# Часть 2 : «Hello World» от eBPF

В предыдущей главе я обсуждала, почему eBPF такой мощный, и это нормально, если вы пока еще не чувствуете что у вас есть конкретное представление о том что на самом деле означает запуск программ eBPF. В этой главе я буду использовать простой пример «Hello World» чтобы дать вам лучшее представление о нем.

Как вы узнаете дальше, читая эту книгу, существует несколько различных библиотек и фреймворков для написания приложений eBPF. В качестве разминки я покажу вам, вероятно, наиболее доступный подход с точки зрения программирования: фреймворк BCC Python. Этот способ предлагает очень простой способ написания базовых программ eBPF. По причинам, которые я обосную в главе 5, это не обязательно тот подход, который я бы рекомендовала в наши дни для рабочих приложений, которые вы собираетесь распространять среди других пользователей, но он отлично подходит для ваших первых шагов.

> Если вы хотите попробовать этот код самостоятельно, он доступен по адресу https://github.com/lizrice/learning-ebpf в каталоге Chapter2. Вы найдете проект BCC по адресу https://github.com/iovisor/bcc и инструкции по установке BCC находятся на https://github.com/iovisor/bcc/blob/master/INSTALL.md .

## BCC's "Hello World"

Ниже приведен полный исходный код `hello.py`, приложения eBPF «Hello World»[1], написанного с использованием библиотеки BCC Python:

```
#!/usr/bin/python
from bcc import BPF

program = r"""
   int hello(void *ctx) {
   bpf_trace_printk("Hello World!");
   return 0;
}
"""
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")
b.trace_print()
```

Этот код состоит из двух частей: самой программы eBPF, которая будет выполняться в ядре[2], и некоторого кода пользовательского пространства, который загружает программу eBPF в ядро и считывает сгенерированную ею трассировку. Как вы можете видеть на рис. 2-1,

1 Первоначально я написала это для доклада под названием «Руководство для начинающих по программированию eBPF». Вы можете найти исходный код вместе со ссылками на слайды и видео по адресу https://github.com/lizrice/ebpf-beginners.

2 Напомним, что по синтаксическим правилам Python всё, что записано между 3-кратными кавычками (""" — открывающие и закрывающие) квалифицируется как «многострочная текстовая строка», а предшествующий квалификатор r или R — признак «необработанной» (raw, сырой) строки, в которой любые спецсимволы не рассматриваются как имеющие какое-то специальное значение. Это будет неоднократно использоваться по всему тексту. (Прим. пер.)

`hello.py` — это часть пользовательского пространства этого приложения, а `hello()` — это программа eBPF, работающая в ядре.

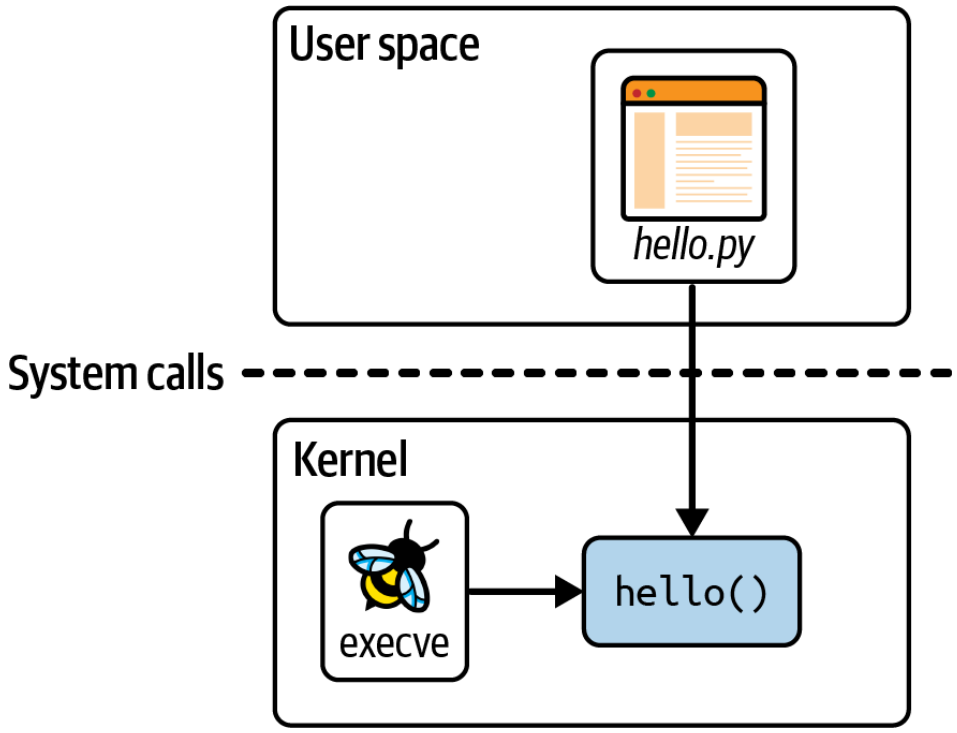


*Рисунок 2-1. Пользовательское пространство и компоненты ядра «Hello World»*

Давайте углубимся в каждую строку исходного кода, чтобы лучше понять его.

Первая строка говорит вам что это код Python, а программа, которая может его запустить, — это интерпретатор Python (`/usr/bin/python`).

Сама программа eBPF написана на C-коде, и вот эта ее часть:

```
int hello(void *ctx) {
    bpf_trace_printk("Hello World!");
    return 0;
}
```

Всё, что делает программа eBPF, — это использует функцию-помощник `bpf_trace_printk()` для написания сообщения. Функции-помощники[3] — еще одна сущность, которая отличает «расширенный» BPF от его «классического» предшественника. Это тот набор функций, которые программы eBPF могут вызывать для взаимодействия с системой. Я буду обсуждать их далее в главе 5. А пока вы можете просто думать об этом как просто о печати строки текста.

Вся программа eBPF определяется как строка с именем `program` в коде Python. Эта программа на C должна быть скомпилирована, прежде чем ее можно будет выполнить, но BCC позаботится об этом за вас. (В следующей главе вы увидите, как самостоятельно компилировать программы eBPF.) Всё, что вам нужно сделать, это передать эту строку в качестве параметра при создании объекта BPF, как в следующей строке:

```
b = BPF(text=program)
```

3 В различной англоязычной литературе для таких функций прижился термин: helper — это просто некоторая условно выделенная группа вызовов API, используемых в рамках отдельной темы рассмотрения. В переводе термин просто «помощник» будет вуалировать смысл изложения, поэтому мы будем, по возможности, использовать в качестве эквивалента: «функция-помощник» (Прим. пер.).

