目录

[一、 BPFtrace简介 3](#_Toc28960)

[1.1 BPFtrace简介 3](#_Toc5800)

[1.2 BPFtrace特性 3](#_Toc19540)

[1.3 BPFtrace和Ftrace比较 3](#_Toc25911)

[1.4 BPFtrace程序格式 3](#_Toc9931)

[1.4.1 程序注释 3](#_Toc20354)

[1.4.2 探针格式 3](#_Toc5746)

[1.4.3 探针通配符 4](#_Toc8723)

[1.4.4 过滤器 4](#_Toc13008)

[1.4.5 代码块 4](#_Toc9964)

[1.4.6 函数 4](#_Toc21515)

[1.4.7 变量 4](#_Toc31494)

[1.4.7.1 内置变量 5](#_Toc23296)

[1.4.7.2 临时变量 5](#_Toc23683)

[1.4.7.3 映射变量 5](#_Toc21271)

[1.5 BPFtrace探针类型 5](#_Toc8117)

[1.6 其他详细介绍 5](#_Toc957)

[二、 环境安装配置 5](#_Toc10543)

[2.1 Linux环境配置 6](#_Toc30378)

[2.1.1 VMware环境配置 6](#_Toc32115)

[2.1.2 Ubuntu环境配置 6](#_Toc24926)

[2.1.2.1 下载操作系统镜像 6](#_Toc15099)

[2.1.2.2 VMware安装Ubuntu20.04 6](#_Toc4984)

[2.2 BPFtrace配置 6](#_Toc14772)

[2.2.1 Ubuntu 19.04以上版本 6](#_Toc22741)

[2.2.2 Ubuntu 16.04及更高版本 6](#_Toc8996)

[2.3 gcc配置 6](#_Toc30104)

[三、 示例程序展示 7](#_Toc12485)

[3.1 安装 bpftrace 7](#_Toc21390)

[3.2 使用“hello world”进行实验 7](#_Toc27247)

[3.3 打印父进程的名字和系统中正在创建的每个新进程的名称 8](#_Toc29576)

[3.4 查询追踪点的所有可用字段 9](#_Toc19643)

[3.5 统计系统调用数 9](#_Toc2715)

[3.5.1 统计全部程序的系统调用数 9](#_Toc16320)

[3.5.1.1 代码 9](#_Toc12914)

[3.5.1.2 运行截图 10](#_Toc18602)

[3.5.2 统计单个程序的系统调用数 11](#_Toc3343)

[3.5.2.1 代码 11](#_Toc29929)

[3.5.2.2 运行截图 12](#_Toc32685)

[3.6 进程写的字节数 12](#_Toc13335)

[3.6.1 代码 12](#_Toc1447)

[3.6.2 运行截图 12](#_Toc15808)

[3.7 进程的读取大小分布（柱状图） 13](#_Toc25479)

[3.8 追踪用户空间程序 14](#_Toc19119)

[3.8.1 获取系统中运行的每个 bash 发出的命令行 14](#_Toc944)

[3.8.1.1 代码 14](#_Toc25972)

[3.8.1.2 运行截图 15](#_Toc20802)

[3.8.2 列出可执行文件bash的所有可用uprobes/uretprobes 15](#_Toc29753)

[3.8.2.1 代码 16](#_Toc19236)

[3.8.2.2 运行截图 16](#_Toc28193)

[3.9 自带脚本拓展程序 16](#_Toc6236)

[3.9.1 代码 17](#_Toc201)

[3.9.2 运行截图 17](#_Toc28695)

[四、 拓展程序 17](#_Toc20129)

[4.1内核级函数探针的使用 17](#_Toc31508)

[4.1.1 基础知识 17](#_Toc30006)

[4.1.1.1 内核级函数探针的定义 17](#_Toc16721)

[4.1.1.2 内核级函数探针的格式 17](#_Toc17062)

[4.1.2基本步骤 18](#_Toc3625)

[4.1.3实现 18](#_Toc31695)

[4.1.3.1 需要嗅探的函数 18](#_Toc17981)

[4.1.3.2 BPFtrace嗅探程序 18](#_Toc24837)

[4.1.3.3 嗅探结果 19](#_Toc17893)

[4.2 用户级函数探针的使用 19](#_Toc19786)

[4.2.1 基础知识 19](#_Toc21506)

[4.2.1.1 用户级函数探针的定义 19](#_Toc27733)

[4.2.1.2 用户级函数探针的格式 20](#_Toc21594)

[4.2.1.3 用户级函数探针的参数 21](#_Toc9725)

[4.2.2 基本步骤 21](#_Toc32239)

[4.3 静态插桩命令的使用 30](#_Toc4537)

[4.3.1基础知识 30](#_Toc15596)

[4.3.1.1 静态插桩命令定义 31](#_Toc6993)

[4.3.1.2 静态插桩命令格式 31](#_Toc25884)

[4.3.2基本步骤 31](#_Toc2554)

[4.3.2.1 tracepoint命令的使用 31](#_Toc16655)

[4.3.2.2 tracepoint命令的使用结果 32](#_Toc25971)

[4.3.2.3 usdt命令的使用 32](#_Toc11086)

[4.3.2.4 usdt命令的使用结果 34](#_Toc20697)

[4.4基于时间的BPF 34](#_Toc2642)

[4.4.1基于时间的采样——profile 34](#_Toc29004)

[4.4.2基于时间的输出——interval 36](#_Toc3356)

[4.5 事件探针的使用 37](#_Toc26275)

[4.5.1 基础知识 37](#_Toc13725)

[4.5.1.1 事件探针的定义 37](#_Toc20148)

[4.5.1.2 用户级函数探针的格式 37](#_Toc26448)

[4.5.2 基本步骤 40](#_Toc20485)

1. BPFtrace简介

1.1 BPFtrace简介

bpftrace 是 Linux 高级追踪工具和语言。该工具基于 eBPF 和 BBC 实现了通过探针机制采集内核和程序运行的信息，然后用图表等方式将信息展示出来，帮助开发者找到隐藏较深的 Bug、安全问题和性能瓶颈。

bpftrace 使用 LLVM 作为后端，将脚本编译为 BPF 字节码，并利用 BCC 与 Linux BPF 系统交互，以及现有的 Linux 跟踪功能：内核动态跟踪（kprobes）、用户级动态跟踪（uprobes）、和跟踪点。bpftrace 语言的灵感来自 awk 和 C，以及 DTrace 和 SystemTap 等前置探测器。

1.2 BPFtrace特性

插桩所需要的额外系统开销较低

在生产环境中只能由root用户使用

工具应尽量简单、短小。更复杂的工具需要借助BCC实现。

新手教学文档和帮助文档齐全、完整。

1.3 BPFtrace和Ftrace比较

BPFtrace提供一种和c语言以及awk非常相似的编程语言，而Ftrace则使用一种自有语法来实现对内核的探测功能。由于Ftrace所需要的依赖比BPFtrace少，因此Ftrace更适用于嵌入式Linux环境，BPFtrace适合个人计算机或服务器环境。Ftrace在某些功能上经过专门的性能优化，因而在一些特定事件上Ftrace比BPFtrace更加高效。

1.4 BPFtrace程序格式

1.4.1 程序注释

bpftrace注释方式和c语言一样：

1. *//单行注释*
2. */\**
3. \*多行注释
4. \*/

1.4.2 探针格式

标识符的类型由探针类型决定。内核级别的探针可以使用“pbftrace -l”命令查看，这个命令会以标准格式完整的打印出bpftrace支持的跟踪点。用户态探针格式为‘uprobe：用户程序二进制文档的绝对地址：跟踪的函数名’。例如：

1. 探针类型：标识符1：标识符2：......
2. kprobe：vfs\_read              *//内核探针*
3. uprobe：/home/qz/a.out:add    *//用户探针*

1.4.3 探针通配符

1. kprobe：vfs\_\*

这会对bpftrace支持的所有以vfs\_开头的内核函数进行插桩。通配符不仅可以在探针插桩上使用，而且也能在查找跟踪点上使用。例如：

1. bpftrace -l 'vfs\_\*'

1.4.4 过滤器

过滤器是一个布尔表达式，当检查内容为真时才会继续执行后面的程序。例如下面的过滤条件：

1. /pid/
2. /pid == 1234/
3. /pid == 100 || pid == 1234/   *//过滤器允许使用布尔运算符*

1.4.5 代码块

一个代码块可以是单条语句，也可以是多条语句，多条语句需要使用分号将语句分隔。例如：

1. bpftrace -e ‘探针 {语句1； 语句2；......}‘

1.4.6 函数

除了第一个例子中的printf()之外，还有time()、exit()、str()等，更多支持的函数参考：<https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md>

1.4.7 变量

变量分为内置变量、临时变量和映射变量

1.4.7.1 内置变量

由bpftrace预先定义好，可以直接读取信息的变量。例如表示进程ID的pid变量，表示进程名的comm表示以纳秒为单位的nsecs等。

1.4.7.2 临时变量

可以用于临时存储数据、计算的变量。以‘‘$’’作为前缀。例如：

1. $a = 1;
2. $b = "qwer";
3. $c = (struct task\_struct \*)ts;

1.4.7.3 映射变量

使用映射表存储数据，以“@”作为前缀，同时也可以用作全局存储，在不同语句间传递数据。

1. probe1 {@a = 10;}
2. probe2 {$b = @a;}
3. probe3 {@ns[pid] = nsecs;} *// 定义关联数组，相当于定义一个哈希表，名称是ns，键是pid。*

1.5 BPFtrace探针类型

kprobe——内核函数的开始处  
 kretprobe——内核函数的返回处  
 uprobe——用户级函数的开始处  
 uretprobe——用户级函数的返回处  
 tracepoint——内核静态追踪点  
 usdt——用户级静态追踪点  
 profile——基于时间的采样  
 interval——基于时间的输出  
 software——内核软件事件  
 hardware——处理器级事件

