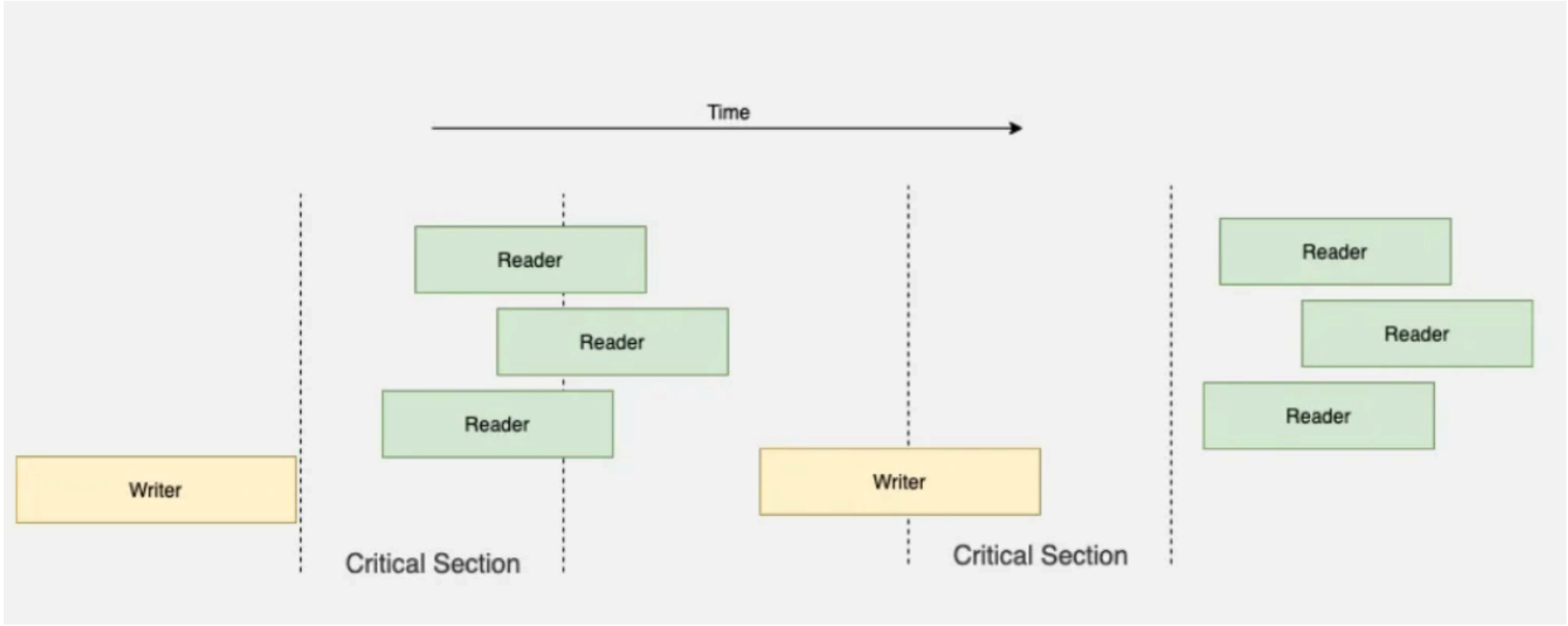
RCU 机制及 Userspace RCU 实现

钟鸣 12/07/2022

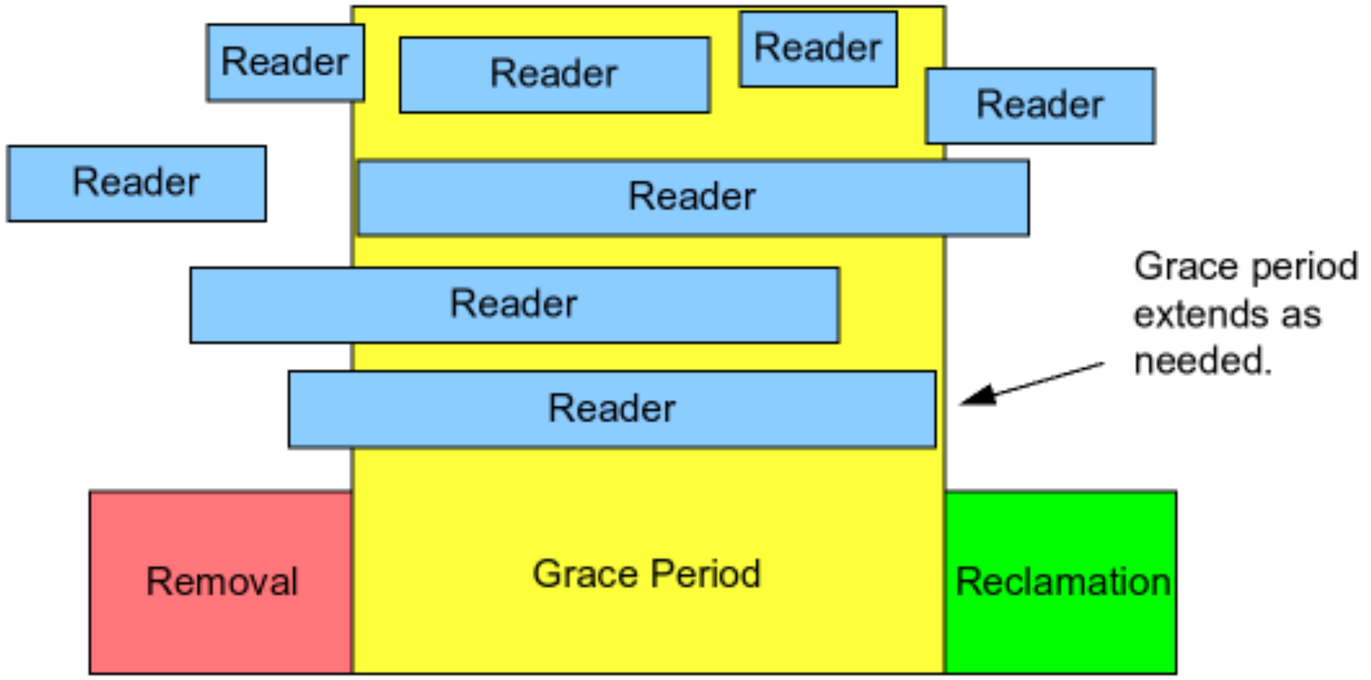
读写锁


Time
Reader
Reader
Reader
Writer
Critical Section
Writer
Critical Section
Reader
Reader
Reader