Программы eBPF должны быть привязаны к событию, и для этого примера я выбрала присоединение к системному вызову `execve`, который является системным вызовом, используемым для выполнения программы. Всякий раз, когда что-либо или кто-либо запускает новую программу, выполняемую на этой машине, вызывается функция `execve()`, которая активирует программу eBPF. Хотя имя `execve()` является стандартным API в Linux, имя самой функции, реализующей его в ядре, зависит от конкретной архитектуры чипа, но BCC дает нам удобный способ найдите имя функции для машины, на которой мы работаем:

```
syscall = b.get_syscall_fnname("execve")
```

Теперь `syscall` представляет имя функции ядра, к которой я собираюсь подключиться, используя `kprobe` (вы познакомились с концепцией `kprobes` в главе 1)[4]. Вы можете присоединить функцию `hello` к этому событию, например так:

```
b.attach_kprobe(event=syscall, fn_name="hello")
```

На этом этапе программа eBPF загружается в ядро и привязывается к событию, поэтому программа будет активироваться всякий раз, когда на машине запускается новый исполняемый файл. Всё, что осталось сделать в коде Python, — это прочитать трассировку, выводимую ядром, и записать ее на экран:

```
b.trace_print()
```

Эта функция `trace_print()` будет бесконечно повторяться (пока вы не остановите программу, например, с помощью Ctrl+C), отображая любую трассировку.

Рисунок 2-2 иллюстрирует этот код. Программа Python компилирует код C, загружает его в ядро и прикрепляет к системному вызову `execve kprobe`. Всякий раз, когда какое-либо приложение на этой (виртуальной) машине вызывает `execve()`, оно активирует программу eBPF `hello()`, которая записывает строку трассировки в определенный псевдофайл. (Я расскажу где находится этот псевдофайл позже в этой главе.) Программа Python считывает это сообщение трассировки из псевдофайла и отображает его пользователю.

4 Существует более производительный способ присоединения программ eBPF к функциям, доступный начиная с версии ядра 5.5, который использует `fentry` (и соответствующий `fexit` вместо `kretprobe` для выхода из функции). Я расскажу об этом позже в книге, а сейчас я использую `kprobe` чтобы сделать пример в этой главе максимально простым.

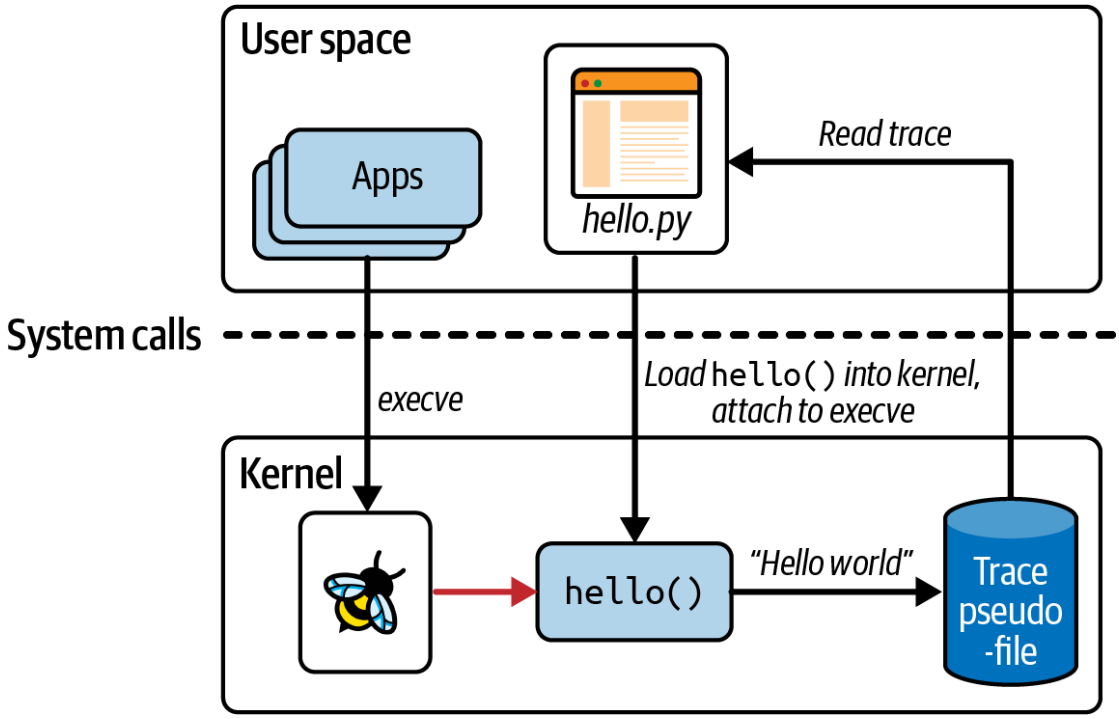


*Рисунок 2-2. «Hello World» в действии*

## Выполнение «Hello World»

Запустите эту программу, и, в зависимости от того что происходит на (виртуальной) машине, которую вы используете, вы можете увидеть, что трассировка генерируется сразу же, потому что другие процессы могут выполнять программы[5] с системным вызовом `execve`. Если вы ничего не видите, откройте второй терминал и выполните любые команды, которые вам нравятся[6], и вы увидите соответствующую трассировку, сгенерированную «Hello World»:

```
# hello.py
...
b' bash-5412 [001] .... 90432.904952: 0: bpf_trace_printk: Hello World'
```

> Поскольку eBPF такой мощный, для его использования требуются особые привилегии. Привилегии автоматически назначаются пользователю `root`, поэтому проще всего способ запуска программ eBPF — от имени пользователя `root`, возможно, с помощью `sudo`. Для ясность я не буду включать `sudo` в примеры команд в этой книге[7], но если вы когда-нибудь увидите ошибку «Операция не разрешена», первое что нужно проверить это не пытаетесь ли вы запускать программы eBPF как непривилегированный пользователь.

---

5 Я довольно часто использую VScode remote для подключения к виртуальной машине в облаке. Это запускает множество скриптов узла на виртуальной машине, которая генерирует много трассировки из этого приложения «Hello World».

6 Некоторые команды (например, `echo`) могут быть встроенными командами оболочки `bash`, которые запускаются как часть процесса оболочки, а не как новая программа. Они не вызовут событие `execve()`, и поэтому трассировка не будет сгенерирована.

7 Мы будем, как это общепринято в Linux, различать привилегированные команд (выполняемые от `root`) от не привилегированных по значку «приглашения» командного интерпретатора `bash`: `$` - это не привилегированная команда, `#` - это привилегированная команда (Прим. пер.)

Уровень привилегий `CAP_BPF` был представлен в версии ядра 5.8 и дает достаточно привилегия выполнять некоторые операции eBPF, такие как создание определенных видов карт. Однако вам, вероятно, потребуются и дополнительные возможности:

• CAP_PERFMON и CAP_BPF необходимы для загрузки трассировки программ;

• CAP_NET_ADMIN и CAP_BPF необходимы для загрузки сетевые программы;

Более подробная информация об этом содержится в сообщении блога «Введение в CAP_BPF» Милана Ландаверде (https://mdaverde.com/posts/cap-bpf/).

Как только программа `hello` eBPF загружается и прикрепляется к событию, она активизируются событиями, генерируемыми уже ранее существовавшими процессами[8]. Это должно усилить пару моментов, которые вы уже узнали в главе 1:

- Программы eBPF можно использовать для динамического изменения поведения системы. Нет необходимости перезагружать машину или перезапускать выполняющиеся процессы. Код eBPF начинает действовать, как только он прикрепляется к событию.

- Нет необходимости что-либо менять в других приложениях чтобы они были видны eBPF. Везде, где у вас есть доступ к терминалу на этой машине, если вы запустите на ней исполняемый файл, который будет использовать системный вызов `execve()`, и если у вас есть программа `hello`, прикрепленная к этому системному вызову, она будет запущена для генерации трассировки. Аналогичным образом, если у вас есть скрипт, который запускает исполняемые файлы, он также запускает программу `hello` eBPF.

