RCU(Read Copy Update)에 대한 이해

RCU에 대한 많은 문서들이 있지만 실제로 RCU를 정확히 이해하고 있는 사람은 많지 않다. RCU는 여러 면의 장점을 가지는 반면, 제대로 알고 사용하지 못하는 경우, critical한 bug를 만들어 낼 수 있다. 그러므로 본 문서는 RCU에 관한 글 중, 가장 이해하기 쉽도록 설명된 LWN(http://lwn.net/Articles/262464/)의 기사를 의역한 문서임을 밝힌다. 원문을 보고자 할 경우 해당 URL을 참조하면 될 것이다.

RCU는 updater들과 cuncurent하게 reader들을 동작하도록 하여 시스템의 scalability를 향상시키는 lock free 메커니즘이다. Update들과 동시에 reader들도 동작할 수 있도록 하기 위한 rwlock이 있긴 하지만 rwlock은 update가 없을 때에만 reader들의 수행을 허락한다. 반면 RCU는 single update와 multiple reader들 사이의 concurrency를 지원한다는 점이 다르다. 즉 update와 reader가 동시에 수행할 수 있도록 한다는 것이다. 이 말을 보고 의아해 할 사람이 많을 것이다. 그럼 지금부터 RCU를 살펴보기로 하자.

RCU는 3가지의 기본적인 원리를 바탕으로 동작한다.

1. Publish-Subscribe mechainsm (삽입을 위해)
2. Wait For Pre-Existing RCU Readers to Complete (삭제를 위해)
3. Maintain Multiple Versions of Recently Updated Objects (리더를 위해)

지금부터 설명하는 아래의 예제들을 볼 때 주의해야 할 것은 update operation들은 mutex로 이미 atomic하게 실행되고 있다고 가정하고 있다는 것이다. 사실은 아래의 예제들은 update operation에 race condition이 없는 single update를 위해 작성된 code이다. 하지만 일반적으로는 multiple update들로 인해 race condition이 발생할 수 있음으로 update operation은 mutex에 의해 보호되어야 한다.

Publish-Subscribe Mechanism

RCU의 중요한 요소는 데이터가 동시에 변경되고 있다고 하더라도 안전하게 data를 scan할 수 있는 점이다.
그러므로 reader와 동시에 insertion 허용하기 위해서 RCU는 publish-subscribe 메커니즘을 사용한다.

```
1 struct foo {
 2 int a;
 3 int b;
 4 int c;
 5 };
 6 struct foo *gp = NULL;
 7
 8 /* . . . */
 9
 10 p = kmalloc(sizeof(*p), GFP_KERNEL);
 11 p->a = 1;
```

```
12 p->b = 2;
13 p->c = 3;
14 gp = p;
```

위의 코드는 일반적으로 문제가 없어 보인다. 하지만 Alpha와 같은 특정 CPU의 misordering이나 value-speculation compiler optimization과 같은 것들에 의해 p가 초기화 되기 이전에 14 line이 먼저 실행 될 수도 있다. 그렇게 되면 p의 필드들이 초기화 되기 전에 14라인이 실행되게 된다면 concurrent한 리더들은 초기화되지 않은 값들을 보게 된다. 그러므로 CPU와 컴파일러에게 11-14 line까지의 코드에 명시된 순서대로 실행하라고 명령해야 한다. 위의 순서를 보장하기 위해 memory barrier가 필요하다. 하지만 memory barrier를 사용하는 것은 그리 쉽지 않다. 그래서 RCU는 rcu_assign_pointer()와 같은 API로 memory barrier를 캡슐화시켜 publication semantics를 만들었다. 위의 4줄은 다음과 같이 변환할 수 있다. 여기서 rcu_assign_pointer는 새로운 structure를 publish하는 역할을 하게 된다.

```
 1 p->a = 1;
 2 p->b = 2;
 3 p->c = 3;
 4 rcu_assign_pointer(gp, p);
```

이와 같은 update만으로는 부족하다. Reader들 또한 올바른 순서대로 수행하는 것을 보장해야 한다. 다음의 예제를 보라.

```
1 p = gp;
2 if (p != NULL) {
3 do_something_with(p->a, p->b, p->c);
4 }
```

