RCU(Read Copy Update) -1-

2015-12-22 (http://jake.dothome.co.kr/rcu/) 문영일 (http://jake.dothome.co.kr/author/admin/) 리눅스 커널 (http://jake.dothome.co.kr/category/linux/)

RCU History

- RCU는 읽기 동작에서 블러킹 되지 않는 read/write 동기화 메커니즘
- 2002년 커널 버전 2.5.43에서 소개됨
- 2005년 PREEMPT_RCU가 추가됨
- 2009년 user-level RCU도 소개됨

기능

- RCU는 read-side overhead를 최소화하는데 목적이 있기 때문에 동기화 로직이 읽기 동작에 더 많은 비율로 사용되는 경우에만 사용한다. 수정 동작이 10%이상인 경우 다른 동기화 로직을 선택.
- RCU는 writing 동작에서는 기존과 같은 동기화 기법을 적절히 사용해야 한다.
- weakly ordered CPU(out of order execution)를 위해 메모리 접근에 대한 order를 적절히 관리해야 하는데 rcu_dereference()를 사용하면 이를 완벽히 수행할 수 있다.
- PREEMPT_RCU
  - 이 옵션을 사용하면 read-side critical section에서 preemption 될 수 있다.
  - read-side critical section에서 sleep 기능이 필요한 경우가 아니면 SRCU 대신 RCU를 사용하는 것이 더 빠르고 사용하기 쉽다.
  - 참고로 tree RCU와 tiny RCU는 non-preemtable로 구현되어 있다.

장/단점

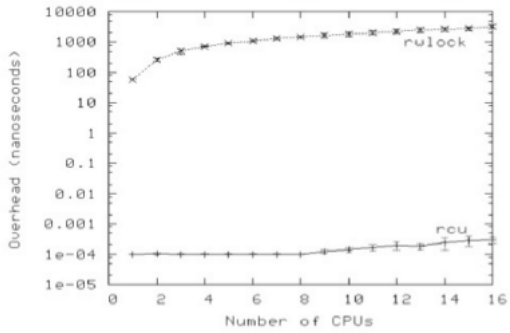
- 장점
  - 성능 향상(특히 read) – zero wait, zero overhead
  - 확장성 좋음
  - deadlock 이슈 없음
  - priority inversion 이슈 없음 (priority inversion & priority inheritance (http://jake.dothome.co.kr/priority-inheritance))
  - unbounded latency 이슈 없음
  - 메모리 leak hazard 이슈 없음
- 단점
  - 사용이 약간 복잡
  - 쓰기 동작에서는 개선 사항 없음

RCU 구현

- Classical RCU – a.k.a tiny RCU
  - CONFIG_TINY_RCU 커널 옵션
  - single 데이터 스트럭처
  - CPU가 많아지는 경우 성능 떨어짐
    - 현재 커널에서 UP만 지원함
  - non-preemtable
- Hierarchical RCU – a.k.a tree RCU
  - CONFIG_TREE_RCU 커널 옵션
  - tree 확장된 RCU 구현
  - non-preemtable
- **Preemptible tree-based hierarchical RCU**
  - CONFIG_PREEMPT_RCU 커널 옵션
  - 최근 linux kernel의 기본 RCU
  - tree 확장된 RCU 구현
  - read-side critical section에서 preemption이 지원
  - SRCU
    - CONFIG_SRCU 커널 옵션
    - read-side critical section에서 sleep 가능
- URCU
  - Userspace RCU (liburcu)

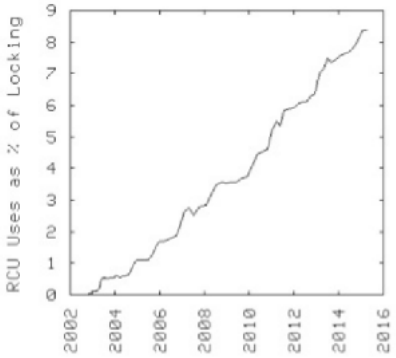
RCU 성능

reader-side에서의 성능 비교 **(rwlock vs RCU)**



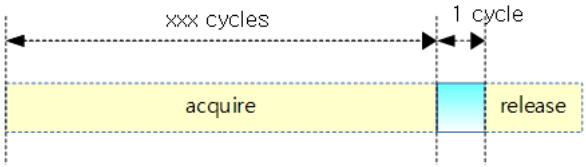
(통계: lwn.net 참고)