Вам не нужно ничего менять в оболочке терминала, скрипте или исполняемых файлах, которые вы запускаете. Выходные данные трассировки показывают не только строку «Hello World», но также некоторую дополнительную контекстную информацию о событии, которое инициировало запуск программы `hello` eBPF. В примере вывода, показанном в начале этого раздела, процесс, выполнивший системный вызов `execve`, имел идентификатор процесса 5412 и выполнял команду `bash`. Для сообщений трассировки эта контекстная информация добавляется как часть инфраструктуры трассировки ядра (которая не является специфичной для eBPF), но, как вы увидите далее в этой главе, контекстную информацию также можно получить внутри самой программы eBPF.

Вам может быть интересно, откуда код Python знает, откуда считывать выходные данные трассировки. Ответ не очень сложный — функция-помощник `bpf_trace_printk()` в ядре всегда отправляет выходные данные в одно и то же предопределенное местоположение псевдофайла: `/sys/kernel/debug/tracing/trace_pipe`. Вы можете убедиться в этом, используя `cat` для просмотра его содержание; вам понадобятся привилегии `root` для доступа к нему.

Одно и то же расположение канала трассировки подходит для простого примера «Hello World» или для базовых целей отладки, но оно очень ограничено. В формате вывода очень мало гибкости, и он поддерживает только вывод строк, поэтому он не очень полезен для передачи структурированной информации. Но, возможно, самое главное что на (виртуальной) машине есть только одно такое место. Если бы у вас одновременно работало

8 Уже выполняющимися к тому времени в системе.

бы несколько программ eBPF, все они записывали бы вывод трассировки в один и тот же канал трассировки, что могло бы сильно запутать оператора-человека.

Есть гораздо лучший способ получить информацию из программы eBPF: использовать карту eBPF.

## Карты BPF

**Комментарий переводчика:** В оригинале для определения и описания таких структур данных, массово упоминаемых по тексту, используется наименование: `map`, `Map`. В различных языках программирования для обозначения таких ассоциативных структур используются самые разные русскоязычные термины: карты, хэш-таблицы, хэши, таблицы и даже массивы… (и в зависимости от контекста все они верные). Применительно к контексту ядра Linux и eBPF, самым адекватным, достаточным и однозначным термином будет: карта — что мы и будем использовать далее по тексту. (Кроме того, такая терминология принята и в ранее изданной книге: Дэвид Калавера, Лоренцо Фонтана, «BPF для мониторинга Linux», изд. Питер, 2020, ISBN: 978-5-4461-1624-9)

Карта — это структура данных, к которой можно получить доступ и из программы eBPF и из пользовательского пространства. Карта — одна из действительно важных функций, отличающих расширенный eBPF от его классического предшественника BPF. (Вы можете подумать что это означает, что их обычно следует называть «карты eBPF», но вы часто будете видеть именно «карты BPF». Как правило, оба термина используются взаимозаменяемо.)

Карты можно использовать для обмена данными между несколькими программами eBPF или для связи между приложением пользовательского пространства и кодом eBPF, работающим в ядре. Типичные варианты использования включают следующее:

- Код пользовательского пространства записывает информацию о конфигурации, которая будет извлечена затем программой eBPF.
- Состояние сохранения программы eBPF для последующего извлечения другой программой eBPF (или будущим запуском этой же программы).
- Программа eBPF записывает результаты или метрики в карту для извлечения приложением пользовательского пространства, которое и будет представлять результаты.

Существуют различные типы карт BPF, определенные в заголовочном файле Linux `uapi/linux/bpf.h`, и некоторая информация о них есть в документации ядра. В общем, все они представляют собой хранилища пар ключей и значений, и в этой главе вы увидите примеры карт для хэш-таблиц, профилировщика и кольцевых буферов, а также массивов программ eBPF.

Некоторые типы карт определены как массивы, которые всегда имеют 4-байтовый индекс в качестве типа ключа; другие карты представляют собой хеш-таблицы, которые могут использовать некоторый произвольный тип данных в качестве ключа. Существуют типы карт оптимизированные для конкретных типов операций, таких как очереди «первым поступил — первым обслужен» (https://docs.kernel.org/bpf/map_queue_stack.html), стеки «первым поступил — последним обслужен» (https://docs.kernel.org/bpf/map_queue_stack.html), хранилище данных наименее недавно использованных (https://docs.kernel.org/next/bpf/map_lpm_trie.html), сопоставление самого длинного префикса (https://docs.kernel.org/next/bpf/map_lpm_trie.html) и фильтры Блума

(https://docs.kernel.org/bpf/map_bloom_filter.html — вероятностная структура данных, предназначенная для получения очень быстрых результатов о том, существует ли элемент).

Некоторые типы карт eBPF содержат информацию об определенных типах объектов. Например, `sockmaps` (https://lwn.net/Articles/731133/) и `devmaps` (https://docs.kernel.org/bpf/map_devmap.html) содержат информацию о сокетах и сетевых устройствах и используются сетевыми программами eBPF для перенаправления трафика. Карта массива программ хранит набор индексированных программ eBPF, и (как вы увидите далее в этой главе) она используется для реализации хвостовых вызовов, когда одна программа может вызывать другую. Существует даже тип map-of-maps (https://docs.kernel.org/bpf/map_of_maps.html) для поддержки хранения информации о самих картах.

Некоторые типы карт имеют варианты для каждого ЦП, то есть ядро использует разные блоки памяти для версии этой карты для каждого ядра ЦП. Это может вызвать у вас вопросы о проблемах параллелизма для карт, которые не относятся к процессору, когда несколько ядер ЦП могут одновременно обращаться к одной и той же карте. Поддержка спин-блокировки для (некоторых видов) карт была добавлена в ядро версии 5.1, и мы вернемся к этой теме в главе 5.

В следующем примере (`chapter2/hello-map.py` в репозитории GitHub) показаны некоторые основные операции с использованием карты типа хэш-таблицы. Пример также демонстрирует некоторые удобные абстракции BCC, упрощающие использование карт.

## Карты типа хеш-таблиц

Как и в предыдущем примере в этой главе, эта программа eBPF будет присоединена к `kprobe` при входе в системный вызов `execve`. Она собирается заполнить хэш-таблицу парами ключ-значение, где ключ — это идентификатор пользователя, а значение — счетчик количества вызовов `execve` процессом, работающим под этим идентификатором пользователя. На практике этот пример покажет, сколько раз каждый пользователь запускал программы.

Во-первых, давайте посмотрим на код C для самой программы eBPF:

```
BPF_HASH(counter_table);
int hello(void *ctx) {
   u64 uid;
   u64 counter = 0;
   u64 *p;
   uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;
   p = counter_table.lookup(&uid);
   if (p != 0) {
      counter = *p;
   }
   counter++;
   counter_table.update(&uid, &counter);
   return 0;
}
```

1). `BPF_HASH()` — это макрос BCC, определяющий карту как хеш-таблицу.

