Selectel

Трассировка и профилирование ядра Linux: краткое введение

Андрей Емельянов

Содержание

Предисловие ... 3

Ядро Linux: основные механизмы трассировки ... 4

- Tracepoints ... 4
- Kprobes ... 5

Ftrace: общая информация ... 8

Трассировщик function ... 9

Трассировщик function_graph ... 12

Фильтрация ... 13

Трассировка событий ... 14

Трассировка ядра с LTTNg ... 16

- Немного истории ... 16
- Установка ... 16
- Основные понятия: сессии, события и каналы ... 18
- Трассировка событий ... 19
- Трассировка системных вызовов ... 21
- Динамическая трассировка ... 22

Предисловие

Многие системные администраторы нередко сталкивались с такой ситуацией: приложение вроде бы запускается, но потом вдруг останавливается или зависает без всяких видимых причин. Обращение к системным журналам тоже не решает проблемы. Нужно более пристально изучить, что происходит в системе.

В таких ситуациях получить ценную информацию и диагностировать возможные причины проблемы можно с помощью профилирования и трассировки ядра.

Профилирование ядра — это выявление «узких мест» в производительности. С помощью профилирования можно определить, где именно произошла потеря производительности в той или иной программе. Специальные программы генерируют профиль — сводку событий, на основе которой можно выяснить, на выполнение каких функций ушло больше всего времени. Эти программы, однако, не помогают выявить причину снижения производительности.

«Узкие места» очень часто возникают при определённых условиях, которые при профилировании выявить невозможно. Чтобы понять, почему произошло то или иное событие, требуется восстанавливать контекст. В свою очередь, для восстановления контекста требуется трассировка.

Под **трассировкой** понимается получение информации о том, что происходит внутри работающей системы. Для этого используются специальные программные инструменты. Они регистрируют все события в системе подобно тому, как магнитофон записывает все звуки окружающей среды. Программы-трассировщики могут одновременно отслеживать события как на уровне отдельных приложений, так и на уровне операционной системы. Полученная в ходе трассировки информация может оказаться полезной для диагностики и решения многих системных проблем.

Трассировку иногда сравнивают с логированием. Сходство между этими двумя процедурами действительно есть, но есть и различия.

Во время трассировки записывается информация о событиях, происходящих на низком уровне. Их количество исчисляется сотнями и даже тысячами. В логи же записывается информация о высокоуровневых событиях, которые случаются гораздо реже: например, вход пользователей в систему, ошибки в работе приложений, транзакции в базах данных и другие.

Как и логи, файлы с данными трассировки можно читать «с листа». Гораздо интереснее и полезнее, однако, извлекать из них информацию, которая относится к работе конкретных приложений.

Для ОС семейства Linux создано множество трассировочно-диагностических инструментов. Об этих инструментах имеется немало подробных публикаций.

В то же время широкой аудитории почти неизвестны «родные» механизмы трассировки, присутствующие в ядре всех современных дистрибутивов Linux. Об этих механизмах мы подробно расскажем в этой книге.

Отдельный раздел мы посвятим ftrace — единственному на сегодняшний день инструменту трассировки, официально включённому в ядро Linux.

Книга будет полезна разработчикам, системным администраторам и всем, кто интересуется операционными системами семейства Linux.

Ядро Linux: основные механизмы трассировки

В ядре Linux существуют три основных механизма, при помощи которых осуществляются процедуры трассировки и профилирования ядра:

- tracepoints — механизм, работающий через статическое инструментирование кода;
- kprobes — механизм динамической трассировки, с помощью которого можно прервать выполнение кода ядра в любом месте, вызвать собственный обработчик и по завершении всех необходимых операций вернуться обратно;
- perf_events — интерфейс доступа к PMU (Performance Monitoring Unit).

Именно на этих механизмах основываются абсолютно все программы для трассировки и профилирования ядра. Ниже мы рассмотрим особенности работы каждого из этих механизмов более подробно.

Tracepoints

Название tracepoints в переводе означает «точка трассировки». Точка трассировки — это специальная вставка в код, регистрирующая определённые системные события. Точка трассировки могут быть активной (это значит, что за ней закреплена некоторая проверка) или неактивной (никакой проверки за ней не закреплено).

Неактивная точка трассировки никакого влияния на работу системы не оказывает; она лишь добавляет несколько байт для вызова трассировочной функции в конце инструментированной функции, а также добавляет структуру данных в отдельную секцию.

Когда точка трассировки активна, при выполнении соответствующего фрагмента кода вызывается трассировочная функция. Данные трассировки записываются в отладочный кольцевой буфер.

Общее число имеющихся точек трассировки в ядре можно узнать с помощью следующей команды:

```
$ perf list tracepoint | wc -l
```

Точки трассировки могут быть вставлены в любое место в коде. Они уже присутствуют в коде многих ядерных функций. Рассмотрим, например, функцию kmem_cache_alloc (взято отсюда):

```
{
    void *ret = slab_alloc(cachep, flags, _RET_IP_);

    trace_kmem_cache_alloc(_RET_IP_, ret,
                            cachep->object_size, cachep->size, flags);
         return ret;
         }
```

trace_kmem_cache_alloc — это и есть точка трассировки. Количество подобных примеров можно умножить, обратившись к исходному коду других функций ядра.

