

eBPF Tracing on Linux Block I/O

摘要

即使用 eBPF 追踪 Linux 内核中的块设备输入/输出。

本报告较为全面地介绍了作为一名初学者，如何使用 eBPF 技术追踪 Linux 内核中发生的块设备 I/O。

首先，我们对 eBPF 技术做了整体的介绍。具体介绍了 eBPF 技术的由来、原理和发展：包括 eBPF 程序在内核中所处的位置、eBPF 挂载的工具：“钩子” hook 等具体原理以及与修改内核源码、插入内核模块相比 eBPF 具有的优点。

为了方便地编写 eBPF 代码，我们介绍了一款基于 eBPF 的开源工具集：BCC。它使用 C 语言和 Python 语言开发，提供了丰富的性能分析、网络监控和故障排查工具，可以帮助开发者更轻松地编写和运行 eBPF 程序。

现在，我们需要使用上述的工具对块设备 I/O 进行追踪，因此我们以 Linux 6.1 内核的 Linux I/O Stack 图为例，介绍了在 Linux 内核中，一个 I/O 请求从发出到完成的流程，具体介绍了文件系统层、块设备层和驱动层的组成和作用。

到这里我们已经做好了前期准备，下面我们正式使用各种工具对 Linux 中的块设备 I/O 做追踪。介绍了追踪（Tracing）的含义后，我们具体使用了 BCC 中自带的两种工具对 Linux 中的块设备 I/O 进行追踪，展示了其运行结果，并对其代码进行了简要分析。

如果 BCC 自带的工具不能满足你的需求，那么便需要自己写 BCC 代码。我们以一个简单的 BCC 代码为例，介绍了开发 BCC 工具的基本流程。

最后，我们介绍了一个基于 eBPF 开发的、已经成功商用的全流程追踪工具集：基于 Anolis OS 的 SysAK 工具集，我们介绍了 SysAK 工具集中三个用于 I/O 追踪的工具的功能。

目录

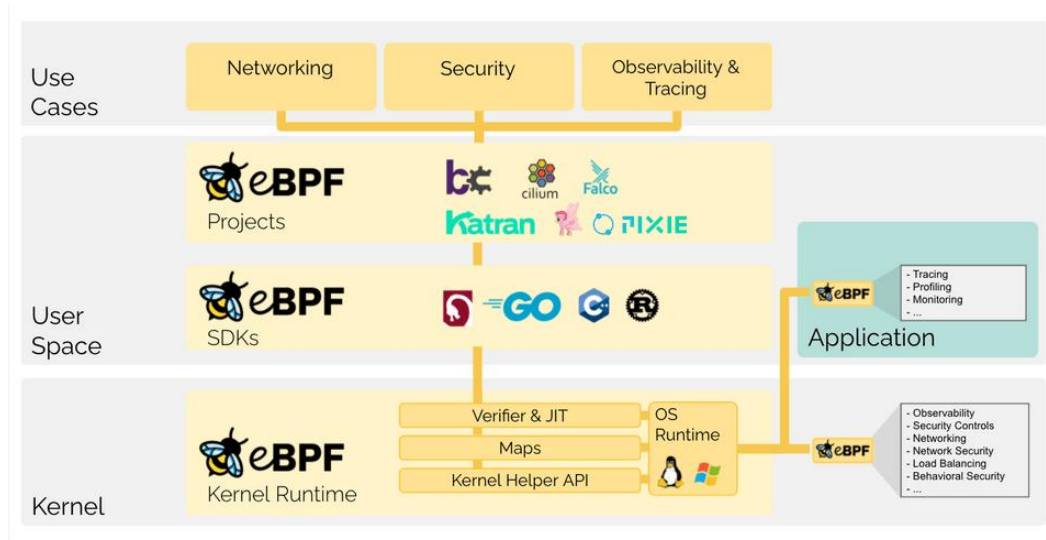
摘要	1
1. eBPF	3
1.1 什么是 eBPF ?	3
1.2 eBPF 的“钩子”	4
1.3 eBPF 的优点	5
2. BCC	5
2.1 什么是 BCC?	5
3. Linux I/O Stack	6
3.1 File System Layer	8
3.2 Block Layer	9
3.3 Driver Layer	9
4. Trace Linux Block I/O by BCC Tools	10
4.1 什么是 Tracing?	10
4.2 biolatency	11
4.3 ext4slower	13
5. Write a BCC Program by Yourself	14
5.1 C 程序	14
5.2 Python 程序	15
5.3 程序运行结果	17
6. 拓展内容：eBPF 在 Anolis OS 上的应用：SysAK 工具集	17
6.1 iofsstat	17
6.2 iowaitstat	18
6.3 fcachetop	19
参考文献	20

1. eBPF

1.1 什么是 eBPF ?

eBPF 是 extend Berkeley Packet Filter 的缩写，其中 Berkeley Packet Filter 指的是伯克利分组过滤器，但如今的 eBPF 的功能远不止分组过滤，因此 eBPF 很大程度上已经成为了一个独立的术语。

eBPF 是一项具有革命性的技术，起源于 Linux 内核，可以在具有特权的上下文中（如操作系统内核）运行受沙盒保护的程序，被用于安全、高效地扩展内核功能，无需更改内核源代码或加载内核模块。



上图是 eBPF 官网中的一张架构图，表明了 eBPF 在整个系统中所处的位置：

User Cases: 是指 eBPF 涵盖的的用例：如为现代数据中心和云原生环境提供高性能网络和负载均衡、以低开销提取细粒度的安全可观察性数据、帮助应用开发人员跟踪应用程序，为性能故障排查提供观察等。

User Space: 是指 eBPF 在用户空间的应用，主要涉及与用户空间工具和库的互动，以便对内核中运行的 eBPF 程序进行加载、管理和与之通信。用户空间应用程序和库可以利用 eBPF 在内核中收集数据、实现策略和优化系统性能：其具体应用包括网络监控和分析、系统性能分析、安全策略实施和应用程序追踪等。