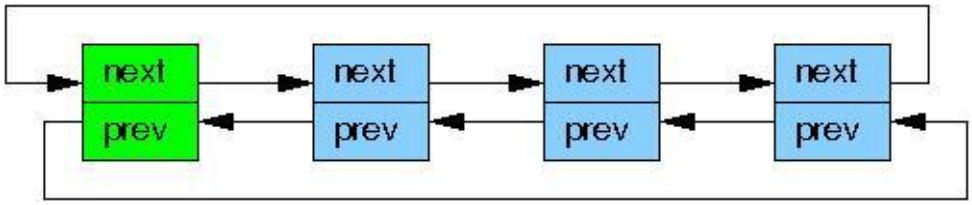
위의 코드가 아무 문제가 없어 보일지 모르지만 DEC Alpha CPU와 같은 경우에서는 p! 의 값을 수행하지 전에 p->a, p->b, p->c의 값을 fetch할 수도 있다. 또한 value-speculation compiler optimization에서도 발생가능한 문제이다. 그러므로 위와 같은 문제를 막기 위해 memory barrier와 컴파일러 지시자를 사용하는 rcu_dereference() primitive가 필요하다.

```
1 rcu_read_lock();
2 p = rcu_dereference(gp);
3 if (p != NULL) {
4 do_something_with(p->a, p->b, p->c);
5 }
6 rcu_read_unlock()
```

;

rcu_dereference() primitive는 명시된 포인터의 값을 subscribing하는 것으로 생각할 수 있다. 이것은 이어지는 dereference operation들은 관련된 publish(rcu_assing_pointer) operation이 발생하기 전에 초기화되었던 값들을 볼 수 있음을 보장하는 primitive다. rcu_read_lock 과 rcu_read_unlock 호출이 반드시 필요하다. 이 호출은 RCU read-side critical section 영역을 정의한다. 이것은 추후 설명한다. 지금은 rcu_read_lock이나 rcu_read_unlock은 절대 spin하거나 block하거나 동시에 수행하고 있는 list_add_rcu()와 같은 것들을 막지않는다는 사실만을 알면 된다. 심지어는 non-CONFIG_PREEMPT 커널에서는 어떤 역할도 하지 않는 코드로 변한다.

위에서 소개한 primitive들은 이미 잘 알려진 list에 포한되어져 더 쉽게 사용할 수 있다.




```
1 struct foo {
 2 struct list_head list;
 3 int a;
 4 int b;
 5 int c;
 6 };
 7 LIST_HEAD(head);
 8
 9 /* . . . */
 10
 11 p = kmalloc(sizeof(*p), GFP_KERNEL);
 12 p->a = 1;
 13 p->b = 2;
 14 p->c = 3;
 15 list_add_rcu(&p->list, &head);
```

사실, 15라인은 동기화 메커니즘을 통해 보호되어야 한다. 왜냐하면 multiple list_add() instance들이 동시에 수행될 수 있기 때문이다. 하지만 서두에서 언급한 바와 같이 이미 보호되고 있다고 생각하고 넘어가자.
우리가 혼동해서는 안될 것이 RCU는 update와 reader들 사이에 concurrency를 향상시키는 것이지 updater들 간에 concurrency를 보장하는 메커니즘이 아니라는 것이다. 그렇기 때문에 그러한 동기화 메커니즘도 이 list_add()로부터 동시에 수행되고 있는 RCU reader들을 막지는 않는다.

RCU-protected list에 subscribing하는 것은 더욱 직관적이다.

```
1 rcu_read_lock();
2 list_for_each_entry_rcu(p, head, list) {
3 do_something_with(p->a, p->b, p->c);
4 }
5 rcu_read_unlock();
```

이미 커널에는 다음과 같은 RCU primitive들이 이미 구현되어 있다. 현재 커널 2.6.28(정확하게 언제 mainline에 merge되었는지는 모르겠지만, radix tree에 대해서도 RCU primitive들이 구현되어 있다. 이것은 suse의 nick에 의해 오랫동안 제안되고 작업되어 온 패치이다. 결국 lockless radix tree는 lockless page cache로 발전하게 되었다. 하지만 최근 정확히는 2009-01-06일 약간의 bug가 발생하였으나 곧 패치되었다.)

