в конфигурации вашего ядра:

```
makeoptions     DEBUG=-g
```

```
options         KDB
options         GDB
```

Обратите внимание, что опция `GDB` отключена по умолчанию в ядрах `GENERIC` для веток -STABLE и -RELEASE, но включена в -CURRENT.

После сборки скопируйте ядро на целевую машину и загрузите его. Подключите последовательный порт целевой машины, у которого на устройстве uart установлены флаги "080", к любому последовательному порту отладочной машины. Подробности о настройке флагов на устройстве uart смотрите в uart(4).

Целевая машина должна быть переведена в режим отладчика GDB, либо из-за паники, либо путем преднамеренного перехода в отладчик. Перед этим выберите бэкенд отладчика GDB:

```
# sysctl debug.kdb.current=gdb
debug.kdb.current: ddb -> gdb
```

Поддерживаемые бэкенды можно вывести с помощью sysctl `debug.kdb.available`. Если конфигурация ядра включает `options DDB`, то ddb(4) будет выбран по умолчанию. Если `gdb` не отображается в списке доступных бэкендов, значит, последовательный порт отладки может быть настроен неправильно.

Затем принудительно войдите в отладчик:

```
# sysctl debug.kdb.enter=1
debug.kdb.enter: 0KDB: enter: sysctl debug.kdb.enter
```

Целевая машина теперь ожидает подключения от удалённого клиента GDB. На машине для отладки перейдите в каталог сборки целевого ядра и запустите `gdb`:

```
# cd /usr/obj/usr/src/amd64.amd64/sys/GENERIC/
# kgdb kernel
GNU gdb (GDB) 10.2 [GDB v10.2 for FreeBSD]
Copyright (C) 2021 Free Software Foundation, Inc.
...
Reading symbols from kernel...
Reading symbols from /usr/obj/usr/src/amd64.amd64/sys/GENERIC/kernel.debug...
(kgdb)
```

Инициализируйте сеанс удалённой отладки (предполагая, что используется первый последовательный порт) с помощью:

```
(kgdb) target remote /dev/cuau0
```

Ваш хостинг GDB теперь получит контроль над целевым ядром:

```
Remote debugging using /dev/cuau0
kdb_enter (why=<optimized out>, msg=<optimized out>) at
/usr/src/sys/kern/subr_kdb.c:506
506                     kdb_why = KDB_WHY_UNSET;
(kgdb)
```

В зависимости от используемого компилятора, некоторые локальные переменные могут отображаться как `<optimized out>`, что не позволяет их напрямую исследовать с помощью `gdb`. Если это вызывает проблемы при отладке, можно собрать ядро с пониженным уровнем оптимизации, что может улучшить видимость некоторых переменных. Это можно сделать, передав `COPTFLAGS=-O1` в make(1). Однако определённые классы ошибок в ядре могут проявляться иначе (или вообще не проявляться) при изменении уровня оптимизации.

Вы можете использовать этот сеанс почти как любой другой сеанс GDB, включая полный доступ к исходному коду, запуск в режиме gud (Grand Unified Debugger) внутри окна Emacs (что даёт автоматическое отображение исходного кода в другом окне Emacs) и т.д.

10.5. Отладка драйвера консоли

Поскольку для работы DDB требуется драйвер консоли, ситуация усложняется, если сам драйвер консоли неисправен. Возможно, вы вспомните о возможности использования последовательной консоли (либо с модифицированными загрузочными блоками, либо указав `-h` в строке `Boot:`), подключив стандартный терминал к первому последовательному порту. DDB работает с любым настроенным драйвером консоли, включая последовательную консоль.

10.6. Отладка взаимоблокировок

Вы можете столкнуться с так называемыми взаимоблокировками — ситуацией, когда система перестает выполнять полезную работу. Чтобы предоставить полезный отчёт об ошибке в такой ситуации, используйте ddb(4), как описано в предыдущем разделе. Включите в отчёт вывод команд `ps` и `trace` для подозрительных процессов.

