Profiling & Tracing WITH PERF

by Julia Evans

Что это?

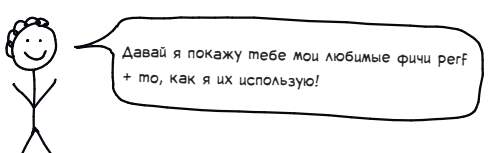
(only Linux!)

perf – это один из моих любимых инструментов дебага в Linux! Он позволяет вам:

- ★ Трассировать вызовы системы быстрее, чем strace
- ★ Очень просто профилировать ваши C, Go, C++, node.js, Rust и Java/JVM программы
- ★ Трассировать или считать почти любое событие ядра ("perf считает сколько пакетов отправляет каждая программа")

Я даже использовала его несколько раз для профилирования Ruby программ!

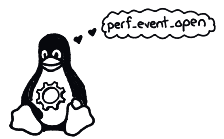
В этом выпуске я расскажу и о том, как использовать основные подкоманды perf, и о том, как устроен perf под капотом и как он работает.



Джулия Эванс
Вот Твиттер: @b0rk
А вот блог: jvns.ca

Содержание:

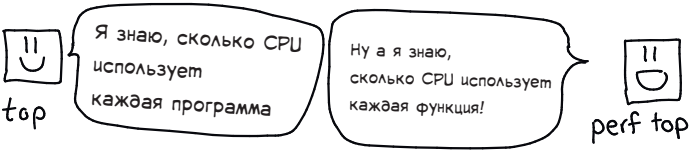
Использование perf



Как работает perf

perf top

Работу с perf я люблю начинать с perf top



Я запускаю 'perf top' на машинах,
если программа использует 100% проца
и я не понимаю, почему.

В качестве примера, давайте проведем профилирование
очень простой программы, которую я написала.
У нее есть единственная функция ('run_ofigennaya_funkciya'),
которая представляет собой бесконечную петлю.

Вот код,
который я запускаю.
Я назвала бинарник
"use_cpu".

```
void run_ofigennaya_funkciya  () {
  int x = 0;
  while (1) {
    x = x + 1;
  }
}
int main() { run_ofigennaya_funkciya  (); }
```

Итак, когда программа работаем, запускаем perf top.
Запускать надо от рута, как и любую подкоманду perf.

```
$ sudo perf top
```

Вывод perf top

Вот, что мы видим, если я запускаю perf top,
когда 'use_cpu' работает на моем ноутбуке:

```
①          ②              ③
100,00%  use_cpu      [.] run_ofigennaya_funkciya
  0,00%  [kernel]     [k] smp_call_function_single
  0,00%  [kernel]     [k] load_balance
```

① % CPU, используемый функцией

② Имя программы или библиотеки

③ Название функции/символа

Нам это подсказывает, что 100% процессорного
времени тратится на run_ofigennaya_funkciya

perf top рассказывает нам о:

- ★ Функциях программ в пользовательском пространстве
- ★ Функциях в ядре

А вот как выглядит ситуация, когда ядро использует много CPU:

```
 27.70%  [kernel]
 11.87%  libxul.so
 10.24%  [kernel]
  6.75%  [kernel]
  3.94%  [kernel]
```

Функции ядра !

```
[k] cpuidle_reflect
[.] _init
[k] _aesni_enc1
[k] end_bio_extent_writepage
[k] find_get_pages_contig
```

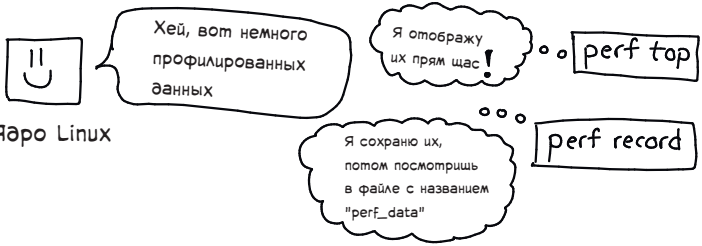
Эта функция производит шифрование ("aes"),
так как я делаю запись
в зашифрованную файловую систему

perf record

Perf top – это крутой способ быстро понять, что происходит. Но частенько я хочу исследовать ситуацию более глубоко.

perf record

Собирает ту же самую информацию, что и perf top, но при этом позволяет сохранить эти данные, чтобы их можно было проанализировать позднее. Сохраняется в текущей директории в виде файла, с названием "perf_data".



