Глава 1

ВВЕДЕНИЕ

Эта глава знакомит с основными терминами и технологиями и показывает применение некоторых инструментов анализа производительности на основе BPF. Представленные здесь технологии будут подробно описаны в последующих главах.

1.1. ЧТО ТАКОЕ BPF И EBPF?

BPF расшифровывается как Berkeley Packet Filter (пакетный фильтр Беркли). Эта малоизвестная когда-то технология, разработанная в 1992 году, создавалась для увеличения производительности инструментов захвата пакетов [McCanne 92]. В 2013 году Алексей Старовойтов предложил существенно переделанную версию BPF [2], развитие которой продолжил с Дэниелем Боркманом. В 2014 она была включена в ядро Linux [3]. Это превратило BPF в механизм выполнения общего назначения, который можно использовать для самых разных целей, включая создание продвинутых инструментов анализа производительности.

Из-за широкой области применения BPF трудно описать. Она позволяет запускать мини-программы для обработки самых разных событий, происходящих в ядре и приложениях. Те, кто знаком с JavaScript, наверняка заметят некоторые сходства: с помощью JavaScript можно запускать на сайте мини-программы для обработки событий в браузере, например щелчков мыши, что дает возможность создавать веб-приложения для самых разных целей. Механизм BPF позволяет ядру запускать мини-программы в ответ на события в системе и в приложениях, таких как дисковый ввод/вывод, что открывает дорогу для новых системных технологий. Он делает ядро полностью программируемым и дает пользователям (включая прикладных программистов) возможность настраивать и контролировать свои системы для решения насущных проблем.

BPF — это гибкая и эффективная технология, состоящая из набора инструкций, хранимых объектов и вспомогательных функций. Она определяет набор виртуальных инструкций, поэтому ее можно считать виртуальной машиной. Эти инструкции исполняются средой выполнения BPF в ядре Linux, которая включает интерпретатор и JIT-компилятор, преобразующие инструкции BPF в машинные инструкции. Инструкции BPF сначала попадают в верификатор,

который проверяет их безопасность и гарантирует, что программа BPF не приведет к сбою или повреждению ядра (что не мешает конечному пользователю писать нелогичные программы, которые могут выполняться, но не иметь смысла). Компоненты BPF подробно описаны в главе 2.

До сих пор технология BPF в основном использовалась для организации сетевых взаимодействий, наблюдаемости и обеспечения безопасности. В этой книге основное внимание уделяется наблюдаемости (трассировке).

Расширенную версию BPF часто обозначают аббревиатурой eBPF (extended BPR — расширенная BPF), но официальной считается прежнее обозначение — BPF, без «e», поэтому в этой книге я буду использовать аббревиатуру «BPF». Ядро содержит только один механизм выполнения BPF (расширенный BPF), который выполняет инструкции как расширенной технологии BPF, так и «классической» BPF[1].

1.2. ЧТО ТАКОЕ ТРАССИРОВКА, ПРОСЛУШИВАНИЕ, ВЫБОРКА, ПРОФИЛИРОВАНИЕ И НАБЛЮДАЕМОСТЬ?

Эти термины используются для классификации методов и инструментов анализа.

Трассировка (tracing) — технология фиксации (записи) происходящих событий, основанная на использовании соответствующих инструментов BPF. Возможно, вам уже приходилось иметь дело с некоторыми специализированными инструментами трассировки. Инструмент strace(1), например, фиксирует и выводит события обращения к системным вызовам. Есть и инструменты, которые не трассируют, а измеряют события, используя фиксированные статистические счетчики, а затем выводят их, как, например, top(1). Отличительная черта трассировщика — это способность фиксировать исходные события и их метаданные. Такая информация может иметь очень большой объем, требующий последующей обработки и обобщения. Программные трассировщики, использующие BPF, могут запускать небольшие программы для обработки событий и формировать статистические метрики «на лету» или выполнять другие действия, чтобы избежать последующей дорогостоящей обработки.

Не у всех трассировщиков в названии есть слово «trace», как strace(1). Например, tcpdump(8) — это еще один специализированный инструмент трассировки сетевых пакетов. (Возможно, его следовало назвать tcptrace?) В ОС Solaris была своя версия tcpdump с именем snoop(1M)[2], названная так потому, что использовалась для прослушивания (snooping) сетевых пакетов. Я был первым, кто разработал

[1] Программы на основе классической версии BPF [McCanne 92] автоматически выполняются под управлением расширенного механизма BPF. Кроме того, развитие классической технологии BPF было прекращено.

[2] Раздел 2 в справочном руководстве Solaris предназначен для команд администрирования и обслуживания (в Linux ему соответствует раздел 8).

и опубликовал множество инструментов трассировки для Solaris, в названиях которых (может и неправильно) использовал «snoop». Поэтому у нас теперь есть execsnoop(8), opensnoop(8), biosnoop(8) и т. д. Прослушивание, вывод событий и трассировка обычно обозначают одно и то же. Эти инструменты будут описаны в следующих главах.