1.6 其他详细介绍

<https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md>

1. 环境安装配置

本次实验的软件环境为Linux操作系统，硬件环境为普通PC机

2.1 Linux环境配置

2.1.1 VMware环境配置

本次实验采用VMware Workstation 16 Pro版本，其他版本如VMware Workstation 15（Pro），或者VMware Workstation 16同样适用，可自行在官方网站下载：

<https://www.vmware.com/products/workstation-player.html>

或者使用以下链接：

<http://www.zhanshaoyi.com/15258.html>

安装教程：

<http://www.zhanshaoyi.com/15261.html>

2.1.2 Ubuntu环境配置

2.1.2.1 下载操作系统镜像

本次实验采用Ubuntu20.04LTS版本，可以在南京大学镜像站下载对应的iso：

<https://mirrors.nju.edu.cn/ubuntu-releases/20.04/>

2.1.2.2 VMware安装Ubuntu20.04

可在下述链接中查看安装方法，非常详细：

<https://zhuanlan.zhihu.com/p/141033713>

2.2 BPFtrace配置

2.2.1 Ubuntu 19.04以上版本

1. sudo apt-get install -y bpftrace

2.2.2 Ubuntu 16.04及更高版本

1. sudo snap install --devmode bpftrace
2. sudo snap connect bpftrace:system-trace

2.3 gcc配置

如果出现了询问，输入y即可

1. sudo apt-get update
2. sudo apt-get install gcc
3. 示例程序展示

3.1 安装 bpftrace

1. sudo apt-get install -y bpftrace

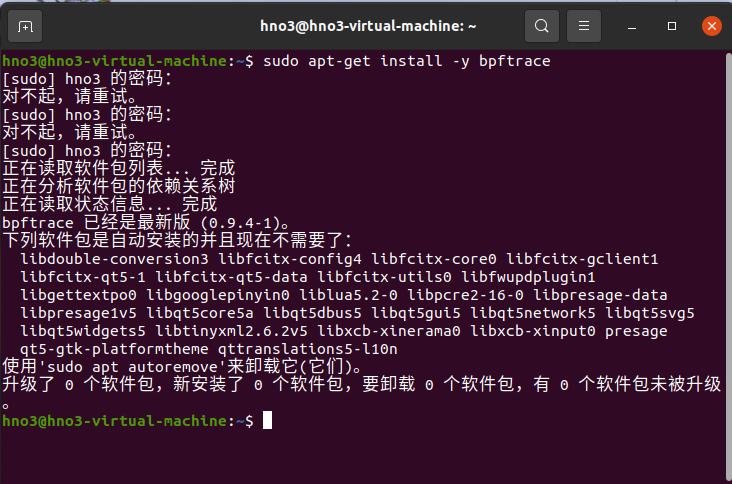


图 3-1

3.2 使用“hello world”进行实验

1. $ sudo bpftrace -e 'BEGIN { printf("hello world\n"); }'

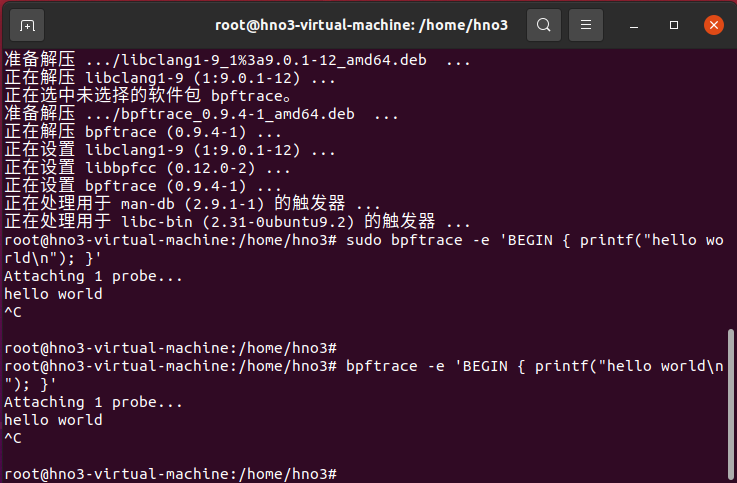


图 3-2

3.3 打印父进程的名字和系统中正在创建的每个新进程的名称

这个例子打印了父进程的名字（comm）和系统中正在创建的每个新进程的名称。t:syscalls:sys\_enter\_execve 是一个内核追踪点，是 tracepoint:syscalls:sys\_enter\_execve 的简写，两种形式都可以使用。comm 是一个bpftrace内建指令，代表进程名；filename 是 t:syscalls:sys\_enter\_execve 追踪点的一个字段，这些字段可以通过 args 内建指令访问。

1. $ sudo bpftrace -e 't:syscalls:sys\_enter\_execve { printf("%s called %s\n", comm, str(args->filename)); }'

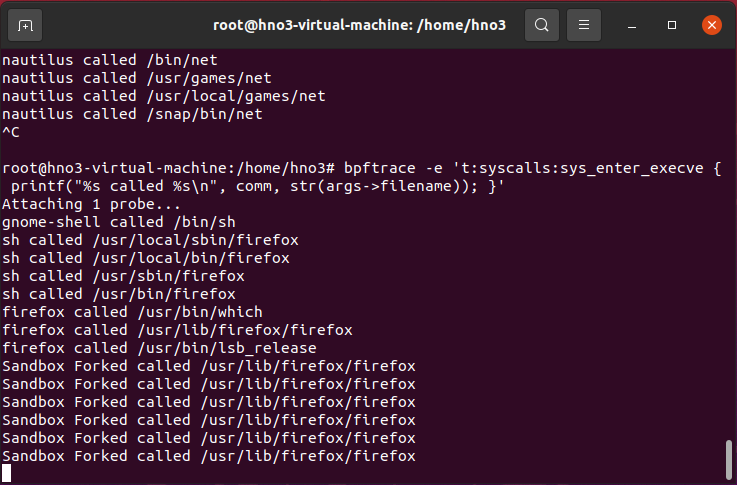


图 3-3

3.4 查询追踪点的所有可用字段

1. $ bpftrace -lv "t:syscalls:sys\_enter\_execve"

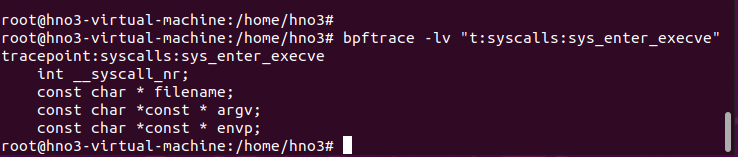


图 3-4

3.5 统计系统调用数

3.5.1 统计全部程序的系统调用数

3.5.1.1 代码

映射 是保存计数、统计数据和柱状图的特殊 BPF 数据类型，你可以使用映射统计每个系统调用正在被调用的次数：

1. $ sudo bpftrace -e 't:syscalls:sys\_enter\_\* { @[probe] = count(); }'

一些探针类型允许使用通配符匹配多个探针，你也可以使用一个逗号隔开的列表为一个操作块指明多个连接点。上面的例子中，操作块连接到了所有名称以 t:syscalls:sysenter\_ 开头的追踪点，即所有可用的系统调用。

bpftrace 的内建函数 count() 统计系统调用被调用的次数；@[] 代表一个映射（一个关联数组）。该映射的键 probe 是另一个内建指令，代表完整的探针名。

这个例子中，相同的操作块连接到了每个系统调用，之后每次有系统调用被调用时，映射就会被更新，映射中和系统调用对应的项就会增加。程序终止时，自动打印出所有声明的映射。

3.5.1.2 运行截图

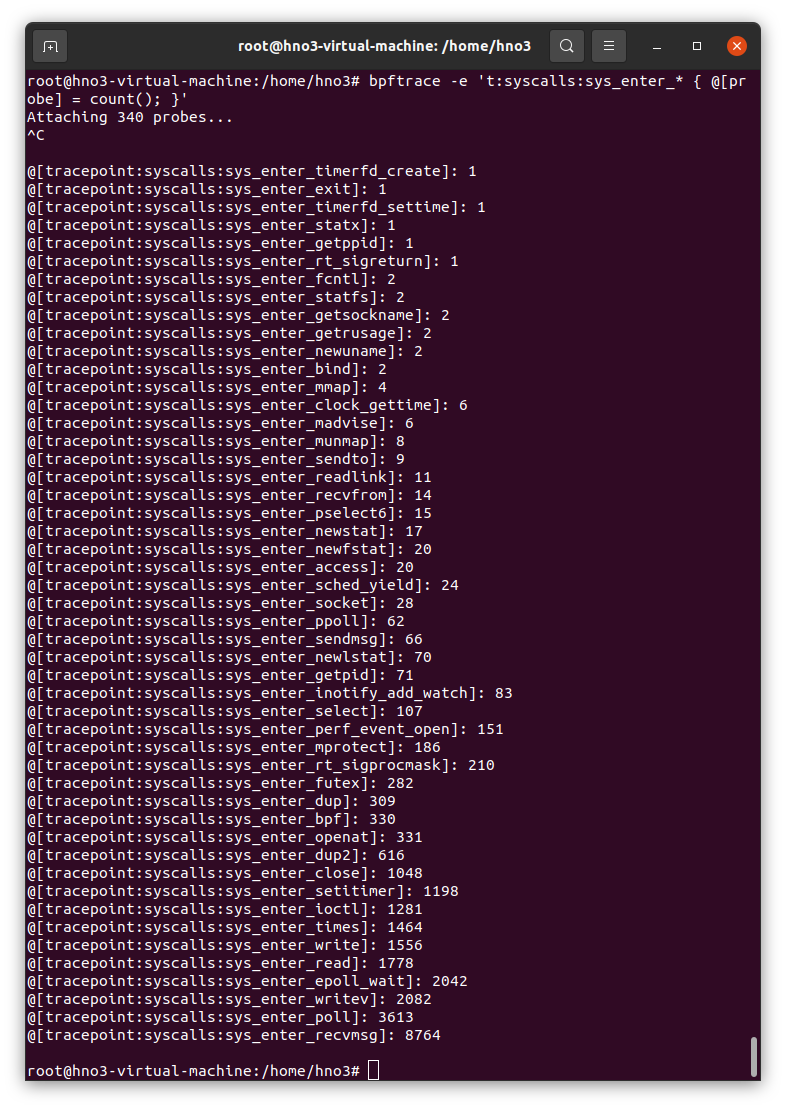


图 3-5-1

3.5.2 统计单个程序的系统调用数

3.5.2.1 代码

下面的例子统计所有的系统调用，然后通过 bpftrace 过滤语法使用 PID 过滤出某个特定进程调用的系统调用：

1. $ sudo bpftrace -e 't:syscalls:sys\_enter\_\* / pid == 1619 / { @[probe] = count(); }'