Написать соответствующий код и вставить точки трассировки, например, в собственный модуль ядра — дело достаточно сложное и, к тому же требующее глубоких знаний языка С и системного программирования в целом (см. примеры кода здесь).

Впрочем, в большинстве случаев такие сложности не нужны: в Linux имеются очень простые и надёжные механизмы, с помощью которых можно обращаться к точкам трассировки и отслеживать системные события. Их мы более подробно рассмотрим ниже.

Kprobes

Механизм kprobes был добавлен в ядро Linux начиная, с версии 2.6.9. Его название представляет собой сокращение от kernel probes. Kprobes — это механизм для динамического инструментирования кода. С его помощью можно прервать исполнение кода в любом месте, вызвать собственный обработчик и вернуться обратно.

Для работы с kprobes нужно написать собственный модуль ядра и зарегистрировать обработчик на любую функцию или вообще на любой адрес. Рассмотрим следующий пример кода:

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kprobes.h>

static unsigned int counter = 0;
int Pre_Handler(struct kprobe *p, struct pt_regs *regs){
    printk(«Pre_Handler: counter=%u\n»,counter++);
    return 0;
}

void Post_Handler(struct kprobe *p, struct pt_
regs *regs, unsigned long flags){
    printk(«Post_Handler: counter=%u\n»,counter++);
}

static struct kprobe kp;

int myinit(void)
{
    printk(«module inserted\n «);
    kp.pre_handler = Pre_Handler;
    kp.post_handler = Post_Handler;
    kp.addr = (kprobe_opcode_t *)0xffffffff8120c3b0;
    register_kprobe(&kp);
    return 0;
}

void myexit(void)
{
    unregister_kprobe(&kp);
    printk(«module removed\n «);
}

module_init(myinit);
module_exit(myexit);
MODULE_AUTHOR(«Andrei»);
MODULE_DESCRIPTION(«KPROBE MODULE»);
MODULE_LICENSE(«GPL»);
```

Make-файл, с помощью которого из этого кода можно собрать модуль ядра, выглядит так:

```
obj-m +=mod1.o
KDIR= /lib/modules/$(shell uname -r)/build
all:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
	rm -rf *.o *.ko *.mod.* .c* .t*
```

По завершении сборки нужно добавить модуль mod1.ko в ядро:

```
$ insmod mod1.ko
```

Сделаем несколько комментариев к приведённому выше фрагменту кода:

```
#include <linux/kprobes.h>
```

Здесь мы ссылаемся на заголовочный файл kprobes.h, в котором описана структура struct_kprobe. Вот наиболее интересные для нас поля этой структуры:

```
struct kprobe {
    .
    .
    kprobe_opcode_t *addr;
    kprobe_pre_handler_t pre_handler;
    kprobe_post_handler_t post_handler;
}
```

В коде мы должны указать адрес интересующей нас функции (в нашем случае это функция sys_open).

Узнать адрес можно с помощью команды:

```
$ cat /proc/kallsyms |grep sys_open
```

Адрес затем нужно указать в функции myinit ():

```
kp.addr = (kprobe_opcode_t *)0xffffffff8120c3b0;
```

Обработчик вызывается перед входом в функцию и после выхода из неё. В нашем фрагменте кода для того используются функции Pre_Handler () и PostHandler () соответственно. Их нужно присвоить указателям в структуре kprobe:

```
kp.pre_handler = Pre_Handler;
kp.post_handler = Post_Handler;
```

Также нам нужно зарегистрировать модуль в ядре (функция register_kprobe (&kp);)

Ftrace: общая информация

Ftrace (сокращение от Function Tracer) представляет собой фреймворк, который предоставляет кольцевой буфер для сбора данных. Собирают эти данные специальные программы-трассировщики.

Ftrace, как и следует из названия, предназначен прежде всего для трассировки функций. Однако его реальные возможности гораздо шире: с его помощью можно отслеживать переключения контекстов, измерять время обработки прерываний, высчитывать время на активизацию заданий с высоким приоритетом.

Работает ftrace на базе файловой системы debugfs, которая в большинстве современных дистрибутивов Linux смонтирована по умолчанию. Чтобы приступить к работе с ftrace, нужно просто перейти в директорию sys/kernel/debug/tracing (она доступна только для root):

```
# cd /sys/kernel/debug/tracing
```

Вот список её содержимого:

```
available_filter_functions  options             stack_trace_filter
available_tracers           per_cpu             trace
buffer_size_kb              printk_formats      trace_clock
buffer_total_size_kb        README              trace_marker
current_tracer              saved_cmdlines      trace_options
dyn_ftrace_total_info       set_event           trace_pipe
enabled_functions           set_ftrace_filter   trace_stat
events                      set_ftrace_notrace  tracing_cpumask
free_buffer                 set_ftrace_pid      tracing_max_latency
function_profile_enabled    set_graph_function  tracing_on
instances                   set_graph_notrace   tracing_thresh
kprobe_events               snapshot            uprobe_events
kprobe_profile              stack_max_size      uprobe_profile
```

Просмотреть список доступных трассировщиков можно так:

```
$ cat available_tracers
blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup function nop
```

Приведём краткую характеристику для каждого трассировщика:

- function — трассировщик функций вызовов ядра без возможности получения аргументов;

- function_graph — трассировщик функций вызовов ядра с подвызовами;
- blk — трассировщик вызовов и событий, связанных с вводом-выводом на блочные устройства; получает почти ту же информацию, что утилита blktrace;
- mmiotrace — трассировщик операций ввода-вывода, отражаемых в память.
- nop — самый простой трассировщик, который, как и следует из названия, ничего не делает (однако в некоторых ситуациях и он может быть полезен, о чём ещё пойдёт речь ниже).

Особенности практической работы с некоторыми из этих трассировщиков мы рассмотрим ниже.

Трассировщик function

Самый простой трассировщик — это, конечно же, function. Чтобы познакомиться с ним поближе и понять, как он работает, откроем любой текстовый редактор и напишем небольшой тестовый скрипт:

```
#!/bin/sh

dir=/sys/kernel/debug/tracing

sysctl kernel.ftrace_enabled=1
echo function > ${dir}/current_tracer
echo 1 > ${dir}/tracing_on
sleep 1
echo 0 > ${dir}/tracing_on
less ${dir}/trace
```

Скрипт очень простой и в целом понятный, однако в нём есть моменты, на которые стоит обратить внимание.

Команда sysctl ftrace.enabled=1 включает трассировку функций. Далее мы устанавливаем текущий трассировщик, записывая его имя в файл current_tracer.

После этого мы записываем 1 в файл tracing_on и тем самым активируем обновление кольцевого буфера. Буквально через одну строку мы его

отключаем (обратите внимание: если в файл tracing_on записать 0, кольцевой буфер не будет очищен; не будет отключен и ftrace).

Зачем мы это делаем? Между двумя командами echo находится команда sleep 1. Мы включаем обновление буфера, выполняем эту команду и затем сразу же его отключаем. Благодаря этому в трассировку будет включена информация обо всех вызовах функций, которые имели место во время выполнения этой команды.

В последней строке скрипта мы выводим данные трассировки на консоль:

```
# tracer: function
#
# entries-in-buffer/entries-written: 29571/29571   #P:2
#
#                               _-----=> irqs-off
#                              / _----=> need-resched
#                             | / _---=> hardirq/softirq
#                             || / _--=> preempt-depth
#                             ||| /     delay
#            TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#               | |       |   ||||       |         |
        trace.sh-1295  [000] ....    90.502874:
mutex_unlock <-rb_simple_write
        trace.sh-1295  [000] ....    90.502875: __fsnotify_parent <-vfs_write
        trace.sh-1295  [000] ....    90.502876: fsnotify <-vfs_write
        trace.sh-1295  [000] ....    90.502876: __srcu_read_lock <-fsnotify
        trace.sh-1295  [000] ....    90.502876: __srcu_read_unlock <-fsnotify
        trace.sh-1295  [000] ....    90.502877: __sb_end_write <-vfs_write
        trace.sh-1295  [000] ....    90.502877: syscall_
trace_leave <-int_check_syscall_exit_work
        trace.sh-1295  [000] ....    90.502878: context_
tracking_user_exit <-syscall_trace_leave
        trace.sh-1295  [000] ....    90.502878: context_
tracking_user_enter <-syscall_trace_leave
        trace.sh-1295  [000] d...    90.502878: vtime_
user_enter <-context_tracking_user_enter
        trace.sh-1295  [000] d...    90.502878: _
raw_spin_lock <-vtime_user_enter
        trace.sh-1295  [000] d...    90.502878: __
vtime_account_system <-vtime_user_enter
        trace.sh-1295  [000] d...    90.502878: get_
vtime_delta <-__vtime_account_system
        trace.sh-1295  [000] d...    90.502879: account_
system_time <-__vtime_account_system
```

