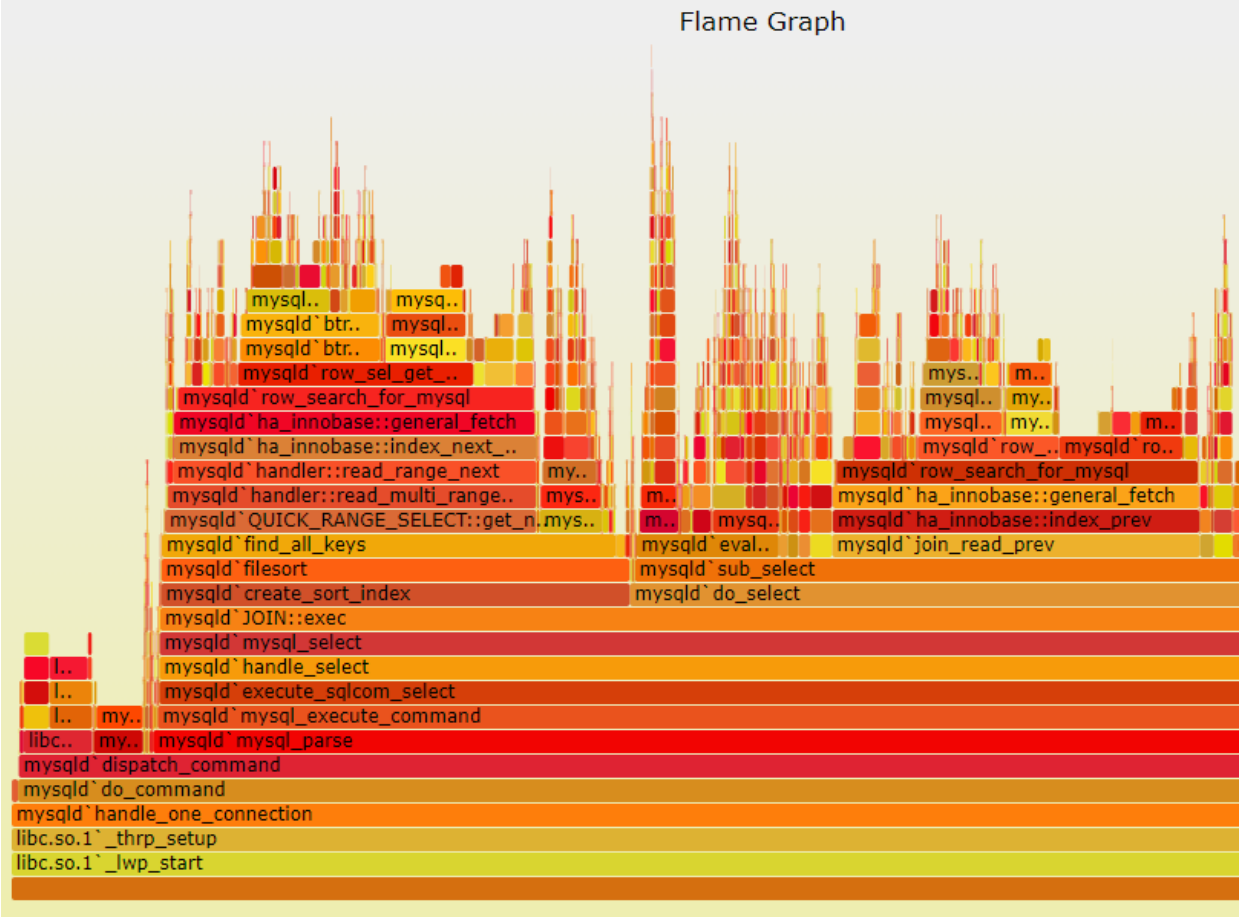
腾讯课堂 零声教育 https://ke.qq.com/course/420945?tuin=137bb271
Darren QQ326873713

主要内容：

- 火焰图能解决什么问题
- 火焰图如何分析cpu性能问题
- 火焰图如分析I/O阻塞问题

网页版本课件：https://www.yuque.com/linuxer/xngi03/dpd36gqgugnn8wtf?singleDoc# 《火焰图生成与分析-课件20230826》

1 火焰图简介



Flame Graph 原图路径：https://queue.acm.org/downloads/2016/Gregg4.svg

火焰图整个图形看起来就像一个跳动的火焰，这就是它名字的由来。
火焰图有以下特征（这里以 on-cpu 火焰图为例）：

- 每一列代表一个调用栈，每一个格子代表一个函数
- 纵轴展示了栈的深度，按照调用关系从下到上排列。最顶上格子代表采样时，正在占用 cpu 的函数。
- 横轴的意义是指：火焰图将采集的多个调用栈信息，通过按字母横向排序的方式将众多信息聚合在一起。需要注意的是它并不代表时间。
- 横轴格子的宽度代表其在采样中出现频率，所以一个格子的宽度越大，说明它是瓶颈原因的可能性就越大。

- 火焰图格子的颜色是随机的暖色调，方便区分各个调用信息。
- 其他的采样方式也可以使用火焰图， on-cpu 火焰图横轴是指 cpu 占用时间，off-cpu 火焰图横轴则代表阻塞时间。
- 采样可以是单线程、多线程、多进程甚至是多 host，进阶用法可以参考附录进阶阅读。

火焰图能做什么：

- 可以分析函数执行的频繁程度(占用cpu时间，或者阻塞的时间)
- 可以分析哪些函数经常阻塞
- 可以分析哪些函数频繁分配内存

以分析程序的性能瓶颈。

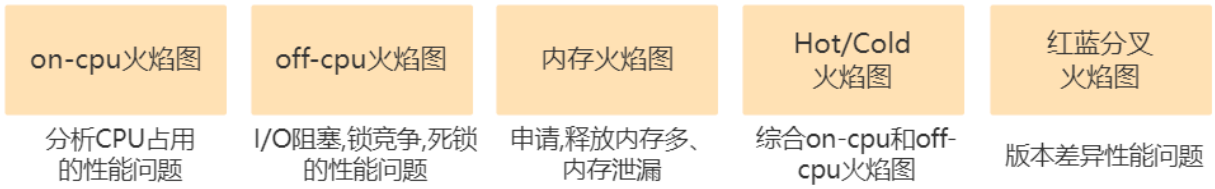
官方博客：https://www.brendangregg.com/flamegraphs.html，火焰图的源资料皆出自该博客。

该博主名称：Brendan Gregg，著作有：

- **性能之巅：洞悉系统、企业与云计算**
- **BPF之巅 洞悉Linux系统和应用性能**

1.1 火焰图类型

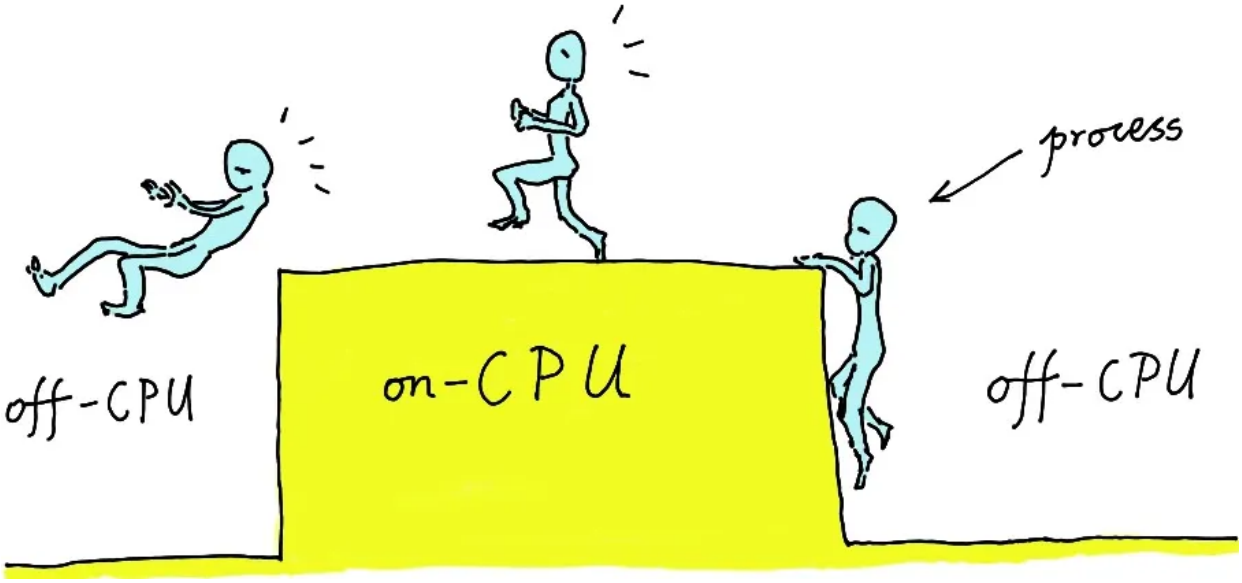
常见的火焰图类型有 On-CPU，Off-CPU，还有 Memory，Hot/Cold，Differential 等等。它们有各自适合处理的场景。