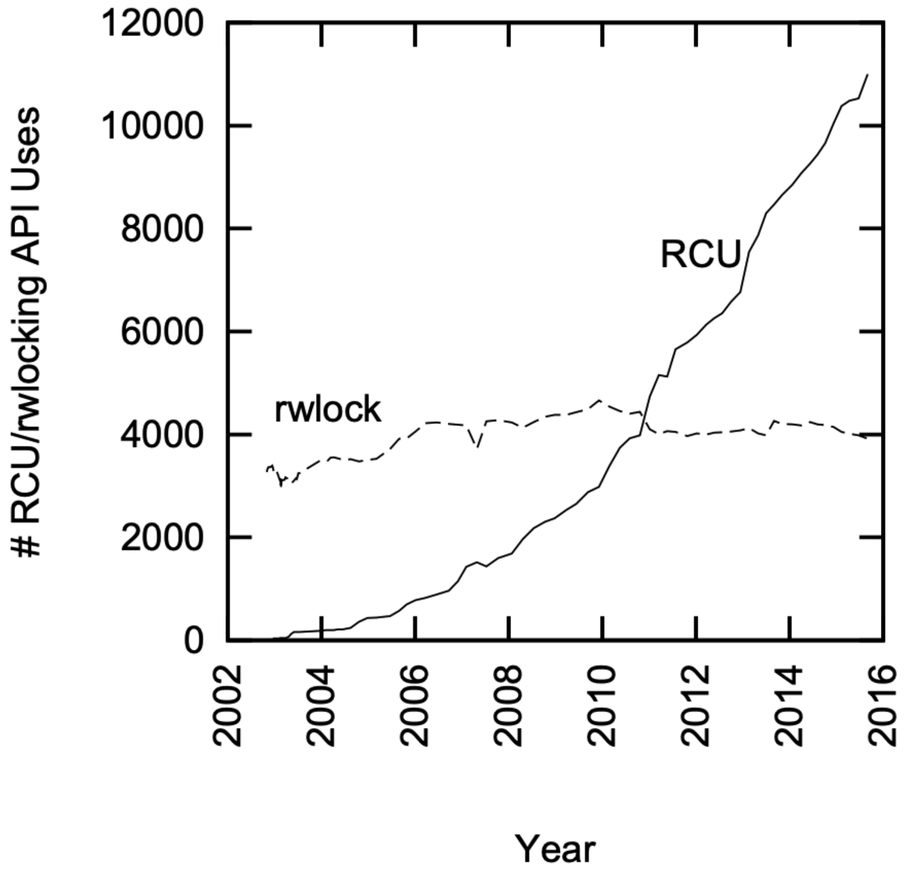
RCU

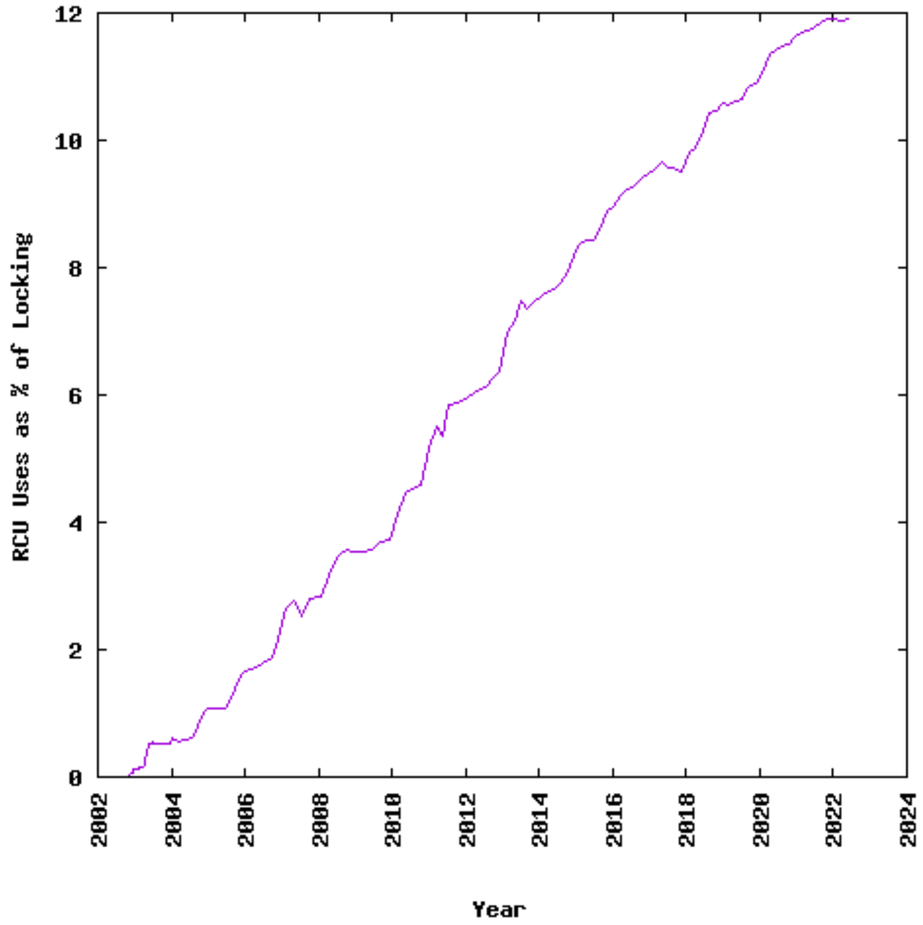
RCU 的基本思想是将共享数据的更新（update）分为修改（removal）和回收（reclamation）两个部分，removal 阶段将数据拷贝一份并进行修改，然后进入容忍期（grace period）等待所有在容忍期前的读者所在的 CPU 发生上下文切换，然后进入 reclamation 阶段完成数据的更新回收

如下图所示，每个读者在所在的 RCU read-side critical sections（蓝色矩形）内不会 