```
        trace.sh-1295  [000] d...    90.502879: cpuacct_
account_field <-account_system_time
        trace.sh-1295  [000] d...    90.502879: acct_
account_cputime <-account_system_time
        trace.sh-1295  [000] d...    90.502879: __
acct_update_integrals <-acct_account_cputime
        trace.sh-1295  [000] d...    90.502879: jiffies_
to_timeval <-__acct_update_integrals
        trace.sh-1295  [000] d...    90.502879: _
raw_spin_unlock <-vtime_user_enter
        trace.sh-1295  [000] d...    90.502879: rcu_
user_enter <-context_tracking_user_enter
        trace.sh-1295  [000] d...    90.502880:
rcu_eqs_enter <-rcu_user_enter
        trace.sh-1295  [000] d...    90.502880: rcu_
eqs_enter_common.isra.47 <-rcu_eqs_enter
        trace.sh-1295  [000] ....    90.502883:
syscall_trace_enter <-tracesys
        trace.sh-1295  [000] ....    90.502883: context_
tracking_user_exit <-syscall_trace_enter
        trace.sh-1295  [000] d...    90.502884: rcu_
user_exit <-context_tracking_user_exit
```

Вывод начинается с информации о количестве записей событий в буфере (entries in buffer) и общем количестве записанных событий (entries written). Разница между этими двумя цифрами — это количество событий, утерянных при заполнении буфера (в нашем случае никаких утерянных событий нет).

Далее идёт перечень функций, включающий следующую информацию:

- идентификатор процесса (PID);
- номер процессорного ядра, на котором выполняется трассировка (CPU#);
- метка времени (TIMESTAMP; указывает время, когда произошёл вход в функцию);
- имя трассируемой функции и имя родительской функции, которая её вызвала (FUNCTION); например, в самой первой строке приведённого нами вывода функцию mutex-unlock вызывает функция rb_simple_write.

Трассировщик function_graph

Трассировщик function_graph работает точно так же, как function, но отслеживает функции более подробно: для каждой функции он указывает точку входа и точку выхода. С помощью этого трассировщика можно отслеживать функции с подвызовами и измерять время выполнения для каждой функции.

Отредактируем приведённый выше скрипт:

```
#!/bin/sh

dir=/sys/kernel/debug/tracing

sysctl kernel.ftrace_enabled=1
echo function_graph > ${dir}/current_tracer
echo 1 > ${dir}/tracing_on
sleep 1
echo 0 > ${dir}/tracing_on
less ${dir}/trace
```

Вывод трассировки, полученный при выполнении этого скрипта, будет выглядеть примерно так:

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0)   0.120 us    |                                 } /* resched_task */
 0)   1.877 us    |                               } /* check_preempt_curr */
 0)   4.264 us    |                             } /* ttwu_do_wakeup */
 0) + 29.053 us   |                           } /* ttwu_do_activate.constprop.74 */
 0)   0.091 us    |                           _raw_spin_unlock();
 0)   0.260 us    |                           ttwu_stat();
 0)   0.133 us    |                           _raw_spin_unlock_irqrestore();
 0) + 37.785 us   |                         } /* try_to_wake_up */
 0) + 38.478 us   |                       } /* default_wake_function */
 0) + 39.203 us   |                     } /* pollwake */
 0) + 40.793 us   |                   } /* __wake_up_common */
 0)   0.104 us    |                   _raw_spin_unlock_irqrestore();
 0) + 42.920 us   |                 } /* __wake_up_sync_key */
 0) + 44.160 us   |               } /* sock_def_readable */
 0) ! 192.850 us  |             } /* tcp_rcv_established */
 0) ! 197.445 us  |           } /* tcp_v4_do_rcv */
 0)   0.113 us    |           _raw_spin_unlock();
```

```
 0) ! 205.655 us  |          } /* tcp_v4_rcv */
 0) ! 208.154 us  |        } /* ip_local_deliver_finish */
```

Как видим, в нём появились графы, которых в выводе трассировщика function не было.

В графе DURATION указано время, затраченное на выполнение функции. Если оно превышает 10 микросекунд, то в графе DURATION отображается знак «+», а если 100 микросекунд — восклицательный знак (!).

Далее идёт графа FUNCTION_CALLS с информацией о вызовах функций.

Начало и конец выполнения каждой функции обозначаются в ней так, как это принято в языке C: фигурная скобка в начале функции и ещё одна — в конце. Функции, которые являются листьями графа и не вызывают никаких других функций, обозначаются точкой с запятой (;).

Фильтрация

Вывод ftrace порой может быть очень большим, и найти в нём нужную информацию крайне затруднительно. Упростить поиск можно с помощью фильтров: в вывод будет попадать информация не обо всех функциях, а лишь о тех, которые нас действительно интересуют. Для этого достаточно просто записать в файл set_ftrace_filter имена нужных функций, например:

```
echo kfree > set_ftrace_filter
```

Чтобы отключить фильтр, нужно записать в этот же файл пустую строку:

```
echo > set_ftrace_filter
```

В результате выполнения команды:

```
echo kfree > set_ftrace_notrace
```

мы получим совершенно противоположный результат: в вывод будет попадать информация обо всех функциях, кроме kfree ().

Ещё одна полезная опция — set_ftrace_pid. Она предназначена для трассировки функций, вызываемых во время выполнения указанного процесса.

Трассировка событий

Через ftrace можно взаимодействовать с механизмом tracepoints. Слово tracepoint в переводе означает «точка трассировки».