Если возможно, рассмотрите проведение дополнительного исследования. Приведенный ниже рецепт особенно полезен, если вы подозреваете, что взаимная блокировка происходит на уровне VFS. Добавьте следующие параметры в файл конфигурации ядра.

```
makeoptions     DEBUG=-g
options     INVARIANTS
options     INVARIANT_SUPPORT
options     WITNESS
options     WITNESS_SKIPSPIN
```

```
options     DEBUG_LOCKS
options     DEBUG_VFS_LOCKS
options     DIAGNOSTIC
```

При возникновении взаимоблокировки, помимо вывода команды `ps`, предоставьте информацию из `show pcpu`, `show allpcpu`, `show locks`, `show alllocks`, `show lockedvnods` и `alltrace`.

Для получения осмысленных трассировок стека для потоковых процессов используйте `thread thread-id` для переключения на стек потока и выполните трассировку с помощью `where`.

10.7. Отладка ядра с помощью Dcons

dcons(4) — это очень простой драйвер консоли, который не связан напрямую с какими-либо физическими устройствами. Он просто читает и записывает символы из буфера в ядре или загрузчике и обратно. Благодаря своей простоте он очень полезен для отладки ядра, особенно с устройством FireWire®. В настоящее время FreeBSD предоставляет два способа взаимодействия с буфером извне ядра с помощью dconschat(8).

10.7.1. Dcons через FireWire®

Большинство контроллеров FireWire® (IEEE1394) основаны на спецификации OHCI, которая поддерживает физический доступ к памяти хоста. Это означает, что после инициализации контроллера хоста мы можем получить доступ к памяти хоста без помощи программного обеспечения (ядра). Мы можем использовать эту возможность для взаимодействия с dcons(4). dcons(4) предоставляет функциональность, аналогичную последовательной консоли. Он эмулирует два последовательных порта: один для консоли и DDB, другой для GDB. Поскольку удалённый доступ к памяти полностью обрабатывается аппаратным обеспечением, буфер dcons(4) остаётся доступным даже при крахе системы.

Устройства FireWire® не только встраиваются в материнские платы. Для настольных компьютеров существуют PCI-карты, а для ноутбуков можно приобрести интерфейс CardBus.

10.7.1.1. Включение поддержки FireWire® и Dcons на целевой машине

Чтобы включить поддержку FireWire® и Dcons в ядре *целевой машины*:

- Убедитесь, что ваше ядро поддерживает `dcons`, `dcons_crom` и `firewire`. `Dcons` должен быть статически связан с ядром. Для `dcons_crom` и `firewire` модули должны подойти.
- Убедитесь, что физический DMA включен. Возможно, потребуется добавить `hw.firewire.phydma_enable=1` в /boot/loader.conf.
- Добавьте параметры для отладки.
- Добавьте `dcons_gdb=1` в /boot/loader.conf, если вы используете GDB через FireWire®.
- Включите `dcons` в /etc/ttys.
- Это необязательно: чтобы принудительно сделать `dcons` высокоуровневой консолью,

добавьте `hw.firewire.dcons_crom.force_console=1` в loader.conf.

Чтобы включить поддержку FireWire® и Dcons в loader(8) на i386 или amd64:

Добавьте `LOADER_FIREWIRE_SUPPORT=YES` в /etc/make.conf и пересоберите loader(8):

```
# cd /sys/boot/i386 && make clean && make && make install
```

Чтобы включить dcons(4) в качестве активной низкоуровневой консоли, добавьте `boot_multicons="YES"` в /boot/loader.conf.

Вот несколько примеров конфигурации. Образец файла конфигурации ядра может содержать:

```
device dcons
device dcons_crom
options KDB
options DDB
options GDB
options ALT_BREAK_TO_DEBUGGER
```

И образец /boot/loader.conf может содержать:

```
dcons_crom_load="YES"
dcons_gdb=1
boot_multicons="YES"
hw.firewire.phydma_enable=1
hw.firewire.dcons_crom.force_console=1
```

10.7.1.2. Включение поддержки FireWire® и Dcons на главной машине

Чтобы включить поддержку FireWire® в ядре на *основной машине*:

```
# kldload firewire
```

Определите EUI64 (уникальный 64-битный идентификатор) контроллера FireWire® и используйте fwcontrol(8) или `dmesg`, чтобы найти EUI64 целевой машины.