而本文接下来将要介绍的 BCC (BPF Compiler Collection) 即是位于用户空间的库。

Kernel: 事实上, eBPF 程序运行于操作系统的内核中, 其运行流程如下: eBPF 程序创建完成并使用 LLVM 中的 clang 编译器编译成 BPF 字节码后, 用户空间应用程序通过调用 bpf 系统调用将 eBPF 程序加载到内核空间; 之后通过验证引擎 (Verifier) 的验证, 保证其不会对内核产生有害的影响 (如内核崩溃或无限循环); 保证程序的安全性后, 通过实时编译器 (JIT Just-In-Time) 将 eBPF 字节码编译成本地机器码; 一旦 eBPF 程序通过验证并经过 JIT 编译 (此步骤是可选的, 但使用的话可以显著提高 eBPF 程序的性能) 后, 它将被附加到内核中的特定事件上, 当这些事件被触发时, 内核将执行 eBPF 程序, 程序可以修改内核数据结构、读取或写入 maps (一种特定的数据结构, 用于存储程序中的数据), 甚至调用特定的 eBPF 辅助函数来完成各种任务。

1.2 eBPF 的“钩子”

在介绍“钩子” (hook) 之前, 我们首先从总体上介绍一下 eBPF 的功能, 下图是 eBPF 的官方 logo:



可以看出, 该 logo 是一只蜜蜂的形象, 以下是我个人对该 logo 的理解 (从原理上来说, 下面的理解没有错误, 但是我目前还没有从其官网上找到对这一理解的书面信息用以确认): 我们将 Linux 内核比作一朵花, 将 eBPF 比作一只蜜蜂, 将我们需要的信息比作花粉, eBPF 可以像蜜蜂采花粉一样在内核的各个位置采集 (甚至修改) 各种信息。

有了上述理解, 加之上文我们提及“eBPF 程序将被附加到内核中的特定事件上, 当这些事件被触发时, 内核将执行 eBPF 程序”, 我们很自然地引出“钩

子”的概念，即 eBPF 通过这些“钩子”“钩”在内核中：

eBPF 是基于事件驱动的，在内核或应用程序经过特定的钩子挂载点时，相应的 eBPF 程序就会运行。而这样的钩子挂载点大致可以分成三种：第一种是预定义的钩子，一般称为“tracepoint”；如果预定义的钩子不能满足需求，那么我们需要自己创建，我们可以分别在内核空间或用户空间跟踪，分别使用 kernel probe（简称 kprobe）和 user probe（简称 uprobe），这两大工具几乎可以用来探查内核和用户空间的各个位置。

1.3 eBPF 的优点

下面介绍 eBPF 相对于修改内核代码和插入内核模块两种方法的优点，也是我们选择 eBPF 技术来做 Linux Block I/O 跟踪的原因：

（1）**安全与稳定**：eBPF 程序在加载到内核之前会经过严格的验证，以确保其不会引发内核崩溃或安全问题。这与内核模块或直接修改内核源码相比，显著提高了系统的安全性。且运行时不会修改内核数据结构，降低了引发内核故障的风险。

（2）**灵活**：eBPF 程序可以动态加载和卸载，使得开发者可以实时地修改、调试和优化程序。

（3）**跨平台兼容**：eBPF 程序具有良好的跨平台兼容性，可以在不同版本的 Linux 内核上运行，而无需对源代码进行修改。这意味着 eBPF 程序可以方便地在不同的 Linux 发行版和内核版本之间迁移。

（4）**隔离**：eBPF 程序运行在一个独立的、受限制的环境中，这有助于降低程序之间的相互干扰。

2. BCC

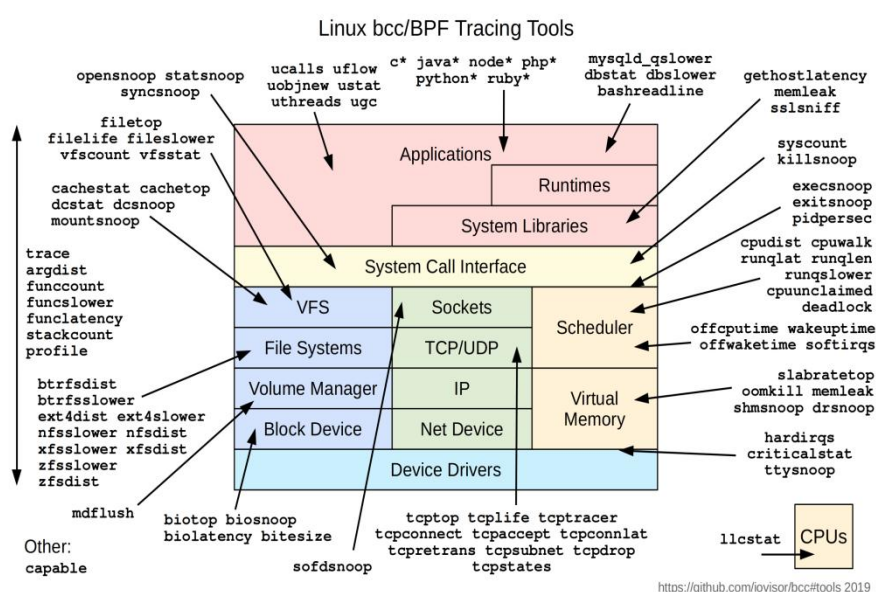
2.1 什么是 BCC?

BCC 是 BPF Compiler Collection 的简称，是一个基于 eBPF 的开源工具集。它使用 C 语言和 Python 语言开发，提供了丰富的性能分析、网络监控和故障

排查工具，旨在帮助开发者更轻松地编写和运行 eBPF 程序。

BCC 提供了一个 C 编程环境，用于编写内核 eBPF 代码，并为用户级接口提供了其他语言如 Python、lua 或 C++。即核心的 eBPF 代码仍然使用 C 语言开发，而使用 BCC 后，开发者可以使用 Python 等语言来编写许多用户级的业务代码，比如将 eBPF 返回的结果使用直方图展示出来等，而这些功能使用 Python 等编程语言是相对简单且高效的。