Термин «трассировка» (tracing) встречается не только в названиях инструментов, но также в описании механизма BPF, когда тот используется для наблюдаемости.

Инструменты **выборки (sampling)** выполняют некоторые измерения, чтобы составить общую картину цели. Их также называют инструментами создания профиля, или профилирования. Есть BPF-инструмент под названием profile(8), который берет выполняемый код по таймеру. Например, он может производить выборку каждые 10 миллисекунд, или, иначе говоря, 100 раз в секунду (на каждом процессоре). Преимущество инструментов выборки состоит в том, что у них обычно более низкий уровень оверхеда, чем у трассировщиков, потому что они оценивают только одно из большого числа событий. С другой стороны, выборка дает только приблизительную картину и может пропускать некоторые важные события.

Под **наблюдаемостью (observability)** понимается исследование системы через наблюдение с использованием специальных инструментов. К инструментам наблюдаемости относятся инструменты трассировки и выборки, а также инструменты, основанные на фиксированных счетчиках. К ним не относятся инструменты бенчмаркинга, которые изменяют состояние системы, экспериментируя с рабочей нагрузкой. Инструменты BPF, представленные в этой книге, все относятся к категории инструментов наблюдаемости и используют BPF для трассировки программ.

1.3. ЧТО ТАКОЕ BCC, BPFTRACE И IO VISOR?

Программировать, используя непосредственно инструкции BPF, довольно сложно, поэтому для языков высокого уровня были разработаны интерфейсы. Для трассировки в основном используются интерфейсы BCC и bpftrace.

Первой инфраструктурой трассировки на основе BPF стала **BCC** (BPF Compiler Collection — коллекция компиляторов для BPF). Она предоставляет среду программирования на C для использования BPF и интерфейсы для других языков: Python, Lua и C ++. Она также дала начало библиотекам libbcc и текущей версии libbpf[1] с функциями для обработки событий с помощью программ BPF. Репозиторий BCC содержит более 70 инструментов на основе BPF для анализа производительности и устранения неполадок. Вы можете установить BCC в своей системе и использовать готовые инструменты, не написав ни строчки кода для BCC. В этой книге вы познакомитесь со многими такими инструментами.

[1] Первая версия libbpf была разработана Ван Нанем (Wang Nan) для использования с perf [4]. Сейчас libbpf — это часть исходного кода ядра.

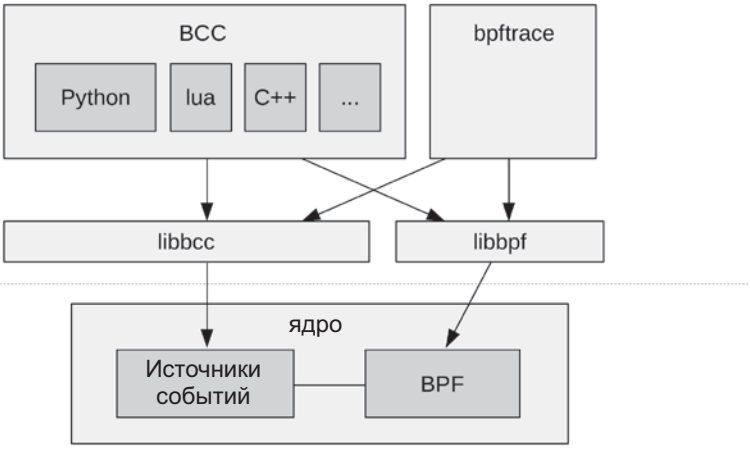

Рис. 1.1. BCC, bpftrace и BPF

bpftrace — более новый интерфейс, предоставляющий специализированный язык высокого уровня для разработки инструментов BPF. Язык bpftrace настолько лаконичен, что мне удалось вставить в эту книгу исходный код инструментов, чтобы показать, какие механизмы они используют и как именно. bpftrace построен на основе библиотек libbcc и libbpf.

Связь между BCC и bpftrace можно видеть на рис. 1.1. Они прекрасно дополняют друг друга: bpftrace идеально подходит для создания однострочных и нестандартных коротких сценариев, а BCC лучше подходит для сложных сценариев и демонов и позволяет использовать другие библиотеки. Например, многие BCC-инструменты на Python используют библиотеку argparse для сложного и точного управления аргументами командной строки.

Сейчас разрабатывается еще один интерфейс BPF — ply [5]. Предполагается, что он должен получиться максимально легковесным и с минимумом зависимостей, чтобы его можно было использовать во встраиваемых системах Linux. Если для вашей среды ply подходит лучше, чем bpftrace, эта книга все равно будет вам полезна в качестве руководства по анализу с использованием BPF. Десятки инструментов на bpftrace, представленные здесь, с таким же успехом могут выполняться с ply, если их переписать с помощью синтаксиса ply. (В будущих версиях ply может появиться прямая поддержка синтаксиса bpftrace.) Основное внимание в этой книге уделяется интерфейсу bpftrace как более развитому и обладающему всеми возможностями, необходимыми для анализа всех целей.

