CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 15

Synchronization III

March 25, 2025

Huaicheng Li

https://people.cs.vt.edu/huaicheng/lkp-sp25/

Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.

Previously: Kernel Synchronization

- Atomic operations
- Spinlocks, reader-writer spinlock (RWLock)
- Semaphore, mutex
- Sequential lock (seqlock)
- completion variable

Spinlock vs. Mutex

Requirement	Recommended lock
Low overhead locking	Spin lock is preferred
Short lock hold time	Spin lock is preferred
Long lock hold time	Mutex is preferred
Need to lock from interrupt context	Spin lock is required
Need to sleep while holding lock	Mutex is required

Preemption Disabling

- When a spin lock is held preemption is disabled
- Some situations need to disable preemption without involving spin locks
- Example: manipulating per-processor data:

```
task A manipulates per-processor variable foo, which is not protected by a lock
task A is preempted
task B is scheduled
task B manipulates variable foo
task B completes
task A is rescheduled
task A continues manipulating variable foo
```

Preemption Disabling

Function	Description
`preempt_disable()`	Disables kernel preemption by incrementing the preemption counter
`preempt_enable()`	Decrements the preemption counter and checks and services any pending reschedules if the count is now zero
`preempt_enable_no_resched()`	Enables kernel preemption but does not check for any pending reschedules
`preempt_count()`	Returns the preemption count

For per-CPU data:

```
int cpu;

/* disable kernel preemption and set “cpu” to the current processor */
cpu = get_cpu();

/* manipulate per-processor data ... */

/* reenable kernel preemption, “cpu” can change and so is no longer valid */
put_cpu();
```

Ordering and Barriers

- Memory reads (load) and write (store) operations can be reordered for performance reasons
  - by the compiler at compile time: compiler optimization
  - by the CPU at run time:
    - » TSO (total store ordering)
    - » PSO (partial store ordering)

Ordering and Barriers

```
/* Following code can be reordered
 * by a compiler (optimization) or a processor (out-of-order execution)
 *
 * Your code             Compiled code
 * ============          ============= */
   a = 4;                b = 5;
   b = 5;                a = 4;

/* Following code will never be reorderd because
 * there is a dependency between a and b.
 *
 * Your code             Compiled code
 * ============          ============= */
   a = 1;                a = 1;
   b = a;                b = a;
```

Memory Barriers

- Instruct the compiler or the processor not to reorder instructions around a given point
- barrier() (i.e., compiler barrier)
  - Prevents the compiler from reordering stores or loads across the barrier

```
/* Compiler does not reorder store operations of a and b
 * However, a processor may reorder the store operations for performance */
a = 4;
barrier();
b = 5;
```

Memory Barrier Instructions

- rmb(): prevents loads from being reordered across the barrier
- wmb(): prevents stores from being reordered across the barrier
- mb(): prevents loads and stores from being reordered across the barrier
- read_barrier_depends(): prevent data-dependent loads to be reordered across the barrier
  - On some architectures, it is much faster than rmb() b/c it is not needed and is, thus, a noop

Memory Barrier Example

- Initial values: a = 1, b = 2
- mb() ensures that a is written before b
- rmb() ensures that b is read before a

Thread 1	Thread 2
`a = 3;`	—
`mb();`	—
`b = 4;`	`c = b;`
—	`rmb();`
—	`d = a;`

Another Memory Barrier Example

- Initial value: a = 1, b = 2, and p = &b
- mb() ensures that a is written before p
- read_barrier_depends() is sufficient b/c the load of *pp depends on the load of p

Thread 1	Thread 2
`a = 3;`	—
`mb();`	—
`p = &a;`	`pp = p;`
—	`read_barrier_depends();`
—	`b = *pp;`

Memory Barrier API

- On SMP kernel they are defined as the usual memory barriers
- On Uniprocessor kernel, they are defined only as a compiler barrier

