

思 路 分 享

eBPF & Linux I/O Stack Tracing

汇报人：赵家乐

CONTENT

- 1 Introduction: eBPF & BCC
- 2 Introduction: Linux I/O Stack
- 3 Trace Linux I/O Stack by eBPF
- 4 How to write a BCC program
- 5 Summary



ONE

Introduction: eBPF & BCC

Part. 01



eBPF



What does eBPF do ?

extend Berkeley Packet Filter

eBPF can run sandboxed programs in the operating system

kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change

kernel source code or load kernel modules.

Strengths of eBPF

Compared with changing kernel source code & loading kernel modules.

The answer to the question why we use eBPF to trace Linux I/O Stack.

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。

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。

安全与稳定

灵活

跨平台兼容

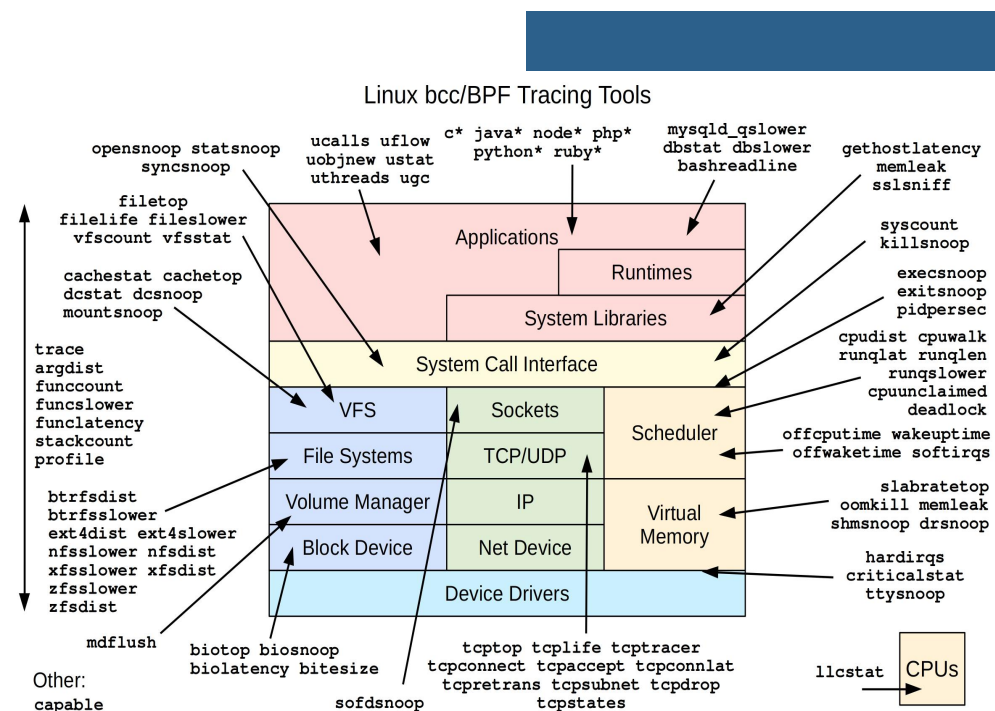
隔离

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。

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。

BPF Compiler Collection

- BCC 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++





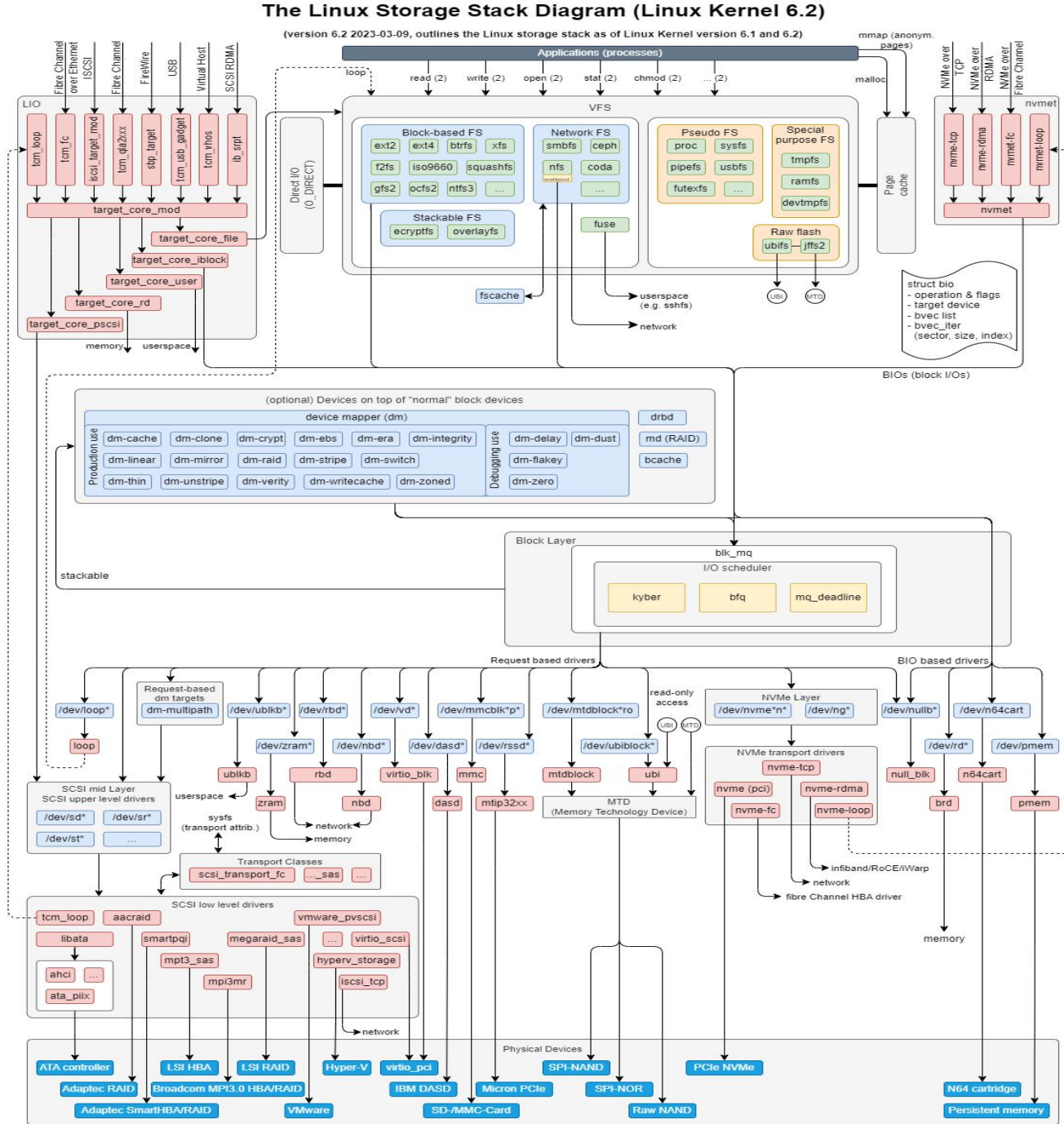
TWO

Introduction:Linux I/O Stack

Part. 02

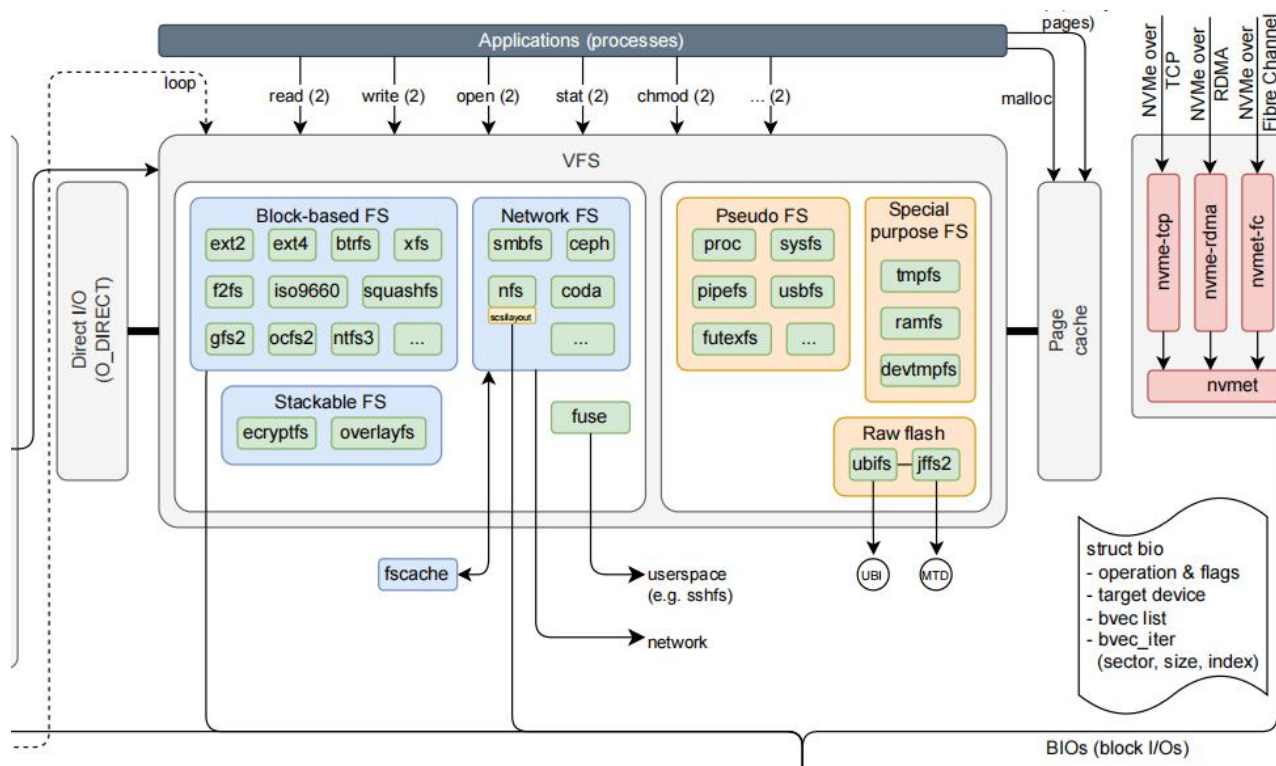


Overall Structure



Linux I/O Stack --- fs layer

将用户空间的I/O操作转化为底层数据结构



Virtual FS

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，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`。

