Spoock

乘物游心 上善若水

perf_event和ringbuf原理介绍和使用

2023-09-16

说明

本篇文章主要是说明 `perf_event` 和 `ringbuf` 的使用、原理已经两者之间的差异。

perf buffer

ebpf中提供了内核和用户空间之间高效地交换数据的机制–perf buffer，它是一种per-cpu的环形缓冲区，当我们需要将ebpf收集到的数据发送到用户空间记录或者处理时，就可以用perf buffer来完成。它还有如下特点：

1. 能够记录可变长度数据记；
2. 能够通过内存映射的方式在用户态读取读取数据，而无需通过系统调用陷入到内核去拷贝数据；
3. 实现epoll通知机制

ebpf提供了专门的map和helper function来使用perf buffer，如下是最常用的两个：

1. `map: BPF_MAP_TYPE_PERF_EVENT_ARRAY`： 此类型map专门用于perfbuffer
2. `helper function: bpf_perf_event_output`： 用于通知用户态拷贝数据；

有关 `perf buffer` 的具体应用，可以参考 perf_event

```
1  struct bpf_map_def SEC("maps/my_map") my_map = {
2      .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
3      .key_size = sizeof(int),
4      .value_size = sizeof(u32),
```

```
    .max_entries = 1024,
};

struct data_t {
    u32 pid;
};

SEC("kprobe/vfs_mkdir")
int kprobe_vfs_mkdir(void *ctx)
{
    bpf_printk("mkdir_perf_event (vfs hook point)%u\n",bpf_get_current_pid_tgid());
    struct data_t data = {};
    data.pid = bpf_get_current_pid_tgid();
    bpf_perf_event_output(ctx, &my_map, BPF_F_CURRENT_CPU, &data, sizeof(data));
    return 0;
};
```

在 BPF_MAP_TYPE_PERF_EVENT_ARRAY 类型的映射中， key_size 和 value_size 的含义与其他类型的映射略有不同。

- key_size ： 这个字段指定了映射的键的大小。对于 BPF_MAP_TYPE_PERF_EVENT_ARRAY 类型的映射，键通常是CPU的编号。因此， key_size 通常被设置为 sizeof(int) 。

- value_size ： 这个字段指定了映射的值的大小。对于 BPF_MAP_TYPE_PERF_EVENT_ARRAY 类型的映射，值是一个文件描述符（file descriptor，简称fd），该文件描述符关联了一个perf event。因此， value_size 通常被设置为 sizeof(u32) 。而不是对应的实际需要传输的结构体的大小，所以在本例中就不能写成 sizeof(struct data_t) 。

上面的代码中还会存在两次内存拷贝的问题。

1. struct data_t data = {}; ，在栈上创建数据，然后将栈上的数据拷贝到 perf buffer 中，是一次内存拷贝；
2. bpf_perf_event_output(ctx, &my_map, BPF_F_CURRENT_CPU, &data, sizeof(data)); ，将 perf buffer 中的数据拷贝到用户空间，是一次内存拷贝；

上面的代码中，因为存在 struct data_t data = {} ，是直接栈中申请的结构体，此时ebpf verify验证器会限制结构体不能超过512字节，影响功能开发

perf buffer heap

刚才说过， struct data_t data = {} 因为是在栈上创建的，所以会存在大小限制，结构体大小不能超过512字节。所以常见的做法是在堆上创建。

```c
struct bpf_map_def SEC("maps/heap") heap = {
    .type = BPF_MAP_TYPE_PERCPU_ARRAY,
    .key_size = sizeof(int),
    .value_size = sizeof(u32),
    .max_entries = 1,
};

struct data_t {
    u32 pid;
};

struct bpf_map_def SEC("maps/perf_map") perf_map = {
    .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    .key_size = sizeof(int),
    .value_size = sizeof(u32),
    .max_entries = 1024,
};

SEC("kprobe/vfs_mkdir")
int kprobe_vfs_mkdir(void *ctx)
{
    bpf_printk("mkdir_perf_event (vfs hook point)%u\n", bpf_get_current_pid_tgid());
    int zero = 0;
    struct data_t *data = bpf_map_lookup_elem(&heap, &zero);
    if (!data) {
        return 0;
    }

    data->pid = bpf_get_current_pid_tgid();
    bpf_perf_event_output(ctx, &perf_map, BPF_F_CURRENT_CPU, data, sizeof(*data));
    return 0;
}
```