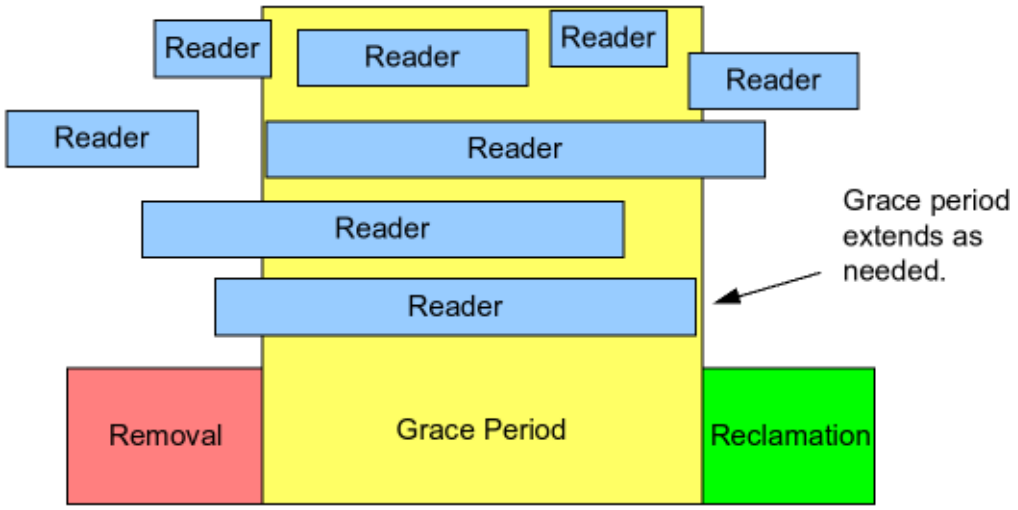
Category	Publish	Retract	Subscribe
Pointers	rcu_assign_pointer() list_add_rcu()	rcu_assign_pointer(..., NULL)	rcu_dereference()
Lists	list_add_tail_rcu() list_replace_rcu()	list_del_rcu()	list_for_each_entry_rcu()
Hlists	hlist_add_after_rcu() hlist_add_before_rcu() hlist_add_head_rcu() hlist_replace_rcu()	hlist_del_rcu()	hlist_for_each_entry_rcu()

그렇다면 여기서 질문이 나올 것이다. 언제 data element들을 교체하거나 제거하는 것이 좋을까?

Wait For Pre-Existing RCU Readers to Complete

RCU의 가장 일반적인 사용 형태는 어떤 것이 끝나기를 기다리기 위한 방법으로 많이 사용된다. 물론 그러한 목적을 달성하기 위한 방법은 많이 있다. 우리는 reference counter를 사용할 수도 있고, rw lock, events 등을 사용할 수도 있다. 하지만, RCU를 사용하는 가장 큰 목적은 performance degradation이나 scalability에 대한 제한, 복잡한 deadlock problem이나 메모리 leak hazard와 같은 문제점에 대한 걱정 없이 사용할 수 있다는 데에 있다. (하지만 그 만큼 사용하기가 간단하지 못하다는 점도 있다 – barrios의 생각)

위와 같은 형태로 RCU를 사용할 경우, 기다리는 것을 "RCU read-side critical section"이라고 말한다. RCU read-side critical section은 rcu_read_lock() primitive에서 시작하고 rcu_read_unlock() primitive로 끝난다. RCU read-side critical section들은 명시적으로 block되거나 sleep 되지 않는한 얼마든지 많은 코드를 포함할 수도 있고 심지어는 nesting될 수도 있다. 이러한 규칙을 잘 따르기만 한다면 기다려야 하는 코드에서 RCU를 사용할 수 있다.



다음의 pseudo code는 RCU가 reader들을 기다리기 위해 사용하는 기본적인 형태를 보여준다.