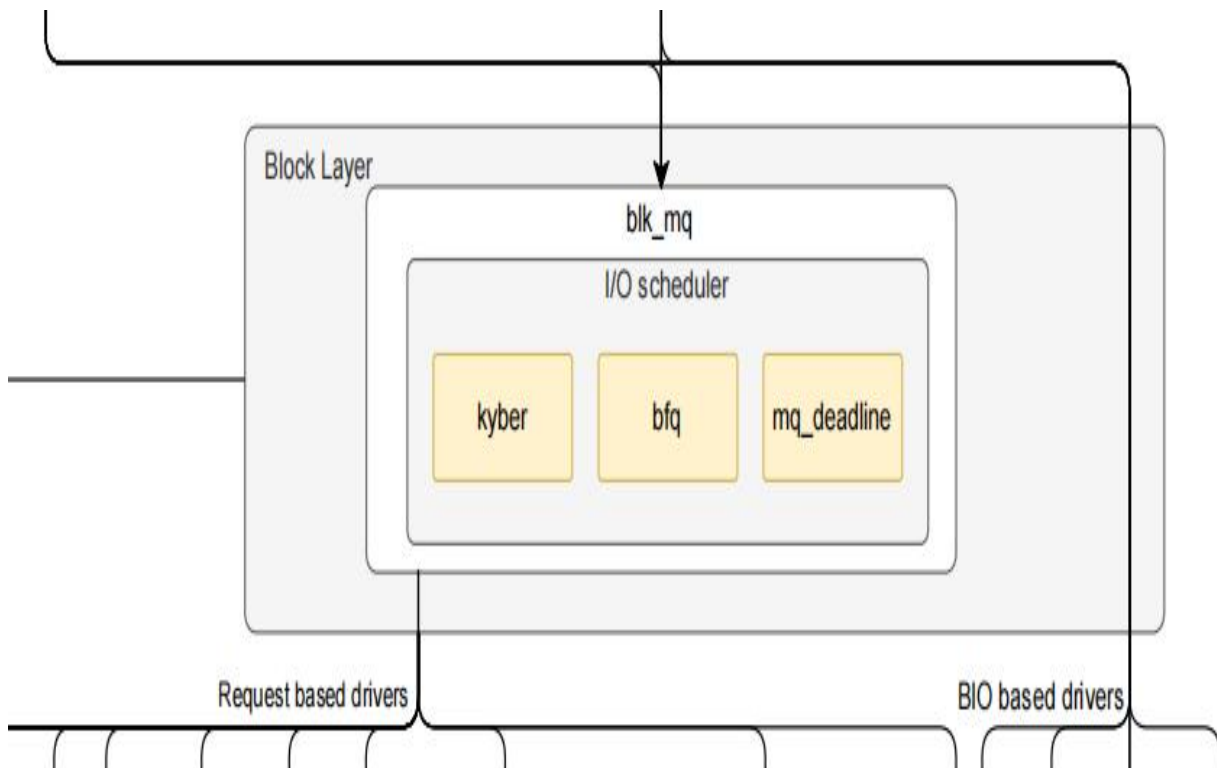
Specific FS: e.g. ext4

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。

Linux I/O Stack --- block layer

将fs层的bio按照一定策略、按照不同的块设备分发给对应的driver程序 (request/bio)