리눅스 커널에서의 사용율 증가

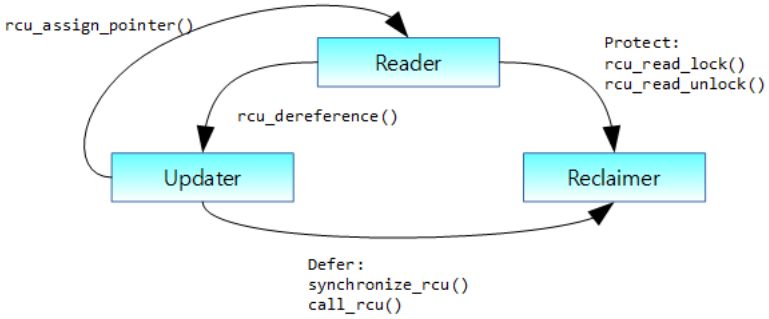


일반 **lock** 획득시의 나쁜 성능

- lock 획득하는데 소요하는 자원이 critical section을 수행하는 것에 비해 수백배 이상 over-head가 발생한다.



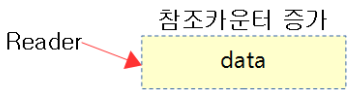
RCU 기본 요소



기본 구조체에서의 RCU 사용

Reader: RCU node를 사용할 때

- rcu_read_lock()에서 참조카운터를 증가하고 reader-side critical section의 마지막인 rcu_read_unlock()에서 참조카운터를 감소시킨다.
- rcu_dereference()는 안전하게 dereference된 RCU-protected pointer 값을 얻어온다.



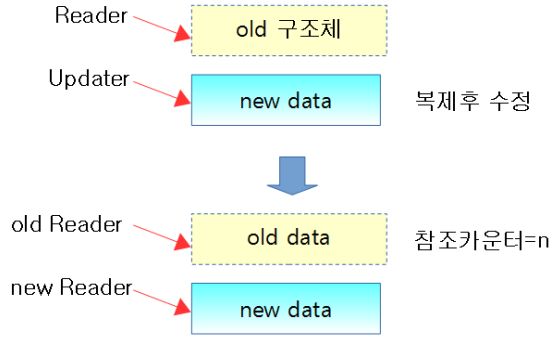
(http://jake.dothome.co.kr/wp-content/uploads/2015/12/rcu5b.png)

```
1  int foo_get_a(void)
2  {
3      int retval;
4      rcu_read_lock();
5      retval = rcu_dereference(gbl_foo)->a;
6      rcu_read_unlock();
7      return retval;
8  }
```

Updater: RCU node의 수정

- 데이터를 수정: 복사 → 수정 → 교체 메커니즘

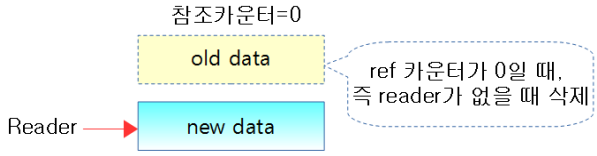
- call_rcu:
  - 모든 RCU read-side critical sections들이 끝나기를 기다린 후 두 번째 인수에 있는 함수를 호출한다.
  - non-blocking 함수로 IRQ context가 등록된 call-back 함수를 호출하는 구조로 사용이 약간 더 복잡하다.
- synchronize_rcu():
  - blocking 함수로 사용이 간단하다.
  - call_rcu()대신 사용하는 경우 synchronize_rcu() 호출 후 kfree()를 사용하여 노드를 free한다.




```
01 void foo_update_a(int new_a)
02 {
03     struct foo *new_fp;
04     struct foo *old_fp;
05     new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
06     spin_lock(&foo_mutex);
07     old_fp = rcu_dereference_protected(gbl_foo,
08          lockdep_is_held(&foo_mutex));
09     *new_fp = *old_fp;
10     new_fp->a = new_a;
11     rcu_assign_pointer(gbl_foo, new_fp);
12     spin_unlock(&foo_mutex);
13     call_rcu(&old_fp->rcu, foo_reclaim);
14 }
```

Reclaimer: RCU node 사용 완료 후 폐기

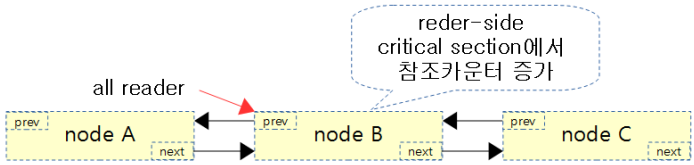
- 폐기될 노드는 grace periods 이후에 삭제한다.




```
1 void foo_reclaim(struct rcu_head *rp)
2 {
3     struct foo *fp = container_of(rp, struct foo, rcu);
4     foo_cleanup(fp->a);
5     kfree(fp);
6 }<span style="font-family: Georgia, 'Times New Roman', 'Bitstream Charter', Times, serif; line-height: 1.5;"> </span>
```

