qemu Coroutine 执行流程分析

——————————————谢龙/2018/2/2

qemu 在处理 block 设备的后端数据时使用了 Coroutine 的技术，本文对 qemu Coroutine 执行流程进行分析。

1. Coroutine 实现机制

对于 OS 来说， Coroutine 不是线程，因为 OS 没有办法直接对 Coroutine 进行调度；但是 Coroutine 有独立的栈，所以所 Coroutine 就是一个在当前线程环境下的一个独立执行流，这个独立执行流的调度是通过代码主动进行切换的。Coroutine 的 ANSI-C 的简单实现可以看这篇文章：https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html 这篇文章介绍如何使用纯 c 的代码（macro）实现这样的一种效果：当一个函数被调用并返回后， 再次调用这个函数时继续从之前函数调用返回时的返回指令的下一条指令处继续执行。

qemu 通过如下 libc 提供的接口实现。

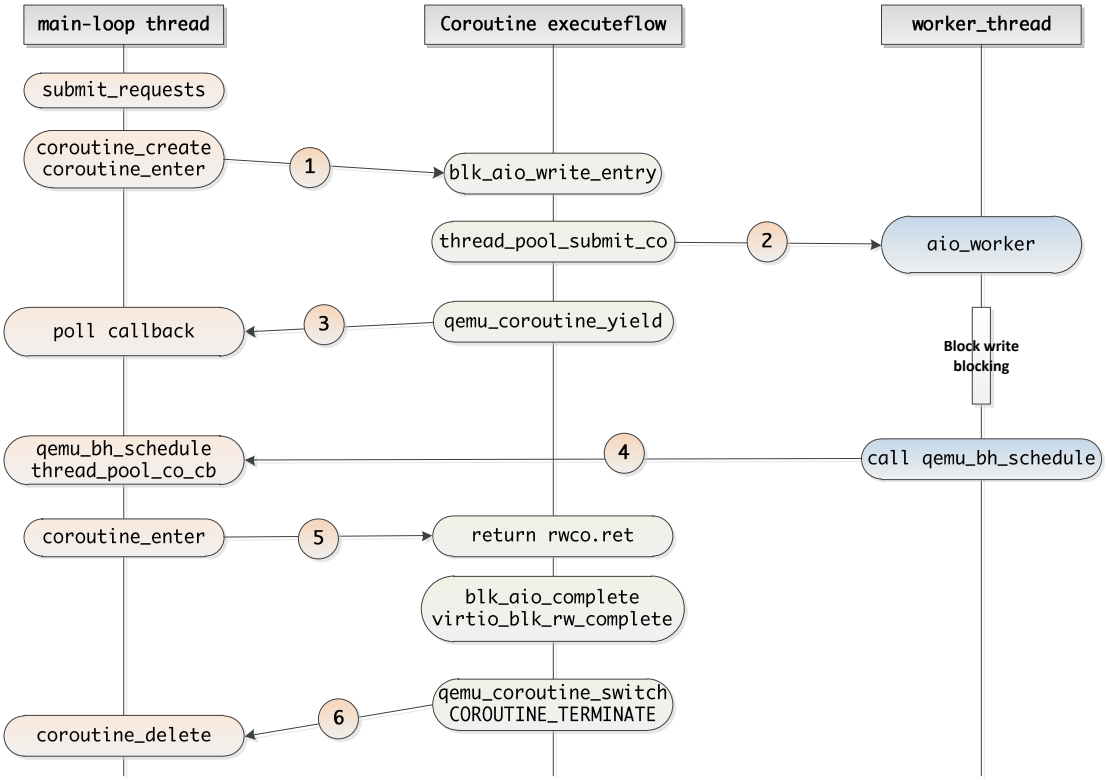
```
,----
| getcontext();
| makecontext();
| swapcontext();
`----
```