Существует три главных способа выбора процесса для профилирования с помощью perf record:

① `perf record КОМАНДА` ← Запускает КОМАНДУ и профилирует ее пока она не завершится

② `perf record PID` ← Профилирует определенный PID, пока не нажмешь ctrl

③ `perf record -a` ← Профилирует все процессы, пока не нажмешь ctrl

А вообще есть и 4-й гибридный способ. Если указать одновременно PID (или -a) и команду – perf record будет профилировать PID до тех пор, пока команда не завершится.

Вот, например так:

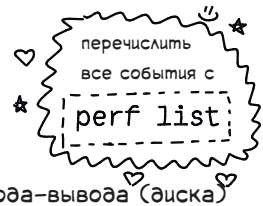
```
perf record -p 8325 sleep 5
            PID     COMMAND
```

Этот полезный трюк позволит профилировать PID 8325 в течении пяти секунд.

Собираем трассированные данные с perf record

Пока что мы собирали профилированные данные с perf: ("какая функция выполняется?).Когда perf собирает профилированную информацию, он делает выборку: он проверяет, какая функция выполняется, скажем, 100 раз в секунду.

Но perf может так же записывать множество различных типов событий. И когда он записывает события, выборки не производится. Так, если вы попросите perf записать системные вызовы, он попытается записать каждый отдельный системный вызов.



Вот немного примеров таких событий:

- системный вызов
- отправка сетевого пакета
- чтение с устройства блочного ввода–вывода (диска)
- переключение контекста/отказы страниц
- кроме того ты можешь превратить любую функцию ядра в событие! (это называется "kprobes")

К примеру, предположим у вас есть какая–то программа, имеющая исходящие сетевые подключения, и вы хотите узнать, что это за программа и зачем ей это.

Благодаря магии perf, каждый раз когда программа обращается к веб–серверу ('connect'), системный вызов будет записан. Кроме того, также будет записана трассировка стека, которая привела к этому вызову.

```
perf record -e syscalls:sys_enter_connect -ag
```

g означает «собирай трассировку стека»

Возможность взять системный вызов/отказ страницы/запись на диск и проследить его вплоть до точного места в коде, которое его вызывает – это очень сильное колдунство.

Анализируем данные perf record

Есть три способа анализа файла "perf.data", сгенерированного perf record:

perf report

Быстрый интерактивный отчет показывающий наиболее используемые функции

```
 100,00%     0,00%  use_cpu  use_cpu            [.] main
 100,00%     0,00%  use_cpu  libc-2.23.so       [.] __libc_start_main
 100,00%   100,00%  use_cpu  use_cpu            [.] run_ofigennaya_funkciya
```

100% времени занято этой функцией

perf annotate

assembly instructions!

perf annotate расскажет, какие инструкции ассемблера вашей программы занимают большую часть времени (осторожнее: одна инструкция может занять все время)

```
         :      Disassembly of section .text:
         :
         :      00000000004004d6 <run_ofigennaya_funkciya>:
         :      run_ofigennaya_funkciya():
    0.00 :        4004d6:       push   %rbp
    0.00 :        4004d7:       mov    %rsp,%rbp
    0.00 :        4004da:       movl   $0x0,-0x4(%rbp)
  100.00 :        4004e1:       addl   $0x1,-0x4(%rbp)
    0.00 :        4004e5:       jmp    4004e1 <run_ofigennaya_funkciya+0xb>
 Percent |      Source code & Disassembly of kcore for cycles:pp
```

эта add-инструкция – это место, где тратится все время

perf script

perf script выводит в виде текста все выборки, которые собирает perf. Полученные данные можно анализировать скриптами. Например flamegraph-скриптом со следующей страницы!