对于多队列模型，请求通常通过blk_mq_ctx和blk_mq_ops结构传递



Request Queue

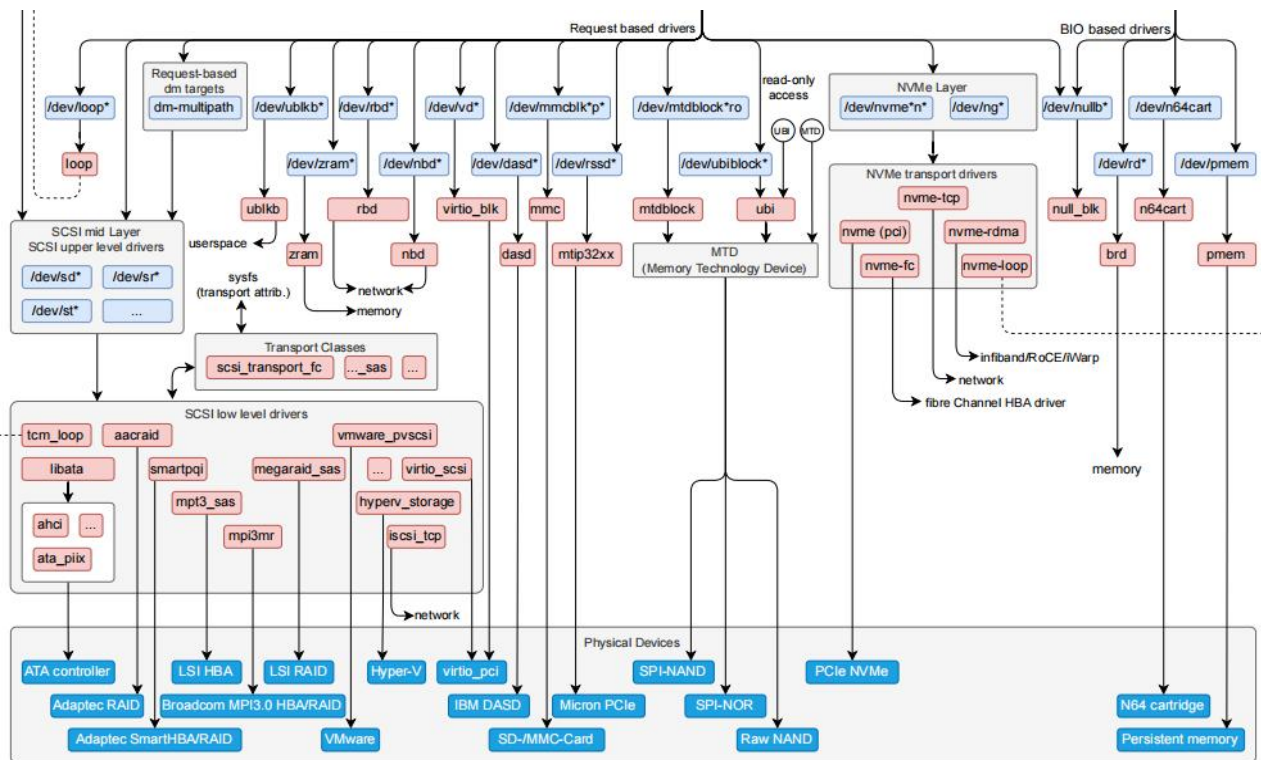
Single-Queue block layer维护了一个全局请求队列，用于存储和管理来自fs layer的I/O请求。这些请求通常是以bio (block I/O) 结构的形式提交的。请求队列可以对I/O请求进行排序、合并和调度，以优化磁盘访问性能。

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。

I/O Scheduler

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、N00P、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。