2). `bpf_get_current_uid_gid()` — это функция-помощник, используемая для получения идентификатора пользователя, запускающего процесс, вызвавший это событие `kprobe`. Идентификатор пользователя хранится в младших 32 битах возвращаемого 64-битного значения. (Старшие 32 бита содержат идентификатор группы, но эта часть замаскирована.)

3). Находится запись в хеш-таблице с ключом, соответствующим идентификатору пользователя. Возвращается указатель на соответствующее значение в хеш-таблице.

4). Если для этого идентификатора пользователя есть запись, устанавливается переменная счетчика в текущее значение в хеш-таблице (на которое указывает p). Если же в хеш-таблице ещё нет записи для этого идентификатора пользователя, указатель будет равен 0, а значение счетчика останется равным 0.

5). Каким бы ни было текущее значение счетчика, оно увеличивается на единицу.

6). Обновляется хеш-таблица новым значением счетчика для этого идентификатора пользователя. Внимательно посмотрите на строки кода, которые обращаются к хеш-таблице:

```
p = counter_table.lookup(&uid);
```

И позже:

```
counter_table.update(&uid, &counter);
```

Если вы думаете: «Это неправильный код C!» вы абсолютно правы. C не поддерживает определение методов для подобных структур[9]. Это отличный пример, когда BCC-версия C очень слабо похожа на C-язык, который BCC переписывает перед отправкой кода компилятору. BCC предлагает несколько удобных ярлыков и макросов, которые он преобразует в «правильный» C.

Как и в предыдущем примере, код C определяется как строка с именем `program`. Программа компилируется, загружается в ядро и подключается к `execve kprobe` точно так же, как и в предыдущем примере «Hello World»:

```
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")
```

На этот раз требуется немного больше работы со стороны Python чтобы прочитать информацию из хеш-таблицы:

```
while True:
   sleep(2)
   s = ""
   for k,v in b["counter_table"].items():
      s += f"ID {k.value}: {v.value}\t"
   print(s)
```

1). Эта часть кода бесконечно циклится, в поиске выходных данных для отображения каждые две секунды.

2). BCC автоматически создает объект Python для представления хеш-таблицы. Такой код перебирает любые значения и выводит их на экран.

---

9 C++ да, но не C.

Когда вы запустите этот пример, вам понадобится второе окно терминала, в котором вы сможете запускать некоторые команды. Вот некоторый пример вывода, который я получила, аннотированный справа командами, которые я запускала в другом терминале:

```
Terminal 1                          Terminal 2
# ./hello-map.py
                                    [blank line(s) until I run something]
ID 501: 1                           ls
ID 501: 1
ID 501: 2                           ls
ID 501: 3          ID 0: 1          sudo ls
ID 501: 4          ID 0: 1          ls
ID 501: 4          ID 0: 1
ID 501: 5          ID 0: 2          sudo ls
```

Этот пример генерирует строку вывода каждые две секунды, независимо от того, произошло что-то или нет. В конце этого вывода хеш-таблица содержит две записи:

- `key=501, value=5`
- `key=0, value=2`

Во втором терминале у меня есть идентификатор пользователя 501. Запуск команды `ls` с этим идентификатором пользователя увеличивает счетчик `execve`. Когда я запускаю `sudo ls`, это приводит к двум вызовам `execve`: один — выполнение `sudo` под идентификатором пользователя 501; другой — выполнение `ls` под идентификатором пользователя `root`, равным 0.

В этом примере я использовала хеш-таблицу для передачи данных из программы eBPF в пространство пользователя. (Я могла бы также использовать здесь карту типа массива, так как ключ был целым числом; а хеш-таблицы позволяют использовать в качестве ключа произвольный тип.) Код пользовательского пространства должен регулярно опрашивать карту. Ядро Linux уже поддерживает подсистему профилирования производительности для отправки данных из ядра в пространство пользователя, а eBPF включает поддержку использования как буферов профилировщика Perf, так и их преемников, кольцевых буферов самого BPF. Давайте взглянем.

**Комментарий переводчика:** Здесь нужно сделать обстоятельное пояснение для понимания всего дальнейшего изложения… В коде ядра Linux с некоторых пор присутствует профилировщик, известный как `Perf` (в общем, это «производительность», но здесь это имя собственное: `Perf`, см. «perf: Linux profiling with performance counters» — https://perf.wiki.kernel.org/index.php/Main_Page, `man perf`[10]). Профилировщик `Perf` использует два буфера для записи различных типов событий. Вспомогательный буфер используется под основным кольцевым буфером (кольцевой буфер содержит в себе этот вспомогательный буфер) для хранения различных видов информации о наблюдаемых событиях. Основной буфер в публикациях именуют: perf-буфер. Согласно документации, размер perf-буфера определяется в числе страниц (виртуальной памяти). Он определяется как 1 + 2 ^ n страниц, где одна страница требуется для хранения метаданных о самом кольцевом буфере, который использует `Perf`. Когда вы пытаетесь записывать события с помощью

10 И ещё более 30 сопутствующих `man` страниц по состоянию на 2023 год: perf-stat(1), perf-top(1), perf-record(1), perf-report(1), perf-list(1), perf-annotate(1),perf-archive(1), perf-bench(1), perf-buildid-cache(1), perf-buildid-list(1), perf-c2c(1), perf-config(1), perf-data(1), perf-diff(1), perf-evlist(1), perf-ftrace(1), perf-help(1), perf-inject(1), perf-intel-pt(1), perf-kallsyms(1), perf-kmem(1), perf-kvm(1), perf-lock(1), perf-mem(1), perf-probe(1), perf-sched(1), perf-script(1), perf-test(1), perf-trace(1), perf-version(1)

команды `perf record ...` (см. `man perf-record`), у вас будет доступна опция `-m`, позволяющая указать количество страниц, на которые будет увеличен/уменьшен размер кольцевого буфера `Perf`.

Везде сквозь книгу вспоминается буфер `perf`, но в русскоязычных публикациях нет однозначно устоявшегося термина для этого нового понятия, поэтому в переводе будет использоваться произвольный (дословный) термин буфер Perf.

## Карты в буфере Perf и кольцевом буфере BPF

В этом разделе я собираюсь описать более сложную версию «Hello World», в которой используются возможности `BPF_PERF_OUTPUT` в BCC, что позволит вам записывать данные, в зависимости от вашего конкретного выбора, в карту буфера Perf.

> Существует и более новая конструкция, называемая «кольцевыми буферами BPF», которая теперь обычно предпочтительнее, чем BPF буферы Perf, но это если у вас используется ядро версии 5.8 или выше. Андрей Накрыко рассуждает о их разнице в своём блоге о кольцевом буфере BPF: https://nakryiko.com/posts/bpf-ringbuf/. Вы увидите пример использования BPF_RINGBUF_OUTPUT в BCC позже, в главе 4.

Кольцевые буферы ни в коем случае не уникальны для eBPF, но я объясню их на тот случай, если вы еще не сталкивались с ними. Вы можете думать о кольцевом буфере как о части памяти, логически организованной в виде кольца, с отдельными указателями «записи» и «чтения». Данные произвольной длины записываются туда, куда указывает указатель записи, при этом информация о длине включается в заголовок для этих данных. Указатель записи перемещается после окончания в конец данных и готов к следующей операции записи.