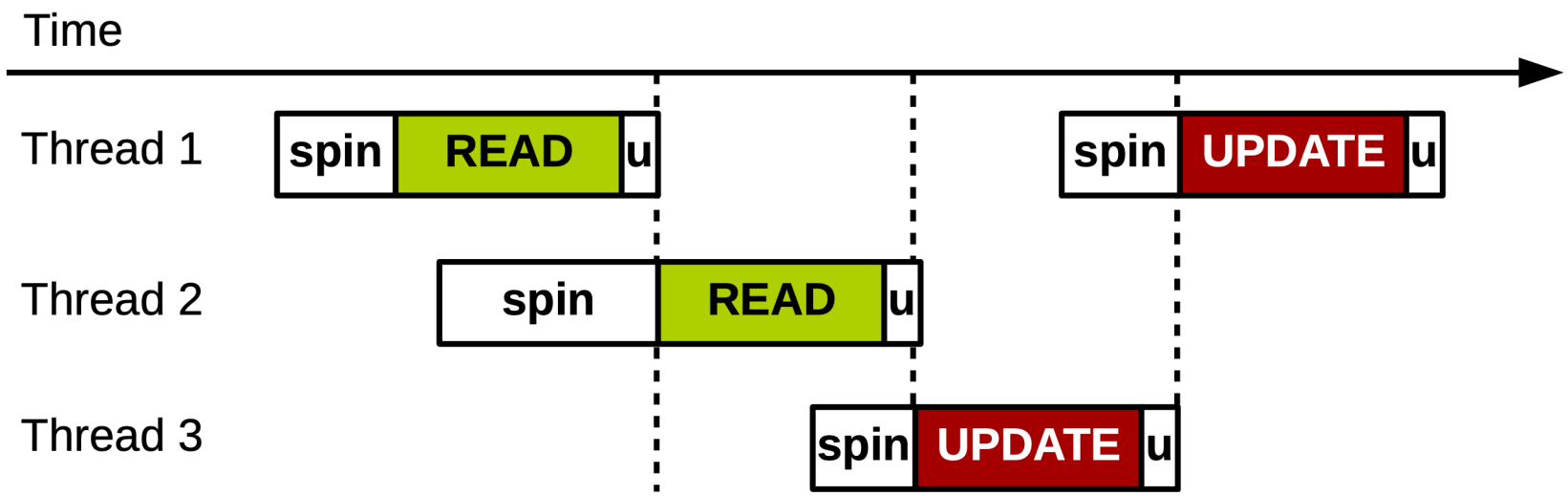
Barrier	Description
`smp_rmb()`	Provides an `rmb()` on SMP, and on UP provides a `barrier()`
`smp_read_barrier_depends()`	Provides a `read_barrier_depends()` on SMP, and provides a `barrier()` on UP
`smp_wmb()`	Provides a `wmb()` on SMP, and provides a `barrier()` on UP
`smp_mb()`	Provides an `mb()` on SMP, and provides a `barrier()` on UP
`barrier()`	Prevents the compiler from optimizing stores or loads across the barrier

Synchronization Primitives

- Protect shared data from concurrent access
- Non-sleeping (non-blocking) synchronization primitives
  - atomic operations, spinlock, read-write lock (rwlock), sequential lock (seqlock)
- Sleeping (blocking) synchronization primitives
  - semaphore, mutex, completion variable