Linux I/O Stack --- driver layer



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。

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。

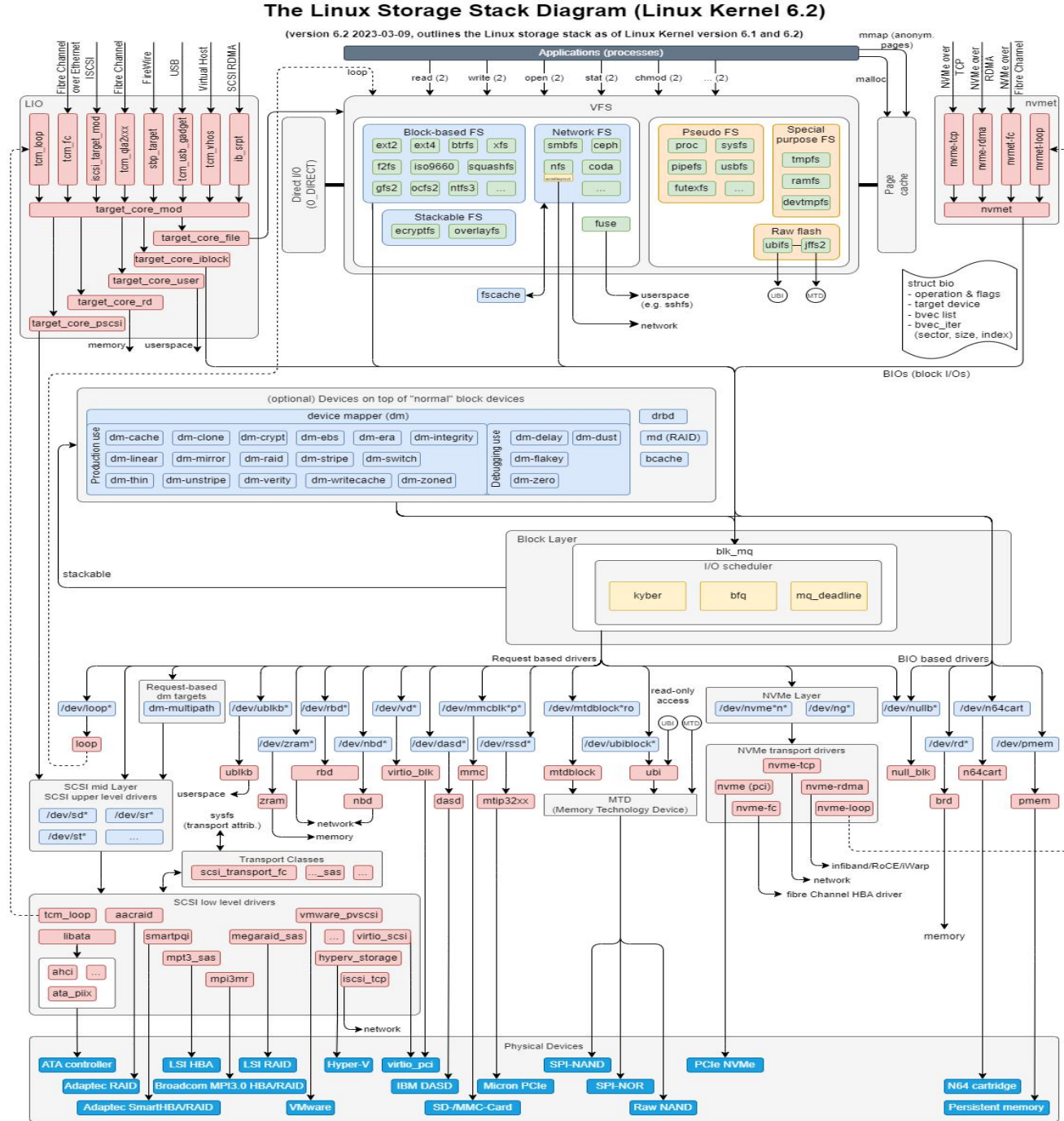
处理设备中断和状态

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fs层。

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fs层操作的完成情况。

Overall Structure





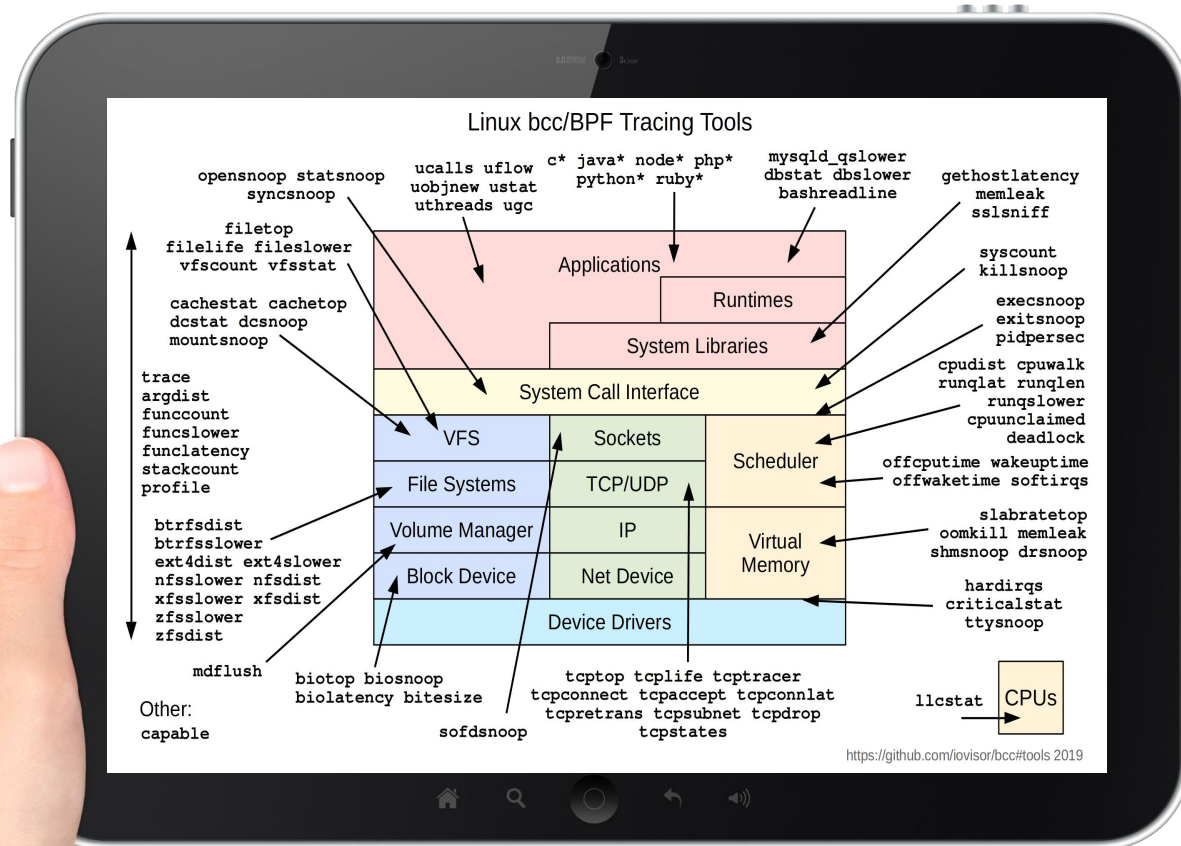
Three

Trace Linux I/O Stack by eBPF

Part. 03



Tracing



What is Tracing ?