下图是 BCC 自带的工具，涉及从用户应用到设备驱动的各个方面的追踪，在下文我们会挑选其中的几个工具具体介绍。



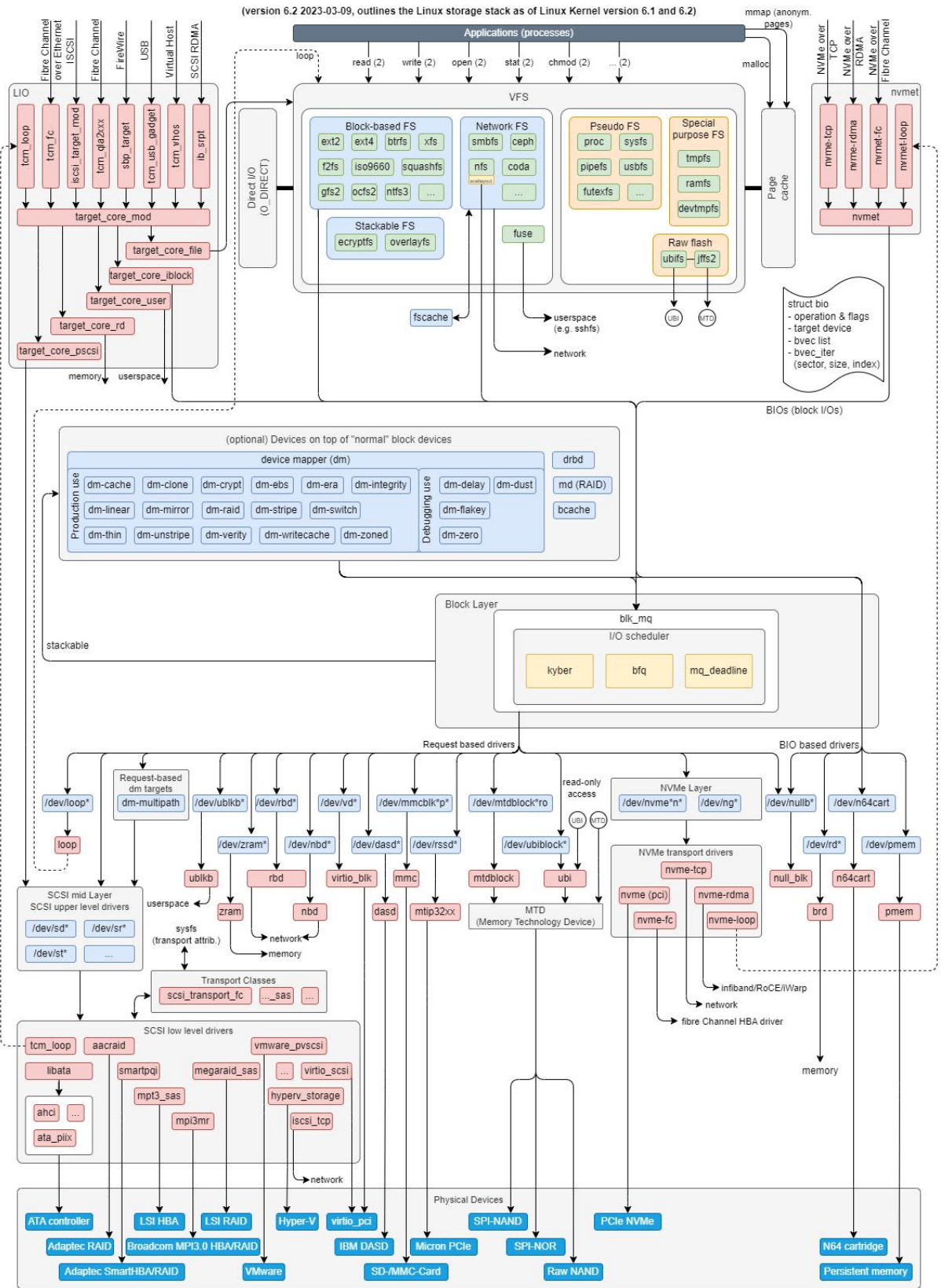
3. Linux I/O Stack

在使用 BCC 追踪 Linux 的块设备 I/O 之前，我们首先向大家介绍 Linux 的输入输出栈，即 Linux 是完成输入输出的全过程。下面首先给出 Linux 6.2 的输入输出总图，其来源为：

https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram#Diagram_for_Linux_Kernel_6.2

The Linux Storage Stack Diagram (Linux Kernel 6.2)

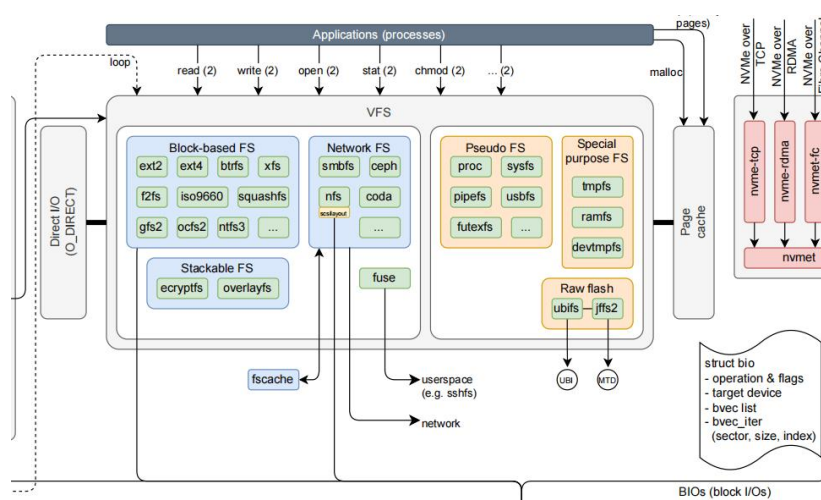
(version 6.2 2023-03-09, outlines the Linux storage stack as of Linux Kernel version 6.1 and 6.2)



The Linux Storage Stack Diagram
https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
 Design Werner Fischer, support by Christoph Hellwig, Richard Weinberger et al.
 License: CC-BY-SA 3.0, see <http://creativecommons.org/licenses/by-sa/3.0/>

我相信大家第一眼看到这张图会不禁感叹其复杂程度，事实也确实如此（而这张图还只是 Linux 内核的 I/O 部分）其中涉及了外部设备的 I/O、软件的 I/O，I/O 过程中还涉及各种优化方法：如 Cache、虚拟化磁盘等。因此我们不会对其每一个分支进行细致的讲解（事实上我也做不到这一点），我们会对来自用户空间（软件）发出一个 I/O 请求，到该请求访问物理磁盘的全流程进行说明（不涉及 Cache）。

