# Linux Kernel Programming
# **RCU**

Sang-Hoon Kim

Systems Software Research Group @ Virginia Tech

April 15, 2017

Virginia Tech

# Who are you?

**Sang-Hoon Kim**

- Ph.D. in Computer Science, Aug. 2016
  - Korea Advanced Institute of Science and Technology (KAIST), South Korea
  - Application-aware Memory Management for Mobile Devices
- Postdoctoral Associate since Nov. 2016
- Interested in system software
  - Distributed systems, memory systems, storage systems, mobile systems
- Working on the Popcorn Linux project

Virginia Tech

# Outline

1. Why RCU?
2. What is RCU?
3. How to use RCU

Virginia Tech

# Outline


1. Why RCU?
2. What is RCU?
3. How to use RCU

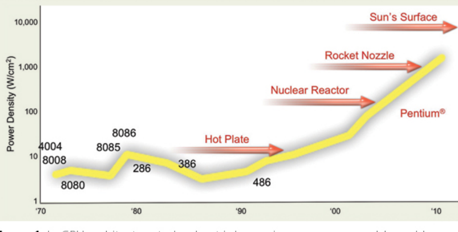

Virginia Tech

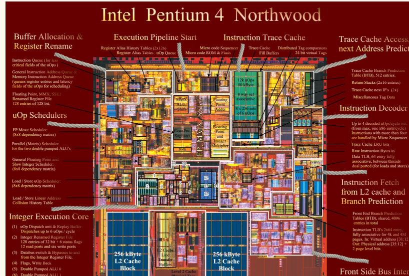
# Once upon a time ...

## Why RCU?

- Single-core era (1990s)
  - Improve the performance by putting more transistors
  - Towards multi-issue super-scalar architectures
  - Diminishing returns in performance, emit more heat

> At this rate, Intel processors will soon be producing more heat per square centimeter than the surface of the Sun!
> *"Discovering Multi-core: Extending the Benefit of Moore's Law", Geoff Koch[2].*

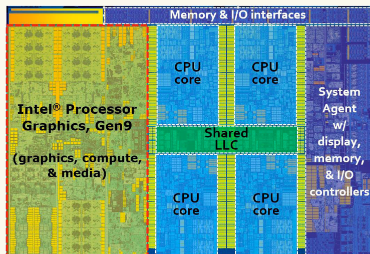


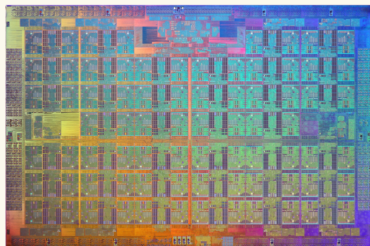
# Once upon a time ...

## Why RCU?

- Multi-core era
  - High-core count machines become common
  - Hyper-threading technology doubles the number of effective cores
  - Intel Xeon E5-2620v4 : 8/16 cores
  - Intel Xeon Phi 7210 : 64/256 cores
  - Cavium Thunder-X : 96 cores




Intel Skylake



Intel Knights Landing

# Contention matters

## Why RCU?

- Contention between contexts becomes the matter
  - Each core can serve a system call
  - Interrupt handlers can preempt the execution
  - Regular kernel code can be preemptible now
    - Can you guarantee no lock is held at any moment?
  - Hard to detect/reproduce deadlocks
- Eummmmmm... Ok, let's go with coarse-grained locks