Аналогично, для операции чтения данные считываются из того места, на которое указывает указатель чтения, с использованием заголовка данных для определения объема данных для чтения. Указатель чтения перемещается в том же направлении что и указатель записи, и таким образом он указывает на следующую доступную для чтения часть данных. Это показано на рисунке 2-3, где показан кольцевой буфер с тремя элементами разной длины, доступными для чтения.

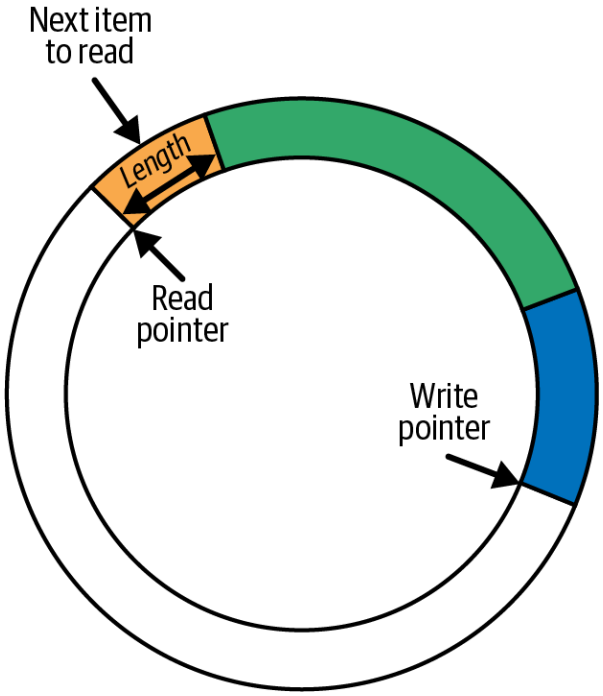


*Рисунок 2-3. Кольцевой буфер*

Если указатель чтения догоняет указатель записи, это просто означает что данных для чтения больше нет. Если же операция записи приведет к тому что указатель записи будет опережать указатель чтения, то данные не будут записаны, а счетчик сбрасывания будет увеличен. Операции чтения включают в результат счетчик сбрасывания чтобы указать, были ли данные потеряны с момента последнего успешного чтения.

Если бы операции чтения и записи происходили с одинаковой скоростью без каких-либо вариаций и всегда содержали бы один и тот же объем данных, вы могли бы, по крайней мере теоретически, обойтись кольцевым буфером, достаточно большим для размещения порции данных такого размера. Но в большинстве приложений время между чтением, записью, или и тем и другим, будет несколько различаться, поэтому размер буфера необходимо настроить с учетом этого.

Исходный код для этого примера вы найдете в `Chapter2/hello-buffer.py` в репозитории Learning eBPF GitHub. Как и в первом примере «Hello World», который вы видели в начале этой главы, эта версия будет выводить строку «Hello World» на экран каждый раз, когда используется системный вызов `execve()`. Он также будет искать идентификатор процесса и имя команды, которая выполняет каждый вызов `execve()`, так что вы получите вывод, аналогичный первому примеру. Это дает мне возможность показать вам еще пару примеров функций-помощников BPF.

Вот программа eBPF, которая будет загружена в ядро:

```
BPF_PERF_OUTPUT(output);

struct data_t {
   int pid;
   int uid;
   char command[16];
   char message[12];
```

```
};

int hello(void *ctx) {
   struct data_t data = {};
   char message[12] = "Hello World";

   data.pid = bpf_get_current_pid_tgid() >> 32;
   data.uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;

   bpf_get_current_comm(&data.command, sizeof(data.command));
   bpf_probe_read_kernel(&data.message, sizeof(data.message), message);

   output.perf_submit(ctx, &data, sizeof(data));
   return 0;
}
```

1). BCC определяет макрос BPF_PERF_OUTPUT для создания карты, которая будет использоваться для передачи сообщений из ядра в пространство пользователя. Я назвала эту карту `output`.

2). При каждом запуске `hello()` код будет записывать данные, необходимые для `struct data_t`. Это определение той структуры, которая содержит поля для идентификатора процесса, имени выполняемой в данный момент команды и текстового сообщения.

3). `data` — это локальная переменная, которая содержит структуру данных для отправки, а `message` содержит строку «Hello World».

4). `bpf_get_current_pid_tgid()` — это функция-помощник, которая получает идентификатор процесса, вызвавшего активацию этой программы eBPF. Она возвращает 64-битное значение с идентификатором процесса в старших 32 битах[11].

5). `bpf_get_current_uid_gid()` — это функция-помощник, которую вы уже видели в предыдущем примере для получения идентификатора пользователя.

6). Точно так же `bpf_get_current_comm()` — это функция-помощник для получения имени исполняемого файла (или «команды»), запущенного в процессе, который выполнил системный вызов `execve`. Это строка, а не числовое значение, как идентификаторы процесса и пользователя, и в C вы не можете просто присвоить строку с помощью оператора присвоения. Вы должны передать адрес поля, куда должна быть записана эта строка, `&data.command`, в качестве аргумента этой функции-помощника.

7). В этом примере сообщение «Hello World» каждый раз неизменно. `bpf_probe_read_kernel()` копирует его в нужное место в структуре данных.

8). На этом этапе структура данных заполняется идентификатором процесса, именем команды и сообщением. Вызов `output.perf_submit()` помещает эти данные в карту.

---

11 Младшие 32 бита — это идентификатор группы потоков. Для однопоточного процесса это ровно то же самое что и идентификатор процесса, но дополнительным потокам процесса будут присвоены другие идентификаторы. В документации к библиотеке GNU C есть хорошее описание различий между идентификаторами процессов и групп потоков.

Ровно как и в первом примере «Hello World», эта программа на C помещается в качестве строки с именем `program` в код Python. А далее следует остальная часть кода Python:

```
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")

def print_event(cpu, data, size):
   data = b["output"].event(data)
   print(f"{data.pid} {data.uid} {data.command.decode()} " + \
         f"{data.message.decode()}")

b["output"].open_perf_buffer(print_event)
while True:
   b.perf_buffer_poll()
```

1). Строки, которые компилируют код C, загружают его в ядро и присоединяют к событию системного вызова (`BPF(text=program)`), не изменились по сравнению с версией «Hello World», которую вы видели ранее.

2). `print_event` — это функция обратного вызова, которая выводит на экран строку данных. BCC делает всю грязную работу, так что я могу ссылаться на карту просто как `b["output"]` и получать данные с нее, используя `b["output"].event()`.

3). `b["output"].open_perf_buffer()` открывает кольцевой буфер Perf. Функция принимает `print_event` в качестве аргумента, чтобы определить что это функция обратного вызова, которая будет использоваться всякий раз, когда есть данные для чтения из буфера.

4). Теперь программа будет бесконечно зацикливаться[12], опрашивая кольцевой буфер Perf. Если есть какие-либо данные, будет вызвано событие `print_event`.

Запуск этого кода дает нам вывод очень похожий на исходный «Hello World»:

```
$ sudo ./hello-buffer.py
11654 node Hello World
11655 sh Hello World
...
```

Как и раньше, вам может понадобиться открыть второй терминал на той же (виртуальной) машине и запускать там некоторые команды чтобы вызвать соответствующий вывод.