В ядре Linux имеется специальный API, с помощью которого можно работать с точками трассировки из пользовательского пространства. В директории /sys/kernel/debug/tracing есть поддиректория events, в которой хранятся доступные для отслеживания системные события. Системное событие в данном контексте — не что иное, как включенные в ядро точки трассировки.

Их список можно просмотреть с помощью команды:

```
# cat available_events
```

На консоль будет выведен длинный список, просматривать который довольно неудобно. Вывести этот же список в более структурированном формате можно так (мы приводим список для ядра версии 3.13.0.24; для других версий сборок ядра он может отличаться):

```
# ls /sys/kernel/debug/tracing/events

block              gpio          mce       random          skb        vsyscall
btrfs              header_event  migrate   ras             sock       workqueue
compaction         header_page   module    raw_syscalls    spi        writeback
context_tracking   iommu         napi      rcu             swiotlb    xen
enable             irq           net       regmap          syscalls   xfs
exceptions         irq_vectors   nmi       regulator       task       xhci-hcd
ext4               jbd2          oom       rpm             timer
filemap            kmem          pagemap   sched           udp
fs                 kvm           power     scsi            vfs
ftrace             kvmmmu        printk    signal          vmscan
```

Все возможные типы событий объединены в поддиректории по подсистемам.

Прежде чем приступать к отслеживанию событий, проверим, включена ли запись событий в кольцевой буфер:

```
# cat tracing_on
1
```

Если после выполнения этой команды на консоль будет выведена цифра 0, выполним:

```
# echo 1 > tracing_on
```

Попробуем отследить, например, системный вызов chroot. Все события, связанные с системными вызовами, хранятся в директории syscalls. В ней, в свою очередь, хранятся директории для входа и выхода из различных системных вызовов. Нас будет интересовать вход в системный вызов chroot. Активируем нужную нам точку трассировки, записав 1 в её enable-файл:

```
echo 1 > /sys/kernel/debug/tracing/events/syscalls/sys_enter_chroot/enable
```

Затем создадим изолированную файловую систему с помощью команды chroot. После выполнения интересующих нас команд отключим запись в кольцевой буфер, чтобы в вывод не попадала информация о лишних и не имеющих отношения к делу событиях:

```
echo 0 > tracing_on
```

Далее просмотрим содержимое кольцевого буфера:

```
cat /sys/kernel/debug/tracing/trace
```

```
# tracer: nop
#
# entries-in-buffer/entries-written: 13644/13644   #P:1
#
#
  _-----=> irqs-off
#                              / _----=> need-resched
#                             | / _---=> hardirq/softirq
#                             || / _--=> preempt-depth
#                             ||| /     delay
#           TASK-PID    CPU#  ||||    TIMESTAMP  FUNCTION
#              | |        |   ||||       |          |

          chroot-11321 [000] ....  4606.265208: sys_chroot(filename: 7fff785ae8c2)
          chroot-11325 [000] ....  4691.677767: sys_chroot(filename: 7fff242308cc)
            bash-11338 [000] ....  4746.971300: sys_chroot(filename: 7fff1efca8cc)
            bash-11351 [000] ....  5379.020609: sys_chroot(filename: 7fffbf9918cc)
```

Трассировка ядра с LTTNg

Немного истории

В прошлом разделе мы уже говорили о том, что абсолютно все инструменты трассировки основываются на встроенных механизмах Linux. Мы подробно разобрали механизм точек трассировки (tracepoints) и даже привели примеры кода. Но мы ничего не сказали о том, что механизм tracepoints был создан в процессе работы над LTTng.

Ещё в 1999 году сотрудник компании IBM Карим Ягмур (Karim Yaghmour) начал работу над проектом LTT (Linux Trace Toolkit). Лежащая в основе LTT заключалась в том, чтобы статически инструментировать наиболее важные фрагменты в коде ядра и благодаря этому получать информацию о работе системы. Несколько лет спустя эта идея была подхвачена и развита Матьё Денуайе в рамках проекта LTTng, что означает Linux Tracing Tool New Generation. Первый релиз LTTng состоялся в 2005 году.

Слова New Generation используются в названии инструмента не просто так: Денуайе внёс огромный вклад в развитие механизмов трассировки и профилирования Linux. Он добавил статическое инструментирование для важнейших ядерных функций: так получился механизм kernel markers, в результате усовершенствования которого появился хорошо знакомый нам tracepoints. Tracepoints активно используется в LTTng (ниже мы покажем, как именно). Именно благодаря этому механизму стало возможным осуществлять трассировку, не создавая дополнительной нагрузки на работу системы.

Помимо tracepoints, в LTTng используется и механизм kprobes, который работает через динамическое инструментирование кода ядра.

Для LTTng создана специальная программа-визуализатор событий — LTTV (Linux Trace Toolkit Visualizer). Имеется также возможность интеграции с IDE Eclipse.