1. modify 해라. (예를 들어 연결 리스트에 한 element를 바꾼다.)
2. 모든 미리부터 존재하고 있었던 RCU read-side critical sections들이 완전히 끝나기를 기다린다. (예를 들면 synchronize_rcu() primitive를 사용하여). 여기에 키는 이어지는 RCU read-side critical section들은 지금 막 제거된 요소의 참조를 얻을 수 있는 방법이 없다는 것이다.)
3. clean up해라. (예를 들면 위에서 교체된 요소를 free한다.)

```
1 struct foo {
2 struct list_head list;
3 int a;
4 int b;
5 int c;
6 };
7 LIST_HEAD(head);
8
9 /* . . . */
10
11 p = search(head, key);
12 if (p == NULL) {
13 /* Take appropriate action, unlock, and return. */
14 }
15 q = kmalloc(sizeof(*p), GFP_KERNEL);
16 *q = *p;
17 q->b = 2;
18 q->c = 3;
19 list_replace_rcu(&p->list, &q->list);
20 synchronize_rcu();
21 kfree(p);
```

19,20,21 line은 위의 3과정을 수행하고 있다. 16~19 line이 우리가 이것을 왜 RCU라고 부르는지 설명하고 있다. concurrent한 reader들을 허락하며 line 16은 Copy, line 17~19는 Update를 하고 있다. synchronize_rcu() primitive가 우리가 제일 궁금해 하는 그것이다. 그 함수는 모든 RCU read-side critical sections들이 완전히 끝나기를 기다려야만 한다. 여기에는 trick이 있다. 이 trick은 rcu_read_lock과 rcu_read_unlock의 사이에서는 절대 block하거나 sleep해서는 안된다는 semantics가 있다고 이전에 언급하였다. 그러므로 CPU가 context switch를 수행하게 되면 이전에 그 CPU에서 수행되었던 모든 RCU read-side critical section period는 완전히 끝났다는 것을 보장할 수 있게 된다. 이러한 규칙을 모든 CPU에 대해 적용하면, 즉 모든 CPU가 한번씩 context switching을 하는 것을 기다린다면, 이러한 메커니즘으로 인하여 synchronize_rcu()가 return되었을 때는 모든 read-side critical section들이 안전하다는 끝났음을 의미하게 된다.

그러므로 RCU의 고전적인 synchronize_rcu()는 개념상 다음과 같이 구현될 수 있다.

```
1 for_each_online_cpu(cpu)
2 run_on(cpu);
```

여기서 run_on()은 현재 쓰레드를 명시된 CPU로 스위칭한다. 그러므로 해당 CPU에서 context switch를 하게 만든다. 이렇게 시스템의 모든 CPU에서 context switch를 하는 것은 이전에 모든 RCU read-side critical sections들이 모두 일을 끝마쳤다는 것을 의미하게 되는 것이다. 이러한 approach는 RCU read-side critical sections에서 preemption이 disable된 곳(non-CONFIG_PREEMPT 와 CONFIG_PREEMPT)에서는 잘 동작하지만, CONFIG_PREEMPT_RT realtimer kernel(-rt)에서는 동작하지 않기때문에 realtime RCU는 reference counter들 기초한 다른 approach를 사용한다. 시간이 된다면 이 문서에 realtime RCU도 추가할 예정이다.

리눅스에서 synchronize_rcu의 실제 구현은 훨씬 더 복잡하다. Interrupt, NMI, CPU hotplug 등을 처리해야 하기 때문이다. synchronize_rcu()의 개념적인 구현을 간단하게 살펴보기는 했지만 다른 질문이 남게 된다.

"RCU reader들은 concurrently updated list를 순회하고 있을 때 정확히 무엇을 보고 있는 것인가"

Maintain Multiple Versions of Recently Updated Objects

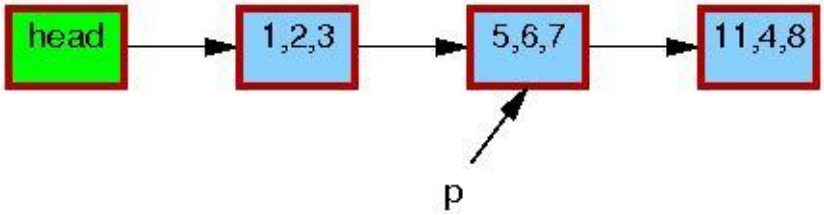
아래 두 예제는 reader가 RCU read-side critical section안에 있는 동안 reader에 의해 reference되는 element를 어떻게 손상되지(intact)않게 보이게 할 것이냐를 잘 보여준다.

```
 1 struct foo {
 2     struct list_head list;
 3     int a;
 4     int b;
 5     int c;
 6 };
 7 LIST_HEAD(head);
 8
 9 /* . . . */
10
11 p = search(head, key);
12 if (p == NULL) {
13     /* Take appropriate action, unlock, and return. */
14 }
15 q = kmalloc(sizeof(*p), GFP_KERNEL);
16 *q = *p;
17 q->b = 2;
18 q->c = 3;
19 list_replace_rcu(&p->list, &q->list);
20 synchronize_rcu();
21 kfree(p);
```