Virginia Tech

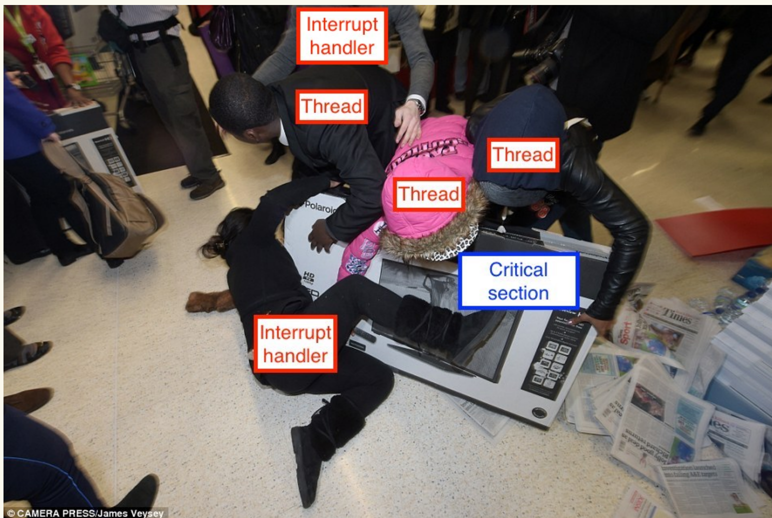
# Contention matters

## Why RCU?

Virginia Tech

# Contention matters

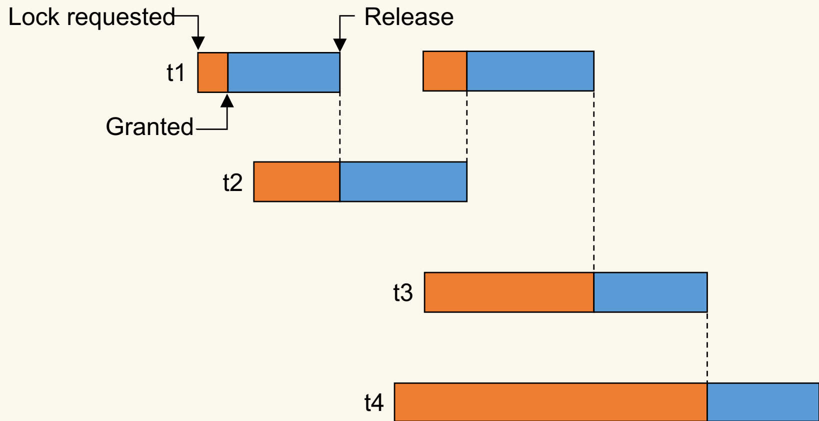
## Why RCU?



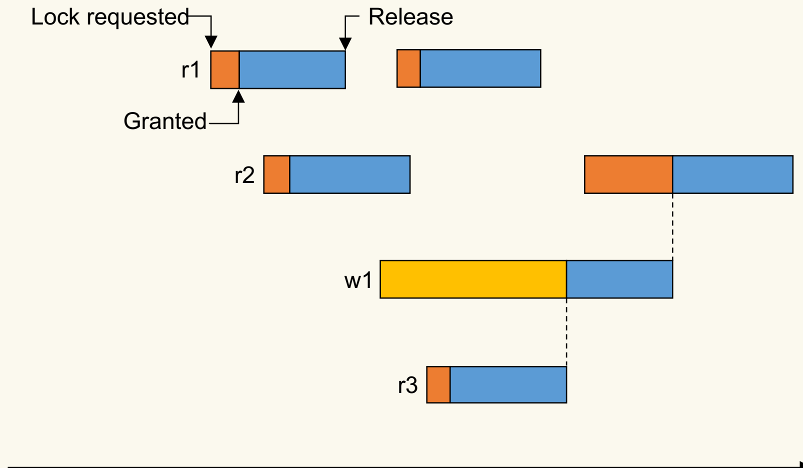
# Recap: Synchronization primitives

- Protect shared data from concurrent access
- Non-sleeping
  - Atomic operations
  - Spinlock
  - Reader-writer spinlock
  - Sequential lock
- Sleeping
  - Mutex
  - Semaphore
  - Completion
  - Wait queue
  - ...
  - `might_sleep();`