通过 `struct data_t *data = bpf_map_lookup_elem(&heap, &zero)` 的方式， `data_t` 就是创建的一个map对象。其中的 `heap` 声明的类型是 `BPF_MAP_TYPE_PERCPU_ARRAY` 。

`BPF_MAP_TYPE_PERCPU_ARRAY` 类型的映射是一种特殊的数组映射，它为每个CPU核心存储一份数据副本。这种映射类型的主要优点是它可以在不需要任何锁的情况下并发地更新数据，因为每个CPU核心都在操作自己的数据副本。

通过 `BPF_MAP_TYPE_PERF_EVENT_ARRAY` 创建map对象，就可以避免512字节大小的限制，但是还是会存在两次拷贝的问题。

1. `BPF_MAP_TYPE_PERCPU_ARRAY` 的优势：

- 并发更新： `BPF_MAP_TYPE_PERCPU_ARRAY` 为每个CPU核心提供了一份数据副本，因此可以在每个核心上并发地更新数据，而不需要任何锁。这提供了高性能的并发访问能力。
- 无锁访问：由于每个CPU核心都有自己的数据副本，所以访问这些副本时不需要进行锁操作，避免了锁竞争的开销。
- 高效的局部性：由于每个CPU核心只访问自己的数据副本，这种映射类型在具有局部性的工作负载中表现出色，避免了对共享数据的频繁访问。

2. `BPF_MAP_TYPE_PERF_EVENT_ARRAY` 的优势：

   - 与性能事件相关： `BPF_MAP_TYPE_PERF_EVENT_ARRAY` 用于与性能事件相关联的数据传输。它允许将数据从内核空间传输到用户空间，并用于性能分析和调试。
   - 与 `perf_event` 子系统集成： `BPF_MAP_TYPE_PERF_EVENT_ARRAY` 类型的映射与Linux内核的 `perf_event` 子系统集成，可以方便地与性能事件相关的数据进行交互。
   - 适用于性能分析：这种映射类型对于需要收集和分析性能数据的应用程序和工具非常有用，可以实时捕获和处理性能事件数据。

总之， `BPF_MAP_TYPE_PERCPU_ARRAY` 适用于需要高性能并发访问的情况，而 `BPF_MAP_TYPE_PERF_EVENT_ARRAY` 适用于与性能事件相关的数据传输和性能分析。选择合适的映射类型取决于你的具体需求和使用场景。

perf buffer 问题

传输速度

perfbuf 为每个 CPU 分配一个独立的缓冲区，这意味着开发者通常需要 在内存效率和数据丢失之间做出折中：

1. 越大的 per-CPU buffer 越能避免丢数据，但也意味着大部分时间里，大部分内存都是浪费的；
2. 尽量小的 per-CPU buffer 能提高内存使用效率，但在数据量陡增（毛刺）时将导致丢数据;

对于那些大部分时间都比较空闲、周期性来一大波数据的场景， 这个问题尤其突出，很难在两者之间取得一个很好的平衡。

ringbuf 的解决方式是分配一个所有 CPU 共享的大缓冲区

- 大缓冲区，意味着能更好地容忍数据量毛刺
- 共享，则意味着内存使用效率更高

另外，ringbuf 内存效率的扩展性也更好，比如CPU数量从16增加到32时：

- perfbuf 的总 buffer 会跟着翻倍，因为它是 per-CPU buffer
- ringbuf 的总 buffer 不一定需要翻倍，就足以处理扩容之后的数据量

事件顺序

perbuf的per-CPU特性导致内核调度器在不同CPU上调度进程时， 对于那些存活时间很短的进程fork,exec,exit会在极短的时间内在不同 CPU 上执行，这样用户态在获取数据时，无法按照事件发生的顺序获取数据。

但对于 ringbuf 来说，因为它是共享的同一个缓冲区。ringbuf 保证 如果事件 A 发生在事件 B 之前，那 A 一定会先于 B 被提交，也会在 B 之前被消费。