3.5.2.2 运行截图

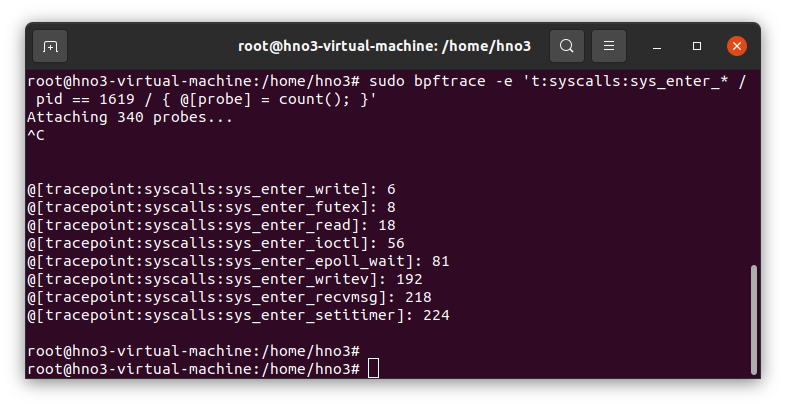


图 3-5-2

3.6 进程写的字节数

3.6.1 代码

让我们使用3.5的概念分析每个进程正在写的字节数，bpftrace 连接操作块到写系统调用的返回探针（t:syscalls:sys\_exit\_write），然后使用过滤器丢掉代表错误代码的负值（/arg->ret > 0/）。

映射的键 comm 代表调用系统调用的进程名；内建函数 sum() 累计每个映射项或进程写的字节数；args 是一个 bpftrace 内建指令，用于访问追踪点的参数和返回值。如果执行成功，write 系统调用返回写的字节数，arg->ret用于访问这个字节数。

1. $ sudo bpftrace -e 't:syscalls:sys\_exit\_write /args->ret > 0/ { @[comm] = sum(args->ret); }'

3.6.2 运行截图

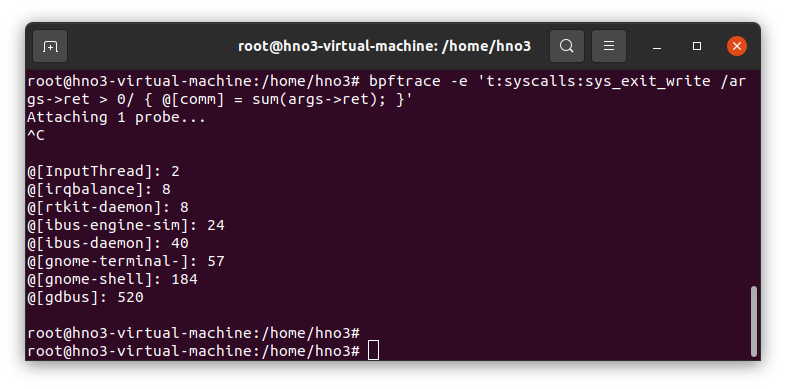


图 3-6

3.7 进程的读取大小分布（柱状图）

1. $ sudo bpftrace -e ‘t:syscalls:sys\_exit\_read { @[comm] = hist(args->ret;) }’

这个命令跟踪了所有进程执行read的统计，并在停止跟踪时统计直方图输出。hist(): 一个map函数，用来描述直方图的参数。输出行以2次方的间隔开始，如[128, 256)表示值大于等于128且小于256。后面跟着位于该区间的个数统计，最后是ascii码表示的直方图。

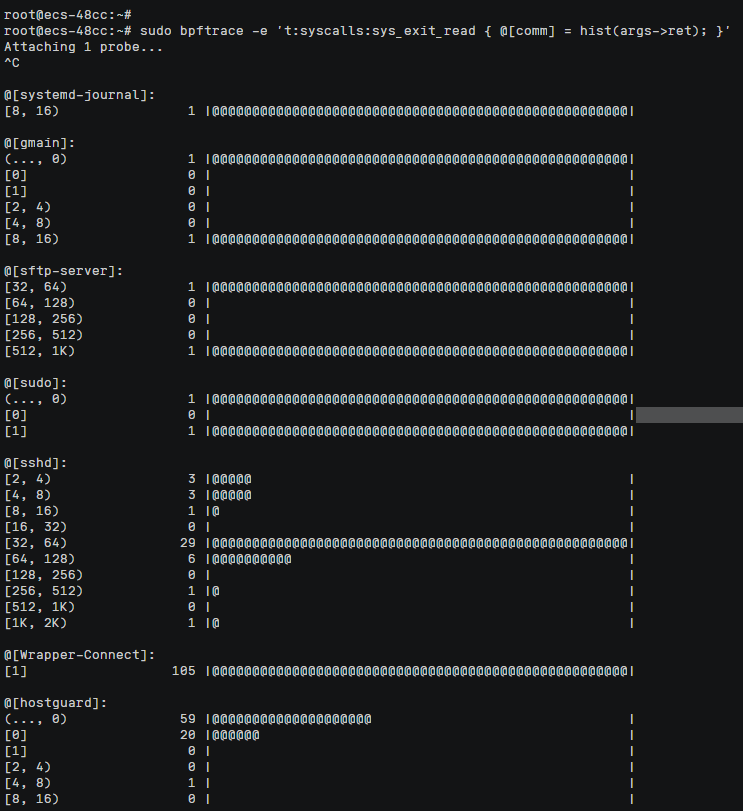


图 3-7

3.8 追踪用户空间程序

3.8.1 获取系统中运行的每个 bash 发出的命令行

3.8.1.1 代码

我们可以通过uprobes/uretprobes和 USDT（用户级静态定义的追踪）探针追踪用户空间程序。下述例子使用探测用户级函数结尾处的uretprobe，获取系统中运行的每个bash发出的命令行：

1. $ sudo bpftrace -e 'uretprobe:/bin/bash:readline { printf("readline: \"%s\"\n", str(retval)); }'

在上述代码中，影号里的内容等价于文件bash.bt:

1. uretprobe:/bin/bash:readline{
2. printf("readline:\"%s\"\n",str(retval));
3. }

uprobe指向用户级函数执行的开始，uretprobe指向执行的结束（返回处）；readline()是/bin/bash的一个函数，返回键入的命令行；retval是被探测的指令的返回值，只能在uretprobe访问。使用uprobes时，你可以用arg0..argN访问参数。同时，在这个例子中retval的类型是char \*指针，需要调用str()将char \*指针转化成一个字符串，以便输出。

3.8.1.2 运行截图

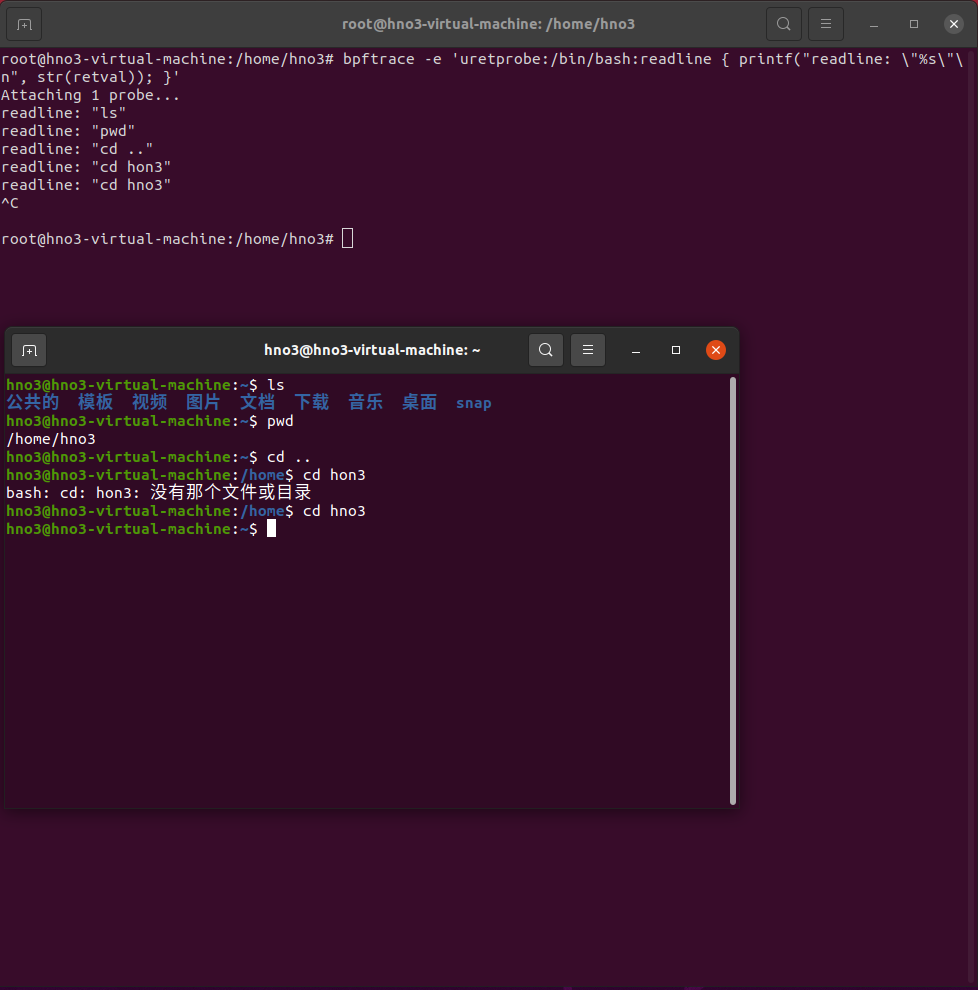


图 3-8-1

3.8.2 列出可执行文件bash的所有可用uprobes/uretprobes

3.8.2.1 代码

1. $ sudo bpftrace -l "uprobe:/bin/bash"