关于上述函数的说明可以直接 man makecontext， linux manual 里面有详细的说明。需要注意的时， qemu 并不是完全依靠上述的机制实现 Coroutine， 还结合 sigsetjmp()/siglongjmp()这几个接口。那么问题是 qemu 中实现 Coroutine 时为什么还用到 sigsetjmp()/siglongjmp()? 答案从 qemu 协程实现源码中可以找到。大致意思就是说：
先说为什么不直接使用 sigsetjmp()/siglongjmp()实现 Coroutine? 因为 sigsetjmp()/siglongjmp()只是工作在当前的栈上， 它只是提供了突破正常函数调用栈的限制， 而没有形成一个独立的执行流。那再回到我们提到的主要问题，即使用 sigsetjmp()/siglongjmp() 辅助实现 Coroutine， 主要原因是在调用 ucontext 这些接口时会保存 signal masks，这会存在一个系统调用的开销。 而 sigsetjmp()/siglongjmp()并不会保存 signal mask，所以结合这两者可以实现一个轻量级的 Coroutine.

qemu 从 commit: 00dccaf1f848290d979a4b1e6248281ce1b32aaa 引入了协程。

2. qemu Coroutine 执行流程

下图就是 qemu 在执行 block 操作时协程调度执行流：



步骤	说明
1	main-loop 线程接收到虚机的 block-req，创建一个 Coroutine,然后将执行流切入到 Coroutine 中执行；
2	Coroutine 继续将 block-req 提交到线程池里面的一个线程去执行；
3	Coroutine 将执行流切换回 main-loop；注意：此处可以看到 main-loop 是不会阻塞的；
4	当 worker_thread 完成 block-req 的执行后，需要将返回结果通知调用方即 Coroutine；如何通知？通过 qemu_bh_schedule 提交给 qemu 的 BH 去执行；
5	在 BH 中执行的 callback 为：thread_pool_co_cb， 这个函数中再次将执行流切换到 Coroutine 中。
6	Coroutine 此时已经获得 block-req 的执行结果了， 一方面将返回结果返给调用方，另一方面调用 block 设备注册的回调，更新 block 设备的状态信息。然后 Coroutine 将执行流切换回 main-loop，main-loop 中删除协程占用的资源，这样完成了一个 block-req 的处理。

3. 为什么使用 Coroutine 实现？

先看看如果对于上面的流程不用协程我们需要怎么处理？

不用协程的话，对于每个提交给 worker_thread 执行的 block-req， 在 main-loop 中都要有记录（暂且称之为“请求记录表”），当 worker_thread 执行完成后，需要查询请求记录表找到相应的 block-req， 更新这个 block-req 的状态，做一些收尾的工作。这样实际也实现了我们需要的功能；但是这样势必需要我们保存一个请求记录表，而且要去频繁的查询。与 Coroutine 相比，一方面代码不简洁，另外一方面估计性能也有损失（因为需要查询）。