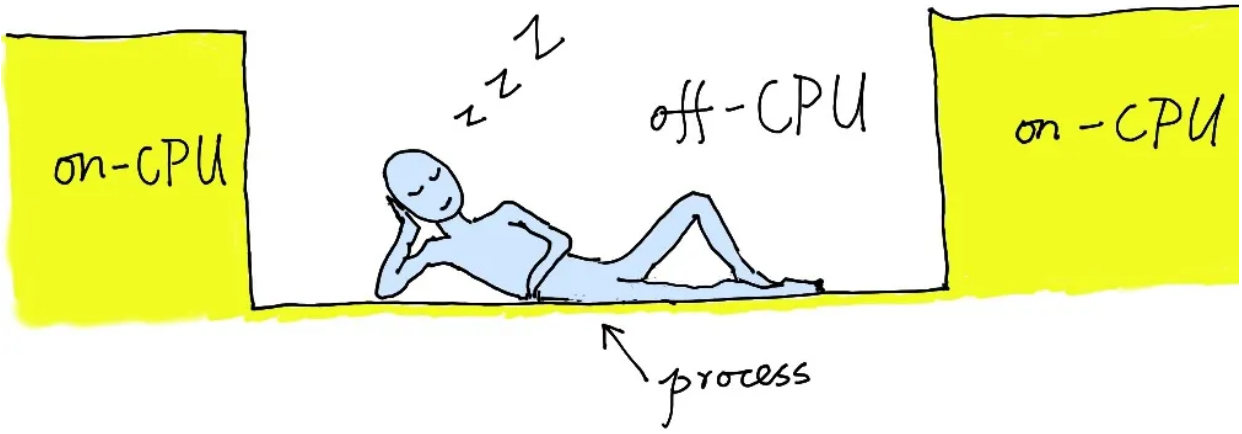


火焰图类型	横轴含义	纵轴含义	解决问题	采样方式
on-cpu火焰图	cpu占用时间	调用栈	找出cpu占用高的问题函数；分析代码热路径	固定频率**采样cpu调用栈**
off-cpu火焰图	阻塞时间	调用栈	i/o、网络等阻塞场景导致的性能下降；锁竞争、死锁导致的性能下降问题	固定频率**采样阻塞事件调用栈**
内存火焰图	内存申请/释放函数调用次数	调用栈	内存泄漏问题；内存占用高的对象/申请内存多的函数；虚拟内存或物理内存泄漏问题	有四种方式：跟踪malloc/free；跟踪brk；跟踪mmap；跟踪页错误

火焰图类型	横轴含义	纵轴含义	解决问题	采样方式
Hot/Cold 火焰图	on-cpu和off-cpu综合展示	调用栈	需要结合cpu占用以及阻塞分析的场景；off-cpu火焰图无法直观判断的场景	on-cpu火焰图和off-cpu火焰图结合
红蓝分叉火焰图	红色表示上升，蓝色表示下降	调用栈	处理不同版本性能回退问题	对比两个on-cpu火焰图

1.2 什么时候使用 On-CPU 火焰图? 什么时候使用 Off-CPU 火焰图呢?





取决于当前的瓶颈到底是什么：

- 如果是 CPU 则使用 On-CPU 火焰图，（先看cpu是不是快到百分百），

- 如果是 IO 或锁则使用 Off-CPU 火焰图（如果cpu占用率不高，就需要用off-cpu）.
- 如果无法确定, 那么可以通过压测工具来确认：
  - 通过压测工具看看能否让** CPU 使用率趋于饱和**, 如果能那么使用 On-CPU 火焰图
  - 如果不管怎么压, **CPU 使用率始终上不来**, 那么多半说明程序被 IO 或锁卡住了, 此时适合使用 Off-CPU 火焰图.
- 如果还是确认不了, 那么不妨 On-CPU 火焰图和 Off-CPU 火焰图都搞搞, 正常情况下它们的差异会比较大, 如果两张火焰图长得差不多, 那么通常认为 CPU 被其它进程抢占了.

1.3 火焰图分析技巧

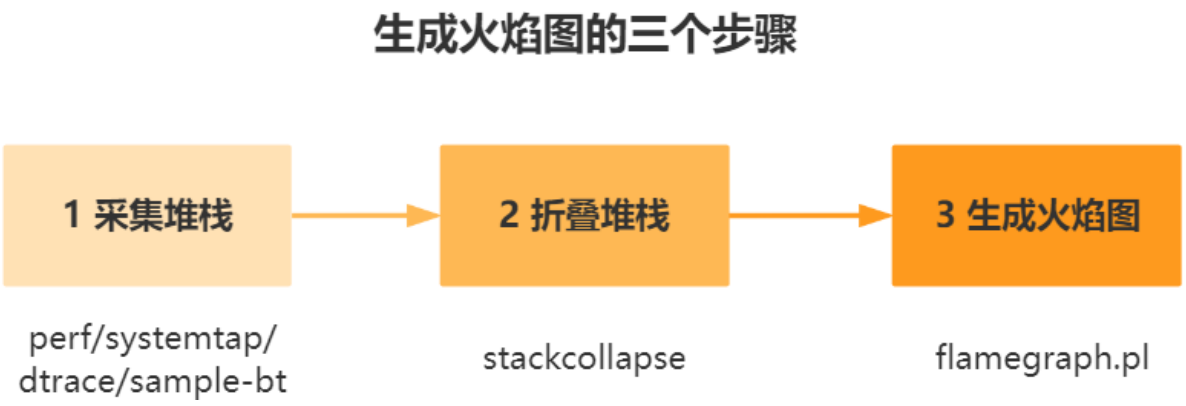
1. 纵轴代表调用栈的深度（栈桢数），用于表示函数间调用关系：下面的函数是上面函数的父函数。
2. 横轴代表调用频次，一个格子的宽度越大，越说明其可能是瓶颈原因。
3. 不同类型火焰图适合优化的场景不同：
   1. **比如 on-cpu 火焰图适合分析 cpu 占用高的问题函数**；
   2. off-cpu 火焰图适合解决阻塞和锁抢占问题。
4. 无意义的事情：横向先后顺序是为了聚合，跟函数间依赖或调用关系无关；火焰图各种颜色是为方便区分，本身不具有特殊含义
5. 多练习：进行性能优化有意识的使用火焰图的方式进行性能调优（如果时间充裕）

2 如何绘制火焰图-- 采集需要root权限

2.1 生成火焰图的流程

2.1.1 生成火焰图的三个步骤

生成和创建火焰图需要如下三个步骤



流程	描述	脚本
采集堆栈		
cpu tick		
1000HZ		

流程	描述	脚本
使用**perf**/systemtap/dtrace等工具抓取程序的运行堆栈	perf/systemtap/dtrace	
折叠堆栈		
堆栈采集的时间，结束时间等等	trace 工具抓取的系统和程序运行每一时刻的堆栈信息, 需要对他们进行分析组合, 将重复的堆栈累计在一起, 从而体现出负载和关键路径	FlameGraph中的stackcollapse程序
生成火焰图 （展示信息用的）	分析 stackcollapse 输出的堆栈信息生成火焰图	flamegraph.pl

2.2 安装火焰图必备工具

2.2.1 安装火焰图FlameGraph脚本

Brendan D. Gregg 的 Flame Graph 工程实现了**一套生成火焰图的脚本。**Flame Graph 项目位于 GitHub上

> https://github.com/brendangregg/FlameGraph

当GitHub网络不通畅的时候可以使用码云的链接：

> git clone https://gitee.com/mirrors/FlameGraph.git

用 git 将其 clone下来

不同的 trace 工具抓取到的信息不同, 因此 Flame Graph 提供了一系列的 stackcollapse（堆栈折叠） 工具.

stackcollapse	描述
stackcollapse.pl	for DTrace stacks
stackcollapse-**perf.pl**	for Linux **perf_events** “perf script” output
stackcollapse-pmc.pl	for FreeBSD pmcstat -G stacks
stackcollapse-**stap.pl**	for SystemTap stacks
stackcollapse-instruments.pl	for XCode Instruments
stackcollapse-vtune.pl	for Intel VTune profiles
stackcollapse-ljp.awk	for Lightweight Java Profiler
stackcollapse-jstack.pl	for Java jstack(1) output
stackcollapse-gdb.pl	for gdb(1) stacks

stackcollapse	描述
stackcollapse-go.pl	for Golang pprof stacks
stackcollapse-vsprof.pl	for Microsoft Visual Studio profiles