block（除了 sleepable rcu 或 preemptible rcu 等），写者保证每个在 grace period 前开始执行的读者在 grace period 内完成：



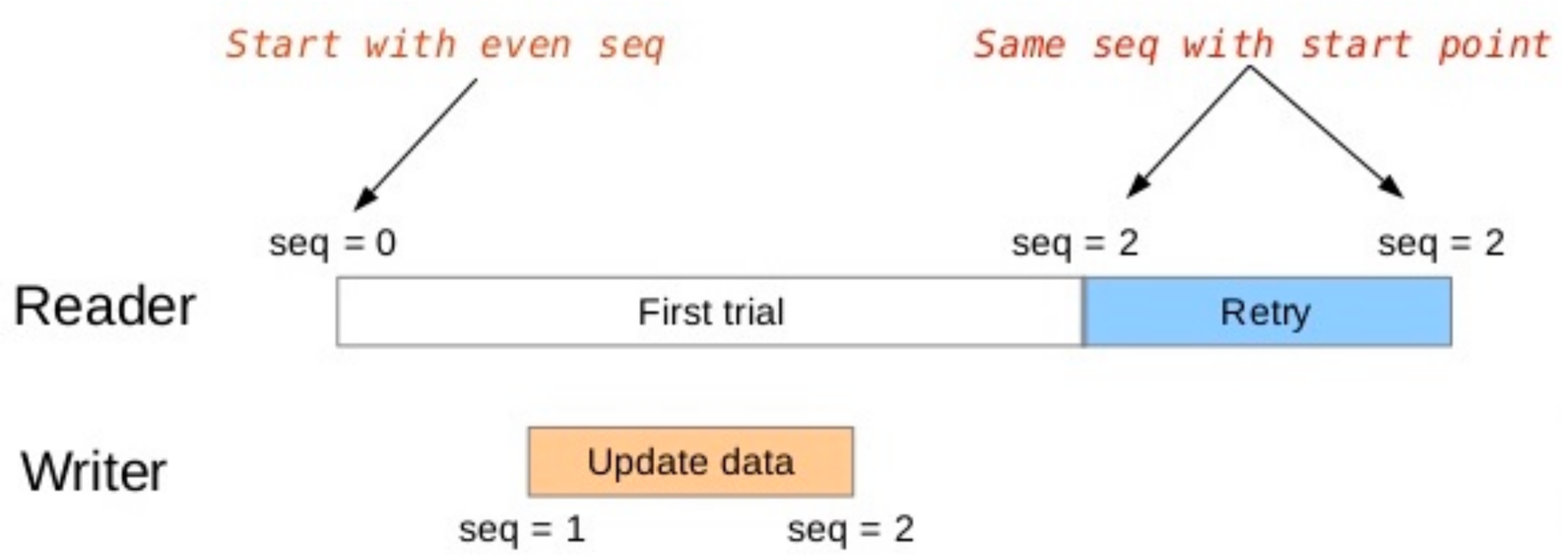
这样做的目的是将读者的开销减小写者的开销增加，从而在读多写少的场景下相比于读写锁获得更好的性能，对比 Readers-writer lock 如 Reader-Writer Spin Locks 这种读者优先的锁，RCU 不会造成 writer starvation，读者写者可以切换，读者的开销少