инструкция

символ

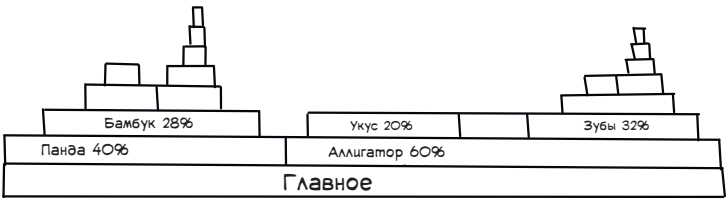
```
use_cpu 23001 19774.727477:     349732 cycles:pp:
        4e1 run_awesome_function (/home/bork/work/perf-zine/use_cpu)
        4f5 main (/home/bork/work/perf-zine/use_cpu)
      20830 __libc_start_main (/lib/x86_64-linux-gnu/libc-2.23.so)
 8fe258d4c544155 [unknown] ([unknown])
```

трассировка стека

flamegraph'ы

Flamegraph – это отличный способ визуализации профилированных данных, изобретенный и популяризированный Бренданом Греггом.

Вот как они выглядят:



Они составлены из множества (обычно тысяч) трассировок стека, отобранных с программы. График наверху означает, что 40% трассировок происходят из [main, panda] и 32% с [main, alligator, teeth]

Чтобы сгенерировать flamegraph, иди на

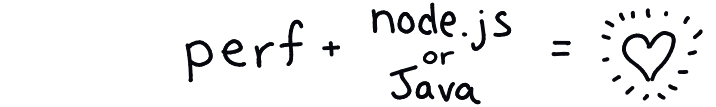
github.com/brendangregg/Flamegraph

и добавьте его в свой ПУТЬ. После того как вы это сделали, вот как сгенерировать флеймграф.

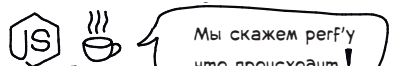
```
$ sudo perf script | stackcollapse-perf.pl
                   | flamegraph.pl > graph.svg
```

Откройте это в своем браузере!

(это тот же 'perf script', что и на прошлой странице!)

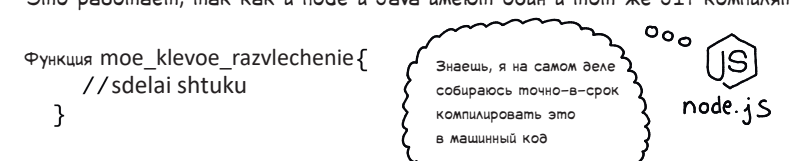


Обычно с интерпретируемыми языками как, например, node.js perf расскажет какая функция интерпретатора запущена, но сможет сделать того же для функции Javascript . Но:



"точно в срок"

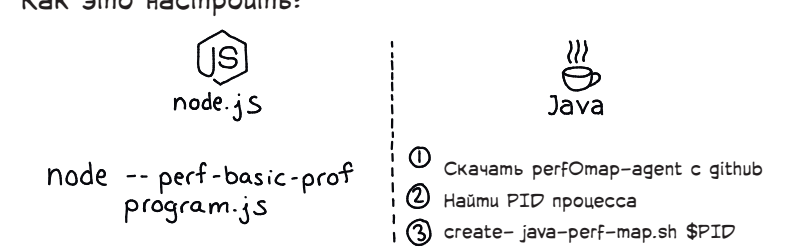
Это работает, так как и node и Java имеют один и тот же JIT компилятор



node общается с perf путем создания файла

/tmp/perf-$PID.map

Как это настроить:



Почему моя трассировка стека содержит функции ядра?

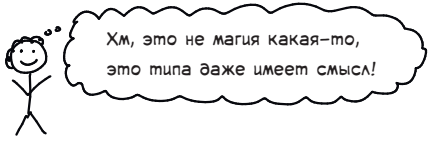
Иногда от perf можно получить трассировку стека, в которой будут смешаны функции вашей программы (например, _ _getdents 64) и функции ядра (например, btrfs_real_readdir). Это нормально!

Пример:

```
find 27968 97997.204322:     707897 cycles:pp:
            7fffc034eac7 read_extent_buffer ([kernel.kallsyms])
            7fffc032e4f7 btrfs_real_readdir ([kernel.kallsyms])
            7fff81229eb8 iterate_dir ([kernel.kallsyms])
            7fff8122a359 sys_getdents ([kernel.kallsyms])
            7fff81850fc8 entry_SYSCALL_64_fastpath ([kernel.kallsyms])
                   c88eb __getdents64 (/lib/x86_64-linux-gnu/libc-2.23.s
```

Обычно это означает что ваша программа сделала системный вызов или же наблюдался отказ страницы и это подсказывает вам, какая именно функция ядра была вызвана в качестве результата того системного вызова.