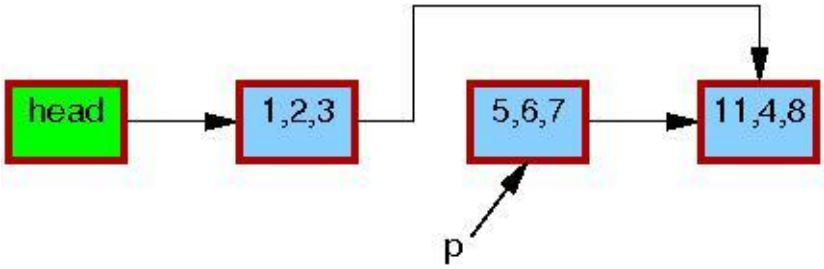
deletion 예제를 만들어보면 위의 예제에서 우리는 11-21 line 다음과 같이 바꿀 수 있다.

```
1 p = search(head, key);
2 if (p != NULL) {
3     list_del_rcu(&p->list);
4     synchronize_rcu();
5     kfree(p);
6 }
```

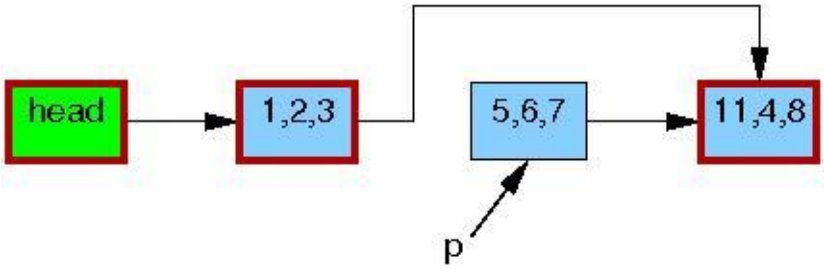
이것을 list 그림과 같이 살펴보면 다음과 같다.



빨간색 border는 현재 reader가 해당 element에 대해서 reference를 하고 있다는 것을 의미한다.
line 3에 list_del_rcu가 호출되고 나서 아래 그림과 같이 5,6,7을 포함하고 있는 element는 list로 부터 제거된다.



그러므로 reader들은 timing에 따라 제거된 element를 볼 수도 있고 보지 못할 수도 있다. 막 제거된 element에 pointer를 fetch하자마자 interrupt, ECC memory error, CONFIG_PREEMPT_RT kernel에서 preemption으로 인해 delay되었던 reader들은 list의 old version element를 보게 되는 것이다. 그러므로 우리는 지금 2가지의 다른 list를 갖게 된다. 하나는 5,6,7을 포함한 list이고 다른 하나는 그 element가 포함되지 않은 list이다. 5,6,7의 element의 border는 여전히 빨간색이다. 즉, reader는 아직도 reference하고 있다는 것이다. 여기서 주의해야 할 것이 RCU reader는 RCU read-side critical section이 종료된 이후에도 계속 5,6,7을 포함하는 element에 대한 reference를 유지하고 있으면 안 된다는 것이다. 그러므로 line 4에 synchronize_rcu가 호출되고 나면 결국 모든 pre-existing reader들은 모두 완료됐음을 보장하게 된다. 그러므로 아래와 같이 black border로 바뀌게 되고 우리는 다시 list의 single version만을 가질 수 있게 되는 것이다.