3.1 File System Layer

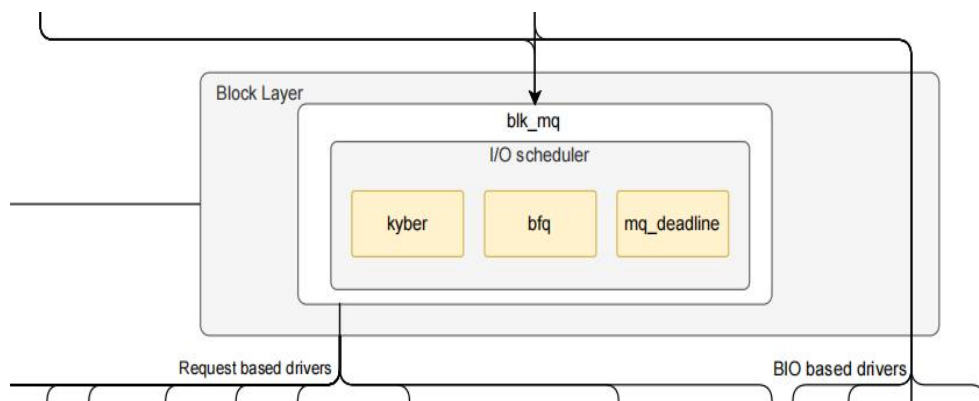


文件系统层将用户空间的 I/O 操作转化为底层数据结构。

File System Layer 具体分为两个部分，第一部分是虚拟文件系统（Virtual File System），VFS 是抽象层，提供统一的接口，使不同的文件系统能够透明地与用户空间程序协同工作。当用户空间程序发起一个 I/O 操作时，系统调用会首先到达 VFS。VFS 会检查文件的路径名，并在其内部挂载表中找到相应的文件系统（如 ext4）。然后，VFS 会将系统调用转发给 ext4 提供的相应函数，如读取文件的 ext4_read 或写入文件的 ext4_write。

第二部分则是具体实现的文件系统，我们以 ext4 文件系统为例，当 ext4 文件系统需要读取或写入磁盘上的数据时，它会创建一个或多个 bio（block I/O）结构。bio 结构包含了对磁盘块进行操作的必要信息，如操作类型（读或写）、块设备、目标磁盘扇区、数据缓冲区等。创建 bio 后，ext4 文件系统会将其提交给 block layer 进行处理。

3.2 Block Layer

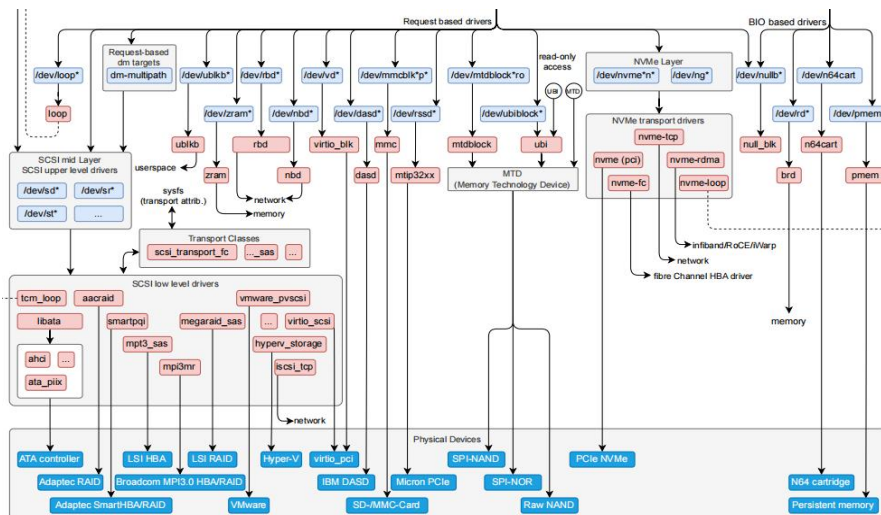


Block Layer 会将文件系统层的 **bio** 结构体按照一定策略、按照不同的块设备分发给对应的 **driver** 程序（以 request/bio 的形式），对于多队列模型，请求通常通过 `blk_mq_ctx` 和 `blk_mq_ops` 结构传递。

Block Layer 按功能可以分成两个部分，第一部分时请求队列：在较老的 Linux 内核中，使用的是单请求队列，Single-Queue Block Layer 维护了一个全局请求队列，用于存储和管理来自 fs layer 的 I/O 请求。这些请求通常是以 `bio` (block I/O) 结构的形式提交的。请求队列可以对 I/O 请求进行排序、合并和调度，以优化磁盘访问性能。在 Linux 5.0 后，多请求队列便成为了默认选项，Multi-Queue Block Layer 引入了多个独立的请求队列和 I/O 调度器实例，每个 CPU 核心或硬件队列都可以有一个独立的请求队列。

第二部分则是 I/O 调度器，负责决定请求队列中的 I/O 请求的执行顺序。I/O 调度器可以是基于不同算法的，如 CFQ（完全公平队列）、Deadline、NOOP、BFQ 等。I/O 调度器的目标是在满足性能和公平性要求的前提下，优化磁盘访问。

3.3 Driver Layer



驱动层非常复杂，因为不同的物理设备需要使用不同的驱动程序，因此我们下面只介绍驱动层的通用功能：

- (1) **解析请求：**Driver Layer 收到来自 Block Layer 的请求后，需要解析请求的类型（读取、写入等）、目标设备、起始扇区和数据缓冲区等信息。
- (2) **与硬件设备通信：**根据解析出的请求信息，Driver Layer 会与相应的硬件设备进行通信。这通常涉及与设备控制器交互，以发送命令和数据。
- (3) **处理设备中断和状态：**在 I/O 操作过程中，物理设备可能会产生中断以通知驱动程序操作的完成或错误状态。驱动程序需要处理这些中断，根据设备的状态更新请求的状态，并在适当的时候通知 Block Layer 和文件系统层。
- (4) **完成 I/O 操作：**当设备完成 I/O 操作，驱动程序需要将结果返回给 Block layer。这可能包括操作成功的确认、操作失败的错误代码、读取到的数据等。然后，Block Layer 会根据 I/O 结果更新请求的状态，并通知文件系统层操作的完成情况。

4. Trace Linux Block I/O by BCC Tools

4.1 什么是 Tracing?

