无锁队列详细分解 — 顶层设计

原创 DPDK开源社区 2016-12-09

作者 马良

无锁队列是一个非常经典的并行计算数据结构，已经有很多相关的文章以及论文对其进行了探讨。它在DPDK中是一个非常基础且关键的组件，其中包含了很多非常特定的优化技巧。本文试图从顶层设计和具体实现分别来阐述DPDK无锁队列的优点以及正确使用的边界条件。在开始之前，我们还需要先进行两个知识点的铺垫。

1 RTE_Ring 数据结构

```
struct rte_ring {

 ›   /* Ring producer status. */

 ›   struct prod {

 ›   ›   uint32_t watermark;      /**< Maximum itemsbefore EDQUOT. */

 ›   ›   uint32_t sp_enqueue;     /**< True, if single producer. */

 ›   ›   uint32_t size;           /**< Size of ring.*/

 ›   ›   uint32_t mask;           /**< Mask (size-1)of ring. */

 ›   ›   volatile uint32_thead;   /**< Producer head.*/

 ›   ›   volatileuint32_t tail;   /**< Producer tail.*/

 ›   } prod __rte_cache_aligned;
 ›   /* Ring consumer status. */

 ›   struct cons {

 ›   ›   uint32_t sc_dequeue;     /**< True, if single consumer. */

 ›   ›   uint32_t size;           /**< Size of thering. */
```

```
 ›   ›   uint32_t mask;          /**< Mask (size-1)of ring. */

 ›   ›   volatileuint32_t head;  /**< Consumer head.*/

 ›   ›   volatileuint32_t tail;  /**< Consumer tail.*/


 ›   } cons __rte_cache_aligned;


›   void*ring[] __rte_cache_aligned;
 };
```

整个数据结构分为3个主要部分：生产者状态信息prod；消费者状态信息 cons；消息队列本身(循环 Ring Buffer)每个单元存储着指向报文内容的指针(64bits)。

每一部分都是Cache Line(64Bytes) 对齐的， 这样就保证CPU可以最有效的 。访问这些数据(不对齐会导致更多的缓存/内存读取操作)， 尤其是各个部分之间的数据是互相隔离的，这样不会导致互相干扰。所有的生产者线程只会竞争prod占用的cache line，所有的消费者线程只会竞争cons占用的cache line， ring buffer虽然是共享的，但是实际的访问是通过 prod 和cons两个数据结构来协调控制。在burst size 是32(一次处理32个报文)的情况下，消费者线程很少会和生产者线程竞争同一Cache Line。

2 CAS 操作

CAS 是 Compare and swap的简称， 这是一个同步原语。它的伪代码如下：

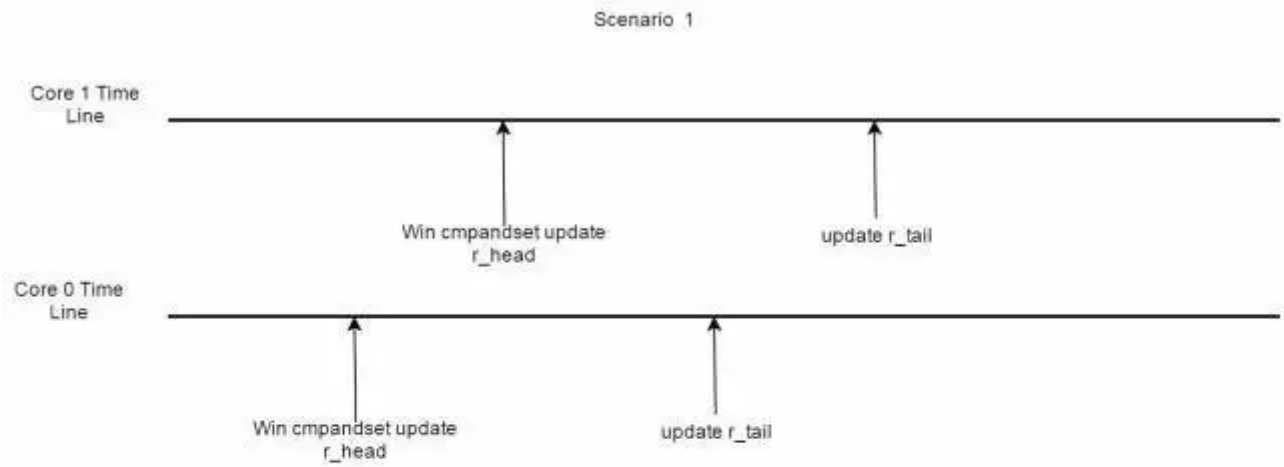
```
bool compare_and_swap (int *accum, int *dest, int newval)
{  if (  *accum == *dest ) {
       *dest = newval;       returntrue;
  }  returnfalse;
}
```