3.8.2.2 运行截图

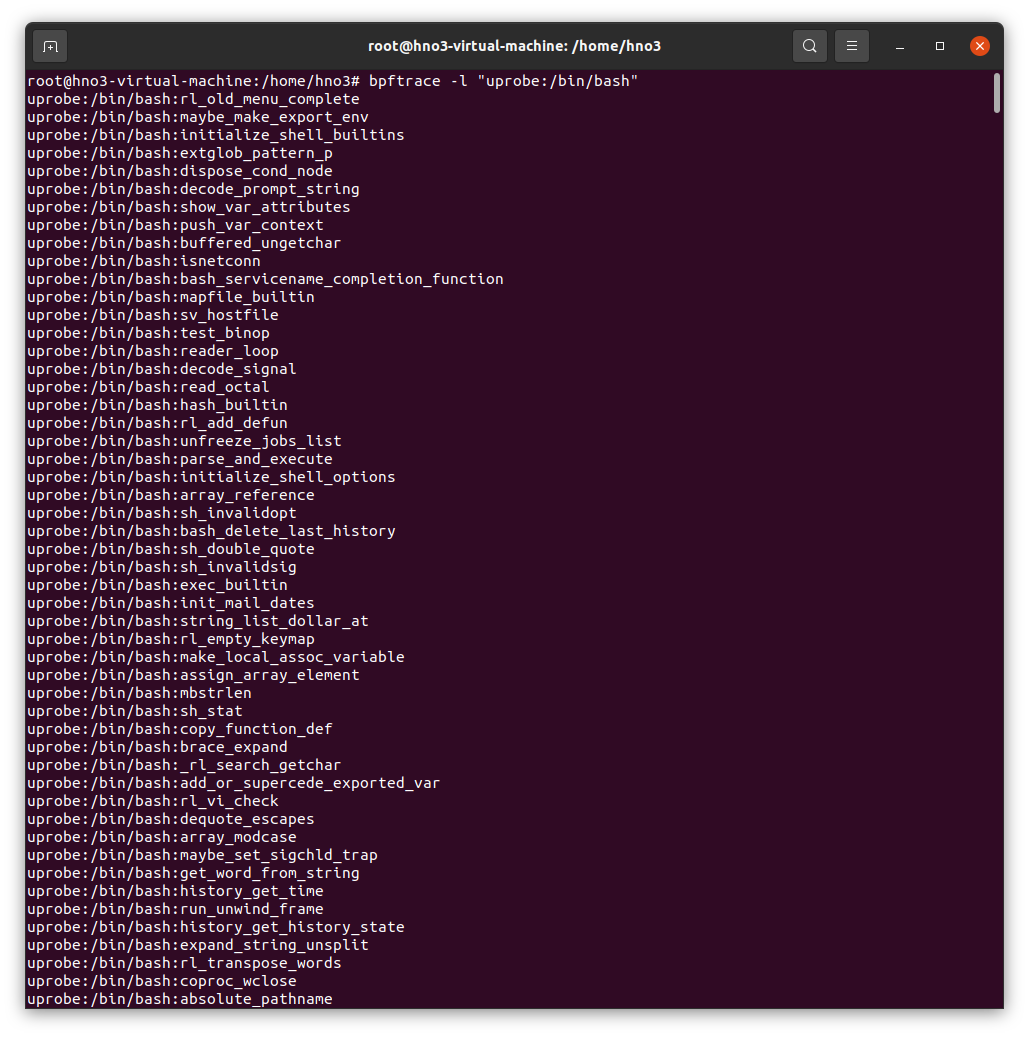


图 3-8-2

3.9 自带脚本拓展程序

bpftrace软件包附带了许多有用的脚本，可以在/usr/sbin目录找到。这些脚本中，你可以找到：

* killsnoop.bt——追踪kill()系统调用发出的信号
* tcpconnect.bt——追踪所有的TCP网络连接
* pidpersec.bt——统计每秒钟（通过fork）创建的新进程
* opensnoop.bt——追踪open()系统调用
* bfsstat.bt——追踪一些VFS调用，按秒统计

你可以直接使用这些脚本，比如接下来在例子中使用的killsnoop.bt,追踪kill()系统调用发出的信号

3.9.1 代码

1. $ sudo /usr/sbin/killsnoop.bt

3.9.2 运行截图

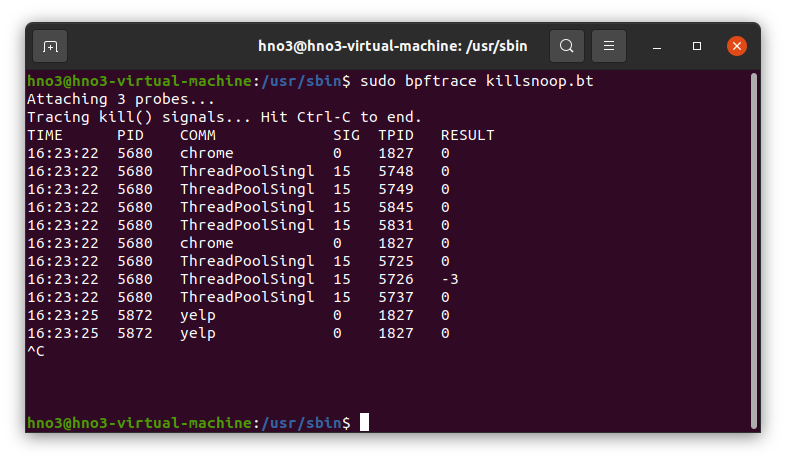


图 3-9

1. 拓展程序

4.1内核级函数探针的使用

4.1.1 基础知识

4.1.1.1 内核级函数探针的定义

kprobe探针可对内核函数开始位置进行动态插桩，kretprobe探针可对内核函数结束位置进行动态插桩。

4.1.1.2 内核级函数探针的格式

1. kprobe:function\_name[+offset]
2. kretprobe:function\_name

其中，function\_name是需要插桩的内核函数名，offset是地址的偏移量。

function\_name表示一个内核函数。

function\_name[+offset]表示一个内存地址，是function的内存地址加上偏移量offset，指向一条内存指令。

4.1.2基本步骤

内核级函数探针的使用，无外乎就是以下几个步骤：

1. 选择一个需要嗅探的函数（内核态）
2. 写一个相应的嗅探文件，例如test.bt
3. 在一个终端中执行嗅探文件
4. 按下Ctrl+C结束嗅探
5. 在一个终端中查看嗅探结果

4.1.3实现

4.1.3.1 需要嗅探的函数

1. vfs\_read

4.1.3.2 BPFtrace嗅探程序

1. *//初始化探针，可以理解为只要执行嗅探程序，就首先触发，但只会触发一次*
2. BEGIN
3. {
4. printf("使用内核级函数探针kprobe和kretprobe");
5. printf("显示进程调用vfs\_read函数的开始时间和持续时间");
6. }
7. *//显示进程调用vfs\_read函数的开始时间*
8. kprobe:vfs\_read
9. {
10. *//@start的映射表为map类型，tid是线程id，nsecs是当前时间*
11. @start[tid] = nsecs;
12. }
14. *//用直方图显示进程调用vfs\_read函数的持续时间*
15. kretprobe:vfs\_read
16. /@start[tid]/
17. {
18. *//保存函数的执行时间*
19. $duration\_us = (nsecs - @start[tid]) / 1000;
20. *//生成直方图*
21. @us = hist($duration\_us);
22. *//删除映射表*
23. delete(@start[tid]);
24. }
25. *//一种探针类型，可以理解为嗅探程序结束时，就触发，但只会触发一次*
26. END
27. {
28. printf("\n嗅探结束\n");
29. }

嗅探的思路是kprobe探针会在每个线程在进入read函数的时候，都用@start映射表保存进入的时间。而kretprobe探针会在每一个线程从read函数返回的时候，用@duration\_us保存执行时间，然后使用hist生成直方图。

4.1.3.3 嗅探结果



图4-1-3-3

我们可以看到线程6128，4369，4006，6301四个线程进入read函数的时间保存在@start中，然后显示出来。并且us会将四个线程的持续时间的直方图画出。即四个线程会执行0微秒的read函数由221条，执行1微秒的read函数由138条等等。

4.2 用户级函数探针的使用

4.2.1 基础知识

4.2.1.1 用户级函数探针的定义

uprobe 探针检测用户级函数执行的开始，而 uretprobe 探针检测结束（返回）。

4.2.1.2 用户级函数探针的格式

4.2.1.2.1 简介

1. uprobe:library\_name:function\_name[+offset]
2. uprobe:library\_name:address
3. uretprobe:library\_name:function\_name

其中，uprobe类似于int，是一个类型标识符

library\_name是需要嗅探的文件的绝对路径，例如：/home/hno3/maze

function\_name是文件里的函数的名称，例如，maze文件里有一个main函数：/home/hno3/maze:main

offset是一个偏移量，function\_name[+offset]就是一个内存地址，等于function\_name的地址加上偏移量，这个内存地址里是一条汇编指令

address是文件里的函数的内存地址，可以代替函数名

第三条格式的意思是嗅探在library\_name文件中的function\_name函数

第二条格式的意思是嗅探在library\_name文件中的address这个内存地址里保存的函数

第一条格式的意思是嗅探在library\_name文件中的function\_name[+offset]这个内存地址里保存的汇编指令

4.2.1.2.2 示例

1. *# bpftrace -e '*
2. *uretprobe:/bin/bash:readline { printf("read a line\n");}'*
3. Attaching 1 probe...
4. read a line
5. read a line
6. read a line
7. ^C

这个例子中，我们嗅探了在/bin/bash路径下的bash文件里的readline函数，对应上述的第三个格式

1. *# objdump -tT /bin/bash | grep main*
2. ...
3. 000000000002ec00 g    DF .text  0000000000001868  Base        main
4. ...
5. *# bpftrace -e 'uprobe:/bin/bash:0x2ec00 { printf("in here\n"); }'*
6. Attaching 1 probe...