Tracing is event-based recording.

A hallmark of a tracer is its ability to record raw events and event metadata.

biolateny

```
root@jiale-virtual-machine:/usr/share/bcc/tools# ./biolateny -D
Tracing block device I/O... Hit Ctrl-C to end.
```

^C

disk = sda

usecs	: count	distribution
0 -> 1	: 0	
2 -> 3	: 0	
4 -> 7	: 0	
8 -> 15	: 0	
16 -> 31	: 0	
32 -> 63	: 0	
64 -> 127	: 33	****
128 -> 255	: 169	*****
256 -> 511	: 325	*****
512 -> 1023	: 29	***
1024 -> 2047	: 6	

```
root@jiale-virtual-machine:/usr/share/bcc/tools# ./biolateny
Tracing block device I/O... Hit Ctrl-C to end.
```

usecs	: count	distribution
0 -> 1	: 0	
2 -> 3	: 0	
4 -> 7	: 0	
8 -> 15	: 0	
16 -> 31	: 0	
32 -> 63	: 0	
64 -> 127	: 9	*
128 -> 255	: 73	*****
256 -> 511	: 312	*****
512 -> 1023	: 108	*****
1024 -> 2047	: 39	*****
2048 -> 4095	: 4	
4096 -> 8191	: 0	
8192 -> 16383	: 1	

```
^Croot@jiale-virtual-machine:/usr/share/bcc/tools# ./biolateny -Q
Tracing block device I/O... Hit Ctrl-C to end.
```

^C

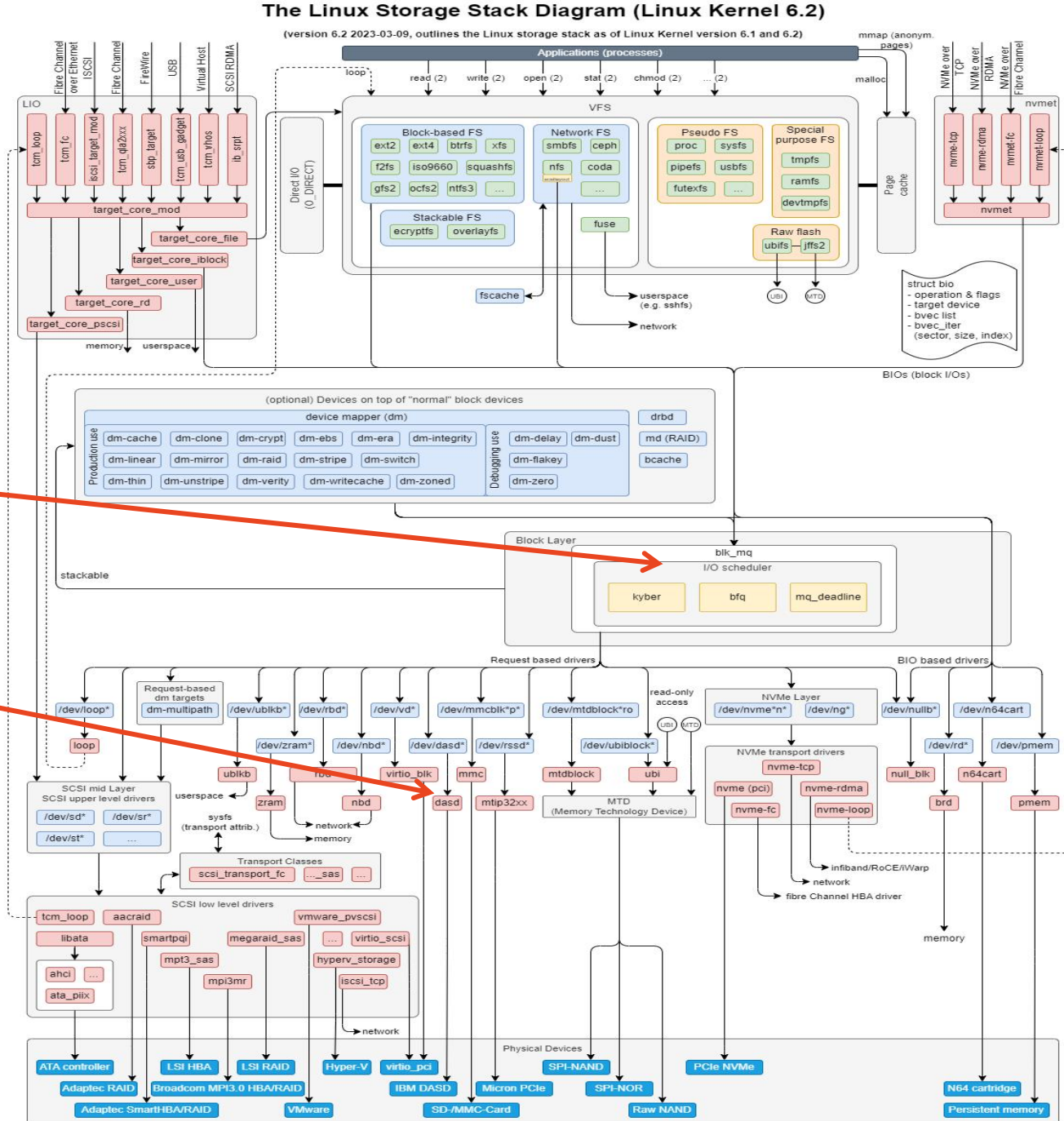
usecs	: count	distribution
0 -> 1	: 0	
2 -> 3	: 0	
4 -> 7	: 0	
8 -> 15	: 0	
16 -> 31	: 0	
32 -> 63	: 0	
64 -> 127	: 0	
128 -> 255	: 18	*
256 -> 511	: 423	*****
512 -> 1023	: 230	*****
1024 -> 2047	: 181	*****
2048 -> 4095	: 23	**
4096 -> 8191	: 15	*