Запустите dconschat(8), с:

```
# dconschat -e \# -br -G 12345 -t 00-11-22-33-44-55-66-77
```

Следующие комбинации клавиш могут быть использованы после запуска dconschat(8):

`~` + `.`	Отсоединиться

~	ALT BREAK
~	ПЕРЕЗАГРУЗИТЬ (RESET) целевую машину
~	Приостановить dconschat

Присоедините удалённый GDB, запустив kgdb(1) с сеансом удалённой отладки:

```
kgdb -r :12345 kernel
```

10.7.1.3. Некоторые общие рекомендации

Вот несколько общих советов:

Чтобы в полной мере использовать скорость FireWire®, отключите другие медленные драйверы консоли:

```
# conscontrol delete ttyd0       # serial console
# conscontrol delete consolectl # video/keyboard
```

Существует режим GDB для emacs(1); вот что нужно добавить в ваш .emacs:

```
(setq gud-gdba-command-name "kgdb -a -a -a -r :12345")
(setq gdb-many-windows t)
(xterm-mouse-mode 1)
M-x gdba
```

10.7.2. Dcons с KVM

Мы можем напрямую читать буфер dcons(4) через /dev/mem для работающих систем и в дампе памяти для систем после аварии. Это даёт аналогичный вывод команде `dmesg -a`, но буфер dcons(4) содержит больше информации.

10.7.2.1. Использование Dcons с KVM

Для использования dcons(4) с KVM:

Дамп буфера dcons(4) работающей системы:

```
# dconschat -1
```

Дамп буфера dcons(4) аварийного дампа:

```
# dconschat -1 -M vmcore.XX
```

Отладка ядра в реальном времени может быть выполнена через:

```
# fwcontrol -m target_eui64
# kgdb kernel /dev/fwmem0.2
```

10.8. Глоссарий параметров ядра для отладки

В этом разделе представлен краткий глоссарий параметров ядра, указываемых при компиляции и относящихся к отладке:

- `options KDB`: включает фреймворк отладки ядра. Необходим для `options DDB` и `options GDB`. Практически не влияет на производительность. По умолчанию отладчик будет запущен при панике вместо автоматической перезагрузки.
- `options KDB_UNATTENDED`: изменяет значение по умолчанию системной настройки `debug.debugger_on_panic` на 0, что управляет входом в отладчик при панике. Если `options KDB` не вкомпилировано в ядро, поведение по умолчанию — автоматическая перезагрузка при панике; если оно вкомпилировано в ядро, поведение по умолчанию — переход в отладчик, если не вкомпилирована опция `options KDB_UNATTENDED`. Если вы хотите оставить отладчик ядра вкомпилированным в ядро, но желаете, чтобы система перезагружалась, пока вы не готовы использовать отладчик для диагностики, используйте эту опцию.
- `options KDB_TRACE`: изменяет значение по умолчанию системной настройки `debug.trace_on_panic` на 1, что управляет автоматическим выводом трассировки стека при панике. Особенно полезно при использовании с `options KDB_UNATTENDED`, так как позволяет собрать базовую отладочную информацию на последовательной консоли или консоли FireWire, продолжая перезагрузку для восстановления.
- `options DDB`: включает поддержку консольного отладчика DDB. Этот интерактивный отладчик работает на активной низкоуровневой консоли системы, включая видеоконсоль, последовательную консоль или консоль FireWire. Он предоставляет базовые встроенные средства отладки, такие как трассировка стека, список процессов и потоков, вывод состояния блокировок, состояния виртуальной памяти, состояния файловой системы и управления ядром памяти. DDB не требует работы программного обеспечения на второй машине или возможности создания дампа памяти или полных символов отладки ядра, а также предоставляет детальную диагностику ядра во время выполнения. Многие ошибки могут быть полностью диагностированы с использованием только вывода DDB. Эта опция зависит от `options KDB`.
- `options GDB`: включает поддержку удалённого отладчика GDB, который может работать через последовательный кабель или FireWire. При входе в отладчик можно подключить GDB для проверки содержимого структур, генерации трассировки стека и т.д. Некоторые состояния ядра сложнее исследовать, чем в DDB, который способен автоматически создавать полезные сводки состояния ядра, например, автоматически обходить структуры отладки блокировок или управления памятью ядра, но для этого требуется вторая машина с запущенным отладчиком. С другой стороны, GDB объединяет информацию из исходного кода ядра и полных отладочных символов, знает полные определения структур данных, локальные переменные и поддерживает написание скриптов. Эта опция не требуется для запуска GDB на дампе памяти ядра. Данная опция зависит от `options KDB`.