第二个例子中，我们通过反汇编指令objdump -tT得到main函数的内存地址，然后嗅探了在/bin/bash路径下的bash文件里的readline函数，对应上述的第二个格式

1. # objdump -d /bin/bash
2. ...
3. 000000000002ec00 <main@@Base>:
4. 2ec00:       f3 0f 1e fa             endbr64
5. 2ec04:       41 57                   push   %r15
6. 2ec06:       41 56                   push   %r14
7. 2ec08:       41 55                   push   %r13
8. ...
9. # bpftrace -e 'uprobe:/bin/bash:main+4 { printf("in here\n"); }'
10. Attaching 1 probe...
11. ...

在最后的例子中，通过反汇编指令objdump -d反汇编main函数可以得知，main函数的基址为000000000002ec00，下一条汇编指令的地址为2ec04，也就是基址+4，那么我们可以使用：

bpftrace -e 'uprobe:/bin/bash:main+4 { printf("in here\n"); }'

来嗅探这条汇编指令，对应上述的第一个格式

4.2.1.3 用户级函数探针的参数

Uprobe是嗅探用户函数开始时的探针，他的探针参数主要是：arg0, arg1, ..., argN，表示所嗅探的函数传递的参数，比如有一个文件里的用户级函数：

1. int add(int a,int b){
2. return a+b;
3. }

如果此时我调用了add(1,2)，也就是参数a=1，b=2，那么uprobe探针的arg0参数就是1，arg1参数就是2，以此类推

同理，uretprobe是嗅探用户函数结束时的探针，它的探针参数只有一个，就是返回值：retval，用add函数的例子来讲，调用了add(1,2)，那么此时uretprobe探针的参数retval的值就是3。

4.2.2 基本步骤

用户级函数探针的使用，无外乎就是以下几个步骤：

* 选择一个需要嗅探的文件及函数（用户态），例如a.c
* 写一个相应的嗅探文件，例如test.bt
* 在第一个终端中执行嗅探文件
* 在第二个终端中执行被嗅探文件
* 在第一个终端中查看嗅探结果
* 按下Ctrl+C结束嗅探

4.2.3.1 需要嗅探的程序

我们以一个采用dfs深度优先搜索的程序作为嗅探目标，代码如下，注释中详细解释了代码的细节：

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <stdbool.h>
4. #include <string.h>
5. *//起点*
6. #define START 2
7. *//终点*
8. #define END 3
9. *//迷宫长度*
10. #define n 5
11. *//迷宫宽度*
12. #define m 5
13. *//迷宫最大长宽*
14. #define N 50
15. *//迷宫的地图矩阵，2表示起点，1表示墙，过不去，0表示任意通过，3表示终点*
16. int map[n][m] = {
17. {2, 0, 0, 0, 0},
18. {1, 1, 1, 1, 0},
19. {0, 0, 0, 0, 0},
20. {0, 1, 1, 1, 1},
21. {0, 0, 0, 0, 3}};
22. *//标记每个位置是否访问过*
23. int visit[n][m] = {0};
24. *//四个方向*
25. int dir[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
26. *//是否有路*
27. bool have\_way = 0;
28. *//起始点横纵坐标，已走步数*
29. int x = 0, y = 0, step = 0;
30. *//找到路了，终止程序*
31. void stop()
32. {
33. exit(0);
34. }
35. *//深度优先搜索寻路*
36. *// x,y是当前点的坐标，step是已走步数*
37. void dfs(int x, int y, int step)
38. {
39. *//如果在终点*
40. if (map[x][y] == END)
41. {
42. *//输出到达和步数信息*
43. printf("当前在终点[%d,%d],已经走了%d步\n", x, y, step);
44. printf("有出去的路!\n");
45. *//标记有路*
46. have\_way = true;
47. *//终止程序*
48. stop();
49. }
50. *//如果不在终点*
51. else
52. {
53. *//打印当前所在点的位置，和步数信息*
54. printf("当前在点[%d,%d],已经走了%d步\n", x, y, step);
55. int xi, yi;
56. *//四个方向*
57. for (int i = 0; i < 4; i++)
58. {
59. *//计算移动后的坐标*
60. xi = x + dir[i][0];
61. yi = y + dir[i][1];
62. *//如果没超出迷宫范围并且没有访问过，并且不是墙，那么就走*
63. if (xi >= 0 && xi < m && yi >= 0 && yi < n && visit[xi][yi] == 0 && map[xi][yi] != 1)
64. {
65. *//标记前进的点为访问过*
66. visit[xi][yi] = 1;
67. *//步数++*
68. step++;
69. *//继续搜索*
70. dfs(xi, yi, step);
71. *//回溯*
72. step--;
73. visit[xi][yi] = 0;
74. }
75. }
76. }
77. }
78. *//初始化起始点坐标*
79. bool init\_start()
80. {
81. *//遍历迷宫，如果有点值为2，那么就是起始点，并更新起始点坐标*
82. for (int i = 0; i < n; i++)
83. {
84. for (int j = 0; j < m; j++)
85. {
86. if (map[i][j] == START)
87. {
88. x = i;
89. y = j;
90. }
91. }
92. }
93. *//返回更新成功*
94. return true;
95. }
96. int main()
97. {
98. *//初始化起始点坐标*
99. init\_start();
100. *//深度优先搜索寻路*
101. dfs(x, y, step);
102. *//搜完了还没有结束程序，说明没路，输出信息*
103. printf("没有出去的路!\n");
104. }

程序的思路就是用一个深度优先搜索函数在设定的迷宫中去寻找到达终点的路，找到了就输出成功信息，并终止程序，没找到就输出失败信息。

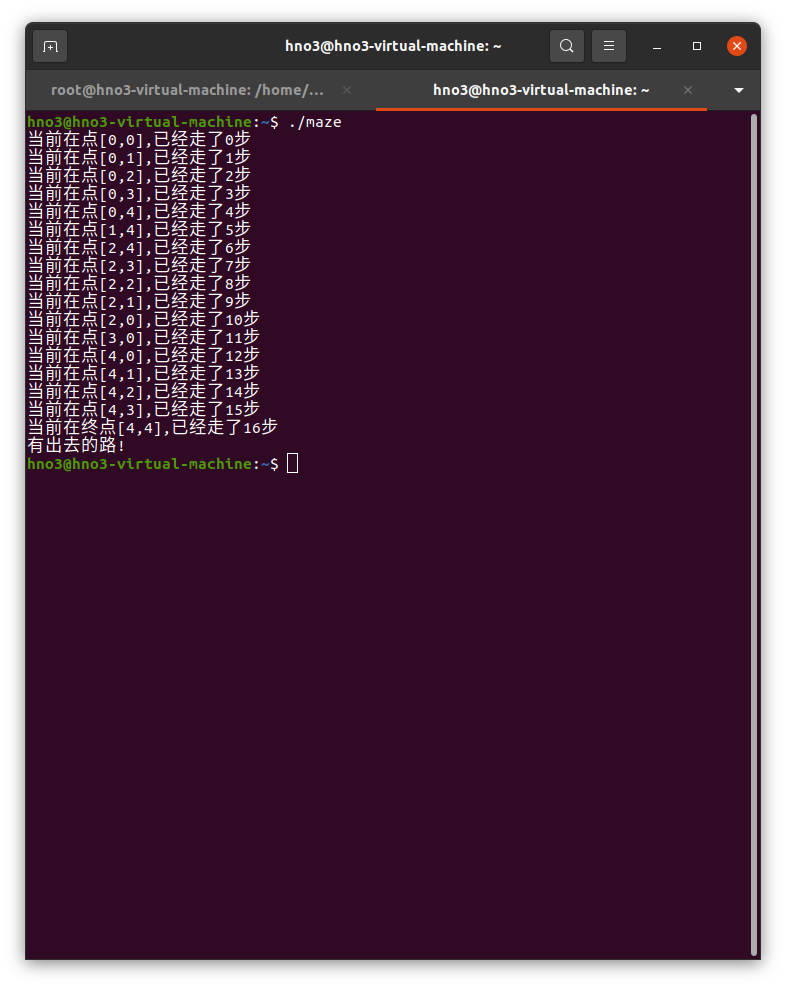
4.2.3.2 BPFtrace嗅探程序

1. *//初始化探针，可以理解为只要执行嗅探程序，就首先触发，但只会触发一次*
2. BEGIN
3. {
4. printf("使用用户态函数探针uprobe和uretprobe");
5. printf("来嗅探解决走迷宫问题的C++程序\n");
6. *//记录dfs函数调用的次数*
7. @count = 0;
8. }
9. *//嗅探dfs函数开始的用户级函数探针，可以理解为每一次执行dfs，就触发一次*
10. uprobe :./maze:dfs
11. {
12. *//调用次数++*
13. @count = @count + 1;
14. *//输出fs函数调用的次数*
15. printf("第%d次调用dfs函数\n", @count);
16. *//输出fs函数传递的参数，分别为横纵坐标和步数*
17. printf("第一个参数为%d,第二个参数为%d,第三个参数为%d\n", arg0, arg1, arg2);
18. }
19. *//嗅探init\_start函数开始的用户级函数探针*
20. uprobe :./maze:init\_start
21. {
22. printf("调用init\_start函数,初始化起点坐标!\n");
23. }
24. *//嗅探init\_start函数结束的用户级函数探针，可以理解为每一次执行init\_start，就触发一次*
25. uretprobe :./maze:init\_start
26. {
27. *//如果返回值为true*
28. if (retval)
29. {
30. printf("init\_start函数调用成功!\n");
31. }
32. *//如果返回值为false*
33. else
34. {
35. printf("init\_start函数调用失败!\n");
36. }
37. }
38. *//嗅探stop函数开始的用户级函数探针*
39. uprobe :./maze:stop
40. {
41. *//如果stop执行了，说明找到了路径，并在dfs函数中终止程序*
42. printf("找到了路径，在dfs函数中结束程序!\n");
43. }
44. *//嗅探main函数开始的用户级函数探针*
45. uprobe :./maze:main
46. {
47. printf("执行main函数!\n");
48. }
49. *//嗅探main函数结束的用户级函数探针*
50. uretprobe :./maze:main
51. {
52. *//如果main函数返回了，说明没找到路径，并在main函数返回时终止程序*
53. printf("没有找到路径，在main函数中结束程序!\n");
54. }
55. *//一种探针类型，可以理解为嗅探程序结束时，就触发，但只会触发一次*
56. END
57. {
58. printf("\n嗅探结束\n");
59. *//清除变量，这样在ctrl+c结束嗅探后不会打印@count的值*
60. clear(@count);
61. }