数据复制

BPF 程序使用 perfbuf 时，必须先初始化一份事件数据，然后将它复制到 perfbuf， 然后才能发送到用户空间。这意味着数据会被复制两次：

- 复制到一个局部变量（a local variable）或 per-CPU array （BPF 的栈空间很小，因此较大的变量无法放到栈上，后面有例子）中
- 复制到 perfbuf 中
  更糟糕的是，如果 perfbuf 已经没有足够空间放数据了，那第一步的复制完全是浪费的。

BPF ringbuf 提供了一个可选的 reservation/submit API 来避免这种问题

- 首先申请为数据预留空间（reserve the space
- 预留成功后
  - 应用就可以直接将准备发送的数据放到 ringbuf 了，从而节省了 perfbuf 中的第一次复制，
  - 将数据提交到用户空间将是一件极其高效、不会失败的操作，也不涉及任何额外的内存复制
- 如果因为 buffer 没有空间而预留失败了，那 BPF 程序马上就能知道，从而也不用再 执行 perfbuf 中的第一步复制

参考：http://arthurchiao.art/blog/bpf-ringbuf-zh/

ringbuf

由于 per-CPU 的存在的两个严重缺陷：

- 内存使用效率低下（inefficient use of memory）
- 事件顺序无法保证（event re-ordering）

因此内核 5.8 引入了 ringbuf 来解决这个问题。 ringbuf 是一个“多生产者、单消费者”（multi-producer, single-consumer，MPSC）队列，可安全地在多个CPU之间共享和操作。perfbuf 支持的一些功能它都支持，包括，

1. 可变长数据（variable-length data records）
2. 通过 memory-mapped region 来高效地从 userspace 读数据，避免内存复制或系统调用

3. 支持 epoll notifications 和 busy-loop 两种获取数据方式

重点说明 `epoll notifications` 和 `busy-loop` 两种获取数据方式，这两种方式和后面的参数配置有关。

1. **epoll notifications**：这种方式使用了Linux的epoll机制。用户空间的应用程序会创建一个epoll实例，并将ringbuf的文件描述符添加到这个epoll实例中。然后，应用程序会调用 `epoll_wait()` 函数来等待新的数据。当内核空间有新的数据写入ringbuf时，会触发一个epoll事件， `epoll_wait()` 函数会返回，应用程序就可以从ringbuf中读取新的数据。这种方式的优点是，当没有新的数据时，应用程序可以进入休眠状态，不会消耗CPU资源。但是，它的缺点是需要处理epoll的相关逻辑，而且在数据到达时需要唤醒应用程序，这会增加一些延迟。

2. **busy-loop**：这种方式是一种轮询机制。用户空间的应用程序会不断地检查ringbuf，看是否有新的数据。如果有新的数据，就立即处理；如果没有新的数据，就继续检查。这种方式的优点是可以最快地处理新的数据，因为它总是在检查新的数据，不需要等待epoll事件。但是，它的缺点是会持续占用CPU资源，即使没有新的数据。

还是先来看如何利用 `ringbuf` 向用户态传输数据。示例程序参考：ringbuf

内核态

```
struct {
        __uint(type, BPF_MAP_TYPE_RINGBUF);
        __uint(max_entries, 1 << 24);
} events SEC(".maps");

struct task_info {
        u32 pid;
        u8 comm[80];
};

SEC("kprobe/vfs_mkdir")
int kprobe_vfs_mkdir(void *ctx)
{
    bpf_printk("mkdir (vfs hook point) by using ringbuf_map\n");
    struct task_info task_data = {};
    u64 id   = bpf_get_current_pid_tgid();
    u32 tgid = id >> 32;
    task_data.pid = tgid;
    bpf_get_current_comm(&task_data.comm, 80);
    bpf_printk("pid: %d\n", tgid);
    bpf_printk("comm: %s\n", task_data.comm);
    bpf_ringbuf_output(&events,&task_data, sizeof(task_data), 0 /* flags */);
```

```
    return 0;
};
```