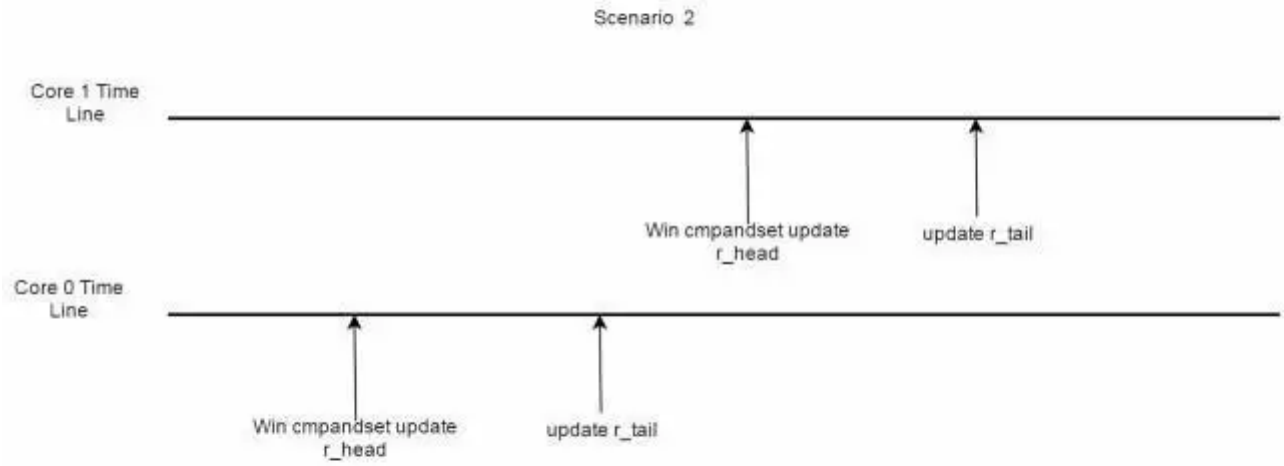
CAS操作是一个隐式总线加锁的指令，DPDK的X86实现如下：

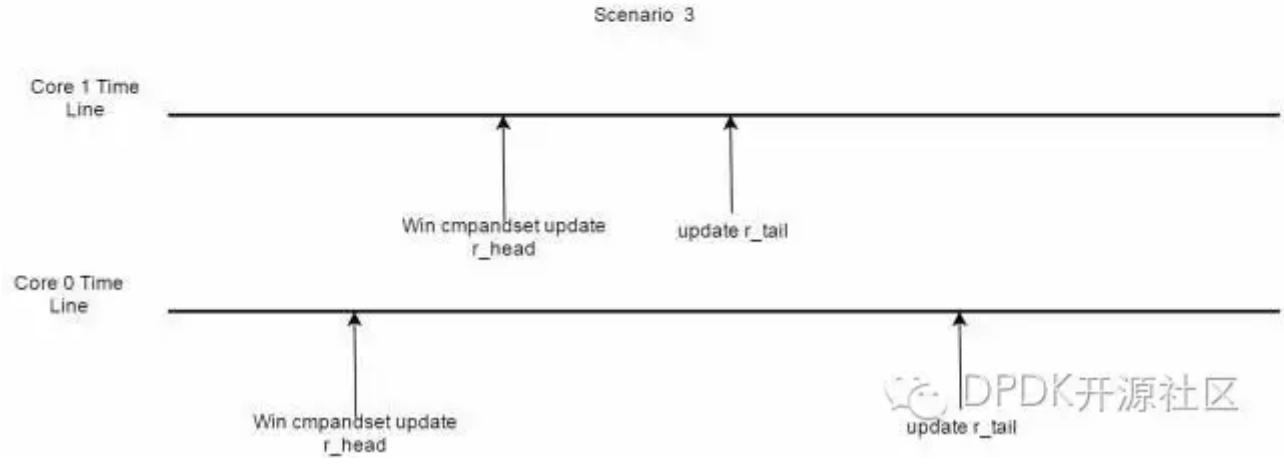
```
staticinlineint                                                rte_atomic32_cmpse
t(volatile uint32_t *dst, uint32_t exp,  uint32_t src)
{
›   uint8_t res;

›   asmvolatile(
›   ›   ›    MPLOCKED
›   ›   ›   "cmpxchgl %[src], %[dst];"                          ›   ›   ›   "sete
 %[res];"                                              ›   ›   ›   : [res] "=a" (res),
  /* output */                        ›   ›   ›     [dst] "=m" (*dst)
```

```
› › › : [src] "r" (src), /* input */ › › › "a"
(exp),
› › › "m" (*dst)
› › › : "memory"); /* no-clobber list */ › return res;

}
```

这段代码的核心是cmpxchg 这条指令，我们将在第二篇文掌中深入讨论这一点。我们现在开始考虑有两个cpu 逻辑core， 同时在竞争队列的使用权。同时向CPU发出cmpxchg指令，总线仲裁器将判断有一个core赢得总线使用权从而获得队列的使用权。任何一个core只需要完成2个动作：赢得使用权后更新状态信息 head；结束使用队列更新状态信息 tail。从时序的角度看一共有三种情景：