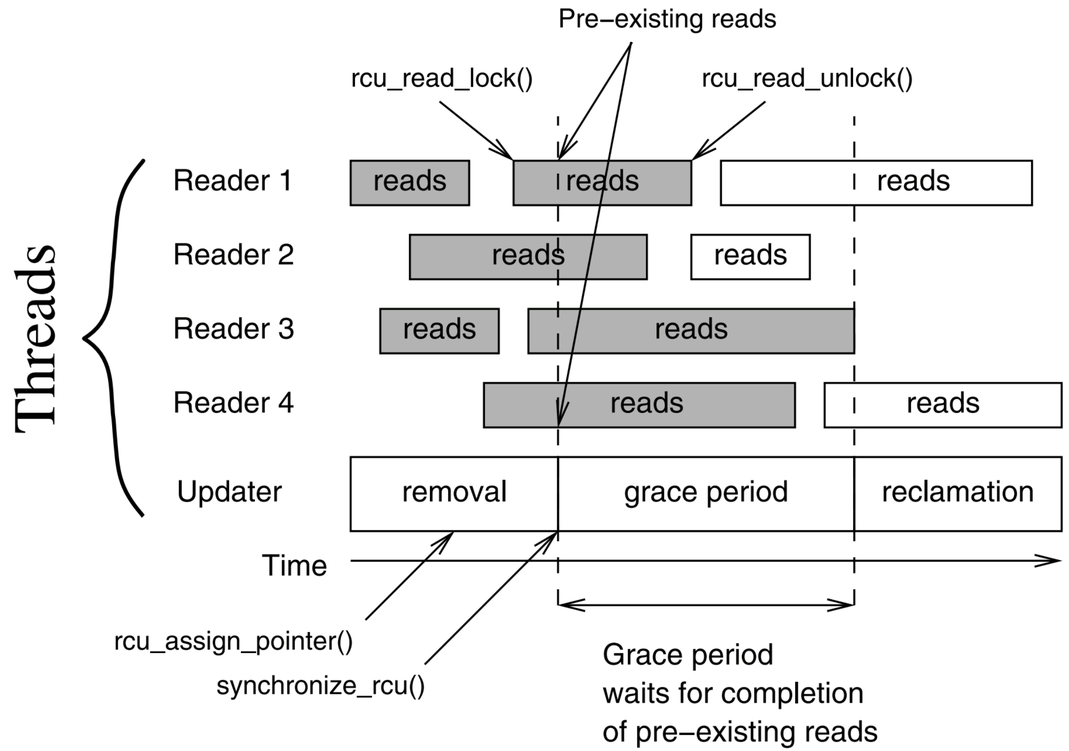




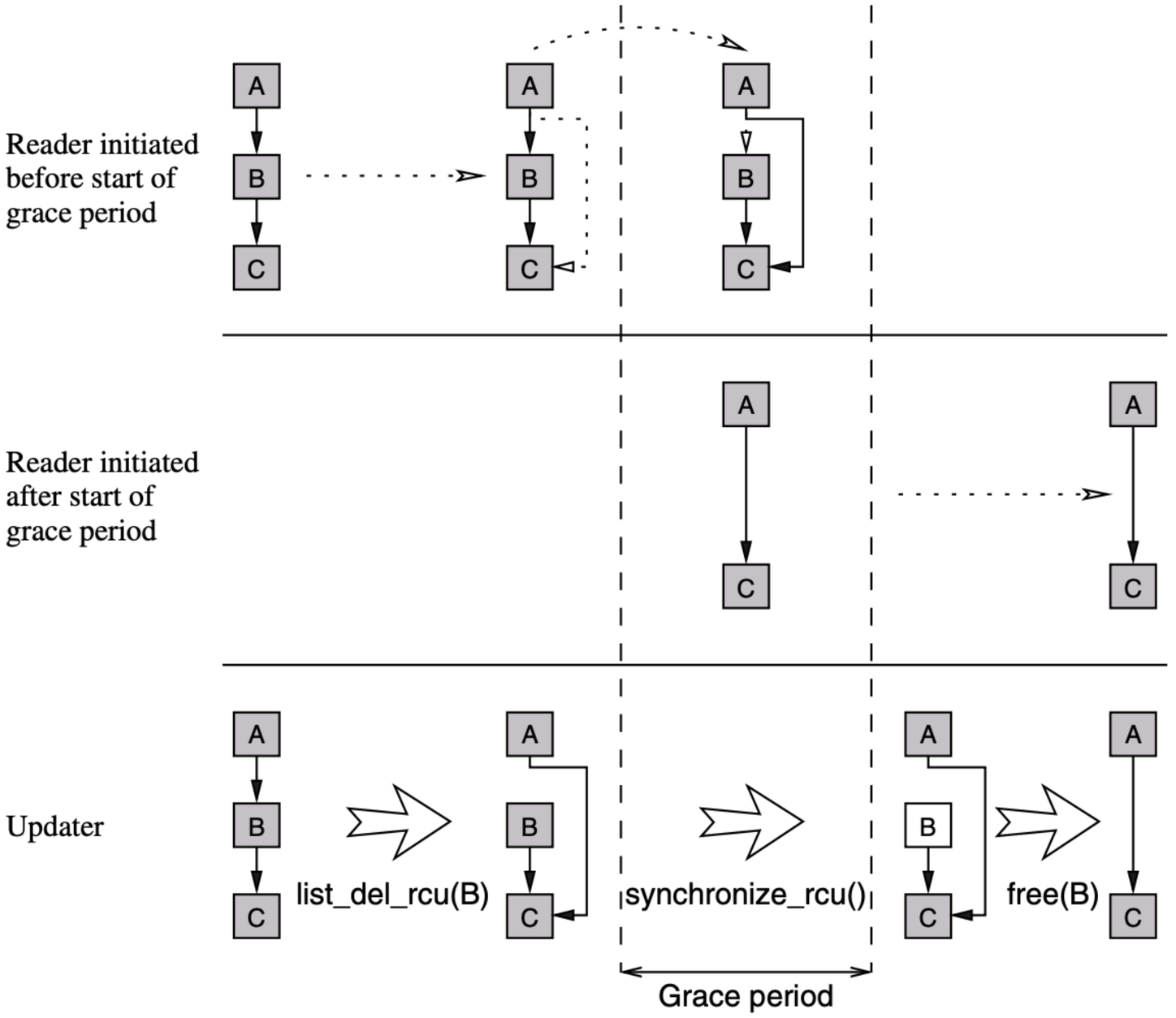
同样适合于读多写少场景的 seqlock，机制上存在不同，seqlock 在写过于频繁时读者可能一直需要循环更新值

```
1.      do {
2.          seq = read_seqbegin(&jiffies_lock);
3.          ret = jiffies_64;
4.      } while (read_seqretry(&jiffies_lock, seq));
```





primitives	**usage**
rcu_read_lock() / rcu_read_unlock()	读者进入/退出 read-side critical section
rcu_assign_pointer()	写者更新数据
rcu_dereference()	读者订阅数据
synchronize_rcu() / call_rcu()	写者同步/异步等待所有 CPU 对旧数据的使用完成


Reader initiated
before start of
grace period
A
B
C
A
B
C
A
B
C
Reader initiated
after start of
grace period
A
C
A
C
Updater
A
B
C
list_del_rcu(B)
A
B
C
synchronize_rcu()
A
B
C
free(B)
A
C
Grace period