查看帮助
./FlameGraph/flamegraph.pl -h

2.2.2 安装火焰图数据采集工具perf

系统级性能优化通常包括两个阶段：性能剖析（**performance profiling**）和代码优化。

- 性能剖析的目标是寻找性能瓶颈，查找引发性能问题的原因及热点代码。
- 代码优化的目标是针对具体性能问题而优化代码或编译选项，以改善软件性能。一般在工作中比较关心的是性能瓶颈，特别是算法。

当在系统全功能启动的时候，算法一般需要将设备的性能用到极限，而在这个过程中不免出现各类性能上的瓶颈，此时需要分析自身的一些性能瓶颈在什么地方就可以用到专门的性能分析工具perf。

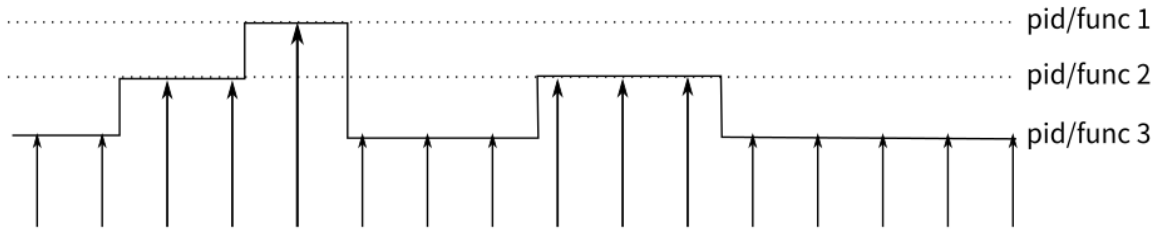
perf 命令(performance profiling的缩写), 它是 Linux 系统原生提供的性能分析工具, 会返回 CPU 正在执行的函数名以及调用栈(stack)

具体用法：

- **perf Examples** https://www.brendangregg.com/perf.html
- **Linux kernel profiling with perf** https://perf.wiki.kernel.org/index.php/Tutorial
- Linux Performance (brendangregg.com) https://www.brendangregg.com/linuxperf.html

perf的原理是这样的：每隔一个固定的时间，就在CPU上（每个核上都有）产生一个中断，在中断上看看，当前是哪个pid，哪个函数，然后给对应的pid和函数加一个统计值，这样，我们就知道CPU有百分几的时间在某个pid，或者某个函数上了。这个原理图示如下：

1秒采集99次，1秒采集1000次（采样次数越高容易影响程序性能）



很明显可以看出，这是一种采样的模式，预期：运行时间越多的函数，被时钟中断击中的机会越大，从而推测，**那个函数（或者pid等）的CPU占用率就越高。**

这种方式可以推广到各种事件，比如ftrace的事件，你可以在这个事件发生的时候上来冒个头，看看击中了谁，然后算出分布，我们就知道谁会引发特别多的那个事件了。

当然，如果某个进程运气特别好，它每次都刚好躲过你发起探测的位置**（这时候就需要考虑提高采样频率）**，你的统计结果可能就完全是错的了。这是所有采样统计都有可能遇到的问题了。

1. 安装perf

安装需要root权限（比如Ubuntu通过sudo su切换到root权限），**perf 采集的时候也需要root的权限**

```
# apt install linux-tools-common
```

2. 测试perf是否可用

```
# perf record -F 99 -a -g -- sleep 10
```

如果报错

> WARNING: perf not found for kernel **4.15.0-48**
>
> You may need to install the following packages for this specific kernel:
> linux-tools-4.15.0-48-generic
> linux-cloud-tools-4.15.0-48-generic
>
> You may also want to install one of the following packages to keep up to date:
> linux-tools-generic
> linux-cloud-tools-generic
>
> apt install linux-tools-generic
> apt install linux-cloud-tools-generic

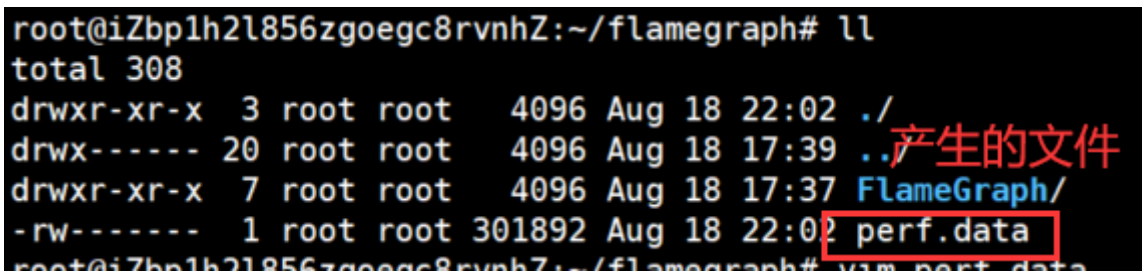
则需要安装linux-tools-generic和linux-cloud-tools-generic，但需选择对应的版本，比如提示的是**4.15.0-48，则我们安装4.15.0-48版本。**

```
# apt-get install linux-tools-4.15.0-48-generic linux-cloud-tools-4.15.0-48-generic
linux-tools-generic linux-cloud-tools-generic
```

再次测试

```
# perf record -F 99 -a -g -- sleep 10
```

如果没有报错则在执行的目录产生`perf.data`



3. perf常用命令

查看帮助文档，perf功能非常强大，我们这里只关注record和report功能，record和report也可以继续通过二级命令查询帮助文档。

perf -h

```
root@ceph-admin:/home/lqf/flamegraph# perf -h

 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated
code
   archive         Create archive with object files with build-ids found in
perf.data file
   bench           General framework for benchmark suites
   buildid-cache   Manage build-id cache.
    ....
   lock            Analyze lock events
   mem             Profile memory accesses
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the profile
```

常用的五个命令：

- perf list：查看当前软硬件环境支持的性能事件
- perf stat：分析指定程序的性能概况
- perf top：实时显示系统/进程的性能统计信息
- **perf record**：记录一段时间内系统/进程的性能事件
- **perf report**：读取perf record生成的perf.data文件，并显示分析数据（生成火焰图用的采集命令）

perf record -h

常用命令

- -e： 指定性能事件（可以是多个，用,分隔列表）
- **-p**： 指定待分析进程的 pid（可以是多个，用,分隔列表， nginx， -p 100,101,102,103）
- -t： 指定待分析线程的 tid（可以是多个，用,分隔列表）
- -u： 指定收集的用户数据，uid为名称或数字
- -**a**：从所有 CPU 收集系统数据
- **-g**：开启 call-graph (stack chain/backtrace) 记录（backtrace调试功能的实现原理就是**利用函数调用栈中的信息来追踪程序执行的路径和调用关系。**）
- -C： 只统计指定 CPU 列表的数据，如：0,1,3或1-2
- -r： perf 程序以SCHED_FIFO实时优先级RT priority运行这里填入的数值越大，进程优先级越高（即 nice 值越小）
- -c： 事件每发生 count 次采一次样

- **-F** ： 每秒采样 n 次
- **-o <output.data>**：指定输出文件output.data，默认输出到**perf.data**

perf report -h

- -i, --input　input file name，可以指定要分析的文件名，默认perf.data

2.3 生成火焰图实践示例

我们需要先写个简单的测试程序，分main、func_a、func_b、func_c、func_d 5个函数：
main: 调用func_a、func_b、func_c
func_a：调用func_d
这里的目的是演示宽度对应函数占用的cpu时间。

2.3.0 编译和执行测试程序

src-flamegraph\test\test.c

```
#include <stdio.h>

void func_d()
{
    for (int i = 5 * 10000; i--;);
}
void func_a()
{
    for (int i = 10 * 10000; i--;);
    func_d();
}
void func_b()
{
    for (int i = 20 * 10000; i--;);
}
void func_c()
{
    for (int i = 35 * 10000; i--;);
}
int main(void)
{
    printf("main into\n");
    while (1)
    {
        for (int i = 30 * 10000; i--;);
        func_a();
        func_b();
        func_c();
    }
    printf("main end\n");
    return 0;
}
```

该程序用于生成测试程序。

编译：gcc -o test test.c

执行：./test

2.3.1 perf 采集数据

通过top指令查看test的pid。

```
# perf record -F 99 -p 5000 -g -- sleep 30
```

perf record 表示采集系统事件, 没有使用 -e 指定采集事件, 则默认采集 cycles(即 CPU clock 周期), -F 99 表示每秒 99 次, **-p 5000是进程号**, 即对哪个进程进行分析, -g 表示记录调用栈, sleep 30 则是持续 30 秒.

> -F 指定采样频率为 99Hz(每秒99次), 如果 99次 都返回同一个函数名, 那就说明 CPU 这一秒钟都在执行同一个函数, 可能存在性能问题.

运行后会产生一个庞大的文本文件. 如果一台服务器有 16 个 CPU, 每秒抽样 99 次, 持续 30 秒, 就得到 47,520 个调用栈, 长达几十万甚至上百万行.