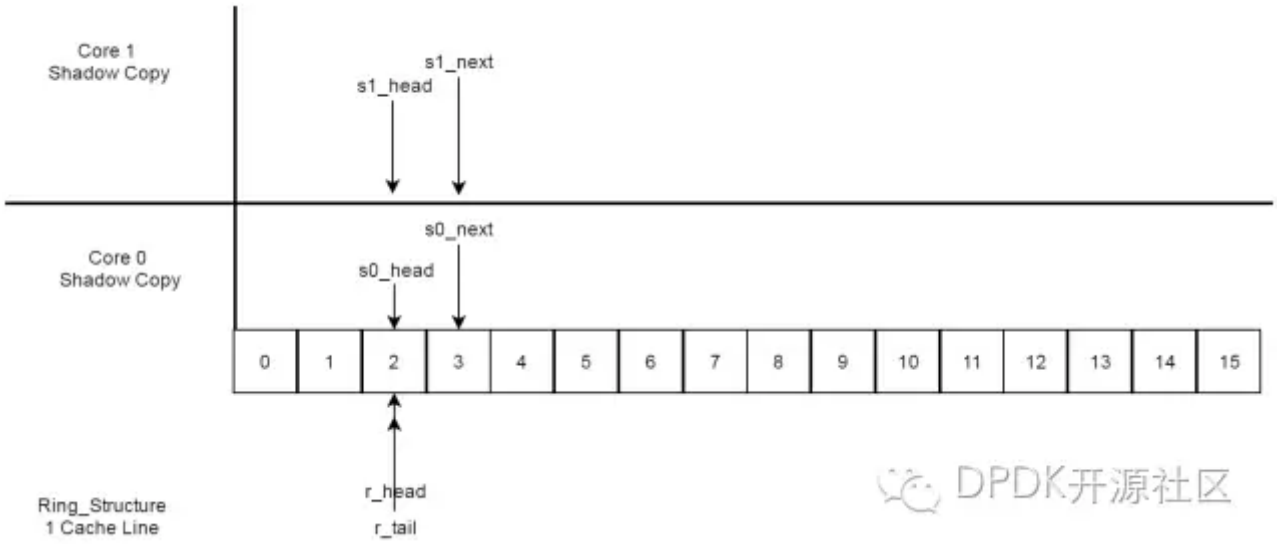




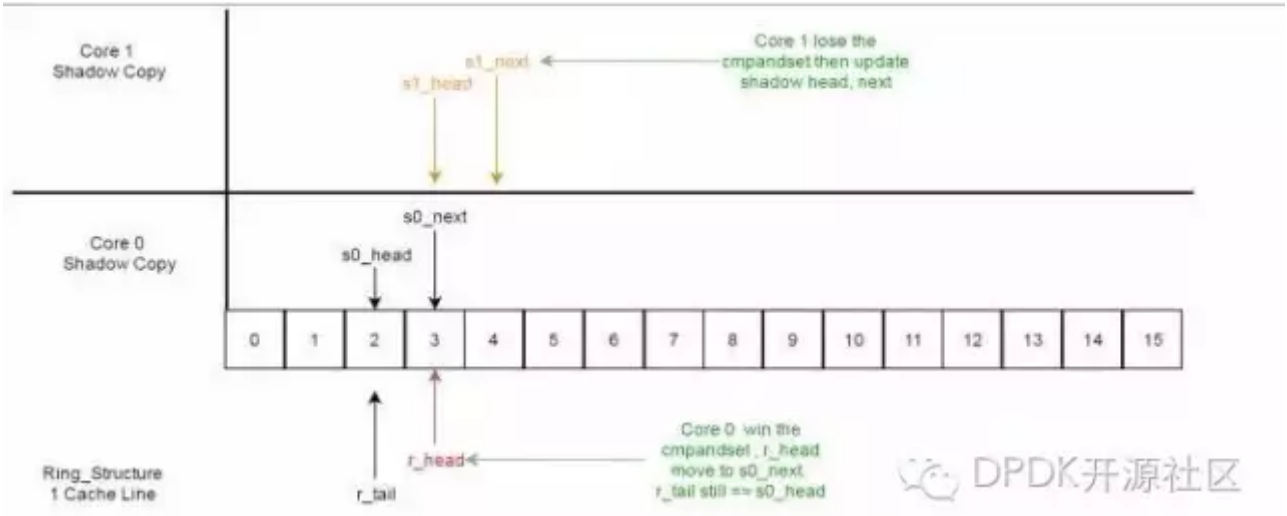


我们将开始详细分解这三种场景：在具体实现中 每个core还保留了队列状态信息的head 指针shadow copy，我命名为s[core number]_head 同时每个core都有自己的next指针指向下一个该core可用的队列偏移，我命名为 s[core number]_next。 RTE_RING 数据结构中的状态控制信息我命名为r_head, r_tail, head 指向当前可用的队列偏移， 而tail指向全局未完成的入队操作起始点偏移。 对于场景1,2 来说处理是一样的。