Большая разница между этим и исходным примером «Hello World» заключается в том, что вместо использования одного центрального канала трассировки данные теперь передаются через карту кольцевого буфера, производя вывод, который был создана этой программой для собственного использования, так как показано на рис. 2-4.

12 Это всего лишь пример кода, так что я не беспокоюсь об очистке прерывания клавиатуры или любых других тонкостях!

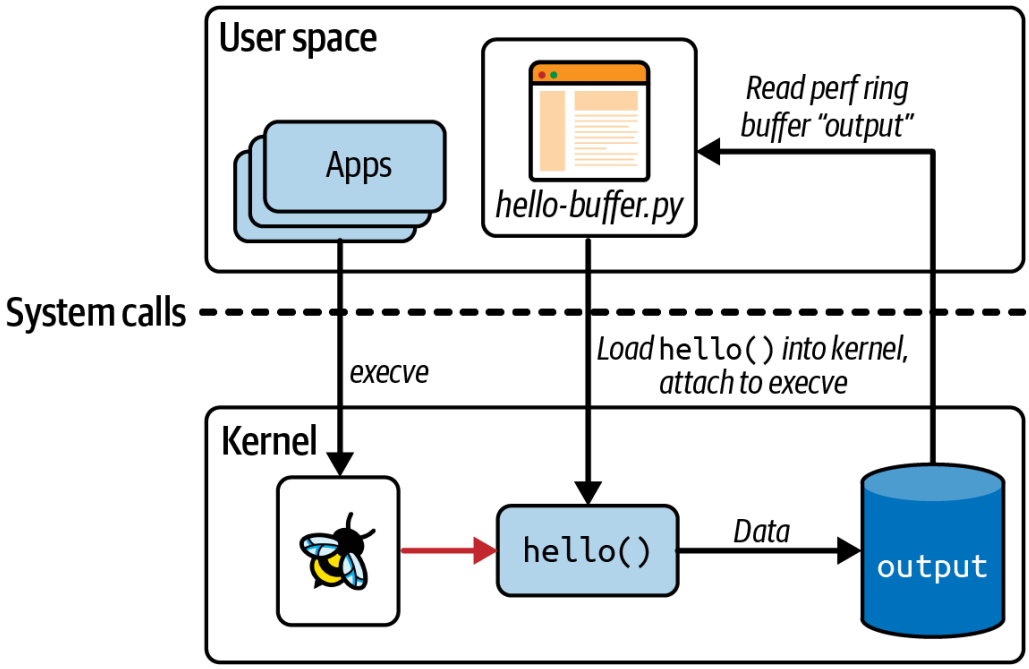


*Рисунок 2-4. Использование кольцевого буфера Perf для передачи данных из ядра в пространство пользователя*

Вы можете убедиться, что информация не поступает в канал трассировки, используя команду `cat /sys/kernel/debug/tracing/trace_pipe`.

Помимо демонстрации использования карты кольцевого буфера, в этом примере показаны некоторые функции-помощники eBPF для получения контекстной информации о событии, которое инициировало активацию программы eBPF. Здесь вы видели, как функции-помощники получают идентификатор пользователя, идентификатор процесса и имя текущей команды. Как вы увидите в главе 7, набор доступной контекстной информации и набор действующих функций-помощников, которые можно использовать для ее извлечения, зависят от типа программы и того события, которое ее вызвало.

Тот факт, что контекстная информация, подобная этой, доступна для кода eBPF, делает его таким ценным для наблюдения. Всякий раз, когда происходит событие, программа eBPF может сообщить не только о том, что событие произошло, но соответствующую информацию о том, что произошло такого что вызвало это событие. Это, сверх того, высокопроизводительно, так как вся эта информация может быть собрана в ядре без необходимости синхронного переключение контекста в пространство пользователя.

В этой книге вы увидите дальше и другие примеры, где функции-помощники eBPF используются для сбора других контекстных данных, а также примеры, когда программы eBPF изменяют контекстные данные или даже полностью блокируют события.

## Вызовы функций

Вы видели, что программы eBPF могут вызывать функции-помощники, предоставляемые ядром, но что, если вы хотите разделить код, который вы пишете, на функции? Как правило, в разработке программного обеспечения считается хорошей практикой[13] включать общий код в функцию, которую можно вызывать из нескольких мест, а не дублировать одни и те же строки снова и снова. Но в ранних реализациях программам eBPF не разрешалось вызывать

13 Этот принцип часто называют «DRY» («Don’t Repeat Yourself», «Не повторяйся»), популяризированный The Pragmatic Programmer (https://media.pragprog.com/titles/tpp20/dry.pdf).

функции, отличные от функций-помощников. Чтобы обойти это, программисты дают компилятору указание «всегда встраивать» (`inline`) свои функции, например:

```
static __always_inline void my_function(void *ctx, int val)
```

Как правило, функция в исходном коде приводит к тому что компилятор порождает инструкцию перехода, которая заставляет выполнение перейти к набору инструкций составляющих вызываемую функцию (а затем снова вернуться назад, когда эта функция завершится).

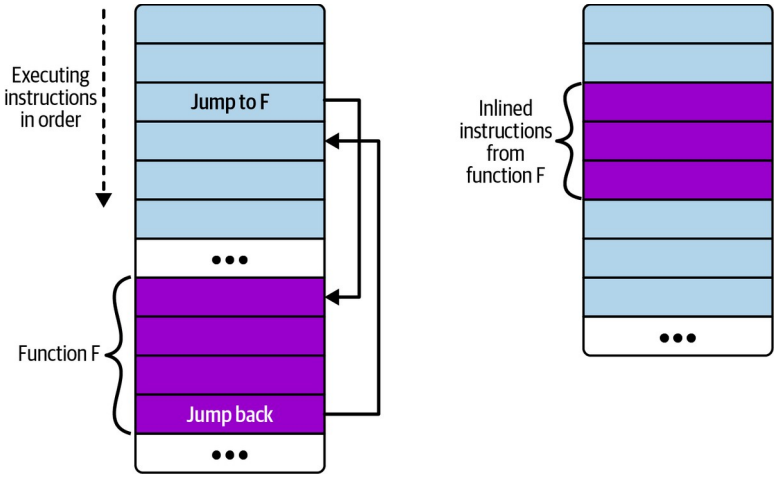


*Рисунок 2-5. Модель невстроенных и встроенных функциональных инструкций*

Это показано на левой стороне рисунка 2-5. Справа показано что происходит когда функция встроена: инструкции перехода нет; вместо этого копия инструкций функции создается непосредственно внутри тела вызывающей функции.

Если встроенная функция вызывается из нескольких мест, это приводит к множеству копий инструкций этой функции в скомпилированном исполняемом файле. (Иногда компилятор может выбрать встроенную функцию в целях оптимизации, и это одна из причин, по которой вы не сможете подключить `kprobe` к определенным функциям ядра. Я вернусь к этому в главе 7.)

Начиная с ядра Linux 4.16 и LLVM 6.0 ограничение, требующее встраивания функций, было снято, чтобы программисты eBPF могли писать вызовы функций более естественным образом. Однако эта функция, называемая «вызовы функций BPF to BPF» или «подпрограммы BPF», в настоящее время не поддерживается инфраструктурой BCC, поэтому давайте вернемся к ней в следующей главе. (Конечно, вы можете продолжать использовать функции с BCC если они встроены.)