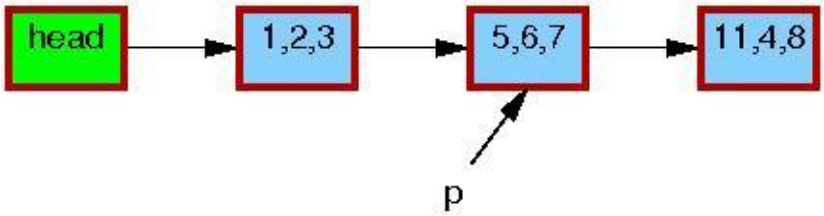


그러므로 이제는 5,6,7을 포함하는 element를 안전하게 free할 될 수 있다.

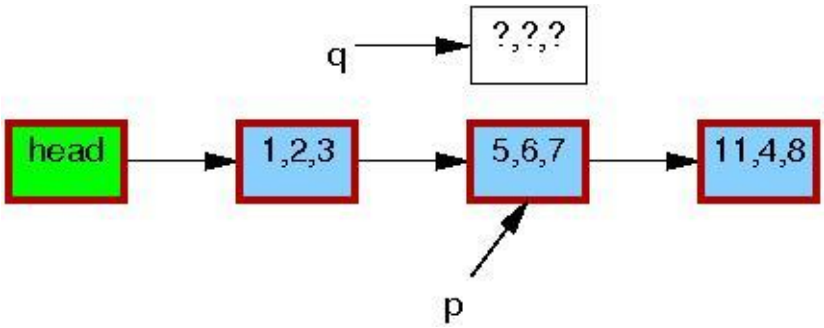
이제 우리는 deletion에 대한 설명을 마쳤다 다음은 replacement이다.
replacement 예제로써 아래와 같은 코드로 시작해보자.

```
1 q = kmalloc(sizeof(*p), GFP_KERNEL);
2 *q = *p;
3 q->b = 2;
4 q->c = 3;
5 list_replace_rcu(&p->list, &q->list);
6 synchronize_rcu();
7 kfree(p);
```

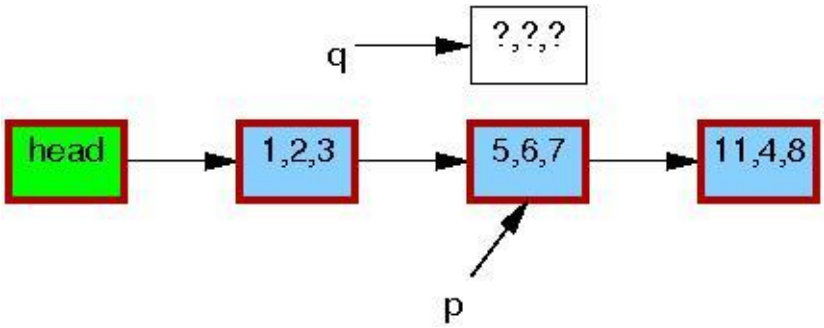
처음은 아래와 같다.



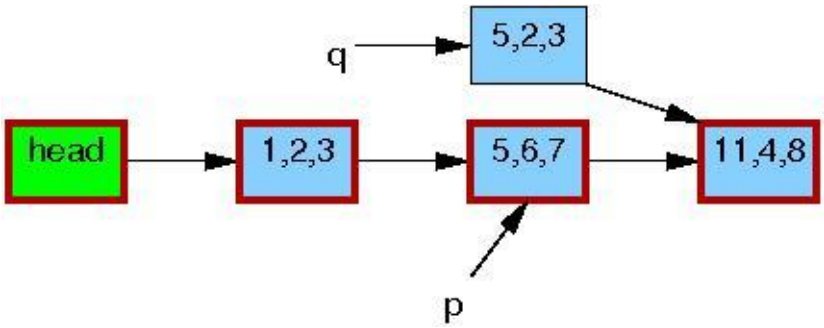
line 1에 kmalloc을 호출하게 되면 다음과 같아진다.



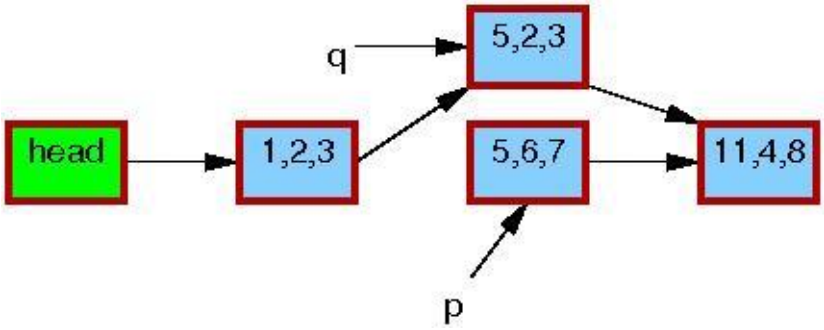
line 2에 old를 new로 copy하게 되면 다음과 같아진다.