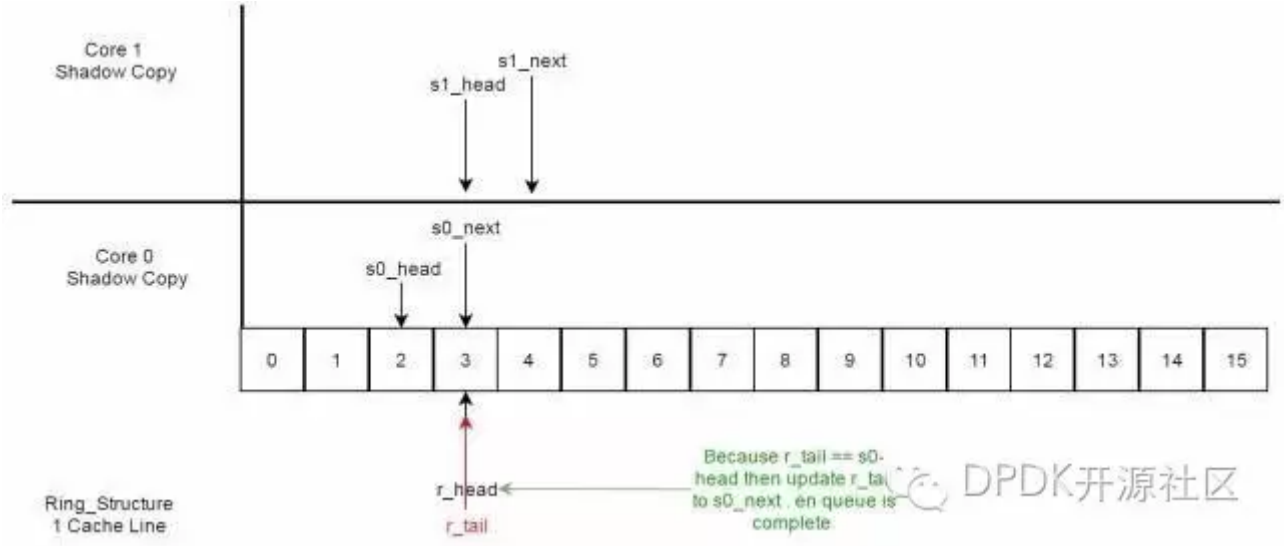
1. 起始点 core 0 core 1 的局部信息和全局信息一致。



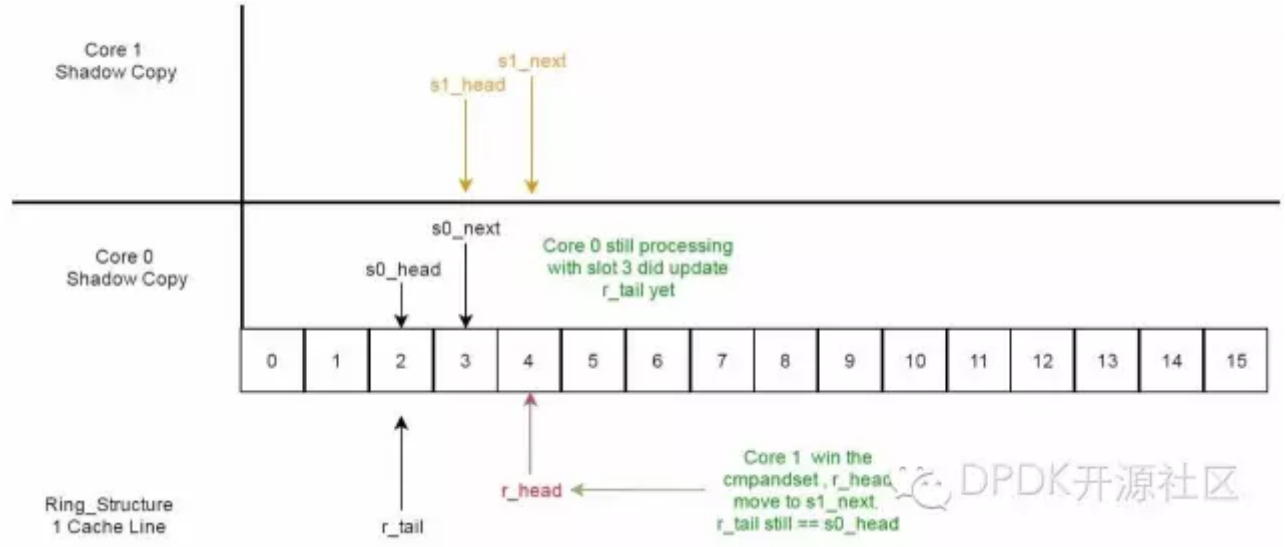
2. 开始竞争队列使用权而core 0胜出，core 0 胜出后，全局的head 更新为 s0_next同时失败的core 1将会再次与全局状态信息head同步， 之后再设置s1_next。