К примеру, в этом случае системный запрос 'getdent' вызвал функцию btrfs_real_readdir (потому что я использую файловую систему btfrs). Прикольненько!



Список читов perf

важные аргументы командной строки:

Какие данные собираем:
- `-F`: выбираем частоту выборки
- `-g`: записываем трассировку стека
- `-e`: выбираем события для записи

На какую программу (программы) смотрим:
- `-a`: всю систему
- `-p`: указываем PID
- `COMMAND`: запускаем эту cmd

perf top: получаем обновления!

```
# Sample CPUs at 49 Hertz, show top symbols:
perf top -F 49

# Sample CPUs, show top process names and segments:
perf top -ns comm,dso
```

} выборка

```
# Count system calls by process, refreshing every 1 second:
perf top -e raw_syscalls:sys_enter -ns comm -d 1

# Count sent network packets by process, rolling output:
stdbuf -oL perf top -e net:net_dev_xmit -ns comm | strings
```

} трассировка события

perf stat: считаем события! Счетчики CPU!

```
# CPU counter statistics for COMMAND:
perf stat COMMAND

# *Detailed* CPU counter statistics for COMMAND:
perf stat -ddd command

# Various basic CPU statistics, system wide:
perf stat -e cycles,instructions,cache-misses -a

# Count system calls for PID, until Ctrl-C:
perf stat -e 'syscalls:sys_enter_*' -p PID

# Count block device I/O events for the entire system, for 10 seconds:
perf stat -e 'block:*' -a sleep 10
```

Чтобы составить список событий – используй *perf list*

perf отчеты

```
# Show perf.data in an ncurses browser:
perf report

# Show perf.data as a text report:
perf report --stdio

# List all events from perf.data:
perf script

# Annotate assembly instructions from perf.data
# with percentages
perf annotate [--stdio]
```

позаимствовано на brendangregg.com/perf.html ,
там есть еще много отличных примеров

★ perf trace : производим трассировку системных вызовов и других событий ★

```
# Trace syscalls system-wide
perf trace
```

```
# Trace syscalls for PID
perf trace -p PID
```

★ perf record : записываем профилированные данные ★

```
# Sample CPU functions for COMMAND, at 99 Hertz:
perf record -F 99 COMMAND

# Sample CPU functions for PID, until Ctrl-C:
perf record -p PID

# Sample CPU functions for PID, for 10 seconds:
perf record -p PID sleep 10

# Sample CPU stack traces for PID, for 10 seconds:
perf record -p PID -g -- sleep 10

# Sample CPU stack traces for PID, using DWARF to unwind stack:
perf record -p PID --call-graph dwarf
```

записываются в файл perf.data

★ perf record : записываем данные трассировки ★

```
# Trace new processes, until Ctrl-C:
perf record -e sched:sched_process_exec -a

# Trace all context-switches, until Ctrl-C:
perf record -e context-switches -a

# Trace all context-switches with stack traces, for 10 seconds:
perf record -e context-switches -ag -- sleep 10

# Trace all page faults with stack traces, until Ctrl-C:
perf record -e page-faults -ag
```

записываются в файл perf.data

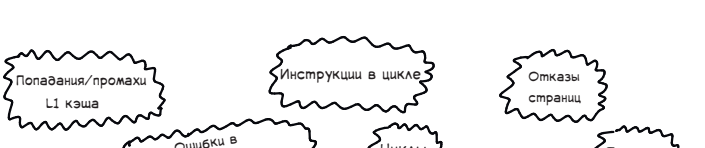
★ Добавляем новые события для трассировки ★

```
# Add a tracepoint for kernel function tcp_sendmsg():
perf probe 'tcp_sendmsg'

# Trace previously created probe:
perf record -e -a probe:tcp_sendmsg

# Add a tracepoint for myfunc() return, and include the retval as a string:
perf probe 'myfunc%return +0($retval):string'

# Trace previous probe when size > 0, and state is not TCP_ESTABLISHED(1):
perf record -e -a probe:tcp_sendmsg --filter 'size > 0 && skc_state != 1' -a

# Add a tracepoint for do_sys_open() with the filename as a string:
perf probe 'do_sys_open filename:string'
```