UserSpace RCU

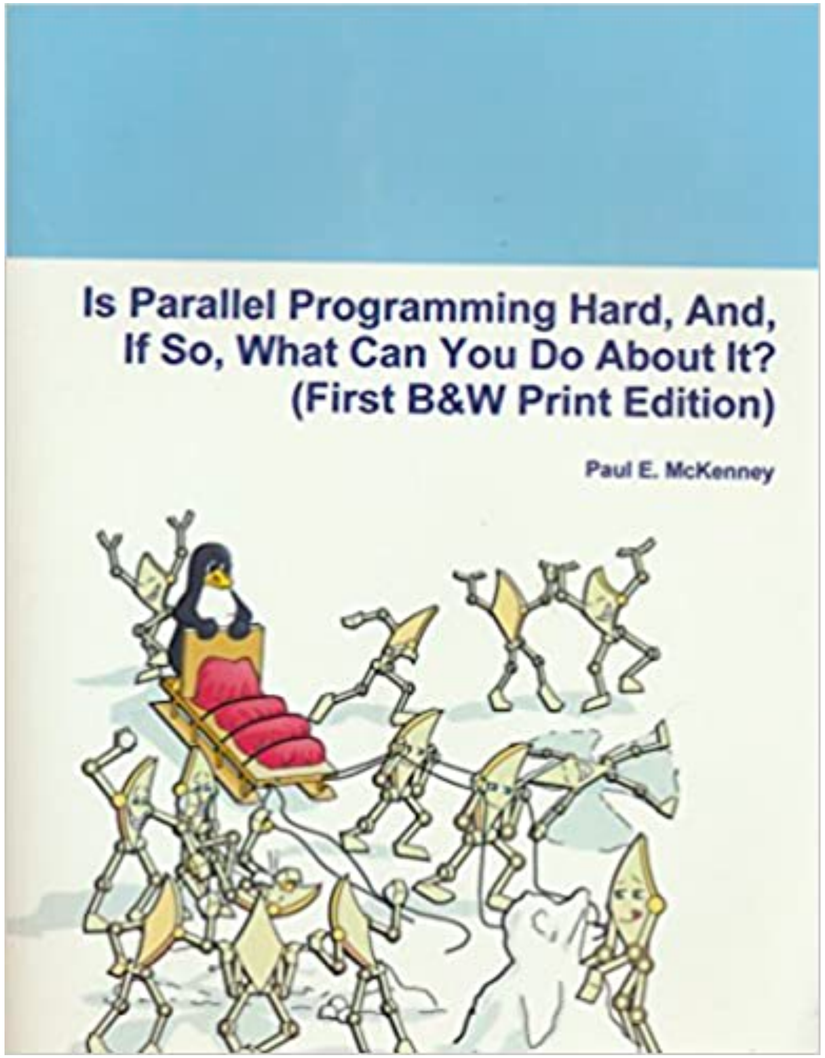
Paul McKenney

FOLLOW

Software Engineer, Facebook
Verified email at fb.com - Homepage

Concurrency RCU Linux kernel Real-Time Systems

TITLE	CITED BY	YEAR
Read-copy update: Using execution history to solve concurrency problems PE McKenney, JD Slingwine Parallel and Distributed Computing and Systems 509518	411	1998
Stochastic fairness queueing PE McKenney IEEE INFOCOM'90, 733,734,735,736,737,738,739,740-733,734,735,736,737,738,739,740	388	1990
Packet recovery in high-speed networks using coding and buffer management N Shacham, P McKenney IEEE INFOCOM'90, 124,125,126,127,128,129,130,131-124,125,126,127,128,129,130,131	361	1990
Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring JD Slingwine, PE McKenney US Patent 5,442,758	307 *	1995
Read-copy update PE McKenney, J Appavoo, A Kleen, O Krieger, R Russell, D Sarma, ... Ottawa Linux Symposium Conference Proceedings, 175	281	2001
User-level implementations of read-copy update M Desnoyers, PE McKenney, AS Stern, MR Dagenais, J Walpole IEEE Transactions on Parallel and Distributed Systems 23 (2), 375-382	220	2011



内存屏障

```
CPU 1                    CPU 2
-------------------      ---------------------
store 1 into a           store 1 into b
load b into x            load a into y
```

userspace-rcu 中使用到了 3 种内存屏障:

编译级的屏障

编译级的屏障保证编译生成的汇编代码的顺序是按照程序文本的顺序，
但实际运行的时候 CPU 仍有可能将它们乱序 (out-of-order)

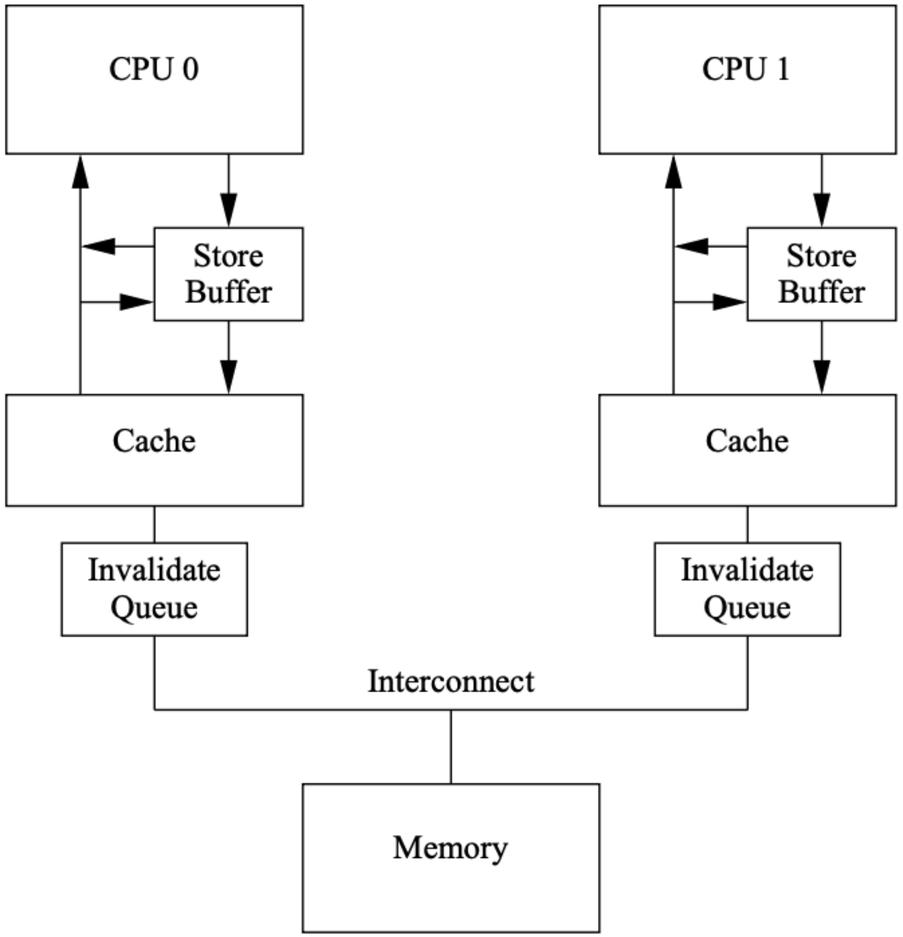
```
1.  /* only compiler barrier */
2.  #define    barrier() __asm__ __volatile__ ("":::"memory")
```