biolateness

- `__blk_account_io_start` 用于跟踪块设备输入/输出 (I/O) 操作的开始时间, 并更新 I/O 统计信息。它是由 Linux 内核中的块设备驱动程序调用的。
- 与其对应, `__blk_account_io_done` 用于跟踪块设备输入/输出 (I/O) 操作的完成时间。
- `blk_start_request` 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。

```
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")
```


Overall Structure



ext4slower

```
root@jiale-virtual-machine:/usr/share/bcc/tools# ./ext4slower -j 1
ENDTIME_us,TASK,PID,TYPE,BYTES,OFFSET_b,LATENCY_us,FILE
1485071784,read_file,40890,R,4096,90112,3466,example.txt
1487058908,read_file,40890,R,4096,28672,1288,example.txt
1492469291,read_file,40890,R,4096,20480,28490,example.txt
1509878951,read_file,40890,R,4096,61440,3544,example.txt
1509949258,read_file,40890,R,4096,77824,1859,example.txt
1509965368,read_file,40890,R,4096,36864,2222,example.txt
1512002198,NetworkManager,865,S,0,0,1274,timestamps.7CJ831
1512004219,NetworkManager,865,S,0,0,1845,NetworkManager
1512622983,journal-offline,434,S,0,0,11677,user-1000.journal
1512624448,journal-offline,434,S,0,0,1416,user-1000.journal
1513864236,systemd-journal,434,S,0,0,1247,user-1000.journal
1526554977,read_file,40890,R,4096,98304,1963,example.txt
1526684429,read_file,40890,R,4096,102400,1896,example.txt
1526694981,read_file,40890,R,4096,61440,1395,example.txt
1526705191,read_file,40890,R,4096,110592,9053,example.txt
1527442364,read_file,40890,R,4096,40960,1554,example.txt
1527517710,read_file,40890,R,4096,94208,1082,example.txt
```

```
^Croot@jiale-virtual-machine:/usr/share/bcc/tools# ./ext4slower -p 40890 1
Tracing ext4 operations slower than 1 ms
TIME      COMM      PID    T BYTES  OFF_KB  LAT(ms)  FILENAME
12:31:59  read_file  40890  R 4096   108     1.59     example.txt
12:31:59  read_file  40890  O 0       0       1.50     example.txt
12:31:59  read_file  40890  R 4096    48     2.03     example.txt
12:31:59  read_file  40890  R 4096    72     2.95     example.txt
12:31:59  read_file  40890  R 4096    24     2.05     example.txt
12:31:59  read_file  40890  R 4096    96     3.17     example.txt
12:31:59  read_file  40890  R 4096    76     1.13     example.txt
12:31:59  read_file  40890  R 4096    88     3.34     example.txt
12:31:59  read_file  40890  R 4096    60     1.33     example.txt
12:32:00  read_file  40890  R 4096   112     1.64     example.txt
12:32:00  read_file  40890  R 4096   116     1.61     example.txt
12:32:01  read_file  40890  R 4096    92     1.01     example.txt
12:32:02  read_file  40890  R 4096    88     1.78     example.txt
12:32:02  read_file  40890  R 4096    44     1.19     example.txt
12:32:02  read_file  40890  R 4096    88     5.18     example.txt
12:32:04  read_file  40890  R 4096    88     1.50     example.txt
12:32:05  read_file  40890  R 4096    64     1.34     example.txt
12:32:06  read_file  40890  R 4096    28     1.14     example.txt
12:32:06  read_file  40890  R 4096    12    10.44     example.txt
12:32:06  read_file  40890  R 4096    16    11.22     example.txt
12:32:06  read_file  40890  R 4096    16     5.81     example.txt
12:32:06  read_file  40890  R 4096   116     2.15     example.txt
12:32:07  read_file  40890  R 4096   116     1.64     example.txt
```