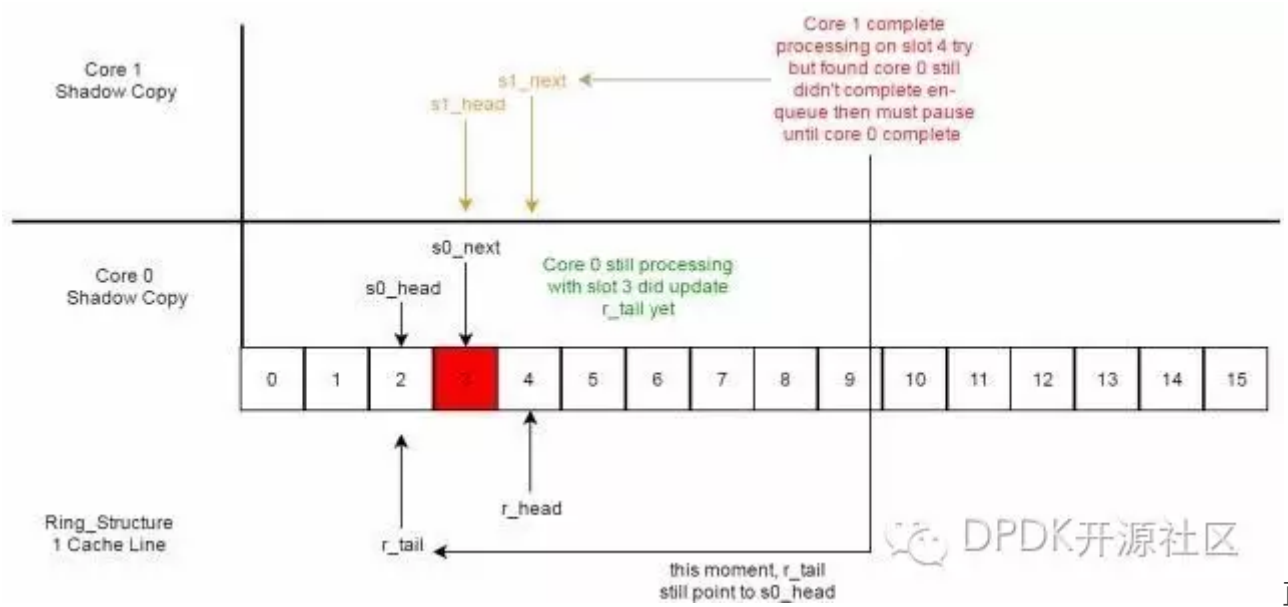
3. core 0 完成操作，入队操作完成。



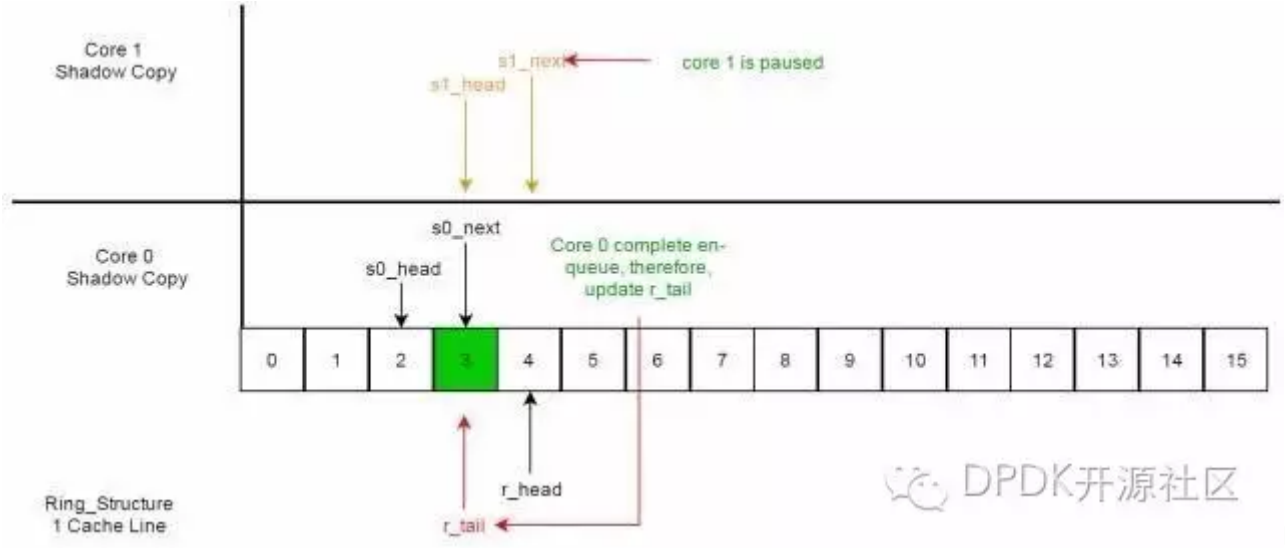
但是，对于第三种情景来说就有些不同。大家可以想象一下，当core0赢得队列使用权之后，core1也赢得了队列使用权， 但是因为某种原因 core0 没有及时的更新tail 那么core1以及完成操作而要更新tail时是怎么样一种情况呢？