Последняя на сегодняшний день стабильная версия LTTng 2.7 вышла в свет в октябре 2015 года. Вот-вот должна выйти версия 2.8, которая на текущий момент находится в статусе релиз-кандидата и доступна для скачивания здесь.

Установка

LTTng включен в репозитории большинства современных дистрибутивов Linux, и установить его можно стандартным способом. В новых версиях

популярных дистрибутивов (например, в недавно вышедшей Ubuntu 16.04) для установки по умолчанию доступна последняя стабильная версия LTTng — 2.7:

```
$ sudo apt-get install lttng-tools lttng-modules-dkms
```

Если вы используете более старую версию Linux, для установки LTTng 2.7 понадобится добавить PPA-репозиторий:

```
$ sudo apt-add-repository ppa: lttng/ppa
$ sudo apt-get update
$ sudo apt-get install lttng-tools lttng-modules-dkms
```

Пакет lttng-tools содержит следующие утилиты:

- babeltrace — утилита для просмотра выводов трассировки в формате CTF (Common Trace Format);
- lttng-crash — утилита для анализа и диагностики причин сбоев ядра;
- lttng-sessiond — демон для управления трассировкой;
- lttng-consumerd — демон, собирающий данные и записывающий их в кольцевой буфер;
- lttng-relayd — демон, передающий данные по сети.

В пакет lttng-modules-dkms входят многочисленные модули ядра, с помощью которых осуществляется взаимодействие со встроенными механизмами трассировки и профилирования. Просмотреть их список можно с помощью команды:

```
$ lsmod |grep lttng
```

Все эти модули можно условно разделить на три группы:

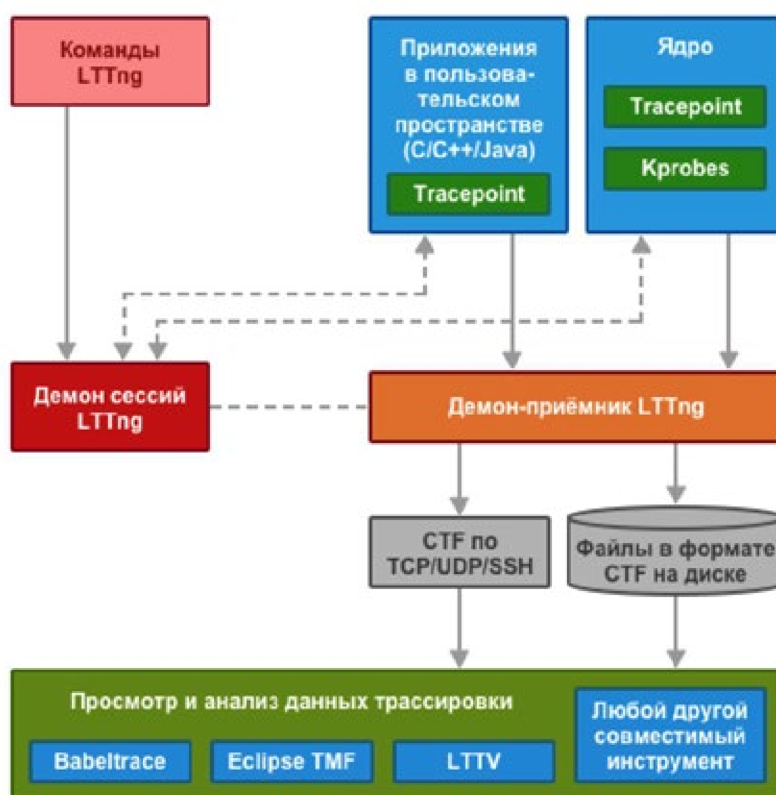
- модули для работы с точками трассировки (tracepoints);
- модули для работы с кольцевым буфером;
- модули-пробы, предназначенные для динамической трассировки ядра.

Из официальных репозиториев для установки доступен также пакет LTTng-UST, с помощью которого осуществляется трассировка событий в пользовательском пространстве. В рамках этой статьи мы его рассматривать не будем. Заинтересованных читателей отсылаем к этой статье, которая может служить вполне неплохим введением в тему.

Все команды LTTng должны выполняться либо от имени root-пользователя, либо от имени пользователя, включённого в специальную группу tracing (она создаётся автоматически при установке).

Основные понятия: сессии, события и каналы

Процесс взаимодействия всех компонентов LTTng можно представить в виде следующей схемы:



Чтобы лучше понять, как всё это работает, сделаем небольшой теоретический экскурс и разберём содержание основных понятий.

Как уже было сказано выше, трассировка в LTTng осуществляется на основе как статического, так и динамического инструментирования кода.

Во время выполнения инструментированного кода вызывается специальная функция-проба (англ. probe, что можно также перевести как «датчик» или «зонд»), которая считывает статус сессии и записывает информацию о событиях в каналы.

Проясним содержание понятий «сессия» и «канал». Сессией называется процедура трассировки с заданными пользователем параметрами. На рисунке выше показана всего одна сессия, но в LTTng можно запускать и несколько сессий одновременно. Сессию можно в любой момент остановить, изменять её настройки и затем запускать снова.