리스트에서의 RCU 사용

Reader: RCU node를 사용할 때

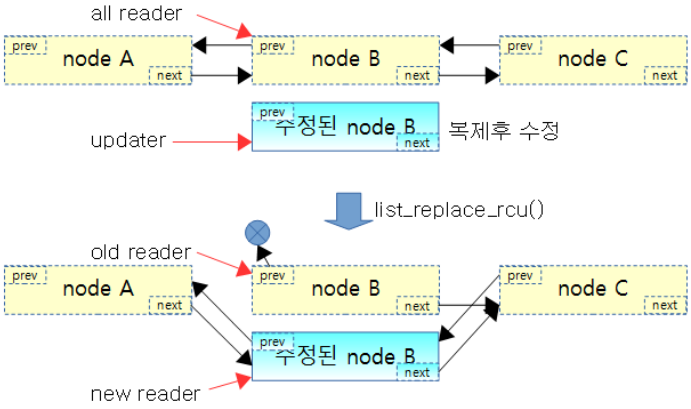


(http://jake.dothome.co.kr/wp-content/uploads/2015/12/rcu13.png)

```
1 void disp_foo(void)
2 {
3     rcu_read_lock();
4     list_for_each_entry_rcu(p, head, list) {
5         disp_foo();
6     }
7     rcu_read_unlock();
8 }
```

Updater: RCU node의 수정

- list_replace_rcu()를 수행한 순간 역방향 연결은 끊어진 반면 순방향 연결은 계속 살아있다.

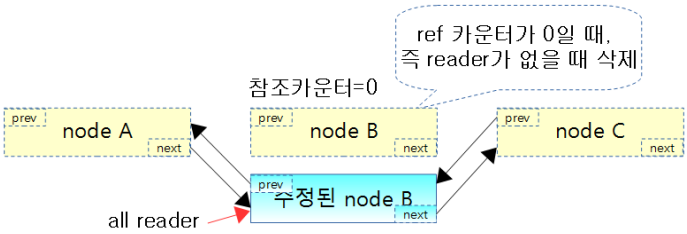



```
01 void foo_update_a(int new_a)
02 {
03     struct foo *new_fp;
04     struct foo *old_fp = search(head, key);
05     if (old_fp == NULL) {
06     /* Take appropriate action, unlock, and return. */
07     }
08     new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
09     *new_fp = *old_fp;
10     new_fp->a = new_a;
11     list_replace_rcu(&old_fp->list, &new_fp->list);
12     call_rcu(&old_fp->rcu, foo_reclaim);
13 }
```

```
1 static inline void list_replace_rcu(struct list_head *old,
2                                     struct list_head *new)
3 {
4         new->next = old->next;
5         new->prev = old->prev;
6         rcu_assign_pointer(list_next_rcu(new->prev), new);
7         new->next->prev = new;
8         old->prev = LIST_POISON2;
9 }
```

Reclaimer: RCU node 사용 완료 후 폐기

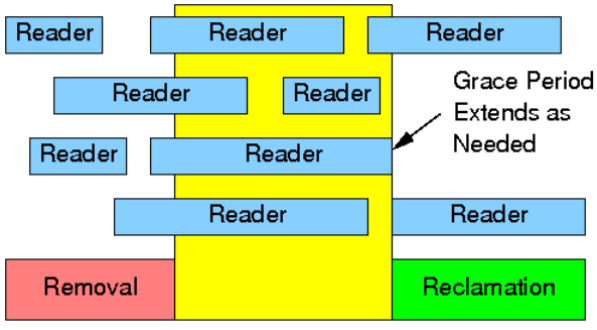
- Grace periods를 지난 후 삭제 대상 노드를 모두 clean-up한다.




```
1 void foo_reclaim(struct rcu_head *rp)
2 {
3     struct foo *fp = container_of(rp, struct foo, rcu);
4     foo_cleanup(fp->a);
5     kfree(fp);
6 }
```

Grace Period

- 노드 변경(삭제)하는 경우 기존 노드가 Reader에 의해 참조되는 경우 사용 완료까지 기다린다. 이 기간을 Grace Period라 한다. (아래 그림에서 노란색 부분)
- Grace Period 기간내에 새롭게 read-side critical section이 진행되는 Reader들은 new 노드(이미 변경된 노드)에 대한 접근을 수행하기 때문에 Grace Period에 관여하지 않는다.