- `options BREAK_TO_DEBUGGER`, `options ALT_BREAK_TO_DEBUGGER`: позволяют сигналу прерывания или альтернативному сигналу на консоли войти в отладчик. Если система зависает без паники, это полезный способ попасть в отладчик. Из-за текущей блокировки ядра сигнал прерывания, сгенерированный на последовательной консоли, значительно надёжнее для входа в отладчик и обычно рекомендуется. Данная опция оказывает незначительное или нулевое влияние на производительность.
- `options INVARIANTS`: включает в ядро большое количество проверок и тестов во время выполнения, которые постоянно проверяют целостность структур данных ядра и инварианты алгоритмов ядра. Эти тесты могут быть затратными, поэтому по умолчанию не включены, но они помогают обеспечить полезное поведение "fail stop", при котором определённые классы нежелательного поведения попадают в отладчик до возникновения повреждения данных ядра, что упрощает их отладку. Тесты включают в себя очистку памяти и проверку использования после освобождения, что является одним из наиболее значимых источников накладных расходов. Эта опция зависит от `options INVARIANT_SUPPORT`.
- `options INVARIANT_SUPPORT`: многие тесты, присутствующие в `options INVARIANTS`, требуют модифицированных структур данных или определения дополнительных символов ядра.
- `options WITNESS`: эта опция включает отслеживание и проверку порядка блокировок во время выполнения, что является неоценимым инструментом для диагностики взаимоблокировок. WITNESS поддерживает граф полученных порядков блокировок по типам блокировок и проверяет граф на каждом получении на наличие циклов (явных или неявных). Если цикл обнаружен, на консоль выводится предупреждение и трассировка стека, указывающие на возможное возникновение взаимоблокировки. WITNESS необходим для использования команд DDB `show locks`, `show witness` и `show alllocks`. Эта отладочная опция создаёт значительную нагрузку на производительность, которую можно несколько уменьшить с помощью `options WITNESS_SKIPSPIN`. Подробная документация доступна в witness(4).
- `options WITNESS_SKIPSPIN`: отключает проверку порядка блокировки spinlock во время выполнения с WITNESS. Поскольку spin-блокировки чаще всего захватываются в планировщике, а события планировщика происходят часто, эта опция может значительно ускорить системы, работающие с WITNESS. Эта опция зависит от `options WITNESS`.
- `options WITNESS_KDB`: изменяет значение по умолчанию системной настройки `debug.witness.kdb` на 1, что приводит к входу в отладчик при обнаружении нарушения порядка блокировок вместо простого вывода предупреждения. Эта опция зависит от `options WITNESS`.
- `options SOCKBUF_DEBUG`: выполнять расширенную проверку согласованности сокетных буферов во время выполнения, что может быть полезно для отладки как ошибок в сокетах, так и состояний гонки в протоколах и драйверах устройств, взаимодействующих с сокетами. Данная опция значительно влияет на производительность сети и может изменить временные параметры в состояниях гонки драйверов устройств.
- `options DEBUG_VFS_LOCKS`: отслеживает точки получения блокировок для lockmgr/vnode, расширяя объём информации, отображаемой командой `show lockedvnods` в DDB. Данная опция оказывает заметное влияние на производительность.

- `options DEBUG_MEMGUARD`: замена для malloc(9), аллокатор памяти ядра, который использует систему VM для обнаружения чтения или записи в освобождённую память. Подробности можно найти в memguard(9). Данная опция значительно влияет на производительность, но может быть очень полезна при отладке ошибок повреждения памяти ядра.
- `options DIAGNOSTIC`: включает дополнительные, более затратные диагностические тесты, аналогичные `options INVARIANTS`.
- `options KASAN`: включает отладчик адресов ядра (Kernel Address Sanitizer). Это включает инструментирование компилятора, которое может использоваться для обнаружения недопустимых обращений к памяти в ядре, таких как использование после освобождения и переполнение буфера. В значительной степени заменяет `options DEBUG_MEMGUARD`. Подробности и список поддерживаемых платформ см. в kasan(9).
- `options KMSAN`: включить отладчик использования памяти ядра (Kernel Memory Sanitizer). Это включает инструментирование компилятора, которое может использоваться для обнаружения использования неинициализированной памяти. Подробности и список поддерживаемых платформ см. в kmsan(9).

Часть IV: Приложения

Приложение A: Библиография

[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.