为了方便和 bpf_perf_event_output 方式对比，下放就是一段通过 bpf_perf_event_output 传输数据的代码例子：

```
struct bpf_map_def SEC("maps/my_map") my_map = {
    .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    .key_size = sizeof(int),
    .value_size = sizeof(u32),
    .max_entries = 1024,
};

struct data_t {
    u32 pid;
};

SEC("kprobe/vfs_mkdir")
int kprobe_vfs_mkdir(void *ctx)
{
    bpf_printk("mkdir_perf_event (vfs hook point)%u\n",bpf_get_current_pid_tgid());
    struct data_t data = {};
    data.pid = bpf_get_current_pid_tgid();
    bpf_perf_event_output(ctx, &my_map, BPF_F_CURRENT_CPU, &data, sizeof(data));
    return 0;
};
```

bpf_ringbuf_output 和 bpf_perf_event_output 有如下几点不同：

1. ringbuf map 的大小（max_entries）可以在 BPF 侧指定了，注意这是所有 CPU 共享的大小
2. bpf_perf_event_output() 替换成了类似的 bpf_ringbuf_output()，后者更简单，不需要 BPF context 参数
3. ringbuf map 对应的类型需要设置为 BPF_MAP_TYPE_RINGBUF

用户态代码

```
m := &manager.Manager{
    Probes: []*manager.Probe{
        &manager.Probe{
            UID:              "MyFirstHook",
            Section:          "kprobe/vfs_mkdir",
            AttachToFuncName: "vfs_mkdir",
            EbpfFuncName:     "kprobe_vfs_mkdir",
        },
    },
```

```
        RingbufMaps: []*manager.RingbufMap{
            &manager.RingbufMap{
                Map: manager.Map{
                    Name: "events",
                },
                RingbufMapOptions: manager.RingbufMapOptions{
                    DataHandler: myDataHandler,
                },
            },
        },
    }
```

相比之前的代码，需要修改的地方不多，只是将 perfbuf map 替换成了 ringbuf map ，并且 ringbuf map 的 DataHandler 回调函数，用来处理内核态传输过来的数据。其中的 DataHandler 的回调函数的原型如下：

```
DataHandler func(CPU int, data []byte, perfMap *RingbufMap, manager *Manager)
```

也和之前的 perfbuf map 的回调函数一样。

实际程序运行得到的结果如下：

```
successfully started, head over to /sys/kernel/debug/tracing/trace_pipe
Generating events to trigger the probes ...
creating /tmp/test_folder
received: pid:1118833,comm:main
removing /tmp/test_folder
```

通过回调函数，成功输出 received: pid:1118833,comm:main ，解析得到 pid 是 1118833 ，对应的进程是 main 。

通过 /sys/kernel/debug/tracing/trace_pipe 得到的结果如下：

```
<...>-1118835 [010] d...1 702590.314924: bpf_trace_printk: pid: 1118833, comm: main
```

trace_pipe 和用户态程序解析结果是一致的，说明数据传输成功。

ringbuf_commit

bpf_ringbuf_output() API 的目的是确保从 perfbuf 到 ringbuf 迁移时无需对 BPF 代码做重大改动，但这也意味着它继承了 perfbuf API 的一些缺点：

1. **额外的内存复制（extra memory copy）**，这意味着需要额外的空间来构建 event 变量，然后将其复制到 buffer。不仅低效， 而且经常需要引入只有一个元素的 per-CPU array，增加了不必要的处理复杂性。
2. **非常晚的 buffer 空间申请（data reservation）**，如果这一步失败了（例如由于用户空间消费不及时导致 buffer 满了，或者有大量 突发事件导致 buffer 溢出了），那上一步的工作将变得完全无效，浪费内存空间和计算资源。

如果能提前知道事件将在第二步被丢弃，就无需做第一步了， 节省一些内存和计算资源，消费端反而因此而消费地更快一些。 但 xxx_output() 风格的 API 是无法实现这个目的的。

为了解决 `xxx_output()` 存在的问题，引入了 入了新的 `bpf_ringbuf_reserve()/bpf_ringbuf_commit()` API

1. 提前预留空间，或者能立即发现没有可以空间了(返回NULL)
2. 预留成功后，一旦数据写好了，将它发送到 userspace 是一个不会失败的操作。也就是说只要 `bpf_ringbuf_reserve()` 返回非空，那随后的 `bpf_ringbuf_commit()` 就永远会成功，因此它没有返回值。