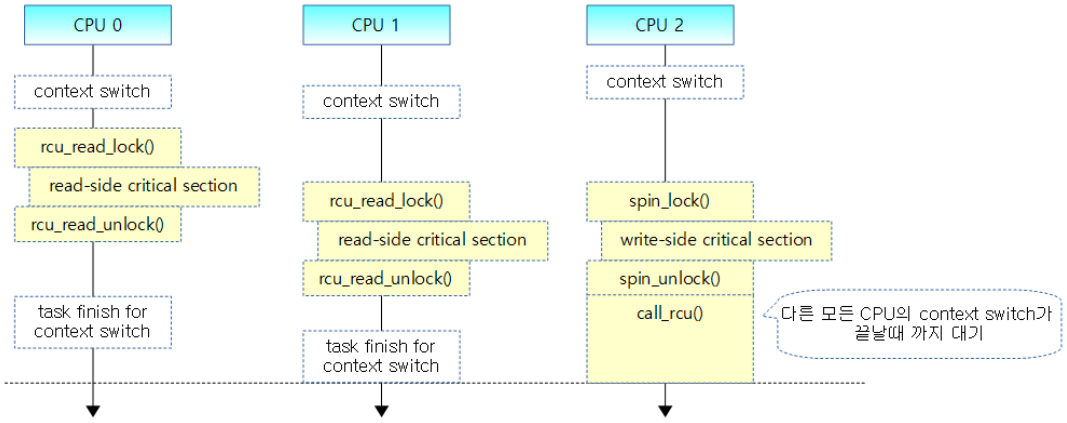


call_rcu() & synchronize_rcu()

- 모든 RCU read-side critical sections들이 끝나기를 기다린다.
- read-side critical section 사용시 규칙: 모든 read-side critical section 즉 rcu_read_lock() 과 rcu_read_unlock()사이에는 block or sleep되면 안된다.

- 모든 CPU의 context switch가 완료되면 RCU read-side critical section period는 완전히 끝난것으로 보장할 수 있게 되므로 모든 read-side critical section들이 안전하게 끝난것을 의미
- 삭제될 old 데이터들은 다른 Reader가 참조하고 있는 동안(Grace Period) 삭제 보류된 채로 있다가 old data를 참조하는 마지막 Reader의 사용이 끝나면 Reclamation을 진행하게 된다.

read-side critical section period 보장



기존 **rwlock** 구현 소스를 **rcu** 구현 소스로 변경 예제

```
1 struct el {
2     struct list_head list;
3     long key;
4     spinlock_t mutex;
5     int data;
6 };
7 spinlock_t  listmutex;
8 struct  el  head;
```

search()

```
01 int search(long key, int *result)
02 {
03     struct list_head *lp;
04     struct el *p;
05     read_lock();
06     list_for_each_entry(p, head, lp) {
07         if (p->key == key) {
08             *result = p->data;
09             read_unlock();
10             return 1;
11         }
12     }
13     read_unlock();
14     return 0;
15 }
```

```
01 int search(long key, int *result)
02 {
03     struct list_head *lp;
04     struct el *p;
05     rcu_read_lock();
06     list_for_each_entry_rcu(p, head, lp) {
07         if (p->key == key) {
08             *result = p->data;
09             rcu_read_unlock();
10             return 1;
11         }
12     }
13     rcu_read_unlock();
14     return 0;
15 }
```

delete()

```
01 int delete(long key)
02 {
03     struct el *p;
04     write_lock(&listmutex);
05     list_for_each_entry(p, head, lp) {
06         if (p->key == key) {
07             list_del(&p->list);
08             write_unlock(&listmutex);
09             kfree(p);
10             return 1;
11         }
12     }
13     write_unlock(&listmutex);
14     return 0;
15 }
```

```
01 int delete(long key)
02 {
```

```
03      struct el *p;
04      spin_lock(&listmutex);
05      list_for_each_entry(p, head, lp) {
06          if (p->key == key) {
07              list_del_rcu(&p->list);
08              spin_unlock(&listmutex);
09              synchronize_rcu();
10              kfree(p);
11              return 1;
12          }
13      }
14      spin_unlock(&listmutex);
15      return 0;
16  }
```