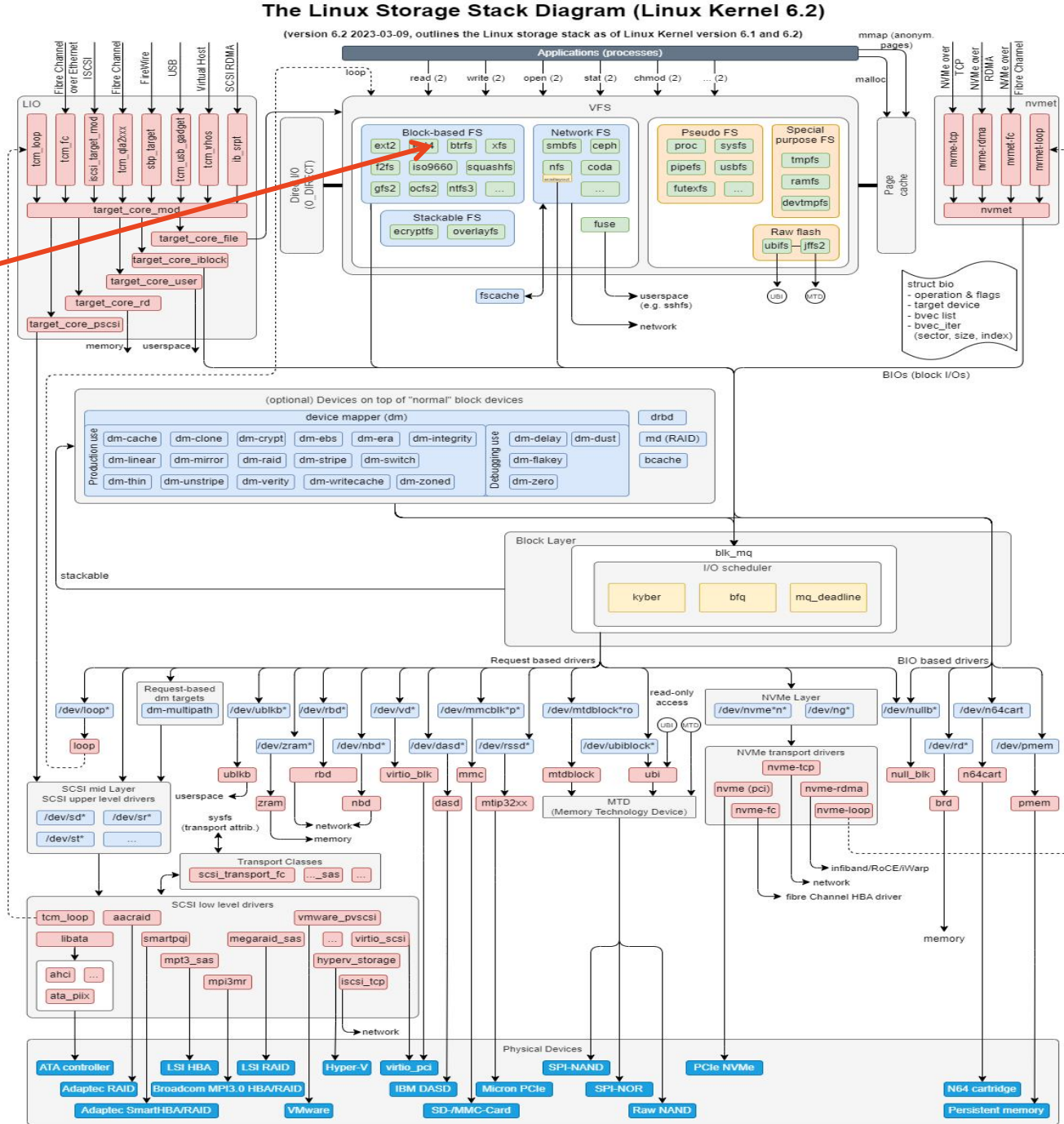
```
^Croot@jiale-virtual-machine:/usr/share/bcc/tools# ./ext4slower 1
Tracing ext4 operations slower than 1 ms
TIME      COMM      PID    T BYTES  OFF_KB  LAT(ms)  FILENAME
12:20:34  read_file  40890  R 4096    60     2.25     example.txt
12:20:37  read_file  40890  R 4096    92     2.58     example.txt
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
```


ext4s lower

- 文件系统通过向内核提供一组操作函数来实现文件访问。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。

```
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")
```

Overall Structure





Four

How to write a BCC program

Part. 04



trace_bio.c

```
#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;
}
```

trace_bio.c

```
// 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;
}
```


trace_bio.py

```
# 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
    ]
```


trace_bio.py

```
# 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()
```

run trace_bio.py

```
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
```



Five

S u m m a r y

Part. 05



Summary



思 路 分 析

Thank you for your listening!

汇报人：赵家乐