# 第 0005 讲 RCU 机制及内存优化屏障

## 一、RCU 机制

1、RCU 重要的应用场景是链表，有效地提高遍历读取数据的效率，读取链有成员数据时候通常只需要 rcu_read_lock()，允许多个线程同时读取链表，并且允许一个线程同时修改链表。

RCU 的意思就是读-复制-更新，它是根据原理命名。写者修改对象的流程为：首先复制生成一个副本，然后更新此副本，最后使用新对象替换旧的对象。在写者执行复制更新的时候读者可以读数据信息啦。

写者删除对象，必须等待所有访问被删除对象读者访问结束的时候，才能够执行销毁操作实现。RCU 优势是读者没有任何同步开销；不需要获取任何的锁，不需要执行原子指令，不需要执行内存屏障。但是写者的同步开销比较大，写者需要延迟对象的释放、复制被修改的对象，写者之间必须使用锁互斥操作方法。

RCU 经常用于读者性能要求比较高的场景。RCU 只能够保护动态分配的数据结构，必须是通过指针访问此数据结

构；受 RCU 保护的临界区内不能 sleep；读写不对称，对写者的性能没有要求，但是读者性能要求比较高。

缺点写者同步开销大，写者之间需要互斥处理操作，我们在应用的时候它比较其他机制更为复杂。

A、读拷贝更新（RCU）模式添加链表项，具体内核源码分析如下：

include > linux > C rculist.h

```
105
106
107 static inline void list_add_rcu(struct list_head *new, struct list_head *head)
108 {
109     __list_add_rcu(new, head, head->next);
110 }
111
```

```
/*
* Insert a new entry between two known consecutive entries.
*
* This is only for internal list manipulation where we know
* the prev/next entries already!
*/
static inline void __list_add_rcu(struct list_head *new,
      struct list_head *prev, struct list_head *next)
{
   if (!__list_add_valid(new, prev, next))
      return;
```

```
    new->next = next;
    new->prev = prev;
    rcu_assign_pointer(list_next_rcu(prev), new);
    next->prev = new;
}
```

B、读拷贝更新（RCU）模式删除链表项，具体内核源码分析如下：

include > linux > C rculist.h > ...

```
160  static inline void list_del_rcu(struct list_head *entry)
161  {
162      __list_del_entry(entry);
163      entry->prev = LIST_POISON2;
164  }
165
```

```
/**
 * list_del - deletes entry from list.
 * @entry: the element to delete from the list.
 * Note: list_empty() on entry does not return true after this,
the entry is
 * in an undefined state.
 */
static inline void list_del(struct list_head *entry)
{
```

```
    __list_del_entry(entry);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}
```

C、读拷贝更新（RCU）模式更新链表项，具体内核源码分析如下：

include > linux > C rculist.h

```
193
194 /**
195  * list_replace_rcu - replace old entry by new one
196  * @old : the element to be replaced
197  * @new : the new element to insert
198  *
199  * The @old entry will be replaced with the @new entry atomically.
200  * Note: @old should not be empty.
201  */
202 static inline void list_replace_rcu(struct list_head *old,
203                                     struct list_head *new)
204 {
205     new->next = old->next;
206     new->prev = old->prev;
207     rcu_assign_pointer(list_next_rcu(new->prev), new);
208     new->next->prev = new;
209     old->prev = LIST_POISON2;
210 }
211
```