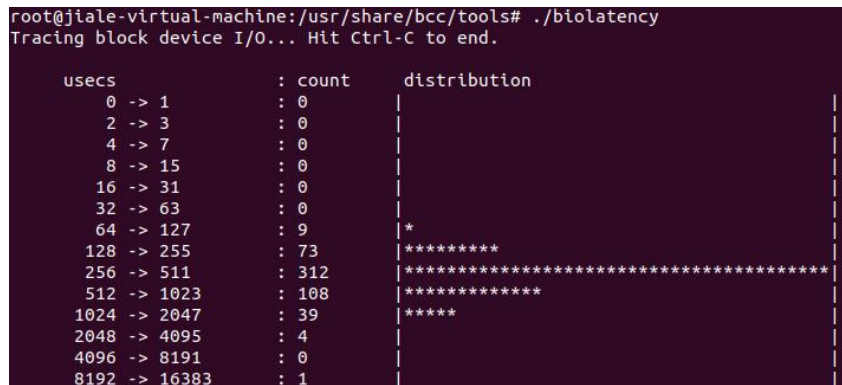
Trace，中文翻译为追踪。是一种基于事件（event）的记录行为，追踪的主要特征是该行为能否记录原始数据和事件的元数据。

而对 Linux Block I/O 的追踪实际上也就是在发生块设备输入/输出时，对其原始

数据和元数据进行记录，如果再将这些信息做筛选、展示，那么便是一种 Linux Block I/O 的追踪工具，而我们下面就选择了几种 BCC 中自带的 Linux I/O Tracing Tools 做具体介绍。

我们在使用这些工具之前，都是开了一个进程，做的是从磁盘中的一个文件中不停读数据的任务。

4.2 biolateness



上图是运行 `biolateness` 工具一段时间后的结果，第一列是时延分布，以微秒为单位，第二列则是运行时间内发生的块设备 I/O 在对应时延分布上的数目，第三列是用图形化的方法展示时延分布：从图中可以看出，耗时 256-511 微秒的块设备 I/O 发生了 312 次，所占比例最多。

下面我们截取其中最重要的代码做分析：

```
205 if args.queued:
206     if BPF.get_kprobe_functions(b'__blk_account_io_start'):
207         b.attach_kprobe(event="__blk_account_io_start", fn_name="trace_req_start")
208     else:
209         b.attach_kprobe(event="blk_account_io_start", fn_name="trace_req_start")
210 else:
211     if BPF.get_kprobe_functions(b'blk_start_request'):
212         b.attach_kprobe(event="blk_start_request", fn_name="trace_req_start")
213         b.attach_kprobe(event="blk_mq_start_request", fn_name="trace_req_start")
214     if BPF.get_kprobe_functions(b'__blk_account_io_done'):
215         b.attach_kprobe(event="__blk_account_io_done", fn_name="trace_req_done")
216     else:
217         b.attach_kprobe(event="blk_account_io_done", fn_name="trace_req_done")
```

在没有附加参数的情况下且内核为 Linux 5.15（我使用的内核版本，5.0 以上默认支持多队列）的情况下，最终运行的代码是 212 和 217 行：

`blk_mq_start_request` 是在多请求队列下是用于将块设备 I/O 请求添加到块设备请求队列中并启动 I/O 操作的时间，这意味着起点是上述 Linux I/O Stack 中的 Block Layer 终点，不需要排队。`blk_account_io_done` 用于跟踪块设备输入/输出 (I/O) 操作的完成时间。`kprobe` 这个术语我们在上面的 **eBPF “钩子”** 部分中介绍过，全称为 kernel probe，因此上述代码即是将 eBPF 程序挂载到这两个函数中，当 `blk_mq_start_request` 开始运行时记录当前时间，当 `blk_account_io_done` 开始运行时记录当前时间，两个时间相减，得到的便是该块设备 I/O 所用的时延。

我们在运行 `biolatency` 工具时加上参数 `-Q` 后，运行结果如下：

```
^Croot@jiale-virtual-machine:/usr/share/bcc/tools# ./biolatency -Q
Tracing block device I/O... Hit Ctrl-C to end.
^C
      usecs          : count      distribution
      0 -> 1         : 0          |
      2 -> 3         : 0          |
      4 -> 7         : 0          |
      8 -> 15        : 0          |
     16 -> 31        : 0          |
     32 -> 63        : 0          |
     64 -> 127       : 0          |
    128 -> 255       : 18         | *
    256 -> 511       : 423        | *****
    512 -> 1023      : 230        | *****
   1024 -> 2047      : 181        | *****
   2048 -> 4095      : 23         | **
   4096 -> 8191      : 15         | *
```

我们可以看到时延主要分布在 256-511 之后，明显比上面的图中的时延要长（我们运行的测试程序相同）。原因如下：

加入 `-Q` 参数后，实际运行的代码是 209 和 217 行，`blk_account_io_start` 跟踪块设备输入/输出操作的开始时间。`blk_account_io_done` 用于跟踪块设备输入/输出 (I/O) 操作的完成时间。因此时间较长（即存在排队时间）。

4.3 ext4slower

```
^Croot@jiale-virtual-machine:/usr/share/bcc/tools# ./ext4slower 1
Tracing ext4 operations slower than 1 ms
TIME      COMM      PID    T BYTES  OFF_KB   LAT(ms)  FILENAME
12:20:34  read_file   40890  R 4096   60      2.25     example.txt
12:20:37  read_file   40890  R 4096   92      2.58     example.txt
12:20:37  read_file   40890  R 4096  116      1.10     example.txt
12:20:37  read_file   40890  R 4096  100      1.14     example.txt
12:20:38  read_file   40890  R 4096  104      1.48     example.txt
12:20:39  read_file   40890  O 0       0       4.33     example.txt
12:20:39  read_file   40890  R 4096   24      2.04     example.txt
12:20:39  read_file   40890  R 4096   24      1.78     example.txt
12:20:39  read_file   40890  R 4096   40      1.50     example.txt
12:20:39  read_file   40890  R 4096   84      3.24     example.txt
12:20:39  read_file   45089  R 4096   44      4.09     example.txt
12:20:39  read_file   40890  R 4096  100      1.25     example.txt
12:20:41  read_file   40890  R 4096   88      2.19     example.txt
12:20:41  read_file   40890  R 4096   96      1.70     example.txt
12:20:53  read_file   40890  R 4096   64     24.56     example.txt
12:21:06  read_file   40890  R 4096    4      2.11     example.txt
12:21:06  read_file   40890  R 4096  112      6.24     example.txt
12:21:06  read_file   40890  R 4096   64      1.41     example.txt
12:21:06  read_file   40890  R 4096  100      3.46     example.txt
12:21:06  read_file   40890  R 4096   64     12.74     example.txt
12:21:06  read_file   40890  R 4096   68      2.09     example.txt
12:21:13  read_file   40890  R 4096   44      1.02     example.txt
12:21:19  read_file   40890  R 4096   92      2.18     example.txt
12:21:19  read_file   40890  R 4096   40      1.33     example.txt
12:21:22  read_file   40890  R 4096   44      1.34     example.txt
```