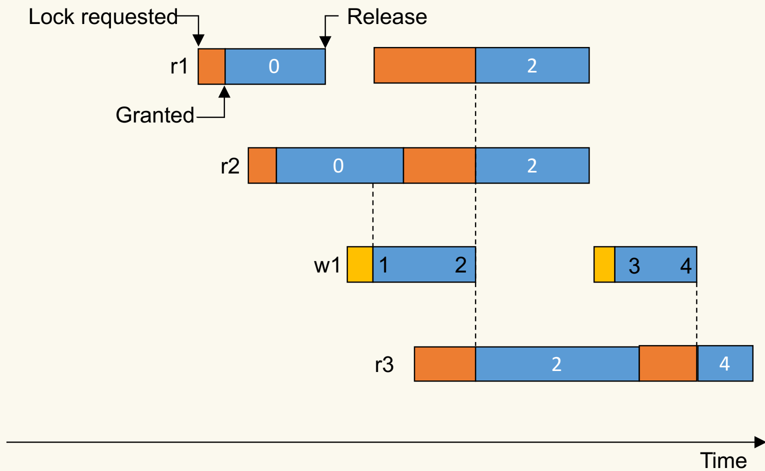
# Case: spinlock



# Case: Reader-writer lock



# Case: Sequential lock



# To sum up

- Spinlock
  - Allow one instance at a time
  - + Simple, good for short critical sections
  - - Spinning costs time and energy
- Reader-writer locks
  - Multiple readers and a writer are exclusive
  - + Applicable to many common cases
  - - Writer might have to wait for a long time
- Sequential lock
  - + Optimized for non-contending common case
  - - Biased too much to writes
  - - Not for busy critical sections
  - - Not for idempotent operations

# Outline

1 Why RCU?

2 What is RCU?

3 How to use RCU

Virginia Tech

# What is RCU?
## Introduction

- **“Read-Copy Update”**
- A synchronization mechamisn added to v2.5.43 in October 2002
- Improve the scalability of the kernel
- Low-overhead and wait-free in read side
  - Readers can be overlapped
  - Writers can be serialized without blocking reads.
- No deadlock between readers and writer
- No lock is required

# Basic concepts
## What is RCU?

- *Publish* a pointer to protect it with RCU
- *Subscribe* to dereference the value of the RCU-protected pointer
- *Replace* the entry to update it
- *Retract* if the pointer is no longer in use

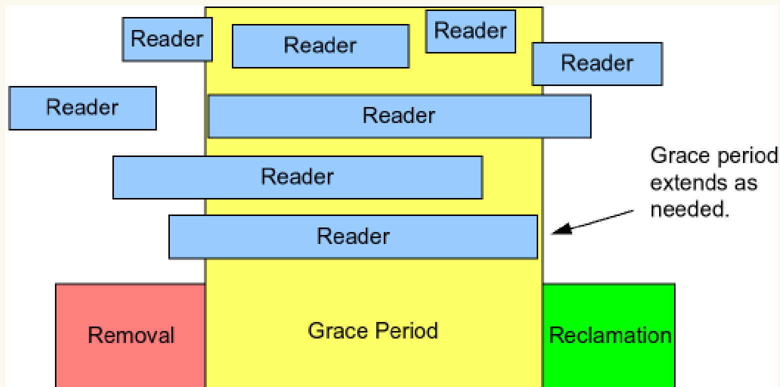
# Use case

## What is RCU?

- Read
  - Subscribe to a RCU-protected pointer
  - End the subscription
- Update
  - Save the pointer to an old structure
  - Create a new structure
  - Copy the data from the old structure into the new one
  - Modify the new copied structure $^{(yes,\ that's\ copy\ update)}$
  - Replace the old pointer to the new structure
  - Wait until no reader lefts using the old structure
    - Or, delegate the deallocation
  - Deallocate the old structure

# Grace period

## What is RCU?




[1]

# Grace period

## What is RCU?

- Actual deallocation can be done only when all readers opened at the removal moment are closed
- Incur some memory overhead
- Readers might see different values at a moment

- Q: What if a reader is blocked?