在整个操作过程当中，有时要防止编译器和 CPU 优化代码执行的顺序。smp_wmb()保证在它之前的两行代码执行完毕之后再执行后两行。

## 2、RCU 层次架构

RCU 根据 CPU 数量的大小按照树形结构来组成其层次结构，称为 RCU Hierarchy。具体内核源码分析如下：

include > linux > C rcu_node_tree.h

```
23
24  /*
25   * Define shape of hierarchy based on NR_CPUS, CONFIG_RCU_FANOUT, and
26   * CONFIG_RCU_FANOUT_LEAF.
27   * In theory, it should be possible to add more levels straightforwardly.
28   * In practice, this did work well going from three levels to four.
29   * Of course, your mileage may vary.
30   */
31
32  #ifdef CONFIG_RCU_FANOUT
33  #define RCU_FANOUT CONFIG_RCU_FANOUT
34  #else /* #ifdef CONFIG_RCU_FANOUT */
35  # ifdef CONFIG_64BIT
36  # define RCU_FANOUT 64
37  # else
38  # define RCU_FANOUT 32
39  # endif
40  #endif /* #else #ifdef CONFIG_RCU_FANOUT */
```

```
/*
 * Define shape of hierarchy based on NR_CPUS, CONFIG_RCU_FANOUT, and
* CONFIG_RCU_FANOUT_LEAF.
 * In theory, it should be possible to add more levels straightforwardly.
* In practice, this did work well going from three levels to four.
* Of course, your mileage may vary.
*/


#ifdef CONFIG_RCU_FANOUT
#define RCU_FANOUT CONFIG_RCU_FANOUT
#else /* #ifdef CONFIG_RCU_FANOUT */
# ifdef CONFIG_64BIT
```

```
# define RCU_FANOUT 64
# else
# define RCU_FANOUT 32
# endif
#endif /* #else #ifdef CONFIG_RCU_FANOUT */


#ifdef CONFIG_RCU_FANOUT_LEAF
#define RCU_FANOUT_LEAF CONFIG_RCU_FANOUT_LEAF
#else /* #ifdef CONFIG_RCU_FANOUT_LEAF */
#define RCU_FANOUT_LEAF 16
#endif /* #else #ifdef CONFIG_RCU_FANOUT_LEAF */


#define RCU_FANOUT_1       (RCU_FANOUT_LEAF)
#define RCU_FANOUT_2       (RCU_FANOUT_1 * RCU_FANOUT)
#define RCU_FANOUT_3       (RCU_FANOUT_2 * RCU_FANOUT)
#define RCU_FANOUT_4       (RCU_FANOUT_3 * RCU_FANOUT)
```

RCU 层次结构根据 CPU 数量决定，内核中有宏帮助构建 RCU 层次架构，其中 CONFIG_RCU_FANOUT_LEAF 表示一个子叶子的 CPU 数量，CONFIG_RCU_FANOUT 表示每个层数最

多支持多少个叶子数量。

## 二、优化内存屏障

1、编译器优化：为提高系统性能，编译器在不影响逻辑的情况下会调整指令的执行顺序。

2、CPU 执行优化：为提高流水线的性能，CPU 的乱序执行可能会让后面的寄存器冲突的汇率指令先于前面指令完成。

## 2、内存屏障

内存屏障是一种保证内存访问顺序的方法，解决内存访问乱序问题：

A、编译器编译代码时可能重新排序汇编指令，使编译出来的程序在处理器上执行速度更快，但是有的时候优化结果可能不符合软件开发工程师意图。

B、新式处理器采用超标量体系结构和乱序执行技术，能够在一个时钟周期并行执行多条指令。一句话总结为：顺序取指令，乱序执行，顺序提交执行结果。

C、多处理器系统当中，硬件工程师使用存储缓冲区、使无效队列协阴缓存和缓存一致性协议实现高效性能，引入处理器之间的内存访问乱序问题。

假设使用禁止内核抢占方法保护临界区：

preempt_desable();

临界区

preempt_enable();

临界区

preempt_desable();

preempt_enable();

preempt_desable();

preempt_enable();

临界区

为了阻止编译器错误重排指令，在禁止内核抢占和开启内核抢占的里面添加编译器优先屏障，具体如下：

include > linux > C preempt.h > ...

```
169
170 #ifdef CONFIG_PREEMPT_COUNT
171
172 #define preempt_disable() \
173 do { \
174     preempt_count_inc(); \
175     barrier(); \
176 } while (0)
177
```

include > linux > C preempt.h > ...

```
187
188 #ifdef CONFIG_PREEMPTION
189 #define preempt_enable() \
190 do { \
191     barrier(); \
192     if (unlikely(preempt_count_dec_and_test())) \
193         __preempt_schedule(); \
194 } while (0)
195
```

GCC 编译器定义的宏

include > linux > C compiler-gcc.h > ...

```
16
17  /* Optimization barrier */
18
19  /* The "volatile" is due to gcc bugs */
20  #define barrier() __asm__ __volatile__("": : :"memory")
21
```

关键字为__volatile_告诉编译器：禁止优化代码，不需要改变 barrier()前面的代码块、barrier()和后面的代码块这 3 个代码块的顺序。

## 3、处理器内存屏障

处理器内存屏障解决 CPU 之间的内存访问乱序问题和处理器访问外围设备的乱序问题。

| 内存屏障类型 | 强制性的内存屏障 | SMP的内存屏障 |
| --- | --- | --- |
| 通用内存屏障 | mb() | smp_mb() |
| 写内存屏障 | wmb() | smp_wmb() |
| 读内存屏障 | rmb() | smp_rmb() |
| 数据依赖屏障 | read_barrier_depends() | smp_read_barrier_depends() |

除数据依赖屏障之外，所有处理器内存屏障隐含编译器优化屏障。