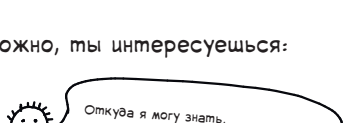
Требуются kernel debug-info

perf stat: счетчики CPU

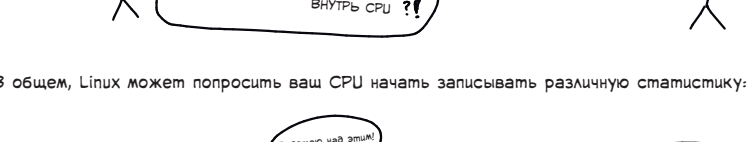
Если вы пишете высокопроизводительные программы, существует множество событий CPU и событий на уровне железа, учет которых может быть вам интересен.



Возможно, ты интересуешься:



В общем, Linux может попросить ваш CPU начать записывать различную статистику:



В качестве примера: вот кусок вывода `"perf stat -ddd ls"`

-d означает "детально"

```
$ sudo perf stat -ddd ls -R /
 Performance counter stats for 'ls -R /':
       3849.615096      task-clock (msec)         #    0.535 CPUs utilized
            26,120      context-switches          #    0.007 M/sec
               342      page-faults               #    0.089 K/sec
     8,583,744,395      cycles                    #    2.230 GHz
    10,337,612,795      instructions              #    1.20  insns per cycle
     1,987,339,660      branches                  #  516.244 M/sec
        20,738,878      branch-misses             #    1.04% of all branches
     2,883,947,626      dTLB-loads                #  749.152 M/sec

       7.192555725 seconds time elapsed
```

10 млрд. инструкций происходят быстро

Статистика предсказания переходов

perf stat : считаем любое событие

На деле, с помощью perf stat можно считать множество различных событий. Эти же события можно записывать с помощью perf record!

Вот пара примеров использования 'perf stat' на ls -R (рекурсивно составляем список файлов, поэтому производит много системных вызовов)

① Считаем переключение контекста между ядром и пользовательским пространством!

```
$ sudo perf stat -e context-switches ls -R /
 Performance counter stats for 'ls -R /':
            20,821      context-switches
```

② Считаем системные вызовы!

wildcard

```
$ sudo perf stat -e 'syscalls:sys_enter_*' ls -R / > /dev/null
     8,028      syscalls:sys_enter_newlstat
    15,167      syscalls:sys_enter_write
   254,755      syscalls:sys_enter_close
   254,777      syscalls:sys_enter_open
   509,496      syscalls:sys_enter_newfstat
   509,598      syscalls:sys_enter_getdents
```

Я запускаю их через sort -n, чтобы получить топовый список

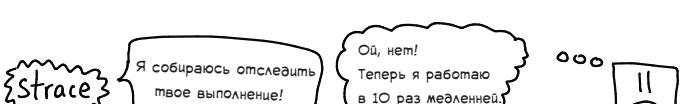
Файлы директории

perf stat производит некоторую дополнительную нагрузку. Подсчет каждого системного вызова на поиск привело к тому, что в ходе моего короткого эксперимента программа замедлилась в 6 РАЗ!!!

Я думаю, пока вы считаете только несколько различных событий (например, только 'syscalls: sys_enter_open' события), все будет хорошо. Тем не менее я не до конца понимаю возникает такая доп. нагрузка.

perf trace

strace – это отличный инструмент дебаггинга в Linux, который производит трассировку системных вызовов. Однако, у него есть одна проблема:



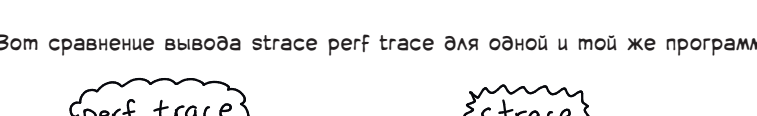
perf trace также производит трассировку системных вызовов, но производит значительно меньше дополнительной нагрузки. Его даже можно безопасно запускать в продакшне, в отличии от strace

Однако есть и два недостатка (для Linux 4.4)