# Under the hood

## What is RCU?

- Observe the time sequenece of publication, subscription, and replacement
- Maintains multiple versions of recently updated objects
- Wait for pre-existing readers to complete
- Reclaim when no subscriber exists

# Outline

1. Why RCU?
2. What is RCU?
3. How to use RCU

# Read-side critical section

## How to use RCU

- A period during which the dereferenced entry is valid
- Dereferenced objects in the section are valid until the section is closed
  - Even though the object is retracted/replaced by other thread
  - Might see a stale value
- Should not block nor sleep within the section
- Might be preempted if `CONFIG_PREEMPT_RCU`
- Can be nested within a context
- Can be overlapped between contexts

Virginia Tech

# rcu_read_lock() / rcu_read_unlock()

## How to use RCU

- rcu_read_lock() opens a read-side critical section
- rcu_read_unlock() closes the read-side critical section
- rcu_read_lock() and rcu_read_unlock() are paired within the context

```
1  void thr_0_level_0(void)
2  {
3    rcu_read_lock();
4    level_1();
5    rcu_read_unlock();
6  }
7  void thr_1_level_0(void)
8  {
9    rcu_read_lock();
10   level_1();
11   rcu_read_unlock();
12 }
13 void level_1(void)
14 {
15   rcu_read_lock();
16   /* ... */
17   rcu_read_lock();
18   /* ... */
19   rcu_read_unlock();
20   /* ... */
21   rcu_read_unlock();
22 }
```

Virginia Tech

# List of RCU APIs

## How to use RCU

| Category | Publish | Retract | Subscribe |
|---|---|---|---|
| **Pointers** | `rcu_assign_pointer()` | `rcu_assign_pointer(..., NULL)` | `rcu_dereference()` |
| **Lists** | `list_add_rcu()`<br>`list_add_tail_rcu()`<br>`list_replace_rcu()` | `list_del_rcu()` | `list_for_each_entry_rcu()` |
| **Hlists** | `hlist_add_after_rcu()`<br>`hlist_add_before_rcu()`<br>`hlist_add_head_rcu()`<br>`hlist_replace_rcu()` | `hlist_del_rcu()` | `hlist_for_each_ehtry_rcu()` |

- APIs in `include/linux/rcupdate.h, rculist.h`

# Publish: `rcu_assign_pointer()`, `list_add_rcu()`

## How to use RCU

- `typeof(p)` **`rcu_assign_pointer`**`(p, typeof(p) v);`
  - Assign a new value `v` to an RCU-protected pointer `p`
  - Return the new value
  - Memory-barrier instructions are performed
- `void` **`list_add_rcu`**`(struct list_head *new, struct list_head *list)`
  - Insert a new list entry `new` into the RCU-protected list `list`
- `void` **`list_replace_rcu`**`(struct list_head *old, struct list_head *new)`
  - Replace `old` with `new`
- No need to `rcu_read_lock()/rcu_read_unlock()`
- May need to serialize concurrent updates

Virginia Tech

# Publish: rcu_assign_pointer(), list_add_rcu()

## How to use RCU

```
1  struct foo {
2    struct list_head list;
3    int a;
4    int b;
5    int c;
6  };
7
8  struct foo *gp = NULL;
9  LIST_HEAD(gl);
10
11 /* ..... */
12
13 struct foo *p = kzalloc(sizeof(*p), GFP_KERNEL);
14
15 INIT_LIST_HEAD(&p->list);
16 p->a = 1;
17 p->b = 2;
18 p->c = 3;
19
20 spin_lock(&gp_mutex);
21 rcu_assign_pointer(gp, p);
22
23 list_add_rcu(&p->list, &gl);
24 spin_unlock(&gp_mutex);
```