BCC и bpftrace не входят в код ядра и размещены в проекте Linux Foundation на Github с названием **IO Visor**:

https://github.com/iovisor/bcc

https://github.com/iovisor/bpftrace

В этой книге под словами *трассировка с использованием BPF* я буду подразумевать инструменты, использующие любой из интерфейсов — BCC и bpftrace.

1.4. ПЕРВЫЙ ВЗГЛЯД НА BCC: БЫСТРЫЙ АНАЛИЗ

Посмотрим на некоторые результаты, возвращаемые разными инструментами. Следующий инструмент следит за новыми процессами и выводит сводную информацию о каждом сразу после его запуска. Этот BCC-инструмент execsnoop(8) трассирует системный вызов execve(2), который является вариантом exec(2) (отсюда и имя). Установка инструментов BCC описана в главе 4, и в следующих главах эти инструменты будут представлены подробнее.

```
# execsnoop
PCOMM          PID    PPID   RET  ARGS
run            12983  4469     0  ./run
bash           12983  4469     0  /bin/bash
svstat         12985  12984    0  /command/svstat /service/httpd
perl           12986  12984    0  /usr/bin/perl -e $l=<>;$l=~/(\d+) sec/;print
$1||0
ps             12988  12987    0  /bin/ps --ppid 1 -o pid,cmd,args
grep           12989  12987    0  /bin/grep org.apache.catalina
sed            12990  12987    0  /bin/sed s/^ *//;
cut            12991  12987    0  /usr/bin/cut -d -f 1
xargs          12992  12987    0  /usr/bin/xargs
echo           12993  12992    0  /bin/echo
mkdir          12994  12983    0  /bin/mkdir -v -p /data/tomcat
mkdir          12995  12983    0  /bin/mkdir -v -p /apps/tomcat/webapps
^C
#
```

В полученном выводе видно, какие процессы запускались, пока происходила трассировка: в основном это настолько кратковременные процессы, что они невидимы для других инструментов. Здесь можно видеть строки, соответствующие запуску стандартных утилит Unix: ps(1), grep(1), sed(1), cut(1) и т. д. Но рассматривая этот вывод на книжной странице, нельзя сказать, насколько быстро выводились показанные строки. Чтобы исправить этот недостаток, добавим параметр командной строки `-t`, который заставит execsnoop(8) выводить время, прошедшее с начала трассировки:

```
# execsnoop -t
TIME(s) PCOMM          PID    PPID   RET  ARGS
0.437   run            15524  4469     0  ./run
0.438   bash           15524  4469     0  /bin/bash
0.440   svstat         15526  15525    0  /command/svstat /service/httpd
0.440   perl           15527  15525    0  /usr/bin/perl -e $l=<>;$l=~/(\d+)
sec/;prin...
0.442   ps             15529  15528    0  /bin/ps --ppid 1 -o pid,cmd,args
[...]
0.487   catalina.sh    15524  4469     0  /apps/tomcat/bin/catalina.sh start
```

```
0.488   dirname         15549  15524    0  /usr/bin/dirname /apps/tomcat/bin/
                                           catalina.sh
1.459   run             15550  4469     0  ./run
1.459   bash            15550  4469     0  /bin/bash
1.462   svstat          15552  15551    0  /command/svstat /service/nflx-httpd
1.462   perl            15553  15551    0  /usr/bin/perl -e $l=<>;$l=~/(\d+)
sec/;prin...
[...]
```

Я сократил вывод (о чем свидетельствует `[...]`), но новый столбец с отметкой времени помогает заметить закономерность: новые процессы запускаются с интервалом в 1 секунду. Просматривая вывод, я могу сказать, что каждую секунду запускается 30 новых процессов, после чего следует пауза в 1 секунду.

В этом выводе показана реальная проблема, которая была в Netflix и которую я анализировал с помощью execsnoop(8). Это происходило на сервере для микробенчмаркинга, но результаты бенчмарков слишком сильно отличались, чтобы им доверять. Я запустил execsnoop(8), когда система должна была простаивать, и обнаружил проблему! Каждую секунду эти процессы запускались и нарушали работу бенмчмарков. Причина крылась в неправильно настроенном сервисе, который пытался запуститься каждую секунду, терпел неудачу и запускался снова. После того как сервис был деактивирован, эти процессы перестали появляться (что было подтверждено с помощью execsnoop(8)) и бенчмарки стали стабильными.

Вывод execsnoop(8) помогает в анализе производительности с использованием методологии под названием *характеристика рабочей нагрузки*, которая поддерживается многими другими инструментами BPF, описанными здесь. Эта методология решает простую задачу: определить величину рабочей нагрузки. Понимания, как распределяется рабочая нагрузка, часто достаточно для решения проблем, без нужды углубляться в исследование задержек или в детальный анализ. Здесь к системе была применена процедура определения рабочей нагрузки. Более подробно с этой и другими методологиями вы познакомитесь в главе 3.

Попробуйте запустить инструмент execsnoop(8) в своих системах и оставьте его поработать в течение часа. Что необычного вы заметили?

execsnoop(8) выводит информацию о каждом событии, однако другие инструменты используют BPF для получения сводной информации. Еще один инструмент, который можно использовать для быстрого анализа, — biolatency(8), который обобщает сведения об операциях ввода/вывода для блочного устройства (дисковый ввод/вывод) в виде гистограммы задержки.