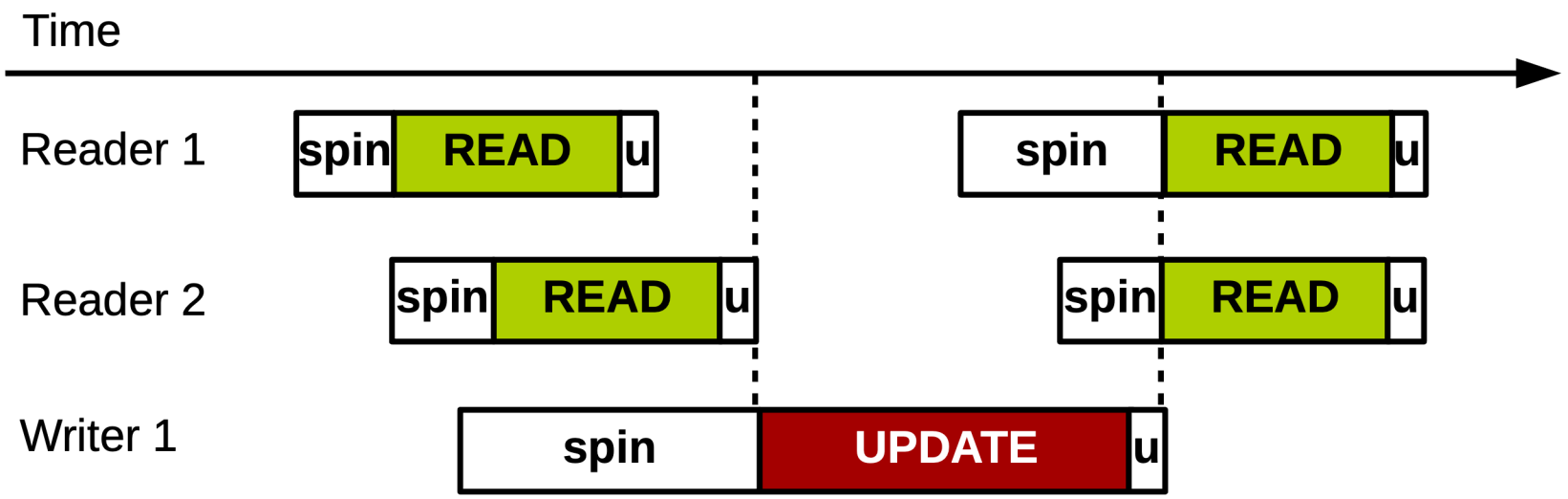
Recap: Spinlock

- Implemented by mutual exclusion



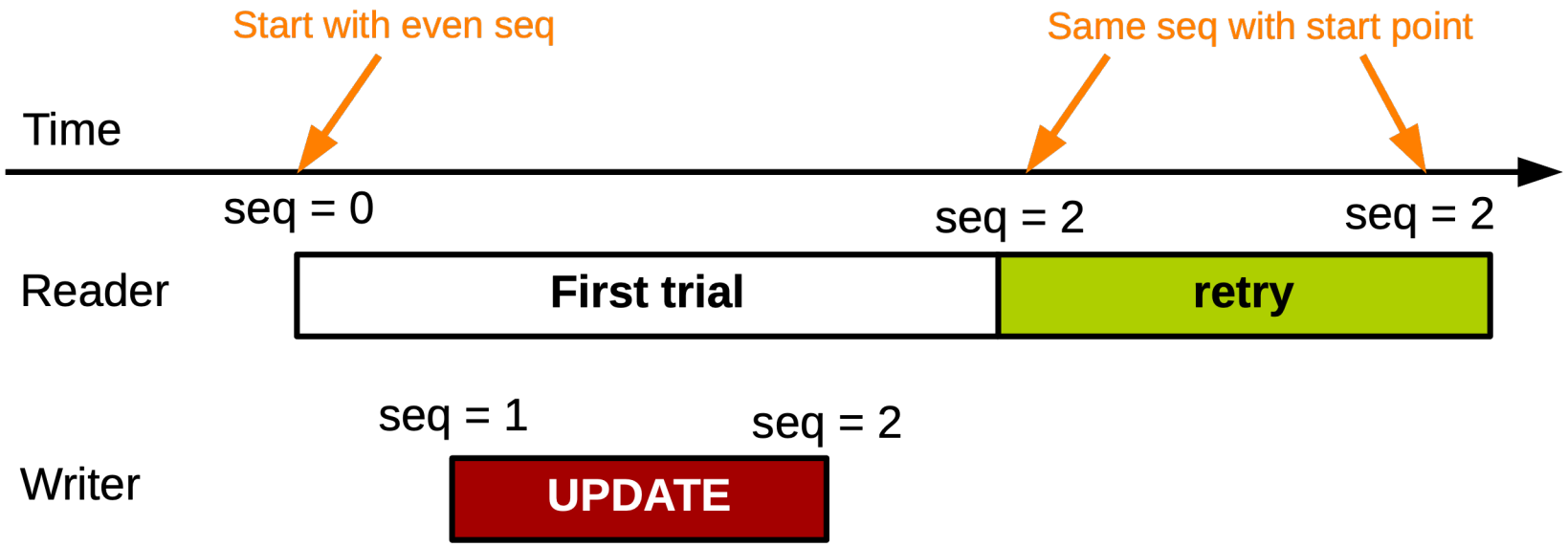
Recap: rwlock

- Allow multiple readers
- Mutual exclusion between readers and a writer
- Linux rwlock is a reader-preferred algorithm



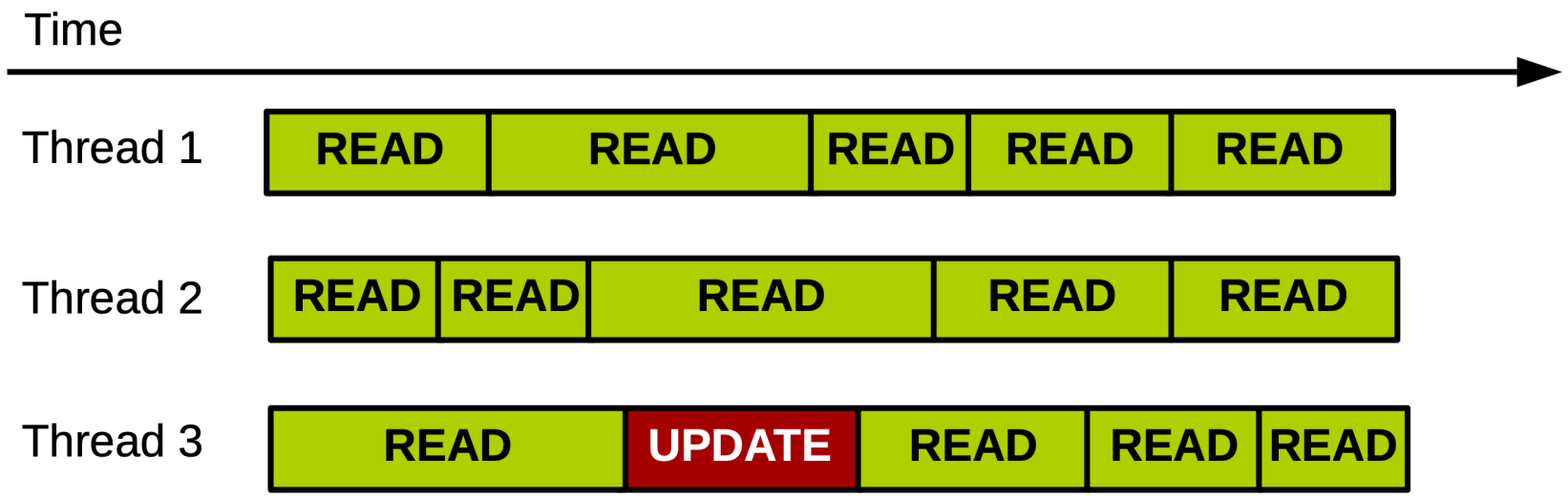
Recap: seqlock

- Consistent mechanism without starving writers



Read-Copy-Update (RCU)

- RCU supports concurrency between multiple readers and a single writer
  - A writer does not block readers!
  - Allow multiple readers with almost zero overhead
  - Optimize for reader performance