请看下图。core1 在续core0后也赢得了队列使用权， core0 还没有更新tail。



core1已经完成操作，试图更新tail，但是发现core0还没更新，只好等待。

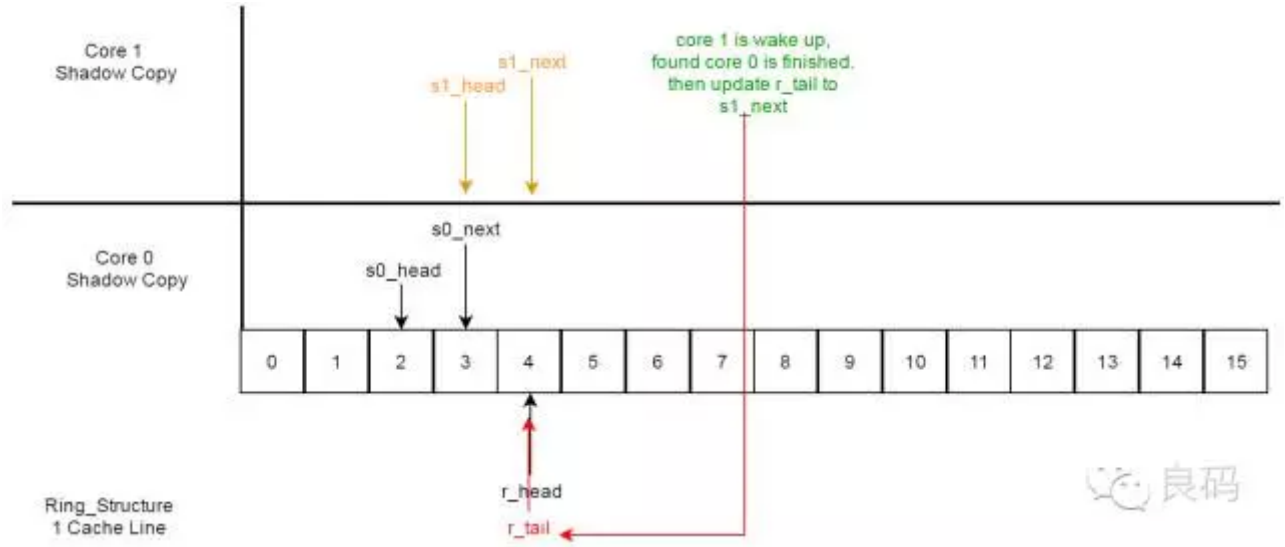


直到core0完成tail更新。



则

皆大欢喜，core1 也可以完成tail的更新。



未完待续

DPDK开源社区

一个有用的社区

投诉