若从内存占用的角度看，协程的其实可以看做是上述请求记录表的另外一种实现方式。

4. 关键代码分析

```
blk_aio_prwv
,----
| /**
|  * submit_requests 通过 blk_aio_pwritev 提交 req： 并提供了当 req 完成后的回调
|  * virtio_blk_rw_complete;
|  */
| blk_aio_pwritev(blk, sector_num << BDRV_SECTOR_BITS, qiov, 0,
|              virtio_blk_rw_complete, mrb->reqs[start]);
|
| BlockAIOCB *blk_aio_pwritev(BlockBackend *blk, int64_t offset,
|                         QEMUIOVector *qiov, BdrvRequestFlags flags,
|                         BlockCompletionFunc *cb, void *opaque)
| {
|       /* 这个函数中封装了 Coroutine 的执行函数 blk_aio_write_entry */
|       return blk_aio_prwv(blk, offset, qiov->size, qiov,
|                         blk_aio_write_entry, flags, cb, opaque);
| }
|
| static BlockAIOCB *blk_aio_prwv(BlockBackend *blk, int64_t offset, int bytes,
|                             QEMUIOVector *qiov, CoroutineEntry co_entry,
|                             BdrvRequestFlags flags,
|                             BlockCompletionFunc *cb, void *opaque)
| {
|       BlkAioEmAIOCB *acb;
|       Coroutine *co;
|
|       bdrv_inc_in_flight(blk_bs(blk));
|       /* 申请一个 BlkAioemaiocb， 这个对象封装了 req 需要的所有信息 */
|       acb = blk_aio_get(&blk_aio_em_aiocb_info, blk, cb, opaque);
|       acb->rwco = (BlkRwCo) {
|              .blk    = blk,
|              .offset = offset,
|              .qiov   = qiov,
|              .flags  = flags,
|              .ret    = NOT_DONE,
|       };
|       acb->bytes = bytes;
|       acb->has_returned = false;
|
|       /**
|        * 创建一个 Coroutine， Coroutine 的入口函数为： coroutine_trampoline，这个函数是对
|        * 此处参数 co_entry 的封装； 在创建 Coroutine 时， qemu 的做法是先将执行流切入到 Coroutine
|        * 中执行一段， 然后再切回来， 执行到哪儿呢？ 执行到 coroutine_trampoline 的 while(ture)... 处。
|        * 为后续 qemu_coroutine_enter 进入 Coroutine 开始真正执行用户提供的 co_entry 做好准备.
|        */
|       co = qemu_coroutine_create(co_entry, acb);
|       qemu_coroutine_enter(co);
|
```

```
|       acb->has_returned = true;
|       if (acb->rwco.ret != NOT_DONE) {
|             aio_bh_schedule_oneshot(blk_get_aio_context(blk),
|                                     blk_aio_complete_bh, acb);
|       }
|
|       return &acb->common;
| }
`----

,----
| int coroutine_fn thread_pool_submit_co(ThreadPool *pool, ThreadPoolFunc *func,
|                                        void *arg)
| {
|       ThreadPoolCo tpc = { .co = qemu_coroutine_self(), .ret = -EINPROGRESS };
|       assert(qemu_in_coroutine());
|       /**
|        * 将 io 请求提交给线程线程池中的线程处理，线程为 worker_thread
|     * worker_thread 调用 aio_worker 处理 req， 处理完成 req 后 qemu_bh_schedule(pool->completion_bh);
|        * 这个 bh 会调用此处的 thread_pool_co_cb.
|        */
|       thread_pool_submit_aio(pool, func, arg, thread_pool_co_cb, &tpc);
|       qemu_coroutine_yield();
|       /**
|        * 执行完成上面的语句后， 即完成了 req 的提交，并将执行流切换回 main_loop.
|        * 当通过上述过程调用 thread_pool_co_cb 时， 会将执行流程再次切入到此处， 返回 io 执行的结果。
|        */
|       return tpc.ret;
| }
|
| /**
|  * 注意：上述的 thread_pool_co_cb 必须有 worker_thread 提交到 bh 执行， 这样可以保证 main-loop 线程的串行化执行。
|  * 如果直接切换回 Coroutine，将会有竞争问题。
|  * 此外， 此处的 qemu_coroutine_yield();有一石两鸟的作用， 第一将执行流引到了 main-loop；第二同时记住了
Coroutine
|  * 的执行点， 使得 thread_pool_co_cb 可以再次回到此处， 当然限制只能通过 main-loop 执行流回到此处。
|  * 回到此处经过层层退出到 Coroutine 的入口 blk_aio_write_entry，然后执行 blk_aio_complete(acb)， 在这个函数中
|  * 会调用 virtioblk 注册的请求完成回调接口 virtio_blk_rw_complete，完成 virtio 队列更新等操作。
|  */
|
| /* 完成上述操作， 那么 Coroutine 如何结束呢？ */
| static void coroutine_trampoline(int i0, int i1)
| {
|       union cc_arg arg;
|       CoroutineUContext *self;
|       Coroutine *co;
|
|       arg.i[0] = i0;
|       arg.i[1] = i1;
```

```
|       self = arg.p;
|       co = &self->base;
|
|       /* Initialize longjmp environment and switch back the caller */
|       if (!sigsetjmp(self->env, 0)) {
|              siglongjmp(*(sigjmp_buf *)co->entry_arg, 1);
|       }
|
|       while (true) {
|              co->entry(co->entry_arg);
|              /**
|               * 此处的 entry 为 blk_aio_write_entry，执行完成这个函数后， 就完整的处理完了一个 blk 的 req；
|               * 注意当前还是处于 Coroutine 的执行流中， 通过下面的调用将 Coroutine 切换回 mail-loop BH 中，实际
|               * 是调用到 qemu_coroutine_enter 这个执行流中， 此处的参数为 COROUTINE_TERMINATE，在
|               * qemu_coroutine_enter 中根据这个标志将 Coroutine delete 了。这样 BH 也执行完成了，
|               * 执行流也回到 main-loop。世界终于清净了...
|               */
|              qemu_coroutine_switch(co, co->caller, COROUTINE_TERMINATE);
|       }
| }
`----
```