Ниже показан вывод инструмента biolatency(8) на промышленной базе данных, которая чувствительна к высокой задержке, так как имеет соглашение об уровне обслуживания по доставке запросов в течение определенного количества миллисекунд.

```
# biolatency -m
Tracing block device I/O... Hit Ctrl-C to end.
```

```
^C
     msecs               : count     distribution
         0 -> 1          : 16335    |****************************************|
         2 -> 3          : 2272     |*****                                   |
         4 -> 7          : 3603     |********                                |
         8 -> 15         : 4328     |**********                              |
        16 -> 31         : 3379     |********                                |
        32 -> 63         : 5815     |**************                          |
        64 -> 127        : 0        |                                        |
       128 -> 255        : 0        |                                        |
       256 -> 511        : 0        |                                        |
       512 -> 1023       : 11       |                                        |
```

После запуска инструмент biolatency(8) включает регистрацию событий блочного ввода/вывода, а их задержки вычисляются и обобщаются механизмом BPF. Когда инструмент завершается (пользователь нажимает Ctrl-C), выводится сводная информация. Я добавил параметр `-m`, чтобы обеспечить вывод информации в миллисекундах.

В этом выводе есть интересная деталь — бимодальный характер распределения задержек, а также наличие выбросов. Наиболее типичный режим (как видно на ASCII-гистограмме) приходится на диапазон от 0 до 1 миллисекунды, которому соответствует 16 355 операций ввода/вывода, зарегистрированных во время трассировки. Скорее всего, к этим операциям относятся попадания в дисковый кэш, а также операции с устройством флеш-памяти. Второй наиболее типичный режим охватывает диапазон от 32 до 63 миллисекунд и включает намного более медленные операции, чем ожидалось, и вероятно, это замедление обусловлено постановкой запросов в очередь. Для более подробного исследования этого режима можно использовать другие инструменты BPF. Наконец, 11 операций ввода/вывода попали в диапазон от 512 до 1023 миллисекунд. Эти очень медленные операции называют *выбросами*. Теперь, когда мы знаем, что они есть, их можно более детально изучить с помощью других инструментов BPF. Для команды базы данных это приоритетная задача: если БД будет блокироваться на этих операциях ввода/вывода, произойдет превышение целевого уровня задержки.

1.5. ОБЛАСТЬ ВИДИМОСТИ МЕХАНИЗМА ТРАССИРОВКИ BPF

Механизму трассировки BPF доступен для наблюдения весь программный стек, что позволяет создавать новые инструменты и производить инструментацию по мере необходимости. Механизм трассировки BPF можно использовать на промышленном сервере без всякой подготовки, то есть без перезагрузки системы или перезапуска приложений в каком-то специальном режиме. Его можно сравнить с рентгеновским зрением: если нужно исследовать какой-то компонент ядра, устройство или

прикладную библиотеку, вы сможете увидеть все это в таком свете, в каком никто и никогда не видел их — вживую на продакшене.

Для иллюстрации на рис. 1.2 показан обобщенный программный стек системы, который я аннотировал названиями инструментов трассировки на основе BPF, предназначенными для наблюдения за различными компонентами. Все эти инструменты созданы в проектах BCC и bpftrace, а также специально для этой книги. Многие из них будут описаны в следующих главах.