Для передачи отладочной информации каждая сессия использует каналы. Канал — это набор событий с определёнными характеристиками и дополнительной контекстной информацией. К числу характеристик (более подробно мы расскажем о них ниже) канала относятся: размер буфера, режим трассировки, период очистки буфера.

Зачем нужны каналы? Прежде всего для того, чтобы поддерживать общий кольцевой буфер, в который при трассировки записываются события и откуда их затем забирает демон-приёмник (consumer daemon). Кольцевой буфер в свою очередь подразделяется на многочисленные секции (sub-buffers) одинакового размера. Данные о событиях записываются в секцию, пока она не будет заполнена. После этого запись данных будет продолжена, но уже в другую секцию. Данные из переполненных секций забирает демон-приёмник и сохраняет их на диске (или передаёт по сети).

В идеале ситуаций, когда секции заполнены и данные записывать некуда, возникать не должно. В реальной практике, однако, такие случаи иногда бывают. В официальной документации прямо сказано, что в LTTng производительность поставлена во главу угла: лучше потерять некоторую часть событий, чем замедлить работу системы. При создании канала можно выбирать один из двух режимов работы при переполнении буфера:

- режим сброса (discard mode) — все новые события не будут записываться до тех пор, пока не будет освобождена одна из секций;
- режим перезаписи (overwrite mode) — самые старые события будут удалены, а на их место будут записываться новые.

Более подробно о настройке каналов можно прочитать в официальной документации.

Трассировка событий

Приступим к изучению LTTng на практике и запустим первую трассировочную сессию.

Просмотреть список доступных для трассировки событий можно так (приводим лишь небольшой фрагмент, на самом деле этот список гораздо больше):

```
$ lttng list --kernel

Kernel events:
-------------
      writeback_nothread (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      writeback_queue (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      writeback_exec (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      writeback_start (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      writeback_written (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      writeback_wait (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      writeback_pages_written (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      …….
      snd_soc_jack_irq (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      snd_soc_jack_report (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      snd_soc_jack_notify (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      snd_soc_cache_sync (loglevel: TRACE_EMERG (0)) (type: tracepoint)
```

Как видим, в этот список включены точки трассировки (tracepoints), написанные специально для LTTng и присутствующие в тех модулях, о которых мы уже говорили выше.

Попробуем отследить какое-нибудь событие, например, sched_switch. Сначала создадим сессию:

```
$ lttng create test_session
Session test_session created.
Traces will be written in /user/lttng-traces/test_session-20151119—134747
```

Итак, сессия создана. Все собранные в ходе этой сессии данные будут записываться в файл `/user/lttng-traces/test_session-20151119-134747`. Затем активируем интересующее нас событие:

```
$ lttng enable-event -k sched_switch
Kernel event sched_switch created in channel channel0
```

Далее выполним:

```
$ lttng start
Tracing started for session test_session
$ sleep 1
$ lttng stop
```

Информация обо всех событиях sched_switch будет сохранятся в отдельном файле. Просмотреть данные его содержимое можно так:

```
$ lttng view
```

© Selectel

Список событий будет слишком большим. Более того, он будет включать слишком много информации. Попробуем конкретизировать запрос и получить информацию только о событиях, которые имели место при выполнении команды sleep:

```
$ babeltrace /user/lttng-traces/test_session-20151119—134747 | grep sleep
[13:53:23.278672041] (+0.001249216) luna sched_switch: {cpu_id = 0},
{prev_comm = «sleep», prev_tid = 10935, prev_prio = 20, prev_state
= 1, next_comm = «swapper/0», next_tid = 0, next_prio = 20}
[13:53:24.278777924] (+0.005448925) luna sched_switch: {cpu_id = 0},
{prev_comm = «swapper/0», prev_tid = 0, prev_prio = 20, prev_state
= 0, next_comm = «sleep», next_tid = 10935, next_prio = 20}
[13:53:24.278912164] (+0.000134240) luna sched_switch: {cpu_id =
0}, {prev_comm = «sleep», prev_tid = 10935, prev_prio = 20, prev_
state = 0, next_comm = «rcuos/0», next_tid = 8, next_prio = 20}
```

Вывод содержит информацию обо всех событиях sched_switch, зарегистрированных в системном ядре за время сессии. Он состоит из нескольких полей. Первое поле — это метка времени, второе — так называемая дельта (количество времени между предыдущим и текущим событием). В поле cpu_id указывается номер CPU, для которого было зарегистрировано событие. Далее следует дополнительная контекстуальная информация.