1. Иногда некоторые системные вызовы упускаются (ну это своего рода преимущества, так как доп. нагрузка от этого сокращается)

2. Он не покажет строки, которые подвергались чтению/записи

Вот сравнение вывода strace perf trace для одной и той же программы



Недавно, я использовала perf trace и он сказал мне, что Docker вызывал 'stat' на

файлов. Это оказалось ОЧЕНЬ ПОЛЕЗНОЙ УЛИКОЙ, которая помогла выяснить, что Docker получает размеры контейнера, смотря на каждый файл. Я использовала perf trace, потому что не хотела иметь дело с доп. нагрузкой от strace!

как работает perf: обзор

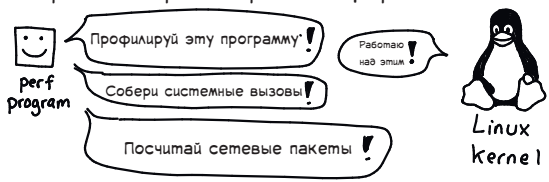
Теперь, когда мы знаем, как использовать perf, давайте посмотрим, как он работает !

Система perf разбита на две части:

1. Программа "perf" в пользовательском пространстве
2. Система в ядре Linux

Когда вы запускаете 'perf record', 'perf stat'или 'perf top', чтобы получить информацию о программе, вот что происходит:

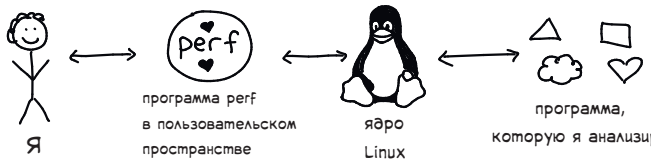
→ perf просит ядро собрать информацию



→ ядро собирает выборки/трассировки/счетчики CPU по программе, о которой спрашивал perf

→ perf отображает эту информацию вам в удобном (надеюсь) виде

Итак, вот большая картинка:



Версии ядра

perf очень плотно работает с ядром Linux. Вот, что это означает:

- нужно устанавливать именно ту версию perf, которая точно соответствует версии вашего ядра.
  На Ubuntu, можно сделать это с помощью:

```
sudo apt-get install linux-tools-$(uname -r)
```

- Фичи perf (и иногда опции командной строки) отличаются в зависимости от версии ядра

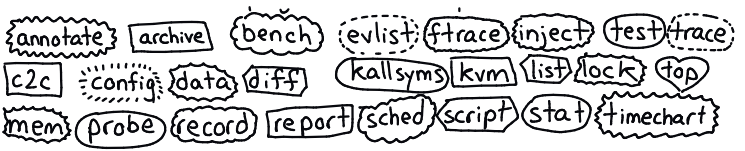
Первая версия perf была представлена в Linux 2.6

Это также означает, что в git репозитории Linux существует раздел документации о perf. Вы можете найти это на github:

github.com/torvalds/linux/tree/master/tools/perf/Documentation

Пара клёвых штук здесь:

- формат файла perf.data
- как использовать встроенный в perf интерпретатор Python для написания скриптов
- страницы с мануалами для всех подкоманд perf



Как работает профилирование с perf

Ядро Linux имеет встроенный профилировщик выборки

Я проверил, какая функция программы запускалась 50 тысяч раз и вот результаты!

Как же Linux узнает, какая функция вашей программы запущена? Ну -- ядро Linux отвечает за создание расписания.

Это означает, что у ядра всегда есть список всех процессов и адресов инструкций CPU, которые конкретный процесс использует сейчас.
Такой адрес называется указатель инструкции.