为了便于阅读, perf record 命令可以统计每个调用栈出现的百分比, 然后从高到低排列.

```
# perf report -n --stdio
```

通过 perf report 命令可以展示采样记录，大概介绍下面板参数

- Samples：采样个数
- Event count：系统总共发生的事件数
- Symbol：函数名，其中 [.] 表示用户空间函数，[k] 表示内核函数
- Shared Objec：函数所在的共享库或所在的程序
- Command：进程名
- Self：该函数的 CPU 使用率
- Children：该函数的子函数的 CPU 使用率

```
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 990  of event 'cpu-clock'
# Event count (approx.): 9999999900
#
# Children      Self       Samples  Command  Shared Object     Symbol
# ........  ........  ............  .......  ..................  ....................
#
   100.00%     0.00%             0  test     [unknown]         [.] 0x0836258d4c544
           |
```

```
          ---0x836258d4c544155
             __libc_start_main
             main
             |
             |--32.63%--func_c
             |
             |--20.10%--func_b
             |
              --16.06%--func_a
                        |
                         --5.66%--func_d

100.00%     0.00%             0  test     libc-2.23.so      [.] __libc_start_ma
          |
          ---__libc_start_main
             main
             |
             |--32.63%--func_c
             |
             |--20.10%--func_b
             |
              --16.06%--func_a
                        |
                         --5.66%--func_d

100.00%    31.21%           309  test     test              [.] main
          |
          |--68.79%--main
          |          |
          |          |--32.63%--func_c
          |          |
          |          |--20.10%--func_b
          |          |
          |           --16.06%--func_a
          |                     |
          |                      --5.66%--func_d
          |
           --31.21%--0x836258d4c544155
                     __libc_start_main
                     main

 32.63%    32.63%           323  test     test              [.] func_c
          |
          ---0x836258d4c544155
             __libc_start_main
             main
             func_c

 20.10%    20.10%           199  test     test              [.] func_b
          |
          ---0x836258d4c544155
             __libc_start_main
```

```
                main
                func_b

    16.06%    10.40%          103  test     test             [.] func_a
              |
              |--10.40%--0x836258d4c544155
              |          __libc_start_main
              |          main
              |          func_a
              |
               --5.66%--func_a
                         func_d

     5.66%     5.66%           56  test     test             [.] func_d
              |
              ---0x836258d4c544155
                 __libc_start_main
                 main
                 func_a
                 func_d
```

2.3.2 折叠堆栈

1. 首先用**perf script** 工具对 perf.data 进行解析

```
# 生成折叠后的调用栈
perf script -i perf.data &> perf.unfold
```

比如:

```
test 32321 161641.198189:   10101010 cpu-clock:
                   553 main (/mnt/hgfs/vip/20230215-flamegraph/src-flamegraph/test/test)
                 20840 __libc_start_main (/lib/x86_64-linux-gnu/libc-2.23.so)
         24e258d4c544155 [unknown] ([unknown])

test 32321 161641.208161:   10101010 cpu-clock:
                   50e func_b (/mnt/hgfs/vip/20230215-flamegraph/src-flamegraph/test/test)
                   56e main (/mnt/hgfs/vip/20230215-flamegraph/src-flamegraph/test/test)
                 20840 __libc_start_main (/lib/x86_64-linux-gnu/libc-2.23.so)
         24e258d4c544155 [unknown] ([unknown])

test 32321 161641.218136:   10101010 cpu-clock:
                   52b func_c (/mnt/hgfs/vip/20230215-flamegraph/src-flamegraph/test/test)
                   578 main (/mnt/hgfs/vip/20230215-flamegraph/src-flamegraph/test/test)
                 20840 __libc_start_main (/lib/x86_64-linux-gnu/libc-2.23.so)
         24e258d4c544155 [unknown] ([unknown])

test 32321 161641.228109:   10101010 cpu-clock:
                   556 main (/mnt/hgfs/vip/20230215-flamegraph/src-flamegraph/test/test)
                 20840 __libc_start_main (/lib/x86_64-linux-gnu/libc-2.23.so)
         24e258d4c544155 [unknown] ([unknown])
```

记录了还是调用栈情况。

2. 然后将解析出来的信息存下来, 供生成火焰图

用 **stackcollapse-perf.pl** 将 perf 解析出的内容 **perf.unfold** 中的符号进行折叠 :

```
# 生成火焰图需要的统计信息
# ./FlameGraph/stackcollapse-perf.pl perf.unfold &> perf.folded
```

./FlameGraph/stackcollapse-perf.pl perf.unfold &> perf.folded

2.3.3 生成火焰图

最后生成 svg 图-绘制火焰图

```
# ./FlameGraph/flamegraph.pl perf.folded > test_oncpu.svg
```

我们也可以使用管道将上面的**流程简化为一条命令**

```
# perf script  | ./FlameGraph/stackcollapse-perf.pl | ./FlameGraph/flamegraph.pl > test_oncpu.svg
```

perf script默认是输入perf.data，如果需要指明输入数据用perf script -i xxx

2.3.4 解析火焰图含义

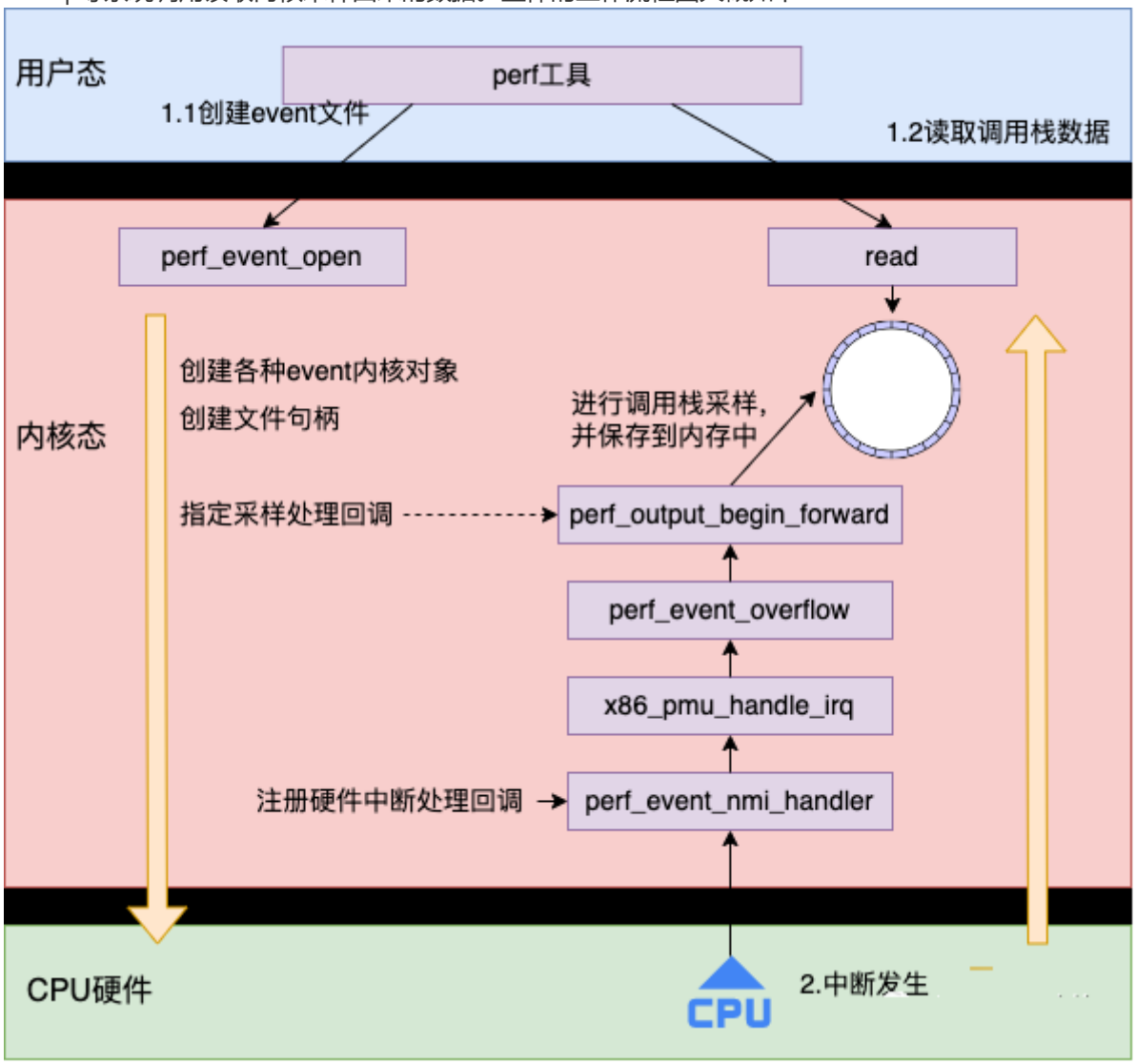
最后就可以用浏览器打开火焰图进行分析啦.
火焰图是基于 stack 信息生成的 SVG 图片, 用来展示 CPU 的调用栈。

- y 轴表示调用栈, 每一层都是一个函数. **调用栈越深, 火焰就越高**, 顶部就是正在执行的函数, 下方都是它的父函数.
- x 轴表示抽样数, 如果一个函数在 x 轴占据的宽度越宽, 就表示它被抽到的次数多, 即执行的时间长. 注意, x 轴不代表时间, 而是所有的调用栈合并后, 按字母顺序排列的.
- 火焰图就是看顶层的哪个函数占据的宽度最大. 只要有 “平顶”(plateaus), 就表示该函数可能存在性能问题。
- 颜色没有特殊含义, 因为火焰图表示的是 CPU 的繁忙程度, 所以一般选择暖色调.