CPU 级的屏障

CPU 级别的屏障保证实际执行的顺序是按照程序文本的顺序，同时保证了多 CPU 间的缓存一致性

CPU 间的缓存一致性（cache coherence）是指被缓存在不同地方的数据保持资料一致性的机制，由于 CPU 的速度往往比主存快得多，CPU 需要将数据缓存到自己的 L1 缓存中，根据 MESI 协议将数据分为4种状态：已修改 Modified (M)，独占 Exclusive (E)，共享 Shared (S)，无效 Invalid (I)

缓存一致性协议下的 CPU 缓存如图所示：



为了减少不必要的 stalls，如图所示用到了 store buffer, store forwarding, invalidate queue

内存屏障可以控制 store buffer 和 invalidate queue，如写屏障 load fence 会标记刷新 store buffer，读屏障 store fence 会标记刷新 invalidate queue，具体例子可见 [Memory Barriers: a Hardware View for Software Hackers, Paul E. McKenney]

x86 Total Store Order 下没有 store-store, load-store 和 load-load 乱序，
故我们只需要考虑 store-load 乱序，如代码所
示，fast_path 和 slow_path 都有对共享数据 a 和 b 的先存后取(store-load)，如果缺少了两个内存屏障就有可能会存
在 read_b 和 read_a 均等于0的情况

```
1.  static volatile int a, b;

3.  static void *fast_path(void *read_b)
4.  {
5.    a = 1;
6.    asm volatile ("mfence" : : : "memory");
7.    *(int *)read_b = b;
8.    return NULL;
9.  }

11.  static void *slow_path(void *read_a)
12.  {
13.    b = 1;
14.    asm volatile ("mfence" : : : "memory");
15.    *(int *)read_a = a;
16.    return NULL;
17.  }
```

x86-TSO 下的 store-load barrier 的实现除了上述的 mfence 外，lock 前缀的命令也可以达到相同效果

```
1.  /* x86-TSO store-load barrier */
2.  #define smp_mb() __asm__ __volatile__ ("mfence":::"memory")

4.  /* x86-TSO store-load barrier for that lacks mfence instruction */
5.  #define smp_mb2() __asm__ __volatile__ ("lock; addl $0,0(%%rsp)":::"memory")
```

membarrier

membarrier 是减少 CPU 级别的内存屏障指令开销的一种屏障，适用场景为，有些用到了屏障的函数被执行的频率比另一些用到了屏障的函数高的多，如下面代码中，若 fast_path 被执行的频率远远高于 slow_path 被执行的频率，那么使用一个编译级内存屏障加 membarrier 的组合可能会比使用两个 store-load 内存屏障的开销要小

被执行频率高的函数 fast_path 使用编译级的内存屏障代替 mfence，被执行频率低的函数 slow_path 使用 membarrier 代替 mfence，membarrier(MEMBARRIER_CMD_GLOBAL) 被调用时会发送一个 inter-processor interrupt 给所有处理器，使它们执行一个内存屏障保证未被执行的存储操作执行完毕(即使 fast_path 中发生了 out-of-order)，虽然 membarrier 调用的开销高，但被调用的频率低，相当于将 fast_path 的成本转移到了 slow_path

```
1.  static volatile int a, b;

3.  static int membarrier(int cmd, unsigned int fl
4.  {
5.    return syscall(__NR_membarrier, cmd, flags,
6.  }

8.  static void *fast_path(void *read_b)
9.  {
10.   a = 1;
11.   asm volatile ("" : : : "memory");
12.   *(int *)read_b = b;
13.   return NULL;
14.  }

16.  static void *slow_path(void *read_a)
17.  {
18.   b = 1;
19.   membarrier(MEMBARRIER_CMD_GLOBAL, 0, 0);
20.   *(int *)read_a = a;
21.   return NULL;
22.  }
```