嗅探程序的思路是嗅探main，dfs，stop函数的开始处，并嗅探main函数的终止处，这样就可以查看搜索的进度并确认是否找到了路径，因为如果找了路径，程序会在在dfs函数中调用stop函数终止程序，反之会在main函数返回时终止程序。

4.2.3.3 嗅探结果

1. 如果迷宫有路通往终点，那么会显示执行和搜索的信息并触发stop探针：

图 4-2-3-3-1

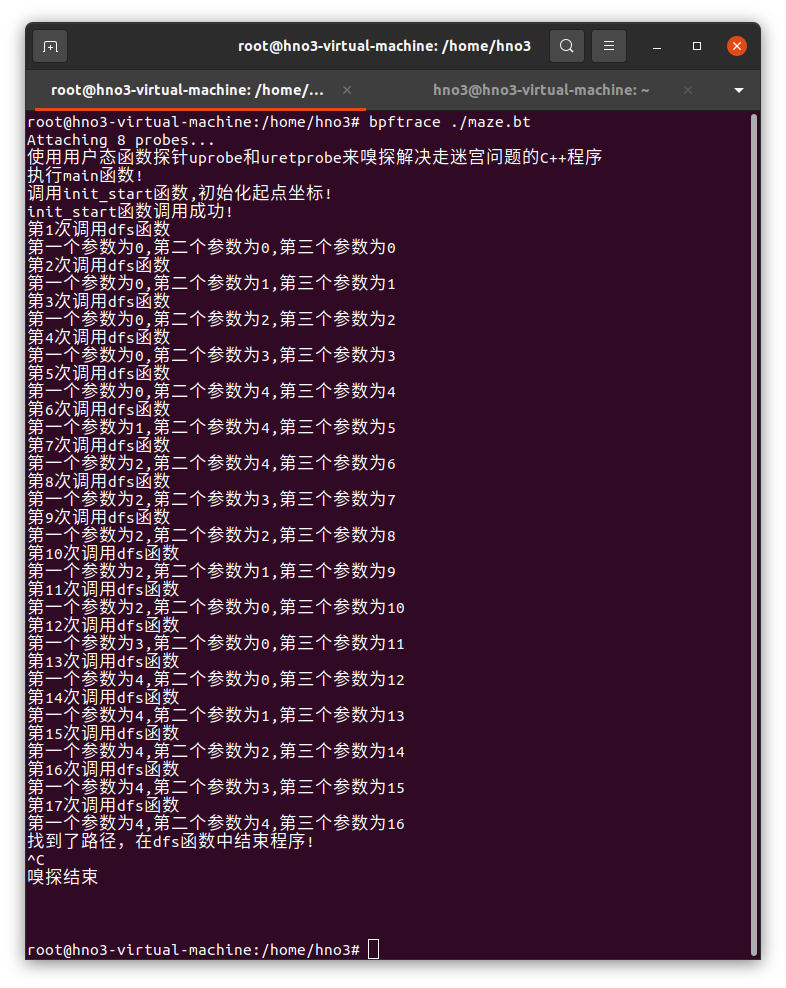


图 4-2-3-3-2

从图中我们可以看到，嗅探程序嗅探到了17次的dfs函数的执行，并打印出参数信息，最后嗅探到了在dfs函数中调用的stop函数的执行，没有嗅探到main函数的返回，说明确实找到了路径，并打印出相关信息。

1. 如果迷宫没有路通往终点，那么会显示执行和搜索的信息，但不会触发stop探针：

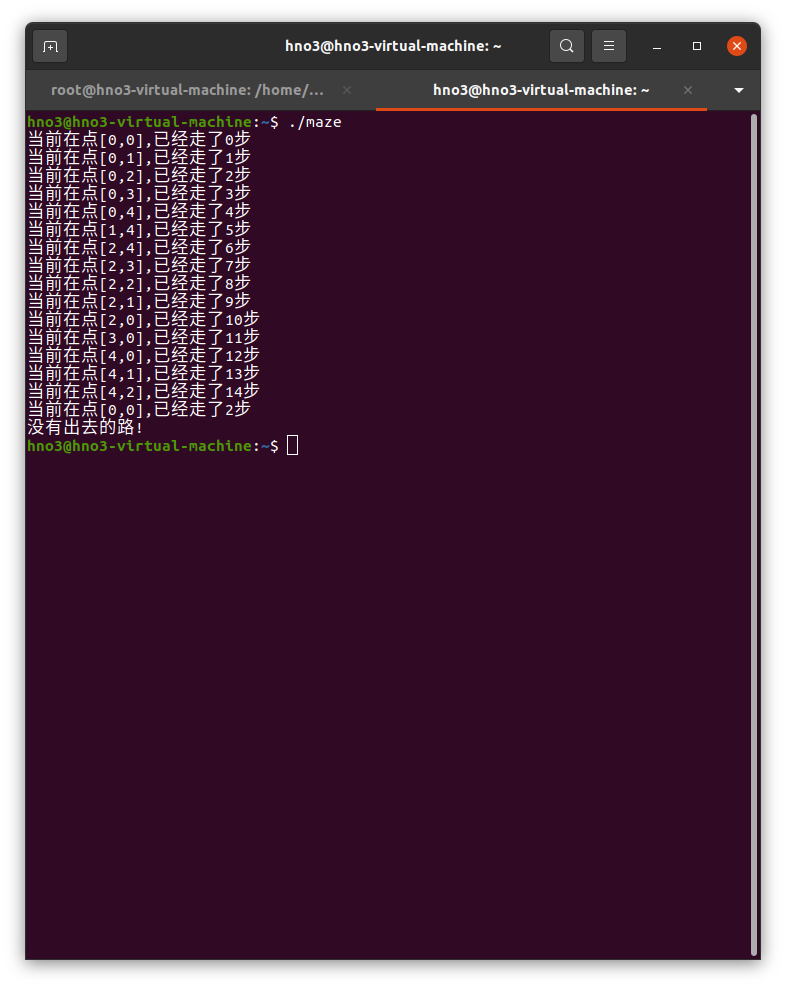


图 4-2-3-3-3

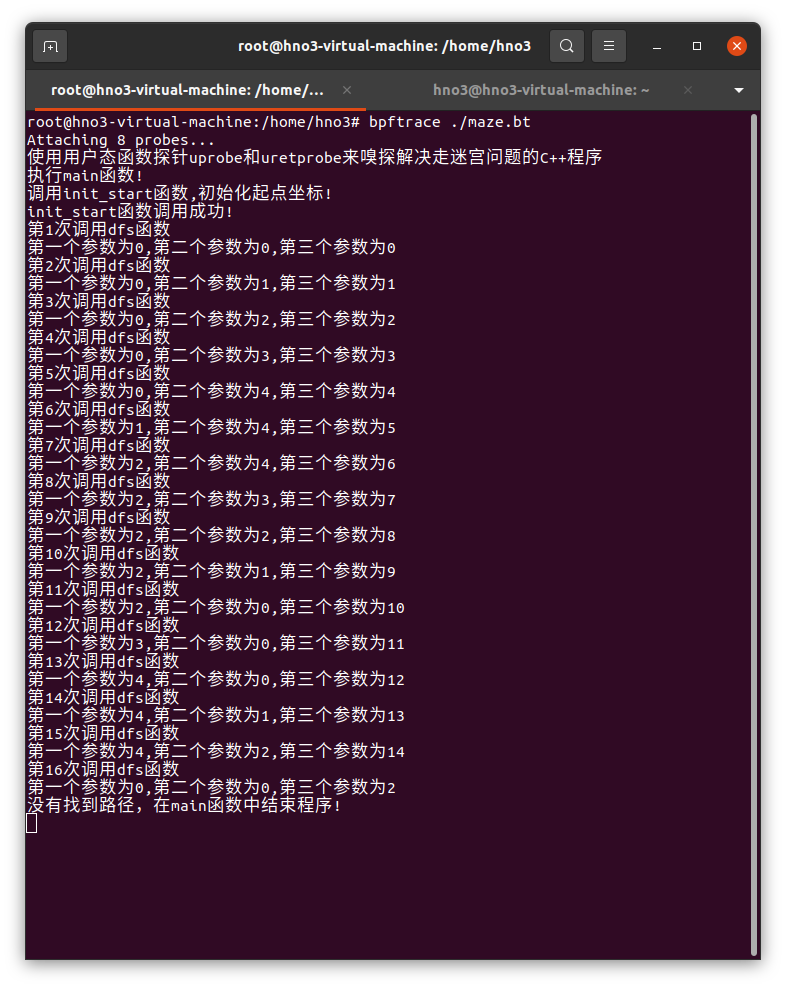


图 4-2-3-3-4

从图中我们可以看到，嗅探程序嗅探到了17次的dfs函数的执行，并打印出参数信息，最后只嗅探到了main函数中的返回，没有嗅探到在dfs函数中调用的stop函数的执行，说明没有找到路径，并打印出相关信息。

4.3 静态插桩命令的使用

4.3.1基础知识

4.3.1.1 静态插桩命令定义

插桩法是内核常用的一种调试手段，利用代码中插桩，执行到此时执行对应的钩子函数（hook）来达到调试的目的。

1. Tracepoint

tracepoint 利用gcc的特性来实现的，在内核代码中提前做了插桩，代码运行到跟踪点（桩）上注册了钩子函数（hook），那么就会执行钩子函数达到调试的目的。