2.4 内核工作过程-补充知识

内核是如何工作的。

perf在采样的过程大概分为两步，一是调用 perf_event_open 来打开一个 event 文件，而是调用 read、mmap等系统调用读取内核采样回来的数据。整体的工作流程图大概如下



更多内容参考：https://mp.weixin.qq.com/s/A19RlLhSgbzw8UU4p1TZNA

3 互动性

火焰图是 SVG 图片，可以与用户互动。

范例：https://queue.acm.org/downloads/2016/Gregg5.svg

（1）鼠标悬浮

火焰的每一层都会标注函数名，鼠标悬浮时会显示完整的函数名、抽样抽中的次数、占据总抽样次数的百分比。下面是一个例子。

> mysqld'JOIN::exec (272,959 samples, 78.34 percent)

off-cpu是否也是如此?

（2）点击放大

在某一层点击，火焰图会水平放大，该层会占据所有宽度，显示详细信息。

左上角会同时显示"Reset Zoom"，点击该链接，图片就会恢复原样。

（3）搜索

按下 Ctrl + F 会显示一个搜索框，用户可以输入关键词或正则表达式，所有符合条件的函数名会高亮显示。

4 局限

两种情况下，无法画出火焰图，需要修正系统行为。

（1）调用栈不完整

当调用栈过深时，某些系统只返回前面的一部分（比如前10层）。

（2）函数名缺失

有些函数没有名字，编译器只用内存地址来表示（比如匿名函数）有些函数没有名字，有些函数在编译时被**优化**了，在火焰图上也不会体现，lambda表达式也有这种问题，导致跟踪困难。。

inline能否分析。

gcc -O -o test2 test2.c -lpthread

```
// gcc编译inline函数报错: 未定义的引用 https://blog.csdn.
inline void func_c2()
{                              -O
    for (int i = 35 * 10000; i--;); // 35
}

void  *thr_fn( void  *arg)
{
    func_c2();
     printf( "sub thread into\n");
    while (1)  // 100
    {
```

先不要做编译优化，很多函数被优化后，是分析不出的。

5 实践-ZeroMQ REQ-REP模型测试

测试两种情况

- hwserver不休眠直接返回 on-cpu火焰图
- hwserver休眠10ms返回 off-cpu火焰图

需要先安装zeromq，参考链接：https://www.yuque.com/docs/share/91aae8cf-f3bb-468f-82d6-3ebefd0e23ad?# 《1-2 ZMQ编译安装和测试》

测试程序：**src-flamegraph\libzmq-test**

请求（REQ）响应（REP）模型, 使用cmake编译。

- 客户端：hwclient.c 请求（REQ）
- 服务端：hwserver .c 响应（REP） 服务端以不同的参数启动
  - ./hwserver 默认不做sleep或者循环占用cpu的模拟
  - ./hwserver 1 收到请求休眠1ms再应答
  - ./hwserver 2 收到请求先做for循环(模拟占用cpu的任务)再应答

重点内容：

- on-cpu和offer-cpu火焰图的生成方式
- on-cpu火焰图分析cpu密集计算的性能问题
- on-cpu火焰图不能分析阻塞导致的性能问题
- off-cpu火焰图分析阻塞导致的性能问题

5.1 on-cpu火焰图

先启动服务端

./hwserver

再启动客户端

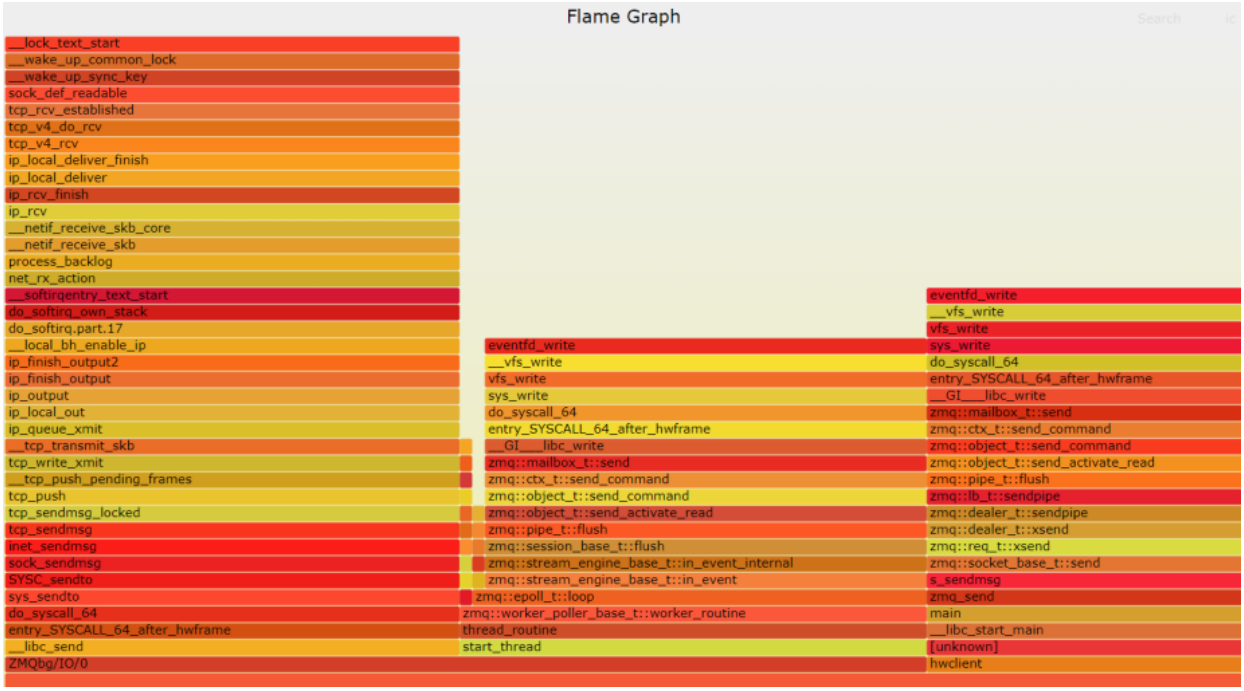
./hwclient

5.1.1 使用perf采集

```
# perf record -F 99 -p 34631 -g -- sleep 120

perf record -F 99 -p 24523 -g -- sleep 10
```

5.1.2 生成火焰图

```
# perf script | ./FlameGraph/stackcollapse-perf.pl | ./FlameGraph/flamegraph.pl > zeromq-req-rep-1-oncpu.svg
```