Вот как выглядит информация ядра Linux:

Команда	PID	Идентификатор потока	Указатель инструкции
python	2379	2379	0x00759d2d
bash	1229	1229	0x00123456
use_cpu	4991	4991	0xabababab
use_cpu	4991	4991	0xabab bbbb

Иногда perf не может понять как превратить адрес указателя инструкции в имя функции. Вот как это выглядит:

загадочный адрес !!

```
0.00%  nodejs           nodejs             [.] 0x0000000000759d20
0.00%  V8 WorkerThread  [kernel.kallsyms]  [k] hrtimer_active
```

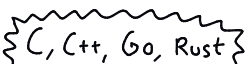
Какие языки умеет профилировать perf

Обычно perf определяет запущенные функции вашей программы следующим образом:

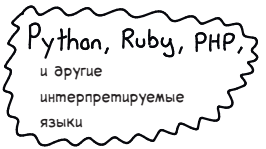
1. получает у программы адрес указателя инструкции
2. получает копию стека программы
3. разворачивает стек, чтобы найти адрес текущего вызова функции
4. использует таблицу символов программы, чтобы выяснить символ, к которому относится этот адрес

Важно понимать, что perf по умолчанию будет выдавать вам символ из таблицы символов программы. Это означает, что perf не сможет сообщить имена функций для стрипанных бинарников.

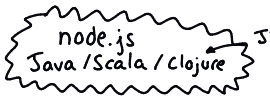
Вот как perf может помочь вам с учетом ограничений языков программирования:



perf расскажет, какая функция запущена



perf даст информацию о интерпретаторе (что тоже может быть полезно!)



может использовать альтернативный способ, чтобы найти "настоящую" функцию (как мы показывали на странице 10)

perf: под капотом

Часто бывает полезным иметь хотя бы базовое представление того, как применяются наши инструменты. Поэтому давайте взглянем на интерфейс инструмента в пользовательском пространстве ('perf'), используемый для общения с ядром Linux. В целом, вот, что происходит:

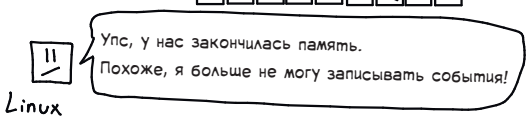
1. perf запускает системный вызов perf_event_open
2. ядро записывает "события" в кольцевой буфер пользовательского пространства
3. perf читает события из этого кольцевого буфера и отображает их вам в определенном виде

Что такое кольцевой буфер?

Для профилирования событий важно использовать ограниченный объем памяти. Поэтому ядро выделяет фиксированный объем памяти.

Каждый из этих участков предназначен для одной записи

Доступная память подходит к концу, так как новые записи появляются быстрее, чем perf может их прочитать



Так что если вы видите предупреждение от perf о том, что события не записываются – теперь вы понимаете почему.

Системный вызов perf_event_open

Чтобы начать выборку или трассировку, perf обращается к ядру Linux с помощью этого системного вызова. Вот алгоритм этого системного вызова:

```
int perf_event_open(struct perf_event_attr *attr,
                    pid_t pid, int cpu, int group_fd,
                    unsigned long flags);
```

PID & CPU на которые смотрим. Возможно значение "все"

А вот здесь находится больше всего аргументов

Не то, чтобы этот мануал был супер-полезным для ежедневного использования perf. Но! Знали ли вы, что CLI тулза perf – не единственная программа, использующая системный вызов perf_event_open?

> Проект 'bcc' – это совокупность средств для написания продвинутых инструментов профилирования с использованием eBPF github.com/iovisor/bcc

С bcc вы сможете сравнительно легко использовать perf_event_open, чтобы создавать собственные кастомные события для профилирования/трассировки! И тогда вы сможете писать код для агрегации и отображения их любым способом, который вы предпочтете.

Поищите BCC_PERF_OUTPUT в документах bcc, чтобы узнать больше.

больше материалов о perf

Спасибо, что читаете! Вот еще немного полезных ресурсов:

блог Брендана Грегга

→ brendangregg.com/perf.html ←

это мой любимый ресурс по perf.
Его блог и статьи тоже очень полезны!

Linux Weekly News
LWN.net

LWN это отличная площадка, где публикуются материалы по Linux!
Иногда там встречаются и статьи про perf

perf'у посвящено множество страниц мануалов.
Например, мануал по perf top

И самое главное: 

- → Выберите программу и попытайтесь профилировать ее!
- → Посмотрите, как ядро ведет себя под различной нагрузкой!
- → Попытайтесь записать / посчитать несколько типов perf-событий и посмотрите, что из этогов выйдет!

Понравилось?
Вот тут можно
посмотреть другие выпуски:
https://jvns.ca/zines

Перевела Команда FirstVDS.ru
https://firstvds.ru/blog/julia_evans