1. usdt

USDT(user-level statically defined tracing)，提供了用户空间版的跟踪点机制，为应用程序添加USDT后，可使用bpftrace对跟踪点进行探测

4.3.1.2 静态插桩命令格式

1. Tracepoint
2. tracepoint:tracepoint\_name

tracepoint\_name是跟踪点的全名，包括冒号，它将冒号分隔成自己的类和事件名称的层次结构。

1. usdt
2. usdt:binary\_path:probe\_name
3. usdt:library\_path:probe\_name
4. usdt:binary\_path:probe\_namespace:probe\_name
5. usdt:library\_path:probe\_namespace:probe\_name

usdt可以通过提供完整路径来检测可执行binaries或shared库。 probe\_name是binary中的USDT探针名称

4.3.2基本步骤

4.3.2.1 tracepoint命令的使用

使用了内核的静态探测点，对于参数的访问方式为args->，可以使用-lv命令查看参数

1. bpftrace -lv tracepoint:syscalls:sys\_enter\_openat
2. root@ecs-c845:~# bpftrace -lv tracepoint:syscalls:sys\_enter\_openat
3. tracepoint:syscalls:sys\_enter\_openat
4. int \_\_syscall\_nr;
5. int dfd;
6. const char \* filename;
7. int flags;
8. umode\_t mode;

在这里args是一个指针，指向该tracepoint的参数。这个结构时由bpftrace根据tracepoint信息自动生成的。

可以使用args->filename进行参数的打印输出。

1. bpftrace -e 'tracepoint:syscalls:sys\_enter\_openat {@[comm]=str(args->filename);} END {print(@); clear(@);}'

这个是tracepoint探针类型(内核静态跟踪)，当进入openat()系统调用时执行该探针。comm是内建变量，代表当前进程的名字。其它类似的变量还有pid和tid，分别表示进程标识和线程标识。

4.3.2.2 tracepoint命令的使用结果

在运行tracepoint命令后，新建终端，进入test文件夹创建并打开test.txt文件。随后结束tracepoint命令。创建并打开vim：

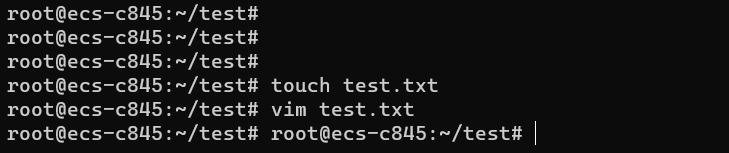


图4-3-2-2-1

追踪到的记录：



图4-3-2-2-2

4.3.2.3 usdt命令的使用

使用usdt需要用户自己编写程序并在程序中加入探测器。这里使用<sys/sdt.h>库进行实现

1. *// 安装工具*
2. sudo apt-get install systemtap-sdt-dev

使用<sys/sdt.h>中的DTRACE\_PROBE、DTRACE\_PROBE1、DTRACE\_PROBE2、DTRACE\_PROBE3和DTRACE\_PROBE4宏声明 SDT 探测器。基于 SDT 探测器的模块名称和函数名称对应于探测器的内核模块和函数。探测器的名称取决于 DTRACE\_PROBEn 宏中指定的名称。如果名称中不包含两个连续下条 (\_)，则探测器的名称与宏中指定的名称相同。如果名称中包含任何两个连续下条，则探测器名称会将该连续下条转换为一个破折号 (-)。

每个 SDT 探测器的参数都是在相应的 DTRACE\_PROBEn 宏引用中指定的参数。参数的数量取决于用于创建该探测器的宏：DTRACE\_PROBE1指定一个参数，DTRACE\_PROBE2指定两个参数，依此类推。声明 SDT 探测器时，可通过不取消引用指针，并且不从探测器参数的全局变量中装入，来将已禁用的探测影响降至最低。

1. #include <sys/sdt.h>
2. #include <sys/time.h>
3. #include <unistd.h>
4. #include <stdio.h>
5. static long
6. myclock() {
7. struct timeval tv;
8. gettimeofday(&tv, NULL);
9. DTRACE\_PROBE2(tracetest, testprobe,  tv.tv\_sec + 0, "hello, word");
10. sleep(1);
11. return tv.tv\_sec;
12. }
13. int main() {
14. for(int i = 0; i < 5; i++) {
15. myclock();
16. }
17. return 0;
18. }

使用该程序可以每隔一秒打印一次当面的时间（以秒为单位），源程序需要编译成可执行文件才可以进行追踪。

1. gcc -o usdt usdt.c

探测命令：

1. bpftrace -e 'usdt:/root/usdt:tracetest:testprobe {printf("now time is: %lld \n", arg0);}'

4.3.2.4 usdt命令的使用结果

执行程序文件：

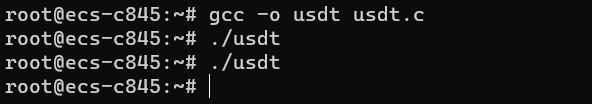


图4-3-2-4-1

Usdt对代码进行监听：

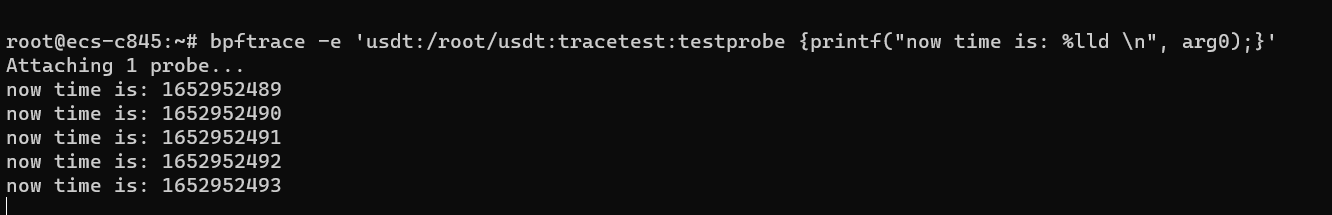


图4-3-2-4-2

4.4基于时间的BPF

4.4.1基于时间的采样——profile

4.4.1.1 profile的语法及意义

Profile会根据给定的频率来采样分析，具体的有4种，分别为：

① profile:ms:rate

以毫秒为单位，常用

② interval:s:rate

以秒为单位，常用于频次较少的采样

③ interval:us:rate

以微秒为单位，由于时间粒度太细一般不会使用

④ interval:hz:rate

以赫兹为单位，即每秒采样多少次，常用

4.4.1.2 profile例子

1. # bpftrace -e 'profile:hz:99 { @[kstack] = count(); }'

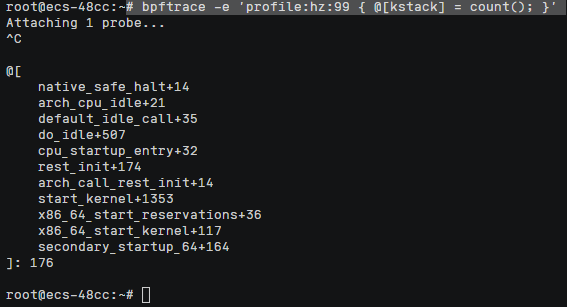


图4-4-1-1

这条命令意为将所有cpu都以99hz的频率采样分析内核调用栈，并将次数统计打印。要注意到99并非随意选取，对于过高的频率，采样过于频繁会影响性能，对于过低的频率，次数过多难以统计；不用100凑整，是为了与cpu频率尽可能错开，避免采样过程正好在多个cpu同步时钟的lockstep阶段。由于所用的云服务器linux系统是单核的，所以只有一个输出。

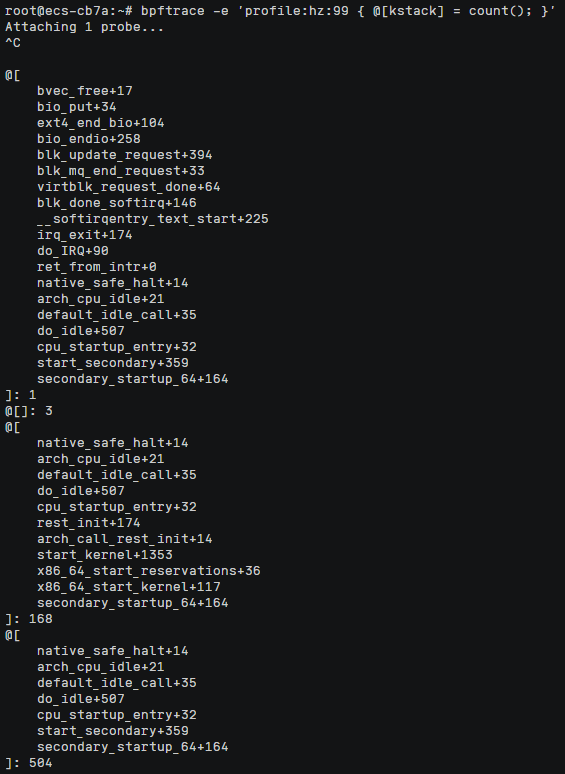


图4-4-1-1

换到一个四核服务器中运行该命令，可以发现在多核系统中调度非常不均衡，cpu2的空闲情况比较严重。

4.4.2基于时间的输出——interval

4.4.2.1Interval的语法及意义

Interval的语法与profile相似，其意义具有明显区别：profile每隔一段时间进行统计，而interval则是统计一段时间内的内容，因此interval在给定的时间结束后会自动输出结果，而profile需要ctrl+c退出示意其停止打点，再输出内容。此外，Interval是只针对一个cpu核的。

4.4.2.2Interval例子

1. # bpftrace -e 'tracepoint:sched:sched\* { @[probe] = count(); } interval:s:5 { exit(); }'