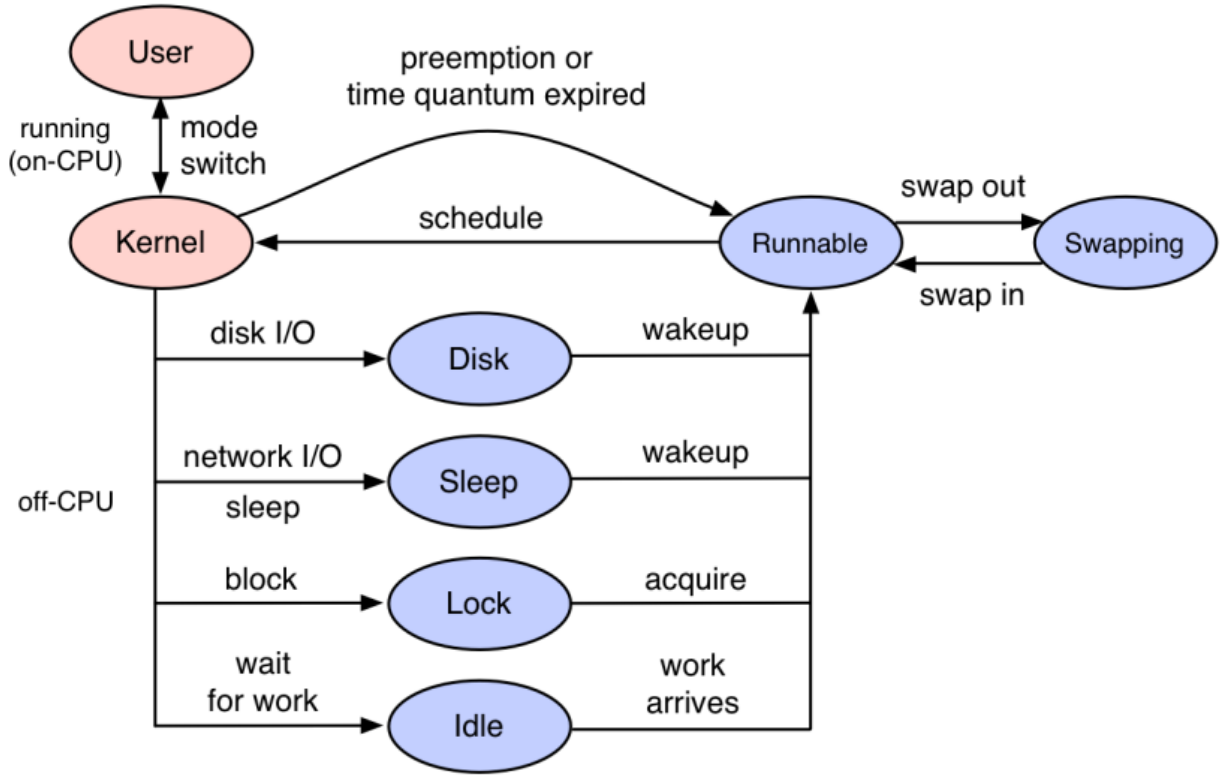


5.2 off-cpu火焰图

https://www.brendangregg.com/blog/2015-02-26/linux-perf-off-cpu-flame-graph.html

Off-CPU Analysis

https://www.brendangregg.com/offcpuanalysis.html



5.2.1 使能sched_schedstats统计

需要在root权限下使能，一定要记得使能：

```
# echo 1 > /proc/sys/kernel/sched_schedstats
```

5.2.2 使用perf采集

需要注意，当阻塞的次数比较多的时候，采集的数据量非常大，所以需要注意采集时间（一般情况30秒左右差不多了）。

-p 是对应要采集程序的进程ID

```
# perf record -e sched:sched_stat_sleep -e sched:sched_switch -e
sched:sched_process_exit -p 23509  -g -o perf.data.raw sleep 30

# perf inject -v -s -i perf.data.raw -o perf.data


perf record -e sched:sched_stat_sleep -e sched:sched_switch -e
sched:sched_process_exit -p 24523  -g -o perf.data.raw sleep 10

perf inject -v -s -i perf.data.raw -o perf.data
```

这里通过perf分别记录：

- **sched:sched_stat_sleep：进程主动放弃 CPU 而进入睡眠的等待事件**
- **sched:sched_switch：进程由于I/O和锁等待等原因被调度器切换而进入睡眠的等待事件**

- **sched:sched_process_exit：进程的退出事件**

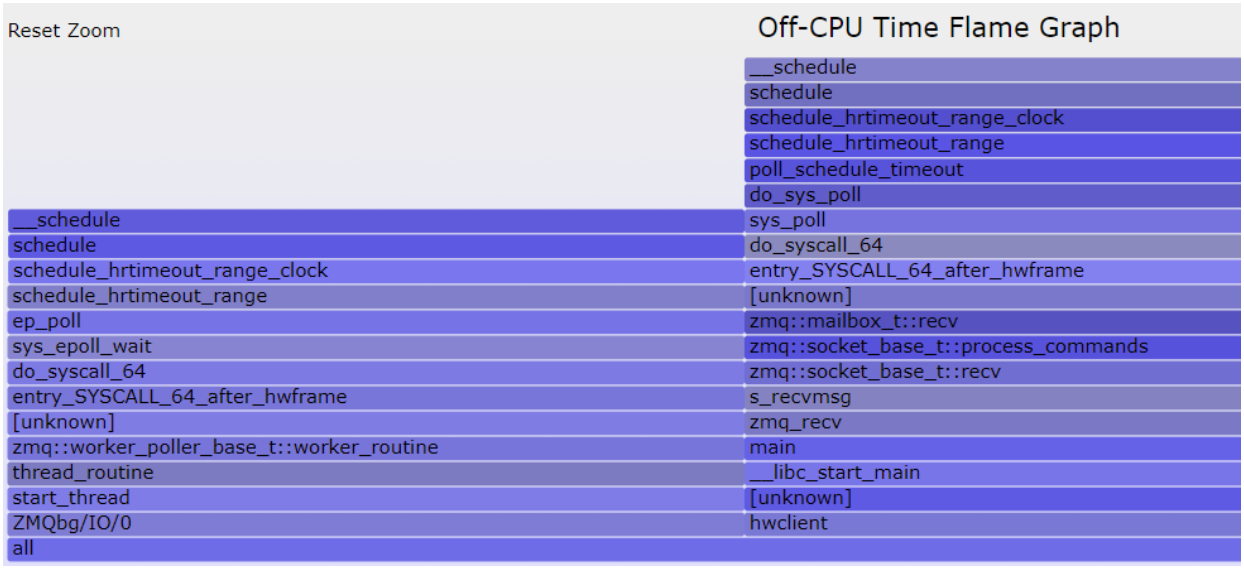
perf inject -v -s：

- -v：表示启用详细模式，会输出更多的调试信息，可帮助排查问题。
- -s：表示显示 symbol 表示。在分析性能数据时，symbol 表示是将地址映射到具体函数或符号名称的过程。

这两个选项主要用于在使用 perf inject 命令时提供更多的调试信息和功能，以便更好地理解和分析性能数据。

5.2.3 生成火焰图

```
perf script -F comm,pid,tid,cpu,time,period,event,ip,sym,dso,trace | awk '
    NF > 4 { exec = $1; period_ms = int($5 / 1000000) }
    NF > 1 && NF <= 4 && period_ms > 0 { print $2 }
    NF < 2 && period_ms > 0 { printf "%s\n%d\n\n", exec, period_ms }' | \
    ./FlameGraph/stackcollapse.pl | \
    ./FlameGraph/flamegraph.pl --countname=ms --title="Off-CPU Time Flame Graph" --
colors=io > zeromq-req-rep-1-offcpu.svg
```



6 实践-nginx http

6.1 配置和启动nginx

1. **修改nginx.conf，配置web页面路径**

主要是修改root /mnt/hgfs/ubuntu/vip/20230826-flamegraph/src-flamegraph/**0voice_webrtc**；（注意自己的实际路径）**对应web页面的路径。**

```
#user  nobody;
worker_processes  1;

#error_log  logs/error.log;
#error_log  logs/error.log  notice;
```

```
#error_log  logs/error.log  info;

#pid        logs/nginx.pid;


events {
    worker_connections  1024;
}


http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;
    #tcp_nopush     on;

    #keepalive_timeout  0;
    keepalive_timeout  65;

    #gzip  on;

    server {
        listen       80;
        server_name  localhost;

        location / {
                        root /mnt/hgfs/ubuntu/vip/20210821-flamegraph/src-
flamegraph/0voice_webrtc;
            index  index.html index.htm;
        }

        #error_page  404              /404.html;

        # redirect server error pages to the static page /50x.html
        #
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
    }
}
```

2. **然后重新启动nginx**

6.2 HTTP压测工具wrk使用指南

6.2.1 安装wk

从github上下载源码

```
git  clone  https://github.com/wg/wrk
```

然后cd到wrk目录，进行安装

```
cd wrk
make
make install
```

可能有一些包没有，导致git，make命令不能顺利执行，安装即可。

6.2.2 基本参数说明

以下是使用wrk查看到的一些基本参数信息

-c	**--connections**	**N**	**跟服务器建立并保持的TCP连接数量**
-d	--duration	T	压测**时间**
-t	--threads	N	使用多少个**线程**进行压测
-s	--script	S	指定**Lua脚本路径**
-H	--header	H	为每一个HTTP请求添加**HTTP头**
--latency			在压测结束后，打印**延迟统计信息**
--timeout		T	**超时**时间
-v	--version		打印正在使用的wrk的详细版本信息