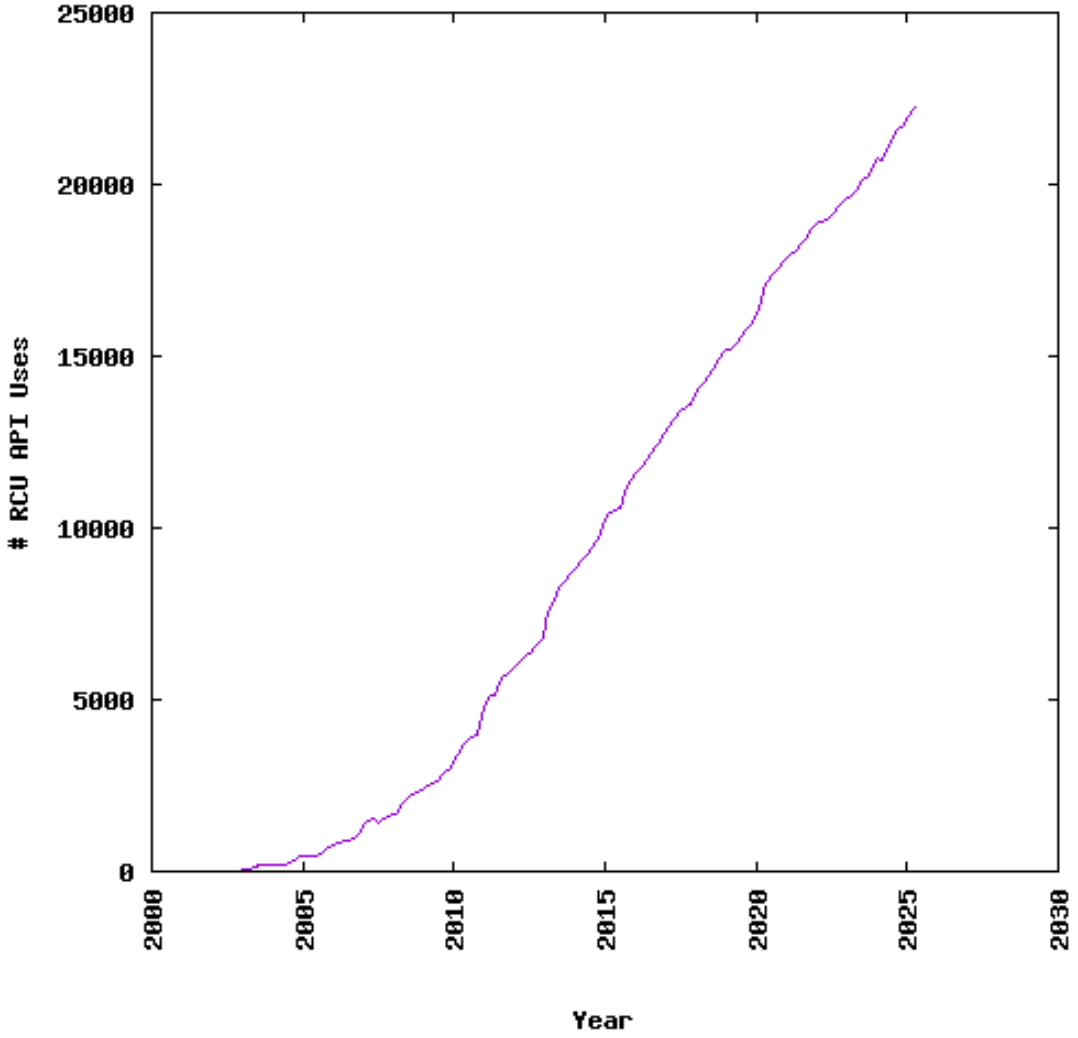


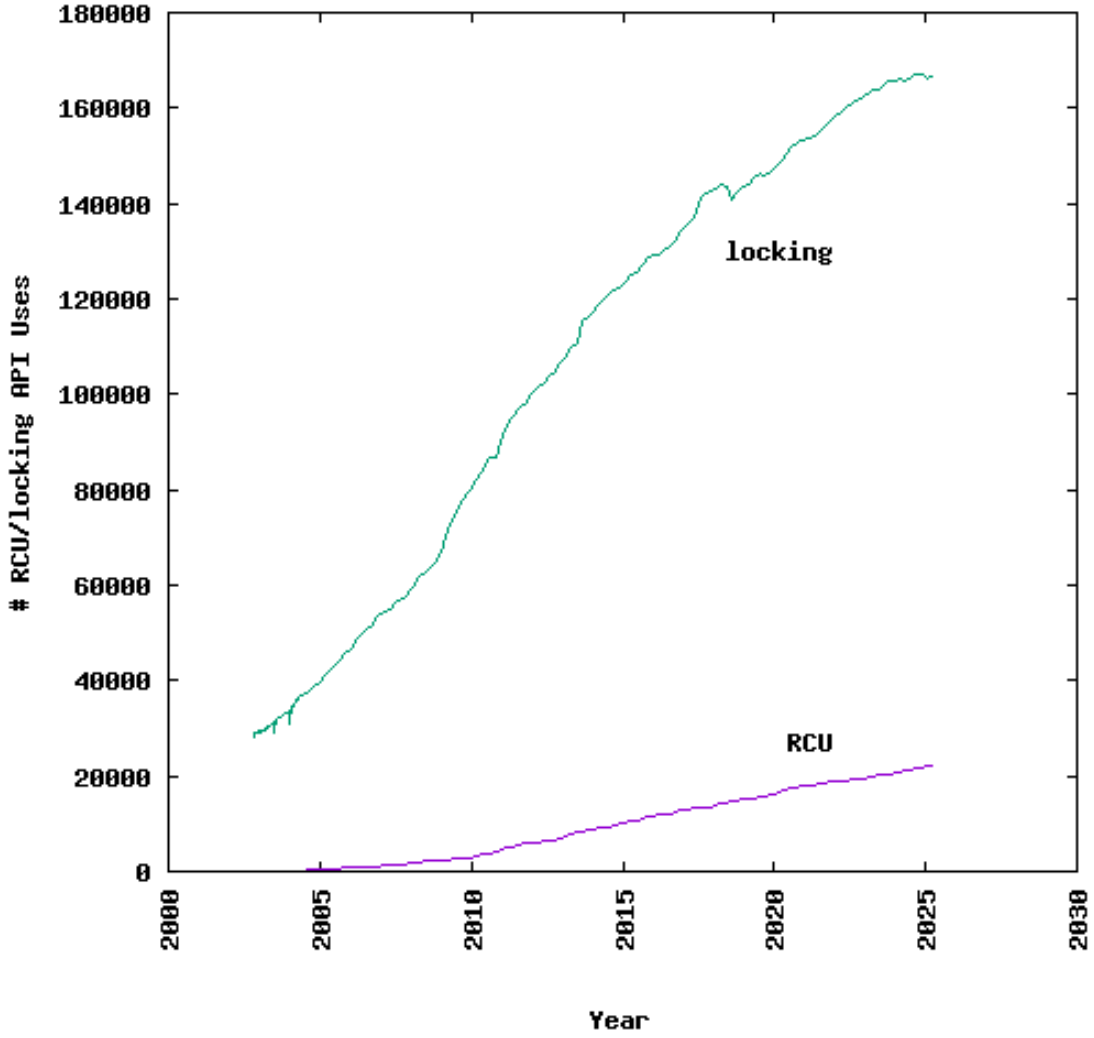
- Only require locks for writes; carefully update data structures so readers see consistent views of data all the time
- RCU ensures that reads are coherent by maintaining multiple version of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete.
- Widely-used for read-mostly data structures
  - Directory entry caches, DNS name database, etc.