ext4slower 可以追踪 ext4 文件系统中 I/O 操作时延慢于指定时间的 I/O 操作的进程、进程描述符、类型、操作字节数、时延、文件名称等。

其中最重要的 probe 挂载代码如下：

```
315 # Common file functions. See earlier comment about generic_file_read_iter().
316 if BPF.get_kprobe_functions(b'ext4_file_read_iter'):
317     b.attach_kprobe(event="ext4_file_read_iter", fn_name="trace_read_entry")
318 else:
319     b.attach_kprobe(event="generic_file_read_iter", fn_name="trace_read_entry")
320 b.attach_kprobe(event="ext4_file_write_iter", fn_name="trace_write_entry")
321 b.attach_kprobe(event="ext4_file_open", fn_name="trace_open_entry")
322 b.attach_kprobe(event="ext4_sync_file", fn_name="trace_fsync_entry")
323 if BPF.get_kprobe_functions(b'ext4_file_read_iter'):
324     b.attach_kretprobe(event="ext4_file_read_iter", fn_name="trace_read_return")
325 else:
326     b.attach_kretprobe(event="generic_file_read_iter", fn_name="trace_read_return")
327 b.attach_kretprobe(event="ext4_file_write_iter", fn_name="trace_write_return")
328 b.attach_kretprobe(event="ext4_file_open", fn_name="trace_open_return")
329 b.attach_kretprobe(event="ext4_sync_file", fn_name="trace_fsync_return")
```

文件系统通过向内核提供一组操作函数来实现文件访问。以读操作为例，当用户程序尝试从文件中读取数据时，内核将调用文件系统的 read_iter 操作函数。ext4_file_read_iter 是 ext4 文件系统中实现的 read_iter 函数。

ext4_file_read_iter 的主要作用是将用户空间缓冲区的数据填充为来自 ext4 文件系统的数据。它从文件系统中读取数据并将其写入用户空间缓冲区。该函数

还处理诸如文件锁、读取缓存、文件映射、权限检查等与读取文件相关的操作。

kretprobe 术语的全程为 kernel return probe，表示返回时的“钩子”，即时间完成时回调对应函数时触发 eBPF 程序。

5. Write a BCC Program by Yourself

一般情况下，一个 BCC 程序由一个 C 程序和一个 Python 程序组成，C 程序是具体的 BPF 代码，写明 BPF 程序被触发后需要做的事情，将 C 程序嵌入 Python 程序中，Python 程序则负责将函数与具体的“钩子”绑定起来，并将 C 程序从内核中取得的数据以开发者希望的方式展示出来。

5.1 C 程序

```
#include <uapi/linux/ptrace.h>
#include <linux/blk_types.h>
#include <linux/bio.h>

// 定义event_data_t结构体，用于存储事件相关数据
struct event_data_t {
    u64 ts;                // 时间戳，单位纳秒
    u32 pid;               // 进程ID
    char comm[TASK_COMM_LEN]; // 进程名
};

// 定义BPF_HASH映射，用于存储每个进程的开始时间
BPF_HASH(start, u32);
// 定义BPF_PERF_OUTPUT，用于将事件数据传递给用户空间
BPF_PERF_OUTPUT(events);

// 定义kprobe处理函数，用于在submit_bio函数开始时获取当前进程的ID和时间戳，并将它们存储在BPF_HASH映射中
int trace_submit_bio_entry(struct pt_regs *ctx, struct bio *bio) {
    u32 pid = bpf_get_current_pid_tgid();
    u64 ts = bpf_ktime_get_ns();

    start.update(&pid, &ts);
    return 0;
}
```

```

// 定义kretprobe处理函数，用于在submit_bio函数返回时计算函数执行的延迟，并将事件数据发送到用户空间
int trace_submit_bio_return(struct pt_regs *ctx, struct bio *bio) {
    u32 pid = bpf_get_current_pid_tgid();
    u64 *tsp = start.lookup(&pid);

    // 如果找到了对应进程ID的开始时间
    if (tsp != 0) {
        struct event_data_t event = {};
        // 计算submit_bio函数执行的延迟
        u64 delta = bpf_ktime_get_ns() - *tsp;
        event.ts = delta;
        event.pid = pid;
        // 获取当前进程的名称
        bpf_get_current_comm(&event.comm, sizeof(event.comm));

        // 将事件数据发送到用户空间
        events.perf_submit(ctx, &event, sizeof(event));
        // 删除已处理的进程ID
        start.delete(&pid);
    }

    return 0;
}

```

如上是我们写好的 C 程序，注释已经非常详细，我们总体上介绍一下其组成：程序由两个函数组成，分别是 `trace_submit_bio_entry` 和 `trace_submit_bio_return`，两大函数的功能注释中已经标明。在写 BPF 代码时，最重要的是结构体的定义，即开发者需要把需要收集的数据集中成一个结构体，然后调用这种函数从内核/用户空间中收集数据，存入结构体，再通过事件 `event` 将数据发送到用户空间。在收集时利用 BPF_HASH 映射，来实现以各个粒度（如进程、容器等）的匹配工作（只是一种方法）。

5.2 Python 程序

如下是我们写好的 Python 程序，也可以理解为用户程序，注释同样十分详细。其首先载入 C 程序，得到一个 `bpf` 对象，然后将该对象挂载到对应的内核/用户态函数上，然后就是将其传回的数据存储起来，再以个性化的方式输出。


```

# trace_bio.py
# 导入bcc (BPF Compiler Collection) 库和相关的工具函数
from bcc import BPF
from bcc.utils import printb
# 导入ctypes库, 用于处理C数据类型和Python数据类型的互操作
import ctypes

# 创建BPF对象, 加载源代码文件"trace_bio.c"
bpf = BPF(src_file="trace_bio.c")

# 为内核函数submit_bio附加kprobe (在函数开始时触发) 和kretprobe (在函数返回时触发), 分别指定处理函数
# trace_submit_bio_entry和trace_submit_bio_return
bpf.attach_kprobe(event="submit_bio", fn_name="trace_submit_bio_entry")
bpf.attach_kretprobe(event="submit_bio", fn_name="trace_submit_bio_return")

# 定义一个名为EventData的ctypes结构体, 用于存储从BPF程序传递过来的数据
class EventData(ctypes.Structure):
    _fields_ = [
        ("ts", ctypes.c_ulonglong), # 时间戳, 单位纳秒
        ("pid", ctypes.c_uint),      # 进程ID
        ("comm", ctypes.c_char * 16), # 进程名
    ]

# 定义处理性能事件的回调函数, 用于处理从BPF程序传递过来的事件数据
def print_event(cpu, data, size):
    # 将传递过来的数据转换为EventData结构体类型
    event = ctypes.cast(data, ctypes.POINTER(EventData)).contents
    # 打印进程名、进程ID和操作的延迟 (纳秒)
    printb(b"[%-16s] PID: %-5d Latency: %-8d ns" % (
        event.comm, event.pid, event.ts))

# 将名为"events"的BPF表与print_event回调函数绑定, 并创建一个perf buffer
bpf["events"].open_perf_buffer(print_event)

# 输出提示信息, 开始追踪bio提交事件
print("Tracing bio submissions... Ctrl-C to stop.")

# 使用循环持续轮询perf buffer, 处理传递过来的事件数据
while True:
    bpf.perf_buffer_poll()