userspace-rcu 应用场景中，读者进入 RCU 临界区是经常发生的，而对 RCU 保护对象的更改可能不经常发生，因此 rcu_read_lock() 调用内存屏障产生的开销都被浪费了，membarrier 将读者（执行频率高）的成本转移到写者（执行频率低）

```
1.  static inline __attribute__((always_inline))
2.  int membarrier(int cmd, unsigned int flags, int cpu_id)
3.  {
4.    return syscall(__NR_membarrier, cmd, flags, cpu_id);
5.  }

7.  #define membarrier_master() membarrier(MEMBARRIER_CMD_PRIVATE_EXPEDITED, 0, 0)

9.  #define membarrier_slave() barrier()

11.  #define membarrier_register() membarrier(MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED, 0, 0)
```

实用函数

RMW(Read-Modifiy-Write) 可能是非原子的，使用 volatile 关键字使变量不被加载到寄存器，在 load 前和 store 后使用编译级内存屏障，但并不能防止 CPU 的重新排序 (reordering)，合并 (merging) 或重新获取 (refetching)

```
1.  #define access_once(x) (*(__volatile__ __typeof__(x) *)&(x))

3.  #define load_shared(x) ({ barrier(); access_once(x); })

5.  #define store_shared(x, v) ({ access_once(x) = (v); barrier(); })
```

在 spin-wait (busy-wait) 循环中使用 pause 指令可以提升性能

```
/* improves the performance of spin-wait loops */
#define PAUSE() __asm__ __volatile__ ("rep; nop":::"memory")

/* example: */
for (i = 0; i < RCU_WAIT_ATTEMPTS; ++i) {
  if (load_shared(node->state) & RCU_WAIT_TEARDOWN)
    break;
  PAUSE();
}
```

统计时钟周期

```
/* counts clock cycles */
static inline __attribute__((always_inline))
uint64_t get_cycles()
{
  unsigned int edx, eax;
  __asm__ __volatile__ ("rdtsc" : "=a" (eax), "=d" (edx));
  return (uint64_t)eax | ((uint64_t)edx) << 32;
}

```

用户态实现

内核中非抢占式 RCU 的 rcu_read_lock() 和 rcu_read_unlock() 只需要将抢占关闭/打开，
而用户态的实现中则需要保存和比较读者的状态

```
1.  void rcu_read_lock(void)
2.  {
3.    unsigned long tmp;
4.    barrier();
5.    tmp = tls_access_reader()->ctr;
6.    /* if reader.ctr low 32-bits is 0 */
7.    if (likely(!(tmp & RCU_GP_CTR_NEST_MASK))) {
8.      store_shared(tls_access_reader()->ctr,
9.        load_shared(gp.ctr));
10.     membarrier_slave();
11.   } else {
12.     store_shared(tls_access_reader()->ctr, tmp + 1);
13.   }
14. }
15.
```

```
1.  void rcu_read_unlock(void)
2.  {
3.    unsigned long tmp;
4.    tmp = tls_access_reader()->ctr;
5.    /* if reader.ctr low 32-bits equals to 1 */
6.    if (likely((tmp & RCU_GP_CTR_NEST_MASK) == 1)) {
7.      membarrier_slave();
8.      store_shared(tls_access_reader()->ctr, tmp - 1);
9.      membarrier_slave();
10.      wake_up_gp(&gp);
11.   } else {
12.     store_shared(tls_access_reader()->ctr, tmp - 1);
13.   }
14.   barrier();
15. }
16.
```

linux 内核中的 gp 是从写者调用 synchronize_rcu()/call_rcu() 开始，在所有 CPU 的 Quiescent State 标志位改变后结束，
userspace rcu 则需要一直检查所有已注册的读者的状态
当循环判断的次数过多时就调用 futex_wait，等待读者结束时 rcu_read_unlock() 中调用 futex_wake 来唤醒

写者更新数据：

```
#define rcu_assgin_pointer(p, v)                                \
  do {                                                          \
    __typeof__(p) ____pv = (v);                                 \
    if (!__builtin_constant_p(v) || ((v) != NULL))              \
      smp_mb();                                                 \
    store_shared(p, ____pv);                                    \
  } while (0)                                                   \

```

读者订阅数据：

其中使用到了 consume 内存序，当前线程依赖于该值的变量的读写不会被重排到 load 前，其他线程对数据依赖变量的写入可被当前线程可见

```
/* use p + 0 to get rid of ther const-ness */
#define rcu_dereference(p) __extension__ ({                     \
      __typeof__(p + 0) ____p1;                                 \
      __atomic_load(&(p), &____p1, __ATOMIC_CONSUME);           \
      (____p1);                                                 \
    })                                                          \

```

References

whatisRCU
What is RCU, Fundamentally?
Introduction to RCU
Userspace RCU
Expediting membarrier()
x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors
Towards Implementation and Use of memory order consume