- **N**代表**数字**参数，支持国际单位 (1k, 1M, 1G)
- **T**代表**时间**参数，支持时间单位 (2s, 2m, 2h)

wrk **-c** 20 **-t** 2 **-d** 20s **--latency** http://192.168.1.34

建立20个**TCP连接**，使用**两个线程**，用时**20秒**，对http://192.168.1.34进行压测。

6.2.3 输出内容

```
lqf@ceph-admin:~/flamegraph/wrk$ ./wrk -c 20 -t 2 -d 20s --latency
http://192.168.0.143
Running 20s test @ http://192.168.0.143
  2 threads and 20 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
              （平均值）（标准差）（最大值）（正负一个标准差所占比例）
```

```
    Latency     9.35ms    1.91ms  81.96ms   96.92%
    （延迟）
    Req/Sec     1.08k    73.16     1.52k    88.50%
    （处理中的请求数）
  Latency Distribution（延迟分布）
     50%    9.03ms
     75%    9.45ms
     90%    9.97ms
     99%   17.26ms（99分位的延迟）
  43019 requests in 20.03s, 170.79MB read（20.03s秒内共处理完成了43019个请求，读取了
170.79MB数据）
Requests/sec:   2148.24     （平均每秒处理完成2148.24个请求）
Transfer/sec:      8.53MB （平均每秒读取数据8.53MB）

lqf@ceph-admin:~/flamegraph/wrk$ ./wrk -c 20 -t 2 -d 20s --latency
http://120.27.131.197
Running 20s test @ http://120.27.131.197
  2 threads and 20 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    44.10ms   69.76ms 509.32ms   92.73%
    Req/Sec   382.88     80.68   470.00     88.14%
  Latency Distribution
     50%   23.98ms
     75%   26.97ms
     90%   58.56ms
     99%  404.74ms
  14363 requests in 20.02s, 5.03MB read
Requests/sec:    717.38
Transfer/sec:    257.11KB
```

- Latency **延迟**时间
- Req/Sec **每秒处理**的请求数
    - **平均值**(Avg)，
    - **标准偏差**(Stdev)，
    - **最大值**(Max)
    - 正负一个**标准差占比**(+/-) Stdev

一般主要关注**Avg和Max。Stdev**如果太大说明样本本身**离散程度**比较高，有可能系统性能波动很大。

6.3 on-cpu火焰图分析nginx

6.3.1 使用wrk压测

压测的时间要大于perf采集的时间

```
./wrk -c 20 -t 2 -d 5M --latency http://192.168.0.143
```

6.3.2 使用perf采集

1. **查找nginx worker进程pid**

```
# ps -ef | grep nginx
root      85453  16908  0 00:03 ?        00:00:00 nginx: master process
/usr/local/nginx/sbin/nginx
nobody    85454  85453  0 00:03 ?        00:00:36 nginx: worker process
root      97907  56681  0 21:18 pts/22   00:00:00 grep --color=auto nginx
```

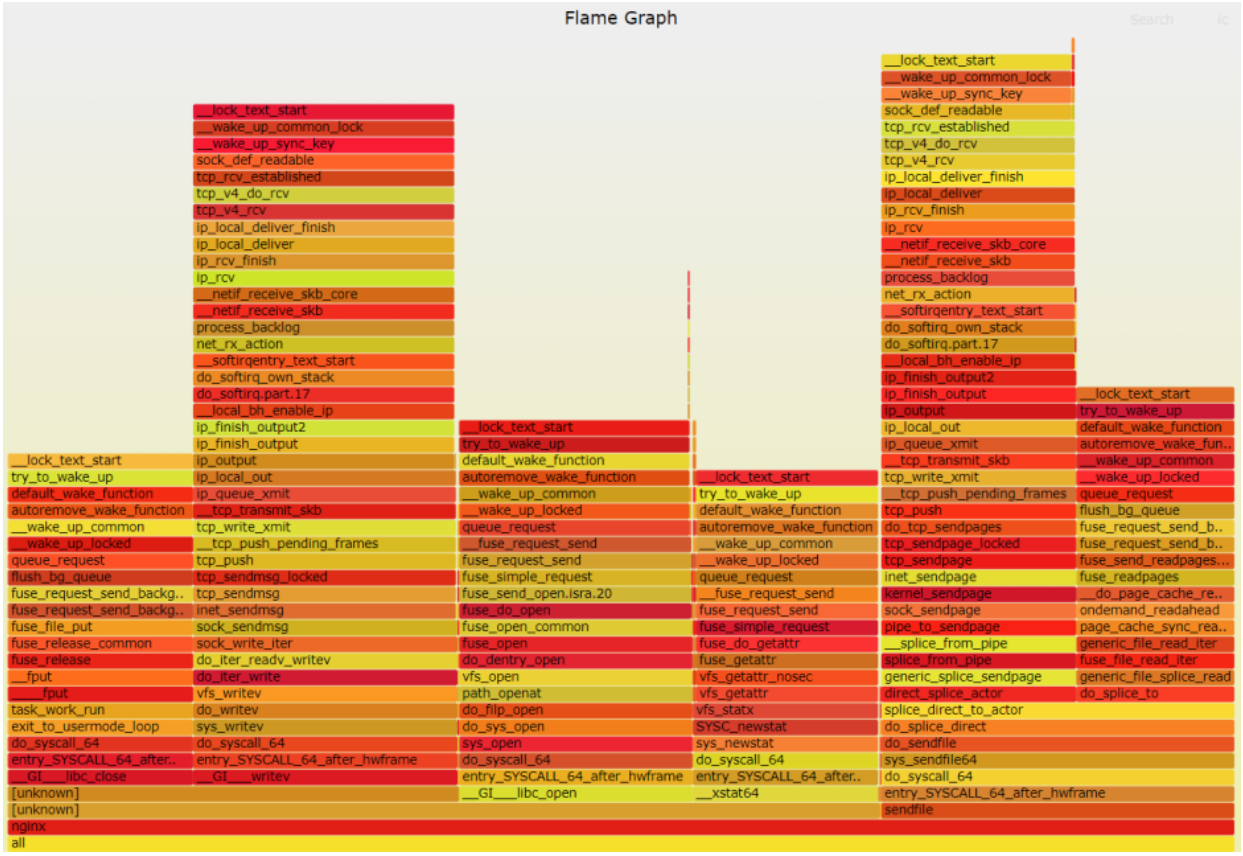
查找到对应work pid为85454

2. **采集数据**

```
# perf record -F 99 -p 33386  -g -- sleep 60
```

6.3.3 生成on-cpu火焰图

```
# perf script | ./FlameGraph/stackcollapse-perf.pl | ./FlameGraph/flamegraph.pl >
nginx-on-cpu.svg
```



6.4 off-cpu火焰图分析nginx

6.4.1 使能sched_schedstats统计

需要在root权限下使能

```
# echo 1 > /proc/sys/kernel/sched_schedstats
```

6.4.2 使用perf采集

1. **查找nginx worker进程pid**

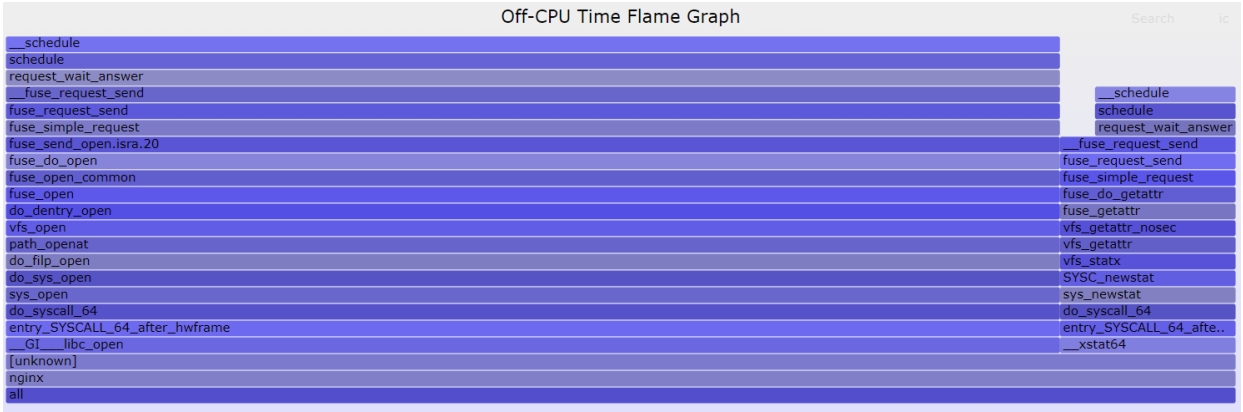
```
# ps -ef | grep nginx
root      85453  16908  0 00:03 ?        00:00:00 nginx: master process
/usr/local/nginx/sbin/nginx
nobody    85454  85453  0 00:03 ?        00:00:36 nginx: worker process
root      97907  56681  0 21:18 pts/22   00:00:00 grep --color=auto nginx
```

查找到对应work pid为85454

```
# perf record -e sched:sched_stat_sleep -e sched:sched_switch -e
sched:sched_process_exit -p 85454 -g -o perf.data.raw sleep 30

# perf inject -v -s -i perf.data.raw -o perf.data
```

6.4.3 生成off-cpu火焰图

```
perf script -F comm,pid,tid,cpu,time,period,event,ip,sym,dso,trace | awk '
    NF > 4 { exec = $1; period_ms = int($5 / 1000000) }
    NF > 1 && NF <= 4 && period_ms > 0 { print $2 }
    NF < 2 && period_ms > 0 { printf "%s\n%d\n\n", exec, period_ms }' | \
    ./FlameGraph/stackcollapse.pl | \
    ./FlameGraph/flamegraph.pl --countname=ms --title="Off-CPU Time Flame Graph" --
colors=io > nginx-off-cpu.svg
```