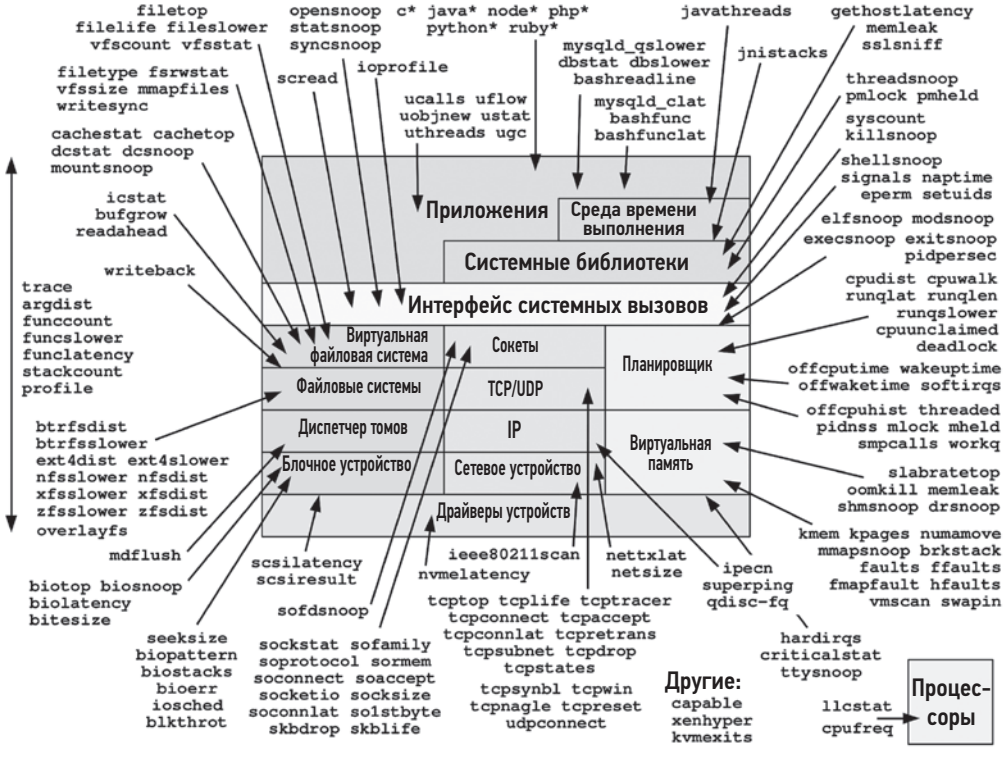

Рис. 1.2. Инструменты оценки производительности на основе BPF и их области видимости

Посмотрите на различные инструменты, которые можно использовать для исследования таких компонентов, как планировщик процессов в ядре, виртуальная память, файловые системы и т. д. Из диаграммы на рис. 1.2 можно заметить, что у механизма BPF нет слепых зон.

В табл. 1.1 перечислены инструменты, традиционно используемые для исследования этих компонентов, а также указана способность механизма BPF наблюдать за этими компонентами.

Таблица 1.1. Традиционные инструменты анализа

Компоненты	Традиционные инструменты анализа	Механизм трассировки BPF
Приложения со средами выполнения своих языков: Java, Node.js, Ruby, PHP	Отладчики среды времени выполнения	Да, с поддержкой среды времени выполнения
Приложения на компилируемых языках: C, C++, Golang	Системные отладчики	Да
Системные библиотеки: /lib/*	ltrace(1)	Да
Интерфейс системных вызовов	strace(1), perf(1)	Да
Ядро: планировщик, файловые системы, сетевые протоколы (TCP, IP и др.)	Ftrace, perf(1) для выборки	Да, и более подробно
Аппаратное обеспечение: процессоры, устройства	perf, sar, счетчики в /proc	Да, прямо или косвенно[1]

Традиционные инструменты могут дать полезную начальную информацию о компоненте, исследование которого можно продолжить на более детальном уровне с помощью инструментов трассировки BPF. В главе 3 кратко описывается базовый анализ производительности с помощью системных инструментов, помогающий получить начальную информацию.

1.6. ДИНАМИЧЕСКАЯ ИНСТРУМЕНТАЦИЯ: KPROBES И UPROBES

Механизм трассировки BPF поддерживает несколько источников событий для обеспечения видимости всего программного стека. Особого упоминания заслуживает поддержка динамической инструментации (иногда ее называют динамической трассировкой) — возможность вставлять контрольные точки в действующее ПО в процессе его выполнения. Динамическая инструментация не порождает оверхед, когда не используется, так как в этом случае ПО выполняется в своем первоначальном виде. Она часто применяется инструментами BPF для определения начала и конца выполнения функций в ядре и в приложении (для любой из десятков тысяч функций, которые обычно имеются в программном стеке). Такое глубокое и всеобъемлющее ви́дение напоминает суперсилу!

[1] BPF может не иметь возможности непосредственно инструментировать прошивку на устройстве, но он может косвенно определять поведение на основе отслеживания событий драйвера ядра или PMC.

Впервые идея динамической инструментации была предложена в 1990-х годах [Hollingsworth 94]. Она основывалась на технологии, используемой отладчиками для вставки точек останова по произвольным адресам команд. Встретив такую точку, динамически инструментированное ПО записывает нужную информацию и автоматически продолжает выполнение, не передавая управление интерактивному отладчику. Тогда же были разработаны первые инструменты динамической трассировки (например, kerninst [Tamches 99]), включавшие языки трассировки, но эти инструменты оставались малоизвестными и редко используемыми. Отчасти это было связано с тем, что их применение сопряжено с большим риском: динамическая трассировка требует изменения инструкций в адресном пространстве, и любая ошибка может привести к немедленному повреждению кода и аварийному завершению процесса или ядра.

Первая поддержка динамической инструментации в Linux была реализована в 2000 году в IBM, она получила название DProbes, но набор исправлений был отклонен[1]. Динамическая инструментация для функций ядра (kprobes), добавленная в Linux в 2004 году и корнями уходящая в DProbes, все еще оставалась малоизвестной и сложной в использовании.

Все изменилось в 2005 году, когда Sun Microsystems выпустила свою версию динамической трассировки DTrace с простым в использовании языком D и включила ее в ОС Solaris 10. В ту пору Solaris была известной операционной системой, славившейся стабильностью работы, поэтому включение в нее пакета DTrace помогло доказать, что динамическая трассировка может быть безопасной для применения в промышленных системах. Это стало поворотным моментом для технологии. Я опубликовал много статей, где показаны реальные случаи использования DTrace, а также разработал и опубликовал множество инструментов DTrace. Кроме того, отдел маркетинга в Sun продвигал не только продажи, но и технологии Sun. Считалось, что это дает дополнительные конкурентные преимущества. Sun Educational Services включили DTrace в стандартные курсы изучения Solaris и разработали специальные курсы по DTrace. Все это способствовало превращению динамической инструментации из малопонятной технологии в широко известную и востребованную.