```

5.3 程序运行结果

```
root@jiale-virtual-machine:/home/jiale/Desktop/ebpf# python3 trace_bio.py
Tracing bio submissions... Ctrl-C to stop.
[gnome-terminal- ] PID: 159177 Latency: 11161 ns
[gnome-terminal- ] PID: 159177 Latency: 27622 ns
[gnome-terminal- ] PID: 159177 Latency: 10410 ns
[gnome-terminal- ] PID: 159177 Latency: 9654 ns
[gnome-terminal- ] PID: 159177 Latency: 8730 ns
[gnome-terminal- ] PID: 159177 Latency: 9179 ns
[vmware-namespac ] PID: 161954 Latency: 10233 ns
[gnome-terminal- ] PID: 159177 Latency: 9979 ns
[gnome-terminal- ] PID: 159177 Latency: 8601 ns
[gnome-terminal- ] PID: 159177 Latency: 9565 ns
[gnome-terminal- ] PID: 159177 Latency: 8631 ns
[gnome-terminal- ] PID: 159177 Latency: 8812 ns
[gnome-terminal- ] PID: 159177 Latency: 9612 ns
[gnome-terminal- ] PID: 159177 Latency: 10410 ns
[gnome-terminal- ] PID: 159177 Latency: 8382 ns
[gnome-terminal- ] PID: 159177 Latency: 11318 ns
[gnome-terminal- ] PID: 159177 Latency: 8383 ns
[gnome-terminal- ] PID: 159177 Latency: 8258 ns
[gnome-terminal- ] PID: 159177 Latency: 11371 ns
[gnome-terminal- ] PID: 159177 Latency: 8423 ns
[gnome-terminal- ] PID: 159177 Latency: 9743 ns
[gnome-terminal- ] PID: 159177 Latency: 9146 ns
[gnome-terminal- ] PID: 159177 Latency: 10986 ns
```

6. 拓展内容：eBPF 在 Anolis OS 上的应用：SysAK 工具集

龙蜥社区对 SysAK 的官方解释如下：SysAK（System Analyse Kit）是龙蜥社区系统运维 SIG，通过对过往百万服务器运维经验进行抽象总结，而提供的一个全方位的系统运维工具集，可以覆盖系统的日常监控、线上问题诊断和系统故障修复等常见运维场景。工具的整体设计上，力图让运维工作回归简单，让系统运维人员不需要深入了解内核就能找出问题的所在。

SysAK 工具（中文名青囊）出现在本次报告中的原因是：该工具是一个国内商业上成功使用 eBPF 技术做 Tracing（甚至不止 Tracing）的案例，可以实现全流程的跟踪。该工具功能较多，包括检测（detect）、监控（monitor）、代码注入（inject）等，涉及的内核部分也非常广泛：包括内存、网络、I/O、cgroup 等，我们下面选择其在 I/O 方面的几种工具进行介绍：

6.1 iofsstat

iofsstat 实现从进程和文件级别统计 I/O 信息：传统的 I/O 统计工具在如下场

景下会略有不足：

- （1） 在磁盘 I/O 被打满的情况下，希望观察是哪个进程贡献了比较多的 I/O，传统的工具只能从整个磁盘角度去统计 I/O 信息，如统计整盘的 iops、bps，但不能统计单个进程所贡献的 iops、bps.
- （2） 系统上统计到某个进程贡献了大量的 IO，希望观察到这些 IO 最终是被哪个磁盘给消费，或者这些 IO 是在访问哪个文件，如果这个进程是来自某个容器，希望依然可以获取访问的文件以及此进程所在的容器.

下图是在 Anolis OS 下运行的结果：

```
[root@localhost out]# ./sysak iofsstat
2023/05/15 00:24:39
device-stat:
r_rqm  w_rqm  r_iops  w_iops  r_bps  w_bps  wait  r_wait  w_wait  util%
sda    1      0      120    0      8.0MB/s 0      3.23   3.23   0      8.2
sr0    0      0      0      0      0      0      0      0      0      0.0
dm-0   0      0      57     0      6.3MB/s 0      5.25   5.25   0      3.6
dm-1   0      0      0      0      0      0      0      0      0      0.0
dm-2   0      0      74     0      3.0MB/s 0      1.22   1.22   0      5.3

comm    pid    iops_rd  bps_rd      iops_wr  bps_wr  device
firefox 3506    66       5.6MB/s     0        0       sda
mozStorage #1 3652    15       664.0KB/s  0        0       sda
IndexedDB #1 3589    11       44.0KB/s   0        0       sda
file:// Content 3610    11       1.3MB/s    0        0       sda
StreamTrans #4 3587    9        904.0KB/s  0        0       sda
Backgro-Pool #1 3554    5        328.0KB/s  0        0       sda
mozStorage #2 3663    4        224.0KB/s  0        0       sda
localStorage DB 3656    3        80.0KB/s   0        0       sda
DOM Worker 3585    2        24.0KB/s   0        0       sda
StreamTrans #1 3553    1        4.0KB/s    0        0       sda
StreamTrans #6 3592    1        4.0KB/s    0        0       sda
StreamTrans #5 3591    1        4.0KB/s    0        0       sda
mozStorage #3 3689    1        32.0KB/s   0        0       sda
```