RCU API 목록

RCU list traversal:

- list_entry_rcu
- list_first_entry_rcu
- list_next_rcu
- list_for_each_entry_rcu
- list_for_each_entry_continue_rcu
- hlist_first_rcu
- hlist_next_rcu
- hlist_pprev_rcu
- hlist_for_each_entry_rcu
- hlist_for_each_entry_rcu_bh
- hlist_for_each_entry_continue_rcu
- hlist_for_each_entry_continue_rcu_bh
- hlist_nulls_first_rcu
- hlist_nulls_for_each_entry_rcu
- hlist_bl_first_rcu
- hlist_bl_for_each_entry_rcu

RCU pointer/list update:

- rcu_assign_pointer
- list_add_rcu
- list_add_tail_rcu
- list_del_rcu
- list_replace_rcu
- hlist_add_behind_rcu
- hlist_add_before_rcu
- hlist_add_head_rcu
- hlist_del_rcu
- hlist_del_init_rcu
- hlist_replace_rcu
- list_splice_init_rcu()
- hlist_nulls_del_init_rcu
- hlist_nulls_del_rcu
- hlist_nulls_add_head_rcu
- hlist_bl_add_head_rcu
- hlist_bl_del_init_rcu
- hlist_bl_del_rcu
- hlist_bl_set_first_rcu

RCU:

- rcu_read_lock
- synchronize_net
- rcu_barrier
- rcu_read_unlock
- synchronize_rcu
- rcu_dereference
- synchronize_rcu_expedited
- rcu_read_lock_held
- call_rcu
- rcu_dereference_check
- kfree_rcu
- rcu_dereference_protected

bh:

- rcu_read_lock_bh
- call_rcu_bh

- rcu_barrier_bh
- rcu_read_unlock_bh
- synchronize_rcu_bh
- rcu_dereference_bh
- synchronize_rcu_bh_expedited
- rcu_dereference_bh_check
- rcu_dereference_bh_protected
- rcu_read_lock_bh_held

sched:

- rcu_read_lock_sched
- synchronize_sched
- rcu_barrier_sched
- rcu_read_unlock_sched
- call_rcu_sched
- synchronize_sched_expedited
- rcu_read_lock_sched_notrace
- rcu_read_unlock_sched_notrace
- rcu_dereference_sched
- rcu_dereference_sched_check
- rcu_dereference_sched_protected
- rcu_read_lock_sched_held

SRCU:

- srcu_read_lock
- synchronize_srcu
- srcu_barrier
- srcu_read_unlock
- call_srcu
- srcu_dereference
- synchronize_srcu_expedited
- srcu_dereference_check
- srcu_read_lock_held
- Initialization/cleanup
- init_srcu_struct
- cleanup_srcu_struct

All: lockdep-checked RCU-protected pointer access

- rcu_access_pointer
- rcu_dereference_raw
- RCU_LOCKDEP_WARN
- rcu_sleep_check
- RCU_NONIDLE

소스 분석

rcu_dereference()

- rcu_dereference()는 안전하게 dereference된 RCU-protected pointer 값을 얻어온다.

include/linux/rcupdate.h

```
#define rcu_dereference(p) rcu_dereference_check(p, 0)
```

```
#define rcu_dereference_check(p, c) ₩
        __rcu_dereference_check((p), rcu_read_lock_held() || (c), __rcu)
```

include/linux/rcupdate.h

```
static inline int rcu_read_lock_held(void)
{
        return 1;
}
```

kernel/rcu/rcupdate.c

DEBUG_LOCK_ALLOC

```
int rcu_read_lock_held(void)
{
        if (!debug_lockdep_rcu_enabled())
                return 1;
        if (!rcu_is_watching())
                return 0;
        if (!rcu_lockdep_current_cpu_online())
                return 0;
        return lock_is_held(&rcu_lock_map);
}
```

* rcu_is_watching(): 레퍼런스 카운터가 0보다 크면 true

```
#define __rcu_dereference_check(p, c, space)                    ₩
({                                                              ₩
        /* Dependency order vs. p above. */                     ₩
        typeof(*p) *________p1 = (typeof(*p) *__force) lockless_dereference(p); ₩
        rcu_lockdep_assert(c, "suspicious rcu_dereference_check() usage"); ₩
        rcu_dereference_sparse(p, space);                       ₩
        ((typeof(*p) __force __kernel *)(________p1));          ₩
})
```