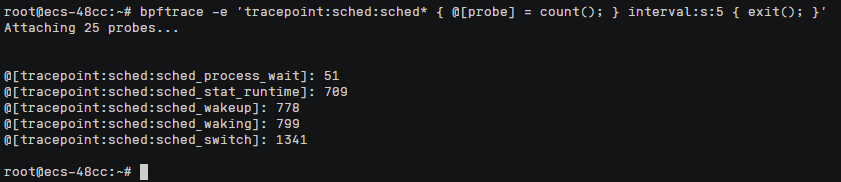


图4-4-2-1

这条命令使用sched探针来统计进程级别的事件，可以看到进程发生交换、唤醒、等待的次数。Interval在这句话中起到一个修饰作用，表明只统计5秒内的时间。

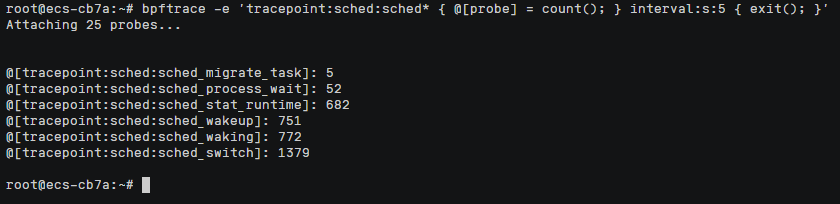


图4-4-2-2

在多核服务器上运行该命令，可以发现多了一个migrate\_task事件，这是多核处理器为了实现负载均衡进行的线程迁移操作。

4.5 事件探针的使用

4.5.1 基础知识

4.5.1.1 事件探针的定义

software探针检测内核软件事件，而hardware探针检测处理器级事件。

4.5.1.2 用户级函数探针的格式

4.5.1.2.1 简介

①内核软件事件探针software语法

1. software: event\_name: count
2. software: event\_name:

event\_name是 Linux 内核提供的预定义软件事件，通常通过 perf 实用程序进行跟踪。它们类似于跟踪点，但只有十几个，它们记录在 perf\_event\_open(2) 手册页中。有以下这些事件：

* cpu-clock 或者 cpu
* task-clock
* page-faults 或者 faults
* context-switches 或者 cs
* cpu-migrations
* minor-faults
* major-faults
* alignment-faults
* emulation-faults
* dummy
* bpf-output

count是探针的触发器，它将为每个计数事件触发一次。如果未提供计数，则使用默认值。

②处理器级事件探针hardware语法

1. hardware: event\_name: count
2. hardware: event\_name:

event\_name是 Linux 内核提供的预定义硬件事件，通常由 perf 实用程序跟踪。它们是使用性能监控计数器 (PMC) 实现的：处理器上的硬件资源。其中大约有十个，它们记录在 perf\_event\_open(2) 手册页中。有以下这些事件：

* cpu-cycles 或者 cycles
* instructions
* cache-references
* cache-misses
* branch-instructions 或者branches
* branch-misses
* bus-cycles
* frontend-stalls
* backend-stall s
* backend-stalls

count是探针的触发器，它将为每个计数事件触发一次。如果未提供计数，则使用默认值。

4.5.1.2.2 示例

①内核软件事件探针software示例

这通过对每 100 个错误的进程名称进行抽样，粗略地计算出谁导致了页面错误。

1. *# bpftrace -e 'software:faults:100 { @[comm] = count(); }'*
2. Attaching 1 probe...
3. ^C
4. @[ls]: 1
5. @[pager]: 2
6. @[locale]: 2
7. @[preconv]: 2
8. @[sh]: 3
9. @[tbl]: 3
10. @[bash]: 4
11. @[groff]: 5
12. @[grotty]: 7
13. @[sleep]: 9
14. @[nroff]: 12
15. @[troff]: 18
16. @[man]: 97

②处理器级事件探针hardware示例

每 1000000 次缓存未命中就会触发一次。这通常表示最后一级缓存 (LLC)。

1. *# bpftrace -e 'hardware:cache-misses:1000000 { @[pid] = count(); }'*

4.5.2 基本步骤

为了便于使用，我们对应地查看内核版本：



图 4-5-2-1

并安装对应的工具：

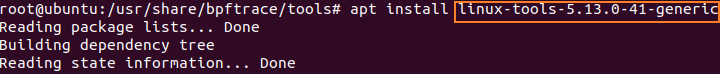


图 4-5-2-2

通过如下命令查看所有的探测指标：

1. *# sudo perf list*

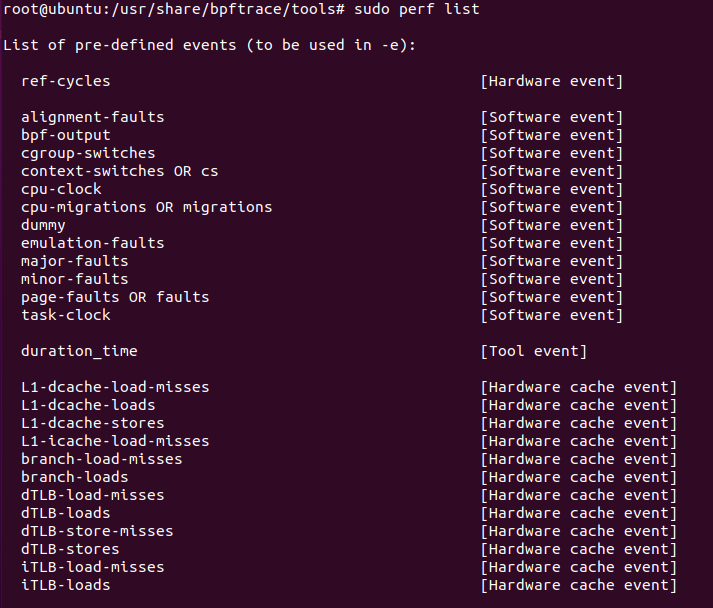


图 4-5-2-3

也可以通过以下命令具体查看某种类型的探测指标，下图展示的是处理器级事件的探测指标：

1. *#*  *bpftrace -lv "hardware.\*"*

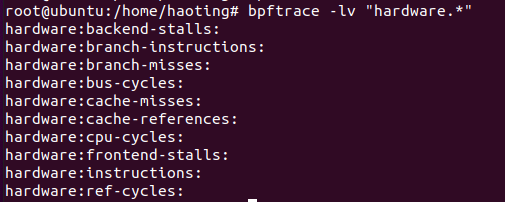


图 4-5-2-4

事件探针的使用，无外乎就是以下几个步骤：

* 选定探测指标
* 写一个相应的嗅探文件，例如test.bt
* 在第一个终端中执行嗅探文件
* 在第二个终端中执行需要进行性能探测的文件，例如a.c
* 在第一个终端中查看嗅探结果
* 按下Ctrl+C结束嗅探

4.5.3.1 性能探测指标

我们选定常见的性能探测指标：

①内核软件级software

* major-faults
* cpu-migrations

②处理器级hardware

* instructions
* branch-misses

4.5.3.2 需要进行性能探测的文件

我们选择探测以下程序的性能：

1. #include<stdio.h>
2. //随机打印 2-n+1 个数（n默认为20）
3. **int** main()
4. {
6. **int** i = 0;
7. **int** n = 20;
8. **do**  ™
9. {
10. i++;
11. **do**
12. {
13. i++;
14. printf("%d\n", i);
15. } **while** (i <= n);
16. } **while**(i<=n);
17. **return** 0;
18. }

4.5.3.3 BPFtrace嗅探程序

1. #!/usr/bin/env bpftrace
2. //初始化探针，可以理解为只要执行嗅探程序，就首先触发，但只会触发一次
3. BEGIN
4. {
5. printf("使用软件事件探针software和硬件事件探针hardware");
6. printf("查看处理器上的软件资源和硬件资源\n");
7. // 上下文切换次数
8. @count\_cpu\_migrations = 0;
10. }
11. //页错误，内存页已经被swap到硬盘上，需要I/O换回
12. software :major-faults:
13. {
14. @[comm] = count();
15. }
17. //嗅探上下文切换次数
18. software :cpu-migrations:
19. {
20. @count\_cpu\_migrations = @count\_cpu\_migrations + 1;
21. printf("上下文切换\n");
22. }

25. //一种探针类型，可以理解为嗅探程序结束时，就触发，但只会触发一次
26. END
27. {
28. printf("上下文共切换%d次\n", @count\_cpu\_migrations);
29. printf("\n嗅探结束\n");
31. clear(@count\_cpu\_migrations);
32. }

4.5.3.4 嗅探结果

切换一次上下文就会触发software: cpu-migrations探针：



图 4-5-3-4-1

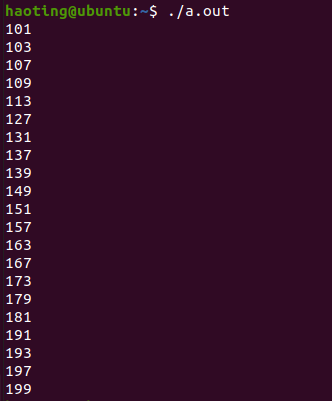


图 4-5-3-4-2

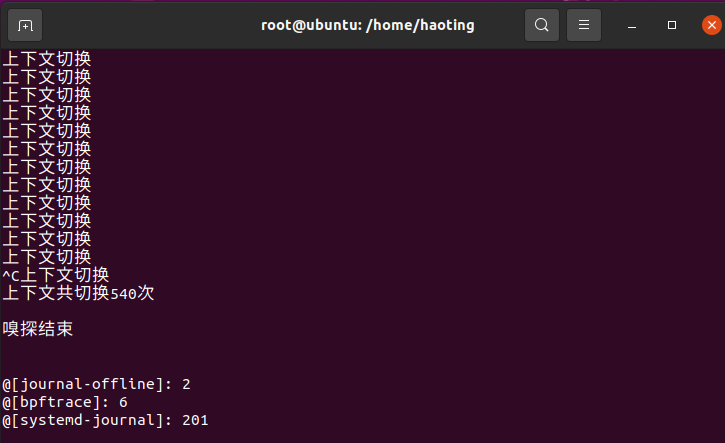


图 4-5-3-4-2

最终我们可以看到，程序执行结束，共切换了540次上下文；其中，software:major-faults探针还检测出journal-offline进程发生了2次页错误，bpftrace进程发送了6次页错误，systemd-journal进程发生了201次页错误。