Помимо этого в eBPF есть еще один механизм декомпозиции сложной функциональности на более мелкие части: хвостовые вызовы.

## Хвостовые вызовы

Как описано на https://ebpf.io/what-is-ebpf/#tail--function-calls, «хвостовые вызовы могут вызывать и выполнять другую программу eBPF и заменять контекст выполнения, аналогично тому, как системный вызов `execve()` работает для обычных процессов». Другими словами, выполнение не возвращается к вызывающей стороне после завершения хвостового вызова.

> Хвостовые вызовы (https://www.baeldung.com/cs/tail-vs-non-tail-recursion) ни в коем случае не являются исключительными для программирования eBPF. Общая мотивация хвостовых вызовов заключается в том чтобы избежать повторного добавления кадров в стек, поскольку функция вызывается рекурсивно, что в конечном итоге может привести к ошибкам переполнения стека. Если вы можете

организовать свой код так, чтобы рекурсивная функция вызывалась в последнюю очередь, фрейм стека, связанный с вызывающей функцией, на самом деле не делает ничего полезного. Хвостовые вызовы позволяют вызывать ряд функций без увеличения стека[14]. Это особенно полезно в eBPF, где размер стека ограничен 512 байтами (https://elixir.bootlin.com/linux/v5.19.17/source/include/linux/filter.h#L86).

Хвостовые вызовы выполняются с помощью функции-помощника `bpf_tail_call()`, которая имеет следующую сигнатуру:

```
long bpf_tail_call(void *ctx, struct bpf_map *prog_array_map, u32 index)
```

Три аргумента этой функции имеют следующий смысл:

- `ctx` — позволяет передавать контекст от вызывающей программы eBPF вызываемой программе.
- `prog_array_map` — карта eBPF типа `BPF_MAP_TYPE_PROG_ARRAY`, которая содержит набор файловых дескрипторов, идентифицирующих набор программ eBPF.
- `index` — указывает, какую из этого набора программ eBPF следует вызывать.

Эта функции-помощник несколько необычна тем, что в случае успеха она никогда не возвращается. Выполняемая в данный момент программа eBPF заменяется в стеке вызываемой программой. Функция-помощник может выйти из строя, например, если указанная программа не существует в карте, и в этом случае вызывающая программа продолжит выполняться.

Код пользовательского пространства должен загрузить все программы eBPF в ядро (как обычно), а также настроить карту массива программ.

Давайте посмотрим на простой пример, написанный на Python с использованием BCC; вы найдете код в репозитории GitHub как `Chapter2/hello-tail.py`. Основная программа eBPF привязана к точке трассировки в общей точке входа для всех системных вызовов. Эта программа использует хвостовые вызовы для отслеживания конкретных сообщений для определенных кодов операций системных вызовов. Если для заданного кода операции нет хвостового вызова, программа отслеживает общее сообщение.

Если вы используете инфраструктуру BCC, для выполнения хвостового вызова вы можете использовать строку немного более простой формы:

```
prog_array_map.call(ctx, index)
```

Прежде чем передать такой код на этап компиляции, BCC перепишет предыдущую строку следующим образом:

```
bpf_tail_call(ctx, prog_array_map, index)
```

Вот исходный код программы eBPF и ее хвостовые вызовы:

```
BPF_PROG_ARRAY(syscall, 300);

int hello(struct bpf_raw_tracepoint_args *ctx) {
   int opcode = ctx->args[1];
   syscall.call(ctx, opcode);
```

14 В данном описании идёт отсылка к тому, что хорошо известно в прикладном программировании как «хвостовая рекурсия» как способ написания рекурсивных функций (Прим. пер.)

```
    bpf_trace_printk("Another syscall: %d", opcode);
    return 0;
}

int hello_execve(void *ctx) {
    bpf_trace_printk("Executing a program");
    return 0;
}

int hello_timer(struct bpf_raw_tracepoint_args *ctx) {
    if (ctx->args[1] == 222) {
        bpf_trace_printk("Creating a timer");
    } else if (ctx->args[1] == 226) {
        bpf_trace_printk("Deleting a timer");
    } else {
        bpf_trace_printk("Some other timer operation");
    }
    return 0;
}

int ignore_opcode(void *ctx) {
    return 0;
}
```

1). BCC предоставляет макрос `BPF_PROG_ARRAY` для простого определения карт типа `BPF_MAP_TYPE_PROG_ARRAY`. Я вызвала системный вызов карты и разрешила 300 записей[15], чего будет достаточно для этого примера.

2). В коде пользовательского пространства, который вы вскоре увидите, я собираюсь присоединить эту программу eBPF к необработанной точке трассировки `sys_enter`, которая срабатывает всякий раз, когда выполняется любой системный вызов. Контекст, передаваемый программе eBPF, прикрепленной к необработанной точке трассировки, принимает форму такой структуры: `bpf_raw_tracepoint_args`.

3). В случае `sys_enter` необработанные аргументы точки трассировки включают код операции `opcode`, определяющий, какой именно системный вызов сейчас выполняется.

4). Здесь мы делаем хвостовой вызов записи в массиве программы, ключ которой (`opcode`) соответствует коду операции. Эта строка кода будет переписана BCC для вызова функции-помощника `bpf_tail_call()` перед передачей исходного кода компилятору.

5). Если хвостовой вызов выше завершится успешно, то следующая строка, отслеживающая значение кода операции, никогда не сработает. Я использовала это чтобы предоставить строку трассировки по умолчанию для тех `opcode`, для которых нет записи программы в карте.

---

15 В Linux около 300 системных вызовов, и, поскольку я не использую в этом примере недавно добавленные системные вызовы, этого достаточно.

6). `hello_execve()` — это следующая программа, которая будет загружена в карту массива программ системных вызовов, и будет выполняться как хвостовой вызов, когда код операции `opcode` указывает, что это системный вызов `execve()`. Он просто сгенерирует строку трассировки, чтобы сообщить пользователю что выполняется новая программа.

7). `hello_timer()` — это еще одна программа, которая будет загружена в массив программ системных вызовов. В этом случае на него будет ссылаться более чем одна запись в программном массиве.

8). `ignore_opcode()` — это программа хвостового вызова, которая вообще ничего не делает. Я буду использовать это для системных вызовов, когда я вообще не хочу чтобы генерировалась какая-либо трассировка.

А теперь давайте посмотрим на код пользовательского пространства, который загружает и управляет этим набором программ eBPF:

```
b = BPF(text=program)
b.attach_raw_tracepoint(tp="sys_enter", fn_name="hello")

ignore_fn = b.load_func("ignore_opcode", BPF.RAW_TRACEPOINT)
exec_fn = b.load_func("hello_exec", BPF.RAW_TRACEPOINT)
timer_fn = b.load_func("hello_timer", BPF.RAW_TRACEPOINT)

prog_array = b.get_table("syscall")
prog_array[ct.c_int(59)] = ct.c_int(exec_fn.fd)
prog_array[ct.c_int(222)] = ct.c_int(timer_fn.fd)
prog_array[ct.c_int(223)] = ct.c_int(timer_fn.fd)
prog_array[ct.c_int(224)] = ct.c_int(timer_fn.fd)
prog_array[ct.c_int(225)] = ct.c_int(timer_fn.fd)
prog_array[ct.c_int(226)] = ct.c_int(timer_fn.fd)

# Ignore some syscalls that come up a lot
prog_array[ct.c_int(21)] = ct.c_int(ignore_fn.fd)
prog_array[ct.c_int(22)] = ct.c_int(ignore_fn.fd)
prog_array[ct.c_int(25)] = ct.c_int(ignore_fn.fd)
...

b.trace_print()
```

1). Вместо присоединения к `kprobe`, как вы видели ранее, на этот раз код пользовательского пространства прикрепляет основную программу eBPF к точке трассировки `sys_enter` вызовом `attach_raw_tracepoint`.