# Subscribe: rcu_dereference()

## How to use RCU

- rcu_dereference(p)
- Fetch an RCU-protected pointer p
  - Does not actually dereference the pointer, but, protect the pointer for later dereferencing
- Returned value is valid only with the enclosing read-side critical section.
- Use a local variable to dereference multiple fields
  - Look ugly
  - rcu_dereference() does not guarantee the same pointer will be returned if an update happened while in the critical section

```
1  rcu_read_lock();
2  p = rcu_dereference(gp)->a;
3  rcu_read_unlock();
4
5  x = p;   /* BUG */
6
7  rcu_read_lock();
8  y = p;   /* BUG */
9
10 p = rcu_dereference(gp)->a;
11 /* gp updated */
12 q = rcu_dereference(gp)->b;
13 /* p and q might not be from
14    the same object  */
15
16 x = rcu_dereference(gp);
17 p = x->a;
18 q = x->b;
19 rcu_read_unlock();
```

# Subscribe: rcu_dereference_protected()

## How to use RCU

- `rcu_dereference_protected(p, c)`
- Fetch a RCU-protected pointer when updates are prevented
- Skip performing memory barrier operations
  - `c`: the condition under which the dereferencing will take place
- Useful for lock-protected copy-update

```
1 spin_lock(&gp_mutex);
2 old = rcu_dereference_protected(
3     gp, lockdep_is_held(&gp_mutex));
4 spin_unlock(&gp_mutex);
```

Virginia Tech

# Subscribe: list_for_each_entry_rcu()

## How to use RCU

- `list_for_each_entry_rcu(pos, head, member)`
- Iterate each entry in `head` as `pos`
- Check other APIs from include/linux/rculist.h

```
1 list_for_each_entry_rcu(p, entry_list, list) {
2   x = p->a;
3   y = p->b;
4 }
```

# Reclaim: synchronize_rcu()

## How to use RCU

- `synchronize_rcu()`
- Blocked until all currently ongoing read-side critical sections are closed
- Safe to reclaim the old data upon the return

```
1  /* ... */
2  spin_lock(&gp_mutex);
3  old = rcu_dereference_protected(gp,
         lockdep_is_held(&gp_mutex));
4  *new = *old;
5  new->a = 0xdead;
6  new->b = 0xbeef;
7  rcu_assign_pointer(gp, new);
8  spin_unlock(&gp_mutex);
9
10 synchronize_rcu();
11
12 kfree(old);
```

# Reclaim: call_rcu()

## How to use RCU

- `void call_rcu(struct rcu_head *head, (void *callback)(struct rcu_head *))`
  - Invokes a callback function after a grace period has elapsed
  - Require to attach `struct rcu_head` in the data structure
  - Should not be blocked
    - Might be called from either softirq or process context
- `kfree_rcu(p, rcu_header)`

```
1  struct foo {
2    struct list_head list;
3    struct rcu_head rcu;
4    int a;
5    int b;
6    int c;
7  };
8
9  void foo_reclaim(struct rcu_head *rp)
10 {
11   struct foo *fp = container_of(rp,
         struct foo, rcu);
12   foo_cleanup(fp->a);
13   kfree(fp);
14 }
```

```
1  void foo_update_1(void)
2  {
3    old = rcu_dereference(gp);
4    /* ... */
5    rcu_assign_pointer(gp, new);
6
7    call_rcu(old->rcu, foo_reclaim);
8  }
9
10 void foo_update_2(void)
11 {
12   old = rcu_dereference(gp);
13   /* ... */
14   rcu_assign_pointer(gp, new);
15
16   kfree_rcu(old, rcu);
```

# Takeaway

- RCU: Read-Copy Update
- Multiple readers + updaters
- Low-overhead and wait-free in read side
- Publisher/subscriber model
- Updated objects are reclaimed after the grace period

Virginia Tech

# Bibliography I

[1] What is RCU, fundamentally?
http://lwn.net/Articles/262464.
Accessed: 2017-03-30.

[2] Koch, G.
Discovering multi-core: Extending the benefit of moore's law.
http://cache-www.intel.com/cd/00/00/22/09/220997_220997.pdf.
Accessed: 2006-05-22.

Virginia Tech