По завершении трассировки сессию нужно удалить:

```
$ lttng destroy
```

Трассировка системных вызовов

Для отслеживания системных вызовов у команды lttng enable-event имеется специальная опция — syscall. В статье, посвящённой ftrace, мы разбирали пример с отслеживанием системного вызова chroot. Попробуем проделать то же самое с помощью LTTng:

```
$ lttng create
$ lttng enable-event -k --syscall chroot
$ lttng start
```

Создадим изолированное окружение с помощью команды chroot, а затем выполним:

```
$ lttng stop
$ lttng view
[12:05:51.993970181] (+?.?????????) andrei syscall_entry_
chroot: {cpu_id = 0}, {filename = «test»}
[12:05:51.993974601] (+0.000004420) andrei
```

```
syscall_exit_chroot: {cpu_id = 0}, {ret = 0}
[12:06:53.373062654] (+61.379088053) andrei syscall_
entry_chroot: {cpu_id = 0}, {filename = «test»}
[12:06:53.373066648] (+0.000003994) andrei syscall_
exit_chroot: {cpu_id = 0}, {ret = 0}
[12:07:36.701313906] (+43.328247258) andrei syscall_
entry_chroot: {cpu_id = 1}, {filename = «test»}
[12:07:36.701325202] (+0.000011296) andrei syscall_
exit_chroot: {cpu_id = 1}, {ret = 0}
```

Как видим, вывод содержит информацию обо всех входах в системный вызов syscall и выходах из него. По сравнению с выводом ftrace он выглядит несколько более структурированным и человекопонятным.

Динамическая трассировка

Выше мы рассмотрели примеры статической трассировки с использованием механизма tracepoints.

Для отслеживания событий в Linux также можно использовать механизм динамической трассировки kprobes (сокращение от kernel probes, как не трудно догадаться). Он позволяет добавлять новые точки трассировки (пробы) в ядро «на лету». Именно на kprobes основывается известный инструмент SystemTap. Работает он так: чтобы добавить в ядро пробу, нужно написать скрипт на специальном С-подобном языке; затем этот скрипт транслируется в код на C, который компилируется в отдельный модуль ядра.

Начиная с версии ядра 3.10, поддержка kprobes была добавлена в ftrace. Благодаря этому стала возможной динамическая трассировка без написания скриптов и добавления новых модулей.

Реализована поддержка kprobes и в LTTng. Поддерживается два вида проб: собственно kprobes («базовые» пробы, которые могут быть вставлены в любое место в ядре) и kretprobes (ставятся перед выходом из функции и дают доступ к её результату). Рассмотрим несколько практических примеров.

В LTTng для установки «базовых» проб используется опция, которая так и называется — probe:

```
$ lttng create
$ lttng enable-event --kernel sys_open --probe sys_open+0x0
$ lttng enable-event --kernel sys_close --probe sys_close+0x0
$ lttng start
$ sleep 1
$ lttng stop
```

Полученный в результате трассировки вывод будет выглядеть так (приводим небольшой фрагмент):

```
….
[12:45:26.842026294] (+0.000028311) andrei sys_
close: {cpu_id = 1}, {ip = 0xFFFFFFFF81209830}
[12:45:26.842036177] (+0.000009883) andrei sys_
open: {cpu_id = 1}, {ip = 0xFFFFFFFF8120B940}
[12:45:26.842064681] (+0.000028504) andrei sys_
close: {cpu_id = 1}, {ip = 0xFFFFFFFF81209830}
[12:45:26.842097484] (+0.000032803) andrei sys_
open: {cpu_id = 1}, {ip = 0xFFFFFFFF8120B940}
[12:45:26.842126464] (+0.000028980) andrei sys_
close: {cpu_id = 1}, {ip = 0xFFFFFFFF81209830}
[12:45:26.842141670] (+0.000015206) andrei sys_
open: {cpu_id = 1}, {ip = 0xFFFFFFFF8120B940}
[12:45:26.842179482] (+0.000037812) andrei sys_
close: {cpu_id = 1}, {ip = 0xFFFFFFFF81209830}
```

В приведённом выводе содержится параметр ip — адрес отслеживаемой функции в ядре.

С помощью опции --function можно выставлять динамические пробы на вход в функцию и выход из неё, например:

```
$ lttng enable-event call_rcu_sched -k --function call_rcu_sched
```

```
/приводим фрагмент вывода/

[15:31:39.092335027] (+0.000000742) cs31401 call_rcu_sched_return: {cpu_
id = 0}, {}, {ip = 0xFFFFFFFF810E7B10, parent_ip = 0xFFFFFFFF810A206D}
[15:31:39.092398089] (+0.000063062) cs31401 call_rcu_sched_entry: {cpu_id
= 0}, {}, {ip = 0xFFFFFFFF810E7B10, parent_ip = 0xFFFFFFFF810A206D}
[15:31:39.092398883] (+0.000000794) cs31401 call_rcu_sched_return: {cpu_
id = 0}, {}, {ip = 0xFFFFFFFF810E7B10, parent_ip = 0xFFFFFFFF810A206D}
```

В приведённом выводе присутствует ещё один параметр: parent_ip — адрес функции, которая вызвала отслеживаемую функцию.

© Selectel

selectel.ru