另外，ring buffer 中预留的空间在被提交之前，用户空间是看不到的， 因此 BPF 程序可以从容地组织自己的 event 数据，不管它有多复杂、需要多少步骤。 这种方式也避免了额外的内存复制和临时存储空间(extra memory copying and temporary storage spaces)

当然 `bpf_ringbuf_commit` 方法还是会存在一个限制，BPF 校验器在校验时（at verification time）， 必须知道预留数据的大小 （size of the reservation），因此不支持动态大小的事件数据。对于动态大小的数据，用户只能退回到用 `bpf_ringbuf_output()` 方式来提交，忍受额外的数据复制开销；

```
bpf_printk("mkdir (vfs hook point) by using ringbuf_map\n");
struct task_info *task_data;
task_data = bpf_ringbuf_reserve(&events, sizeof(*task_data), 0);
if (!task_data) {
    bpf_printk("ringbuf_reserve failed\n");
    return 0;
}
task_data->pid = bpf_get_current_pid_tgid() >> 32;
bpf_get_current_comm(&task_data->comm, sizeof(task_data->comm));
bpf_printk("pid: %d, comm: %s\n", task_data->pid, task_data->comm);
bpf_ringbuf_submit(task_data, 0);
return 0;
```

通过 `bpf_ringbuf_reserve`， 申请内存。之后就可以直接操作 `task_data`， 不需要再拷贝到 `task_data` 中，最后通过 `bpf_ringbuf_submit` 提交数据。

`bpf_ringbuf_submit` 的函数原型如下：

```
1   void bpf_ringbuf_submit_dynptr(struct bpf_dynptr *ptr, u64 flags)
```

`flags` 一共有三个选项：

- `BPF_RB_NO_WAKEUP` ，这个标志位表示在提交数据后不唤醒用户空间的进程。这意味着，即使有新的数据提交到ringbuf，用户空间的进程也不会被唤醒。这个标志位可以用在busy-loop模式中，因为在这种模式下，用户空间的进程总是在检查新的数据，不需要被唤醒。
- `BPF_RB_FORCE_WAKEUP` ，这个标志位表示在提交数据后强制唤醒用户空间的进程。这意味着，只要有新的数据提交到ringbuf，用户空间的进程就会被唤醒。这个标志位可以用在epoll notifications模式中，因为在这种模式下，用户空间的进程在没有新的数据时会进入休眠状态，需要被唤醒。
- 0,当新的数据被提交到ringbuf时，内核会根据当前的条件决定是否唤醒用户空间的进程。

使用 `bpf_ringbuf_commit()` 提交数据，用户态的代码不需要改变，所以这里就不再赘述。

总结

本文针对 `perf buffer` 和 `ringbuf` 的设计思想进行了说明，同时通过代码演示了如何分别使用这两种方式。 `perf buffer` 和 `ringbuf` 各有优劣，具体使用哪种方式，需要根据实际情况进行选择。尤其是 `ringbuf` 还和内核版本有关。

希望本文能够帮助大家理解 `perf buffer` 和 `ringbuf` 的设计思想，以及如何使用。

参考

https://www.cnblogs.com/liuhailong0112/p/17077393.html

https://github.com/cilium/ebpf/blob/main/examples/kprobe_percpu/kprobe_percpu.c

https://elixir.bootlin.com/linux/v4.6/source/samples/bpf/trace_output_kern.c

https://senberhu.github.io/ebpf/Perf-Ring-Buffer%E5%AF%B9%E6%AF%94.html

http://arthurchiao.art/blog/bpf-ringbuf-zh/

https://github.com/anakryiko/bpf-ringbuf-examples/blob/main/src/ringbuf-output.bpf.c

本文标题: perf_event和ringbuf原理介绍和使用

本文作者: Spoock

创建时间: 2023年09月16日

修改时间: 2023年09月16日

引用链接: https://blog.spoock.com/2023/09/16/eBPF-event/

#eBPF

vArmor中的ptrace阻断功能实现分析

vArmor-ebpf-loader项目介绍和功能解读