RCU Author: Paul McKenney

Refer to: https://scholar.google.com/citations?user=k8F7-kUAAAAJ&hl=en

RCU Usage in Linux Kernel

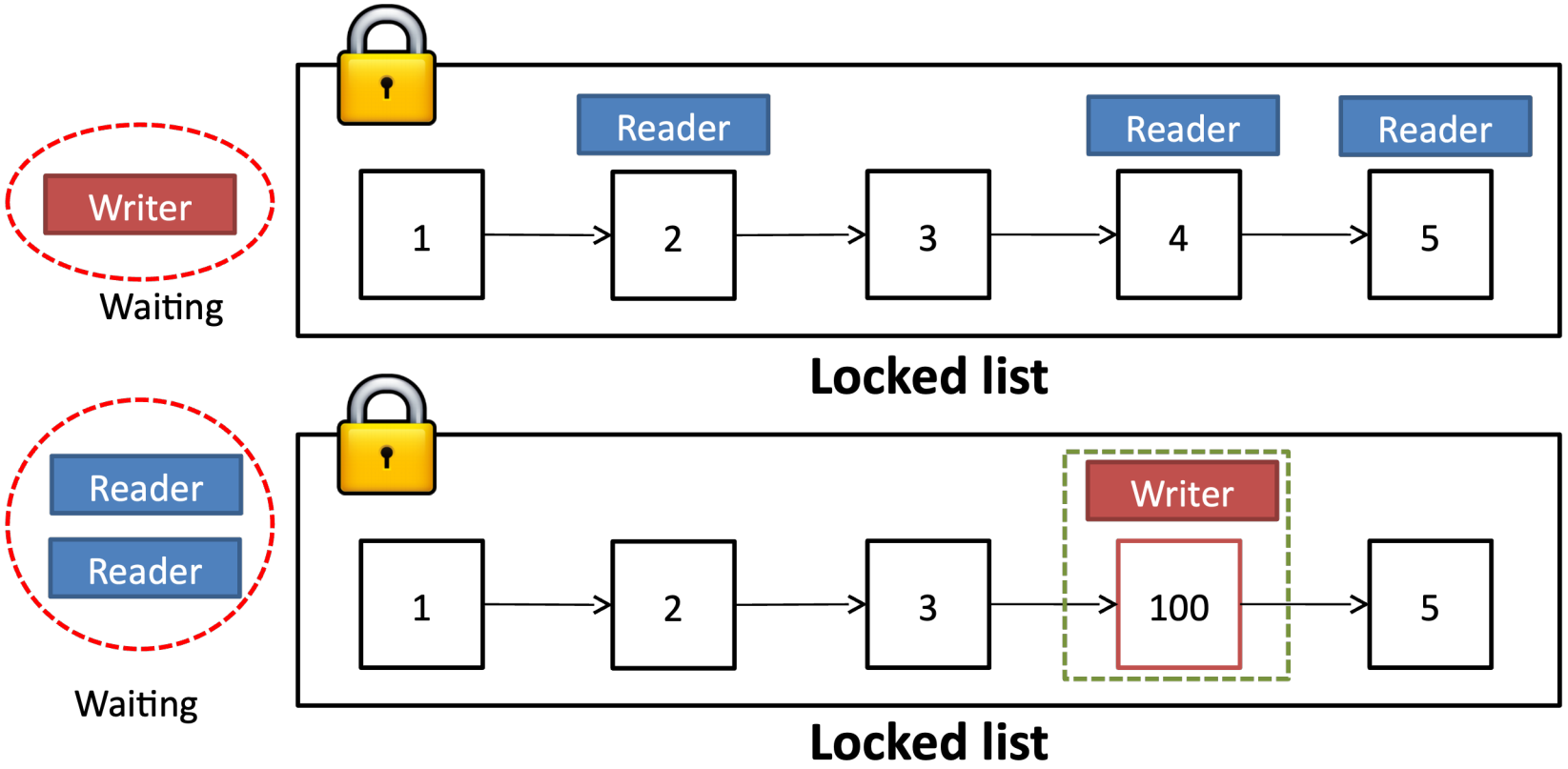




Source: RCU Linux Usage

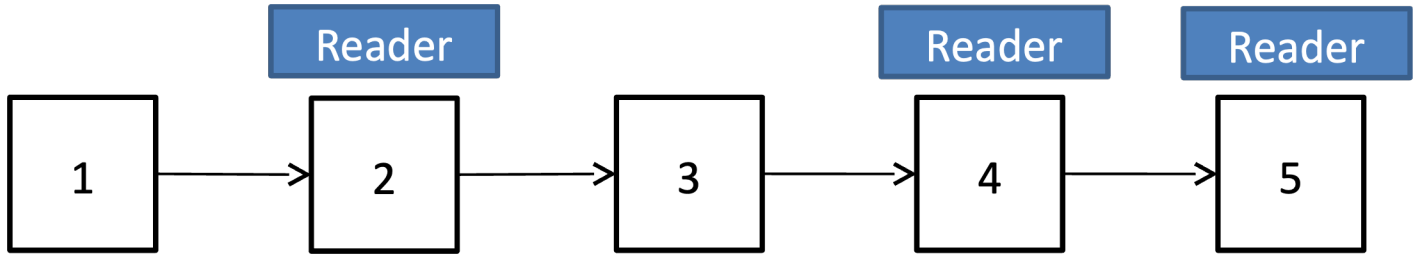
RWLock-based Linked List

- Even using a scalable rwlock, readers and a writer cannot concurrently access the list



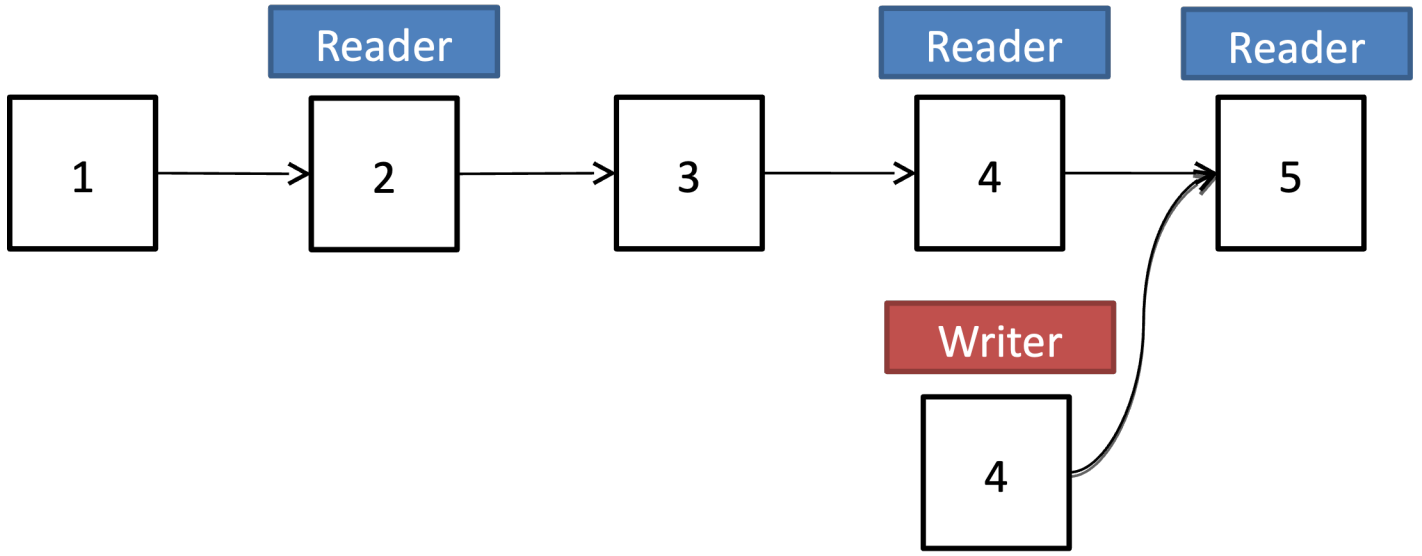
RCU-based Linked List

- Allow concurrent access of readers!