* rcu_lockdep_assert(): for lockdep 디버깅, c=false일때 출력
* rcu_dereference_sparse(): for sparse 체크
* __force, __kernel 등은 gcc 컴파일러 attribute 및 sparse를 참고

```
#define lockless_dereference(p)                                 ₩
({                                                              ₩
        typeof(p) _________p1 = ACCESS_ONCE(p);                 ₩
        smp_read_barrier_depends(); /* Dependency order vs. p above. */ ₩
        (_________p1);                                          ₩
})
```

compiler optimization, week ordered CPU 및 out of memory order 아키텍처에 대응하기 위한 barrier가 포함되어 있다. 단 ARM에서는 smp_read_barrier_depends()가 아무일도 수행하지 않으므로 volatile 효과(compiler optimization을 막는다)만 사용.

include/linux/compiler.h

```
#define __ACCESS_ONCE(x) ({ ₩
        __maybe_unused typeof(x) __var = (__force typeof(x)) 0; ₩
        (volatile typeof(x) *)&(x); })

#define ACCESS_ONCE(x) (*__ACCESS_ONCE(x))
```

include/linux/compiler-gcc.h

```
#define __maybe_unused                  __attribute__((unused))
```

rcu_read_lock()

- read-side critical section 시작

```
static inline void rcu_read_lock(void)
{
        __rcu_read_lock();
        __acquire(RCU);
        rcu_lock_acquire(&rcu_lock_map);
        rcu_lockdep_assert(rcu_is_watching(),
                           "rcu_read_lock() used illegally while
idle");
}
```

PREEMPT_RCU

```
void __rcu_read_lock(void)
{
        current->rcu_read_lock_nesting++;
        barrier();  /* critical section after entry code. */
}
```

```
static inline void __rcu_read_lock(void)
{
        preempt_disable();
}
```

rcu_assign_pointer()

- RCU로 보호되는 포인터(RCU-protecte pointer)에 새로운 값을 할당하기 위해 사용

```
#define rcu_assign_pointer(p, v)                        ₩
        smp_store_release(&p, RCU_INITIALIZER(v))
```

```
#define RCU_INITIALIZER(v) (typeof(*(v)) __force __rcu *)(v)
```

```
#define smp_store_release(p, v)                         ₩
do {                                                    ₩
        compiletime_assert_atomic_type(*p);             ₩
        smp_mb();                                       ₩
        ACCESS_ONCE(*p) = (v);                          ₩
} while (0)
```

참고

- RCU(Read Copy Update) -2- (http://jake.dothome.co.kr/rcu-2) | 문c
- RCU(Read Copy Update)에 대한 이해 (https://app.box.com/shared/x5r7ugx6o6) | 김민찬
- Userspace RCU (http://liburcu.org/) | liburcu.org
- RCU 관련 함수 (http://m.blog.naver.com/like8099/90030483093) | 매화나무
- RCU: The Bloatwatch Edition (http://lwn.net/Articles/323929/) | LWN.net
- Hierarchical RCU (http://lwn.net/Articles/305782/) | LWN.net
- The design of preemptible read-copy-update (http://lwn.net/Articles/253651/) | LWN.net
- Integrating and Validating dynticks and Preemptable RCU (http://lwn.net/Articles/279077/) | LWN.net
- What is RCU, Fundamentally? (http://lwn.net/Articles/262464/) | LWN.net
- RCU requirements part 2 — parallelism and software engineering (http://lwn.net/Articles/652677/) | LWN.net
- RCU requirements part 3 (http://lwn.net/Articles/653326/) | LWN.net
- RCU-walk: faster pathname lookup in Linux (http://lwn.net/Articles/649729/) | LWN.net
- RCU part 3: the RCU API (http://lwn.net/Articles/264090/) | LWN.net
- Read-Log-Update – 다운로드 pdf (http://sigops.org/sosp/sosp15/current/2015-Monterey/printable/077-matveev.pdf)
- Predicate RCU: An RCU for Scalable Concurrent Updates – 다운로드 pdf (http://www.cs.technion.ac.il/~mad/publications/ppopp2015-prcu.pdf)
- rcu_init() (http://jake.dothome.co.kr/rcu_init) | 문c

lock (http://jake.dothome.co.kr/tag/lock/)

답글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다.

댓글

이름 *

이메일 *

웹사이트

댓글 달기

BKL(Big Kernel Lock) (http://jake.dothome.co.kr/bkl/)

Atomic Operation (http://jake.dothome.co.kr/atomic/)

문c 블로그 (2015, 2016)