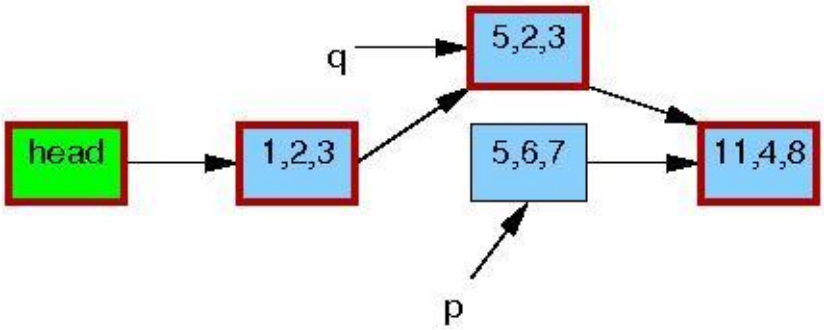
line 3,4는 q->b와 c를 2와 3으로 update한다.



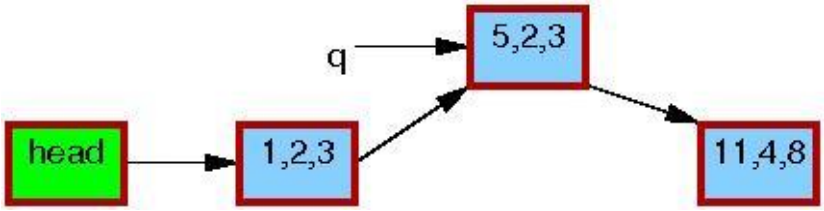
line 5는 replacement를 한다. 그래서 새로운 element는 마침내 reader에게 보이게 될 것이다.
이 시점에 우리는 list의 two version을 가지게 된다. pre-existing reader들은 5,6,7을 보게 될 것이며,
새로운 reader들은 5,2,3을 보게 될 것이다. 하지만 어떤 reader도 well-defined list를 보게 될 것이다.
well-defined list란 data structure는 망가지지 않았다는 것을 의미한다.



Line 6에 synchronize_rcu()가 return되고 나면 grace period가 지난 것이다. 그리고 list_replace_rcu가 호출되기 전 시작되었던 모든 reader들은 complete되었을 것이다. 특별히 5,6,7에 reference를 가지고 있을지도 모르는 모든 reader들은
그들의 RCU read-size critical section이 expire되었다는 것을 보장받을 수 있게 된다. 그러므로 이 시점에는 old element에 reference를 가지고 있는 어떠한 reader도 없을 것이다. 아래 그림에 black border는 그것을 나타낸다.



line 7에 kfree가 호출되고 나서는 결국 이렇게 된다.



지금까지는 가장 기본적인 RCU operation들에 대해 살펴보았다. 이를 잘 이해하고 잘 따라왔다면, 아직 풀리지 않

은 궁금증이 하나 더 있을 것이다. "old element에 대한 reference를 갖는 reader가 무슨 의미가 있을까?" 라는...

하지만 상황은 알고리즘 마다 틀리다. RCU에 대한 설명에서 가장 일반적인 예로 사용하는 것이, 라우팅 테이블이다. 라우팅 테이블은 old한 주소를 가지고 그 주소에 계속해서 패킷을 전송한다 하더라도, 일정 시간이 지나고 나서 라우팅 테이블이 개싱되면 큰 문제가 되지 않는 다는 것이다. 그러한 문제가 발생했던 원인은 컴퓨터 시스템 간의 물리적인 delay로 인한 것이다. 그러므로 알고리즘 자체에서 그러한 stale data를 처리할 수 있어야 한다. 하지만 알고리즘 자체로 풀 수 없는 문제들도 있다. 그럴 때는 stale data를 감내 할 수 있는 몇 가지 방법이 있다.