В 2012 году в Linux была добавлена поддержка динамической инструментации для функций пользовательского уровня в виде uprobes. Механизм трассировки BPF использует и kprobes, и uprobes для динамической инструментации всего программного стека.

Чтобы показать, как можно применять динамическую трассировку, в табл. 1.2 приведены примеры зондов bpftrace, которые используют kprobes и uprobes (bpftrace подробно рассматривается в главе 5).

---

[1] Причины отказа принять DProbes в ядро Linux обсуждаются в первом примере в статье Энди Клина (Andi Kleen) «On submitting kernel patches», где дается ссылка на источник в Documentation/process/submitting-patches.rst [6].

Таблица 1.2. Примеры использования kprobe и uprobe в bpftrace

Зонд	Описание
kprobe:vfs_read	Инструментирует начало выполнения функции ядра vfs_read()
kretprobe:vfs_read	Инструментирует возврат[1] из функции ядра vfs_read()
uprobe:/bin/bash:readline	Инструментирует начало выполнения функции readline() в /bin/bash
uretprobe:/bin/bash:readline	Инструментирует возврат из функции readline() в /bin/bash

1.7. СТАТИЧЕСКАЯ ИНСТРУМЕНТАЦИЯ: ТОЧКИ ТРАССИРОВКИ И USDT

Динамическая инструментация имеет и обратную сторону: инструменты могут переименовываться или удаляться в новой версии ПО. Это называется *проблемой стабильности интерфейса*. После обновления ядра или прикладного ПО иногда оказывается, что инструмент BPF работает не так, как должен. Он может вызывать ошибку, сообщая о невозможности найти функцию для инструментации, или вообще ничего не выводить. Другая проблема в том, что компиляторы могут преобразовывать функции во встраиваемые фрагменты кода с целью оптимизации, делая их недоступными для инструментации через kprobes или uprobes[2].

Одно из решений проблем стабильности интерфейса и встраивания функций — это статическая инструментация, когда стабильные имена событий внедряются в ПО и поддерживаются разработчиками. Механизм трассировки BPF поддерживает точки трассировки для статической инструментации ядра и статически определяемые точки трассировки на уровне пользователя (User-level Statically Defined Tracing, USDT) для статической инструментации на уровне пользователя. Недостаток статической инструментации в том, что поддержка таких точек инструментации становится бременем для разработчиков, поэтому если они существуют, их количество обычно ограниченно.

Эти детали важны, только если вы собираетесь разрабатывать свои инструменты BPF. В таком случае рекомендуется сначала попытаться использовать статическую трассировку (с использованием точек трассировки и USDT), а к динамической трассировке (с помощью kprobes и uprobes) переходить только тогда, когда статическая недоступна.

---

[1] Функция имеет одну точку входа, но может иметь несколько точек выхода: она может вызывать return в нескольких местах. Зонд, настроенный на возврат из функции, реагирует на все точки выхода. (См. главу 2, где объясняется, как это возможно.)

[2] Одно из возможных решений — трассировка по смещению в функции, но этот прием еще менее стабилен, чем трассировка входа в функцию.

В табл. 1.3 приводятся примеры зондов bpftrace для статической инструментации с использованием точек трассировки и USDT. Об упомянутой в этой таблице точке трассировки open(2) рассказано в разделе 1.8.

Таблица 1.3. Примеры точек трассировки и USDTв bpftrace

Зонд	Описание
tracepoint:syscalls:sys_enter_open	Инструментирует системный вызов open(2)
usdt:/usr/sbin/mysqld:mysql:query__start	Инструментирует вызов query__start из /usr/sbin/mysqld

1.8. ПЕРВЫЙ ВЗГЛЯД НА BPFTRACE: ТРАССИРОВКА OPEN()

Начнем знакомство с bpftrace с попытки выполнить трассировку системного вызова open(2). Для этого есть статическая точка трассировки (syscalls:sys_enter_open[1]). Я покажу дальше короткую программу на bpftrace в командной строке: однострочный сценарий.

От вас пока не требуется понимать код этого однострочного сценария; язык bpftrace и инструкции по установке описаны ниже, в главе 5. Но вы наверняка догадаетесь, что делает эта программа, даже не зная языка, потому что он достаточно прост и понятен (понятность языка — признак хорошего дизайна). А пока просто обратите внимание на вывод инструмента.

```
# bpftrace -e 'tracepoint:syscalls:sys_enter_open { printf("%s %s\n", comm,
    str(args->filename)); }'
Attaching 1 probe...
slack /run/user/1000/gdm/Xauthority
slack /run/user/1000/gdm/Xauthority
slack /run/user/1000/gdm/Xauthority
slack /run/user/1000/gdm/Xauthority
^C
#
```

Вывод показывает имя процесса и имя файла, переданное системному вызову open(2): bpftrace трассирует всю систему, поэтому в выводе будет видно любое приложение, вызывающее open(2). Каждая строка вывода соответствует одному системному вызову, и этот сценарий является примером инструмента, который выводит информацию о каждом событии. Механизм трассировки BPF можно использовать не только для анализа промышленных серверов. Например, я запускал его на своем ноутбуке, когда писал эту книгу, и он показывал файлы, которые открывало приложение чата Slack.