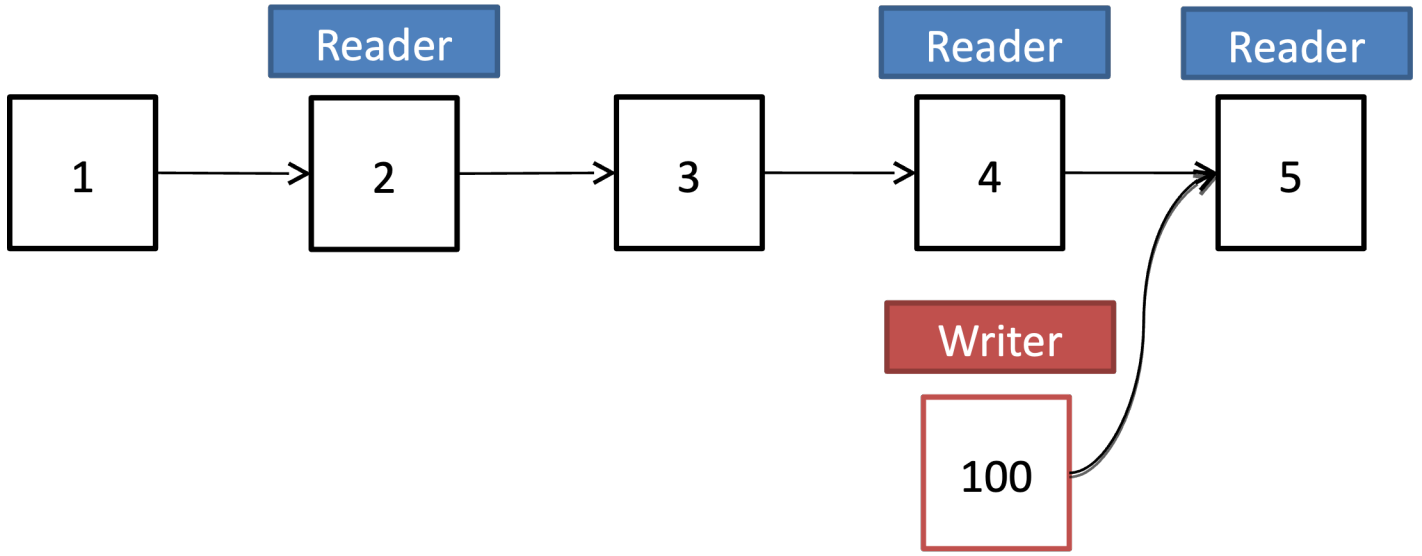
RCU-based Linked List

- A writer copies an element first



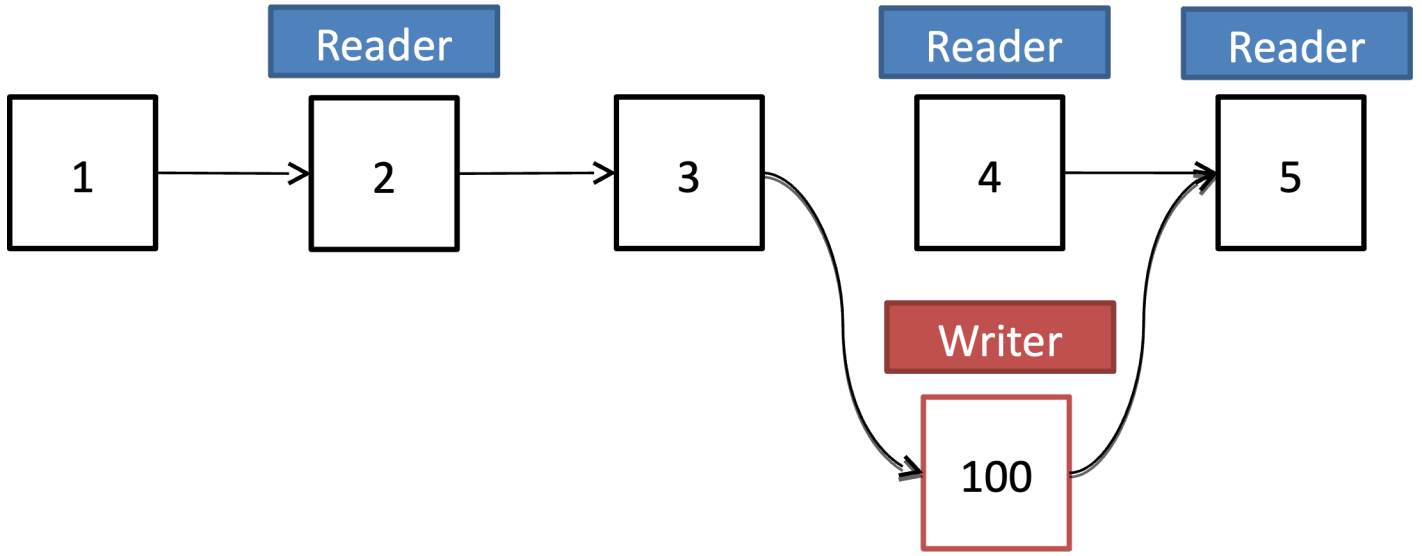
RCU-based Linked List

- And then it updates the element



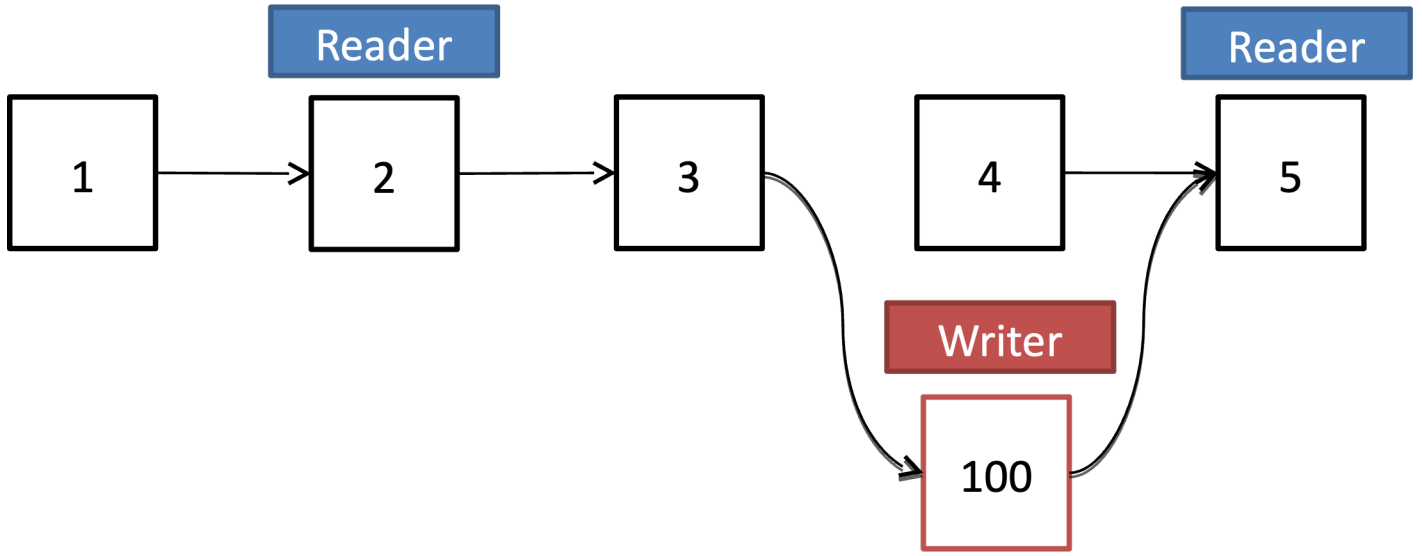
RCU-based Linked List

- And then it makes its change public by updating the next pointer of its previous pointer → New readers will traverse 100 instead of 4.