7 实践-测试线程池队列

- 分析cpu占用情况

- 分析任务队列情况
- 查找程序中sleep延时函数对性能的影响

测试程序：src-flamegraph\threadpool

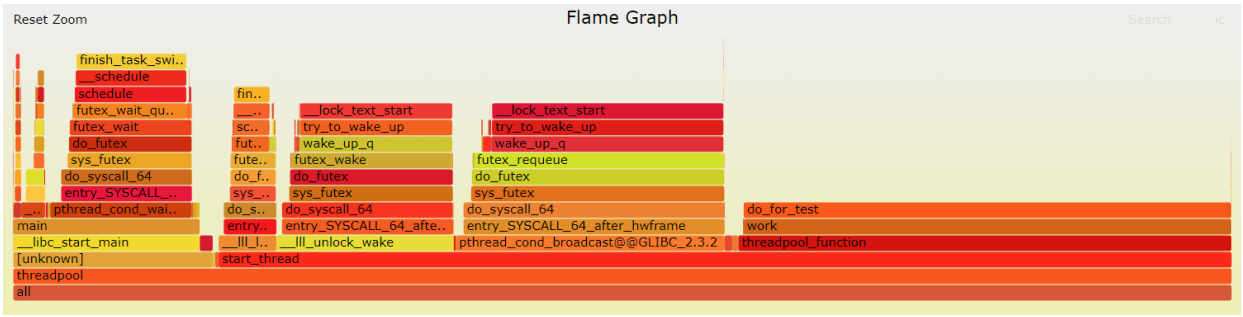
7.1 on-cpu火焰图

7.1.1 使用perf采集

```
# perf record -F 99 -p 36052 -g -- sleep 60
```

7.1.2 生成火焰图

```
# perf script | ./FlameGraph/stackcollapse-perf.pl | ./FlameGraph/flamegraph.pl > threadpool-oncpu.svg
```



7.2 off-cpu火焰图

https://www.brendangregg.com/blog/2015-02-26/linux-perf-off-cpu-flame-graph.html
Off-CPU Analysis
https://www.brendangregg.com/offcpuanalysis.html

7.2.1 使能sched_schedstats统计

需要在root权限下使能

```
# echo 1 > /proc/sys/kernel/sched_schedstats
```

7.2.2 使用perf采集

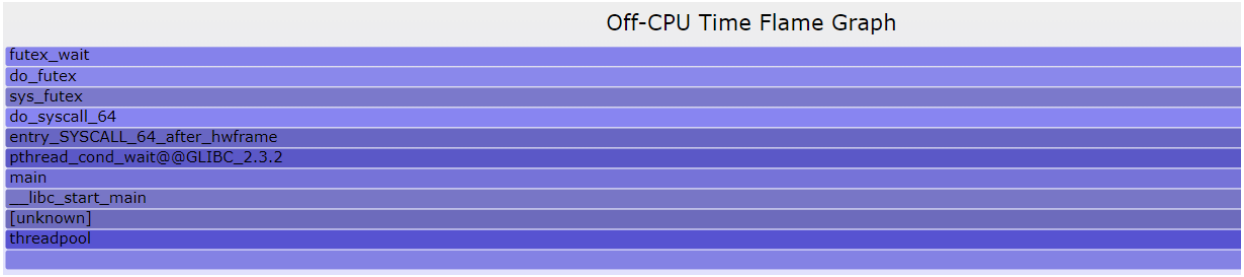
-p 是对应要采集程序的进程ID

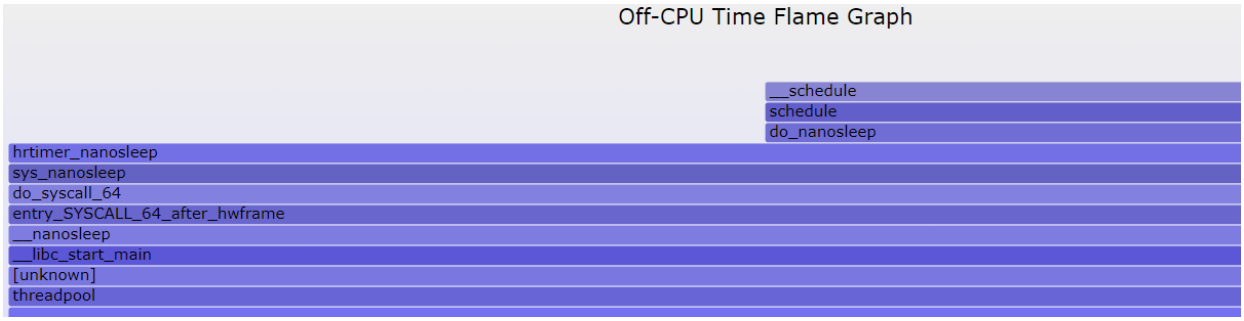
```
# perf record -e sched:sched_stat_sleep -e sched:sched_switch -e sched:sched_process_exit -p 36176 -g -o perf.data.raw sleep 30

# perf inject -v -s -i perf.data.raw -o perf.data
```

7.2.3 生成火焰图

```
perf script -F comm,pid,tid,cpu,time,period,event,ip,sym,dso,trace | awk '
    NF > 4 { exec = $1; period_ms = int($5 / 1000000) }
    NF > 1 && NF <= 4 && period_ms > 0 { print $2 }
    NF < 2 && period_ms > 0 { printf "%s\n%d\n\n", exec, period_ms }' | \
    ./FlameGraph/stackcollapse.pl | \
    ./FlameGraph/flamegraph.pl --countname=ms --title="Off-CPU Time Flame Graph" --
colors=io > threadpool-1-offcpu.svg
```

可以修改程序做一些测试。





7.3 补充-火焰图的局限性

nanosleep

是usleep调用了nanosleep，但火焰图只识别了nanosleep，**在使用火焰图的时候要注意类似的缺陷。**

#0 0x00007ffff78bc360 in nanosleep () from /lib/x86_64-linux-gnu/libc.so.6
#1 0x00007ffff78ede94 in usleep () from /lib/x86_64-linux-gnu/libc.so.6
#2 0x00000000004015d0 in work2 ()
#3 0x000000000040137e in threadpool_function ()
#4 0x00007ffff7bc16ba in start_thread () from /lib/x86_64-linux-gnu/libpthread.so.0
#5 0x00007ffff78f751d in clone () from /lib/x86_64-linux-gnu/libc.so.6

8 实践-测试日志库

测试之前学习的log4cpp

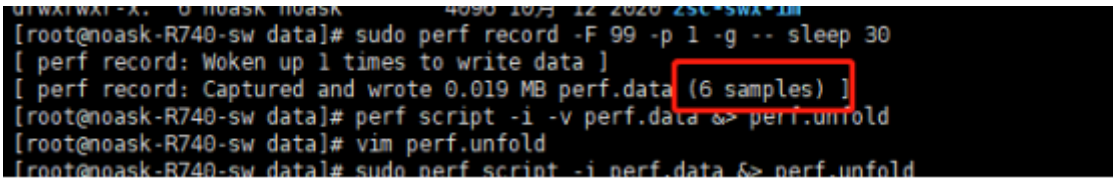
使用oncpu火焰图和offcpu火焰图 lseek

9 FAQ

ERROR: No stack counts found

使用root权限账户执行命令，数据文件属于root用户
The perf. data data has no samples!
如果没有采集到数据解析后的perf.unfold则没有数据
将 sleep 时间 增大一些，同时如果是web 应用则 请求一下。 意思是如果进程没有运行，那么是监控不到 stack 信息的，也就不会有 svg 火焰图了

如何检验是否采集到数据
在结束采集数据后，会显示采集的数据集数量如图所示



如果是off-cpu则大概率是因为：echo 1 > /proc/sys/kernel/sched_schedstats 没有设置。

参考

- 章亦春 动态追踪技术漫谈 https://blog.openresty.com.cn/cn/dynamic-tracing/
- 内存火焰图 https://www.brendangregg.com/FlameGraphs/memoryflamegraphs.html
- perf Examples https://www.brendangregg.com/perf.html
- 火焰图的介绍论文 http://queue.acm.org/detail.cfm?id=2927301
- 火焰图官方主页 http://www.brendangregg.com/flamegraphs.html
- 火焰图生成工具 https://github.com/brendangregg/FlameGraph
- 借助PERF工具分析CPU使用率http://linuxperf.com/?p=36
- 内存火焰图翻译：内存泄漏（增长）火焰图_宋宝华的博客-CSDN博客
- 在Linux下做性能分析3：perf - 知乎 (zhihu.com)
- 剖析CPU性能火焰图生成的内部原理 https://mp.weixin.qq.com/s/A19RlLhSgbzw8UU4p1TZNA