[1] Для доступа к этим точкам трассировки системных вызовов ядро Linux должно быть собрано с включенным параметром CONFIG_FTRACE_SYSCALLS.

Программа для BPF определена в одинарных кавычках. Она была скомпилирована и запущена сразу, как только я нажал Enter для запуска команды bpftrace. bpftrace также активировала точку трассировки open(2). Когда я нажал Ctrl-C, чтобы остановить команду, точка трассировки open(2) была деактивирована и моя маленькая программа для BPF была остановлена и удалена. Вот как работает инструментация в механизме трассировки BPF: активация точки трассировки и выполнение производятся, только пока выполняется команда, и могут длиться всего несколько секунд.

Этот сценарий генерировал вывод медленнее, чем я ожидал: думаю, что я пропустил какие-то события системного вызова open(2). Ядро поддерживает несколько вариантов open, а я трассировал только один из них. С помощью bpftrace можно вывести список всех точек трассировки open, использовав параметр `-l` и подстановочный знак:

```
# bpftrace -l 'tracepoint:syscalls:sys_enter_open*'
tracepoint:syscalls:sys_enter_open_by_handle_at
tracepoint:syscalls:sys_enter_open
tracepoint:syscalls:sys_enter_openat
```

Как мне кажется, вариант openat(2) используется чаще. Мое предположение подтвердил другой однострочный сценарий для bpftrace:

```
# bpftrace -e 'tracepoint:syscalls:sys_enter_open* { @[probe] = count(); }'
Attaching 3 probes...
^C

@[tracepoint:syscalls:sys_enter_open]: 5
@[tracepoint:syscalls:sys_enter_openat]: 308
```

Повторюсь: детали кода этого однострочного сценария я объясню в главе 5. А пока вам важно понимать только вывод. Теперь сценарий выводит количество задействованных точек трассировки, а не события. Результат подтверждает, что системный вызов openat(2) вызывается чаще — в данном случае 308 раз против пяти вызовов open(2). Эта информация вычисляется в ядре программой BPF.

Я могу добавить вторую точку трассировки в свой сценарий и трассировать сразу два системных вызова, open(2) и openat(2). Но этот новый сценарий получится слишком громоздким для командной строки, поэтому лучше сохранить его в выполняемом файле, чтобы его было легче править с помощью текстового редактора. Это уже было сделано: bpftrace поставляется со сценарием opensnoop.bt, который трассирует начало и конец каждого системного вызова и выводит данные в виде столбцов:

```
# opensnoop.bt
Attaching 3 probes...
Tracing open syscalls... Hit Ctrl-C to end.
PID    COMM               FD ERR PATH
2440   snmp-pass           4   0 /proc/cpuinfo
2440   snmp-pass           4   0 /proc/stat
25706  ls                  3   0 /etc/ld.so.cache
25706  ls                  3   0 /lib/x86_64-linux-gnu/libselinux.so.1
```

```
25706  ls                  3   0 /lib/x86_64-linux-gnu/libc.so.6
25706  ls                  3   0 /lib/x86_64-linux-gnu/libpcre.so.3
25706  ls                  3   0 /lib/x86_64-linux-gnu/libdl.so.2
25706  ls                  3   0 /lib/x86_64-linux-gnu/libpthread.so.0
25706  ls                  3   0 /proc/filesystems
25706  ls                  3   0 /usr/lib/locale/locale-archive
25706  ls                  3   0 .
1744   snmpd               8   0 /proc/net/dev
1744   snmpd              -1   2 /sys/class/net/lo/device/vendor
2440   snmp-pass           4   0 /proc/cpuinfo
^C
#
```

В столбцах выводится следующая информация: идентификатор процесса (PID), имя команды процесса (COMM), дескриптор файла (FD), код ошибки (ERR) и путь к файлу, который системный вызов пытался открыть (PATH). Инструмент opensnoop.bt можно использовать для устранения неполадок в ПО, которое будет пытаться открыть файлы, используя неправильный путь, а также для определения местоположения конфигурационных файлов и журналов по событиям обращения к ним. Он также может выявить некоторые проблемы с производительностью, например слишком быстрое открытие файлов или слишком частую проверку неправильных местоположений. У этого инструмента множество применений.

bpftrace поставляется с более чем 20 подобными готовыми инструментами, а BCC — с более чем 70. Помимо помощи в непосредственном решении проблем, эти инструменты показывают код, изучив который можно понять, как трассировать разные цели. Иногда могут наблюдаться некоторые странности, как мы видели на примере трассировки системного вызова open(2), и код инструментов способен подсказать их причины.