指标标识符	含义
r_rqm	Read: Request Queue Length/Magnitude
w_rqm	Write: Request Queue Length/Magnitude
r_iops	Read: I/O Operations Per Second
w_iops	Write: I/O Operations Per Second
r_bps	Read: Bytes Per Second
w_bps	Write: Bytes Per Second
wait(ms)	I/O Wait Time Per Second
r_wait(ms)	Read: Wait Time Per Second
w_wait(ms)	Write: Wait Time Per Second
util(%)	Disk Utilization

6.2 iowaitstat

iowaitstat 工具可以检测 I/O 等待事件，当 I/O 等待时间高时，统计其由哪些

进程贡献:

```
[root@localhost out]# ./sysak iowaitstat -c 1
2023/05/15 01:18:39 -> global iowait%: 0.0
comm          tgid    pid      waitio(ms)    iowait(%)    reasons
kworker/3:0   3198    3198     0.189         0.0          Unknow[stacktrace:(wait_for_common_io.constprop.2+0xf4/0x160 -> io_schedule_timeout)]

2023/05/15 01:18:40 -> global iowait%: 0.0
comm          tgid    pid      waitio(ms)    iowait(%)    reasons

2023/05/15 01:18:41 -> global iowait%: 0.0
comm          tgid    pid      waitio(ms)    iowait(%)    reasons
kworker/3:0   3198    3198     0.851         0.0          Unknow[stacktrace:(wait_for_common_io.constprop.2+0xf4/0x160 -> io_schedule_timeout)]

2023/05/15 01:18:42 -> global iowait%: 0.0
comm          tgid    pid      waitio(ms)    iowait(%)    reasons

2023/05/15 01:18:43 -> global iowait%: 0.0
comm          tgid    pid      waitio(ms)    iowait(%)    reasons
kworker/3:0   3198    3198     0.194         0.0          Unknow[stacktrace:(wait_for_common_io.constprop.2+0xf4/0x160 -> io_schedule_timeout)]

2023/05/15 01:18:44 -> global iowait%: 0.0
comm          tgid    pid      waitio(ms)    iowait(%)    reasons

2023/05/15 01:18:45 -> global iowait%: 0.0
comm          tgid    pid      waitio(ms)    iowait(%)    reasons
Permission    3506    3584     0.713         0.0          Unknow[stacktrace:(wait_on_page_bit+0x11b/0x1f0 -> io_schedule)]
kworker/3:0   3198    3198     0.248         0.0          Unknow[stacktrace:(wait_for_common_io.constprop.2+0xf4/0x160 -> io_schedule_timeout)]

2023/05/15 01:18:46 -> global iowait%: 0.0
comm          tgid    pid      waitio(ms)    iowait(%)    reasons
```

iowaitstat 不仅能够统计进程的 I/O 等待时间，还可以给出原因，上图不能看出原因，如下图是其给出的示例：

2022/09/15 16:29:15 -> global iowait%: 5.71					
comm	tgid	pid	waitio(ms)	iowait(%)	reasons
kworker/u8:3	51242	51242	804.93	3.74	Device queue full
java	57284	57300	143.236	0.67	Device queue full
argusagent	110584	110644	133.787	0.62	Device queue full
kworker/u8:4	35568	35568	63.847	0.3	Device queue full

可以看出其 I/O 等待的原因是设备队列已满。

6.3 fcachetop

fcachetop 可以用于统计系统当前已打开文件的 page cache 占用情况以及 cache 命中率。

```
The top20 Max cached open files:
Name                               Cached pages/size    Total pages    Hit percent    Comm:Pid
/usr/lib64/libpinyin/data/bigram.db 5252/20.52MB         5252           100.0%         ibus-engine-lib:2488
/var/lib/sss/mc/initgroups          2825/11.04MB         2825           100.0%         polkitd:937
/var/lib/sss/mc/passwd              2260/8.83MB          2260           100.0%         udisksd:934
/var/cache/Pa...wkey/AppStream-filenames.solvx 1701/6.64MB          1701           100.0%         packagekitd:1947
/var/lib/sss/mc/group                1695/6.62MB          1695           100.0%         polkitd:937
/home/jiale/.cache/ibus/libpinyin/user_bigram.db1538/6.01MB    1538           100.0%         ibus-engine-lib:2488
/usr/lib64/libpinyin/data/pinyin_index.bin 1278/4.99MB          1278           100.0%         ibus-engine-lib:2488
/etc/udev/hwdb.bin                  979/3.82MB           979            35.28%         systemd-udevd:742
/usr/lib64/libpinyin/data/phrase_index.bin 918/3.59MB           918            100.0%         ibus-engine-lib:2488
/home/jiale/.cache/tracker/meta.db   886/3.46MB           886            100.0%         tracker-miner-a:2590
/var/cache/PackageKit/8.6/hawkey/AppStream.solv 725/2.83MB           725            100.0%         packagekitd:1947
/run/log/jour...26abfec754409ea/system.journal 631/2.46MB           2048           30.81%         systemd-journal:696
/var/cache/PackageKit/8.6/hawkey/BaseOS.solv 441/1.72MB           441            100.0%         packagekitd:1947
/var/cache/Pa.../hawkey/BaseOS-filenames.solvx 373/1.46MB           490            76.12%         packagekitd:1947
/var/lib/sss/secrets/secrets.ldb     314/1.23MB           314            100.0%         sssd:940
/var/cache/Pa...key/PowerTools-filenames.solvx 288/1.12MB           305            94.43%         packagekitd:1947
/home/jiale/.cache/tracker/meta.db-wal 212/848.00KB         884            23.98%         tracker-miner-a:2590
/usr/lib64/libpinyin/data/addon_pinyin_index.bin182/728.00KB         182            100.0%         ibus-engine-lib:2488
/var/lib/sss/db/cache_implicit_files.ldb 173/692.00KB         393            44.02%         sssd:940
/usr/lib64/libpinyin/data/addon_phrase_index.bin149/596.00KB         149            100.0%         ibus-engine-lib:2488
/var/lib/sss/db/timestamps_implicit_files.ldb 113/452.00KB         393            28.75%         sssd:940
Total cached 92.05MB for all open files
```

Name: 文件名称

Cached pages/size: 文件目前占用的 page 页数/大小

Total pages: 当文件全部命中 cache 时，需要的 page 总数

Hit percent: page cache 命中率

Comm:Pid: 进程名称: 进程标识符

参考文献

- [1] 《BPF Performance Tools》 chapter 1/2/3/4/9
- [2] <https://github.com/iovisor/bcc>
- [3] [Linux Storage Stack Diagram - Thomas-Krenn-Wiki-en](#)