2). Далее последовательность вызовов `b.load_func()` возвращают дескрипторы файлов для каждой программы хвостового вызова. Обратите внимание, что хвостовые вызовы должны иметь тот же тип программы, что и их родитель — в данном случае это `BPF.RAW_TRACEPOINT`. Кроме того, следует отметить что каждая программа хвостового вызова является самостоятельной программой eBPF.

3). Код пространства пользователя создает записи в карте системных вызовов `prog_array`. Карта не обязана быть полностью заполнена для каждого возможного кода операции; если нет записи для определенного кода операции, это просто означает что хвостовой вызов не будет выполнен. Кроме того, совершенно нормально иметь несколько разных записей, указывающих на одну и ту же программу eBPF. В этом случае я хочу чтобы хвостовой вызов `hello_timer()` выполнялся для любого набора системных вызовов, связанных с таймером.

4). Некоторые системные вызовы (21, 22, 25 ...) выполняются системой настолько часто, что строка трассировки для каждого из них загромождала бы вывод трассировки до нечитаемости. Я использовала хвостовой вызов `ignore_opcode()` для нескольких таких системных вызовов.

5). Последний оператор предписывает делать вывод трассировки на экран до тех пор, пока пользователь не закроет программу.

Запуск этой программы генерирует вывод трассировки для каждого системного вызова, выполняемого на (виртуальной) машине, если только в коде операции нет записи, которая связывает его с хвостовым вызовом `ignore_opcode()`. Вот пример вывода команды `ls` в другом терминале (некоторые детали опущены для удобочитаемости):

```
# ./hello-tail.py
b' hello-tail.py-2767      ... Another syscall: 62'
b' hello-tail.py-2767      ... Another syscall: 62'
...
b'           bash-2626     ... Executing a program'
b'           bash-2626     ... Another syscall: 220'
...
b'           <...>-2774    ... Creating a timer'
b'           <...>-2774    ... Another syscall: 48'
b'           <...>-2774    ... Deleting a timer'
...
b'              ls-2774    ... Another syscall: 61'
b'              ls-2774    ... Another syscall: 61'
...
```

Конкретные выполняемые системные вызовы не имеют отношения к делу, но вы можете видеть, что различные хвостовые вызовы вызываются и генерируют свои сообщения трассировки. Вы также можете увидеть сообщения по умолчанию «Another syscall» для кодов операций которые не имеют записи в карте программ хвостового вызова.

> Ознакомьтесь с записью в блоге Пола Шеньона (Paul Chaignon) о стоимости хвостовых вызовов BPF для различных версий ядра (https://pchaigno.github.io/ebpf/2021/03/22/cost-bpf-tail-calls.html).

Хвостовые вызовы поддерживаются в eBPF, начиная с версии ядра 4.2, но долгое время они были несовместимы с выполнением вызовов функций BPF из BPF. Это ограничение было снято в ядре 5.10[16].

16 Выполнение хвостовых вызовов из подпрограммы BPF требует поддержки JIT-компилятора, с которым вы познакомитесь в следующей главе. В версии ядра, которую я использовала для написания примеров в этой книге, такая поддержка есть только у JIT-компилятора для x86, хотя поддержка была добавлена позже и в ARM в ядре 6.0 (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?

Тот факт, что вы можете объединить до 33 хвостовых вызовов вместе, в сочетании с ограничением сложности инструкций на программу eBPF в 1 миллион инструкций, означает, что сегодняшние программисты eBPF имеют большую свободу действий для написания очень сложного кода, который полностью выполняется в ядре.

## Резюме

Я надеюсь, что показывая некоторые конкретные примеры программы eBPF, эта глава помогла вам закрепить вашу воображаемую модель кода eBPF, работающего в ядре и запускаемого событиями. Вы также увидели примеры передачи данных из ядра в пространство пользователя с помощью карт BPF.

Использование структуры BCC скрывает многие детали того, как программа создается, загружается в ядро и привязывается к событиям. В следующей главе я покажу вам другой подход к написанию «Hello World», и мы углубимся в эти скрытые детали.

## Упражнения

Вот некоторые дополнительные действия, которые вы могли бы попробовать сделать (или подумать), если хотите чуть глубже изучить «Hello World»:

1. Адаптируйте программу eBPF `hello-buffer.py` для вывода разных сообщений трассировки для нечетных и четных идентификаторов процессов.

2. Измените `hello-map.py`, так чтобы код eBPF активировался более чем одним системным вызовом. Например, `openat()` обычно вызывается для открытия файлов, а `write()` вызывается для записи данных в файл. Вы можете начать с подключения программы `hello` eBPF к нескольким системным вызовам `kprobe`. Затем попробуйте изменять версии программы `hello` eBPF для разных системных вызовов, продемонстрировав что вы можете получить доступ к одной и той же карте из нескольких разных программ.

3. Программа `hello-tail.py` eBPF является примером программы, которая подключается к необработанной точке трассировки `sys_enter`, которая срабатывает всякий раз, когда вызывается любой системный вызов. Измените `hello-map.py`, чтобы показать общее количество системных вызовов, сделанных каждым идентификатором пользователя, подключив его к той же необработанной точке трассировки `sys_enter`.

   Вот пример вывода, который я получила после внесения такого изменения:

   ```
   # ./hello-map.py
   ID 104: 6     ID 0: 225
   ID 104: 6     ID 101: 34     ID 100: 45     ID 0: 332     ID 501: 19
   ID 104: 6     ID 101: 34     ID 100: 45     ID 0: 368     ID 501: 38
   ID 104: 6     ID 101: 34     ID 100: 45     ID 0: 533     ID 501: 57
   ```

4. Макрос RAW_TRACEPOINT_PROBE, предоставляемый BCC, упрощает присоединение к необработанным точкам трассировки, указывая коду BCC

---

id=d4609a5d8c70d21b4a3f801cf896a3c16c613fe1).

пользовательского пространства автоматически прикреплять его к указанной точке трассировки. Попробуйте сделать это в `hello-tail.py`, например:

- Замените определение функции `hello()` на `RAW_TRACEPOINT_PROBE(sys_enter)`.
- Удалите явное присоединение вызова `b.attach_raw_tracepoint()` из кода Python.

  Вы должны увидеть, что BCC автоматически создает присоединение, и программа работает точно так же. Это пример множества удобных макросов, которые предоставляет BCC.

5. Вы можете дополнительно адаптировать `hello_map.py` так чтобы ключ в хеш-таблице идентифицировал конкретный системный вызов (а не конкретного пользователя). Вывод покажет, сколько раз этот системный вызов был вызван вообще во всей системе.