1.9. НАЗАД К BCC: ТРАССИРОВКА OPEN()

Теперь рассмотрим BCC-версию opensnoop(8):

```
# opensnoop
PID    COMM               FD ERR PATH
2262   DNS Res~er #657    22   0 /etc/hosts
2262   DNS Res~er #654   178   0 /etc/hosts
29588  device poll         4   0 /dev/bus/usb
29588  device poll         6   0 /dev/bus/usb/004
29588  device poll         7   0 /dev/bus/usb/004/001
29588  device poll         6   0 /dev/bus/usb/003
^C
#
```

Вывод этого инструмента выглядит очень похожим на вывод более раннего однострочного сценария, по крайней мере он имеет те же столбцы. Но в выводе этого инструмента opensnoop(8) есть то, чего нет в bpftrace-версии: его можно вызвать с разными параметрами командной строки:

```
# opensnoop -h
порядок использования: opensnoop [-h] [-T] [-x] [-p PID] [-t TID]
                                 [-d DURATION] [-n NAME]
                                 [-e] [-f FLAG_FILTER]

Трассирует системные вызовы open()

необязательные аргументы:
  -h, --help            вывести эту справку и выйти
  -T, --timestamp       включить отметку времени в вывод
  -x, --failed          показать только неудачные вызовы open
  -p PID, --pid PID     трассировать только этот PID
  -t TID, --tid TID     трассировать только этот TID
  -d DURATION, --duration DURATION
                        общая продолжительность трассировки в секундах
  -n NAME, --name NAME  выводить только имена процессов, содержащие это имя
  -e, --extended_fields
                        показать дополнительные поля
  -f FLAG_FILTER, --flag_filter FLAG_FILTER
                        фильтровать по аргументу с флагами (например, O_WRONLY)

примеры:
     ./opensnoop           # трассировать все системные вызовы open()
     ./opensnoop -T        # включить отметки времени
     ./opensnoop -x        # показать только неудачные вызовы open
     ./opensnoop -p 181    # трассировать только PID 181
     ./opensnoop -t 123    # трассировать только TID 123
     ./opensnoop -d 10     # трассировать только 10 секунд
     ./opensnoop -n main   # выводить только имена процессов, содержащие "main"
     ./opensnoop -e        # показать дополнительные поля
     ./opensnoop -f O_WRONLY -f O_RDWR # выводить только вызовы для записи
```

Инструменты bpftrace, как правило, проще и выполняют одну конкретную задачу. Инструменты BCC, напротив, обычно сложнее и поддерживают несколько режимов работы. Конечно, можно изменить инструмент для bpftrace, чтобы он отображал только неудачные вызовы open, но уже есть BCC-версия, поддерживающая такую возможность с параметром `-x`:

```
# opensnoop -x
PID    COMM               FD ERR PATH
991    irqbalance         -1   2 /proc/irq/133/smp_affinity
991    irqbalance         -1   2 /proc/irq/141/smp_affinity
991    irqbalance         -1   2 /proc/irq/131/smp_affinity
991    irqbalance         -1   2 /proc/irq/138/smp_affinity
991    irqbalance         -1   2 /proc/irq/18/smp_affinity
20543  systemd-resolve    -1   2 /run/systemd/netif/links/5
20543  systemd-resolve    -1   2 /run/systemd/netif/links/5
20543  systemd-resolve    -1   2 /run/systemd/netif/links/5
[...]
```

Этот вывод показывает повторяющиеся сбои. Такие закономерности могут указывать на неэффективность или неправильную конфигурацию, которые можно исправить.

Инструменты BCC часто имеют несколько параметров для изменения поведения, что делает их более универсальными, чем инструменты bpftrace. Они могут послужить хорошей отправной точкой: с их помощью вы сумеете решить проблему, не написав ни строчки кода для BPF. Но если им не хватит глубины видимости, можно переключиться на bpftrace, имеющий более простой язык для разработки, и создать свои инструменты.

bpftrace впоследствии можно преобразовать в более сложный инструмент BCC, поддерживающий разнообразные параметры, подобно инструменту opensnoop(8), который был показан выше. Инструменты BCC также могут поддерживать различные события: использовать точки трассировки, если они доступны, а в противном случае переключаться на kprobes. Но имейте в виду, что программировать инструменты BCC намного сложнее, и эта тема выходит за рамки книги, где основное внимание уделяется программированию на bpftrace. В приложении C вы найдете ускоренный курс по разработке инструментов BCC.

1.10. ИТОГИ

Инструменты трассировки BPF могут использоваться для анализа производительности и устранения неполадок, и есть два основных проекта, которые предоставляют их: BCC и bpftrace. В этой главе мы познакомились с расширенным механизмом BPF, BCC, bpftrace, а также с динамической и статической инструментацией.

Следующая глава более подробно расскажет об этих технологиях. Если вы спешите приступить к решению проблем, можете пропустить главу 2 и перейти к главе 3 или другой, затрагивающей интересующую вас тему. В этих последующих главах широко используются термины, многие из которых объясняются в главе 2, но их описание также можно найти в глоссарии.