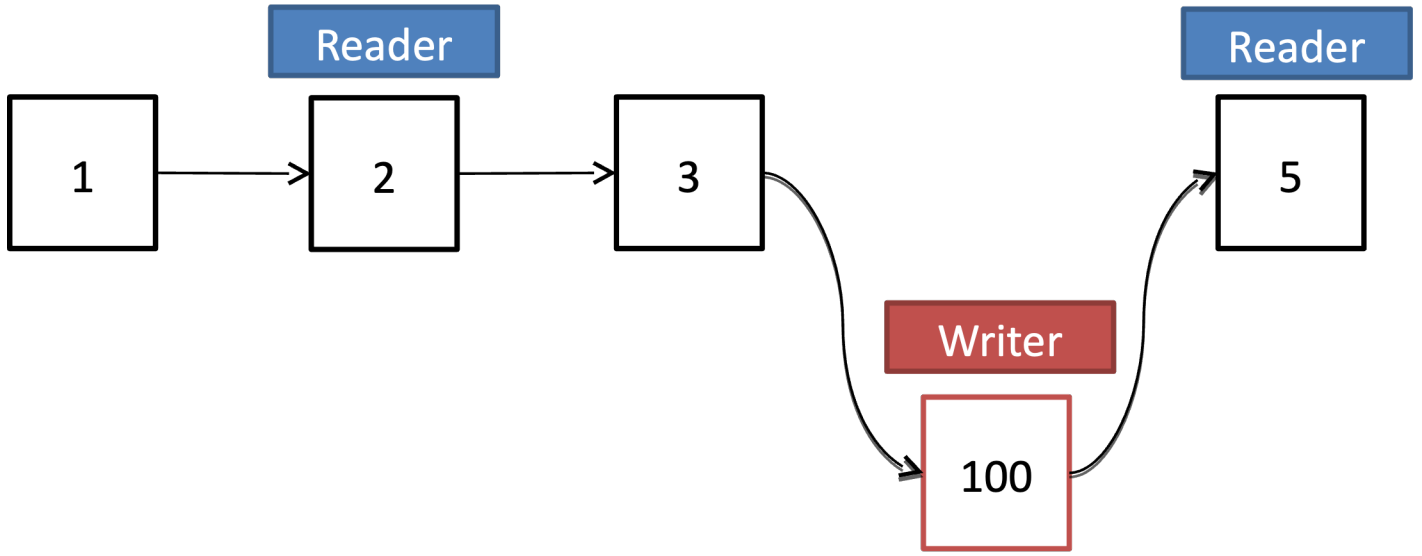
RCU-based Linked List

- Do not free the old node, 4, until no readers access it.



RCU-based Linked List

- When it is guaranteed that there is no reader accessing the old node, free the old node.



RCU API

```c
/* linux/include/linux/rcupdate.h */
/* Mark the beginning of an RCU read-side critical section */
void rcu_read_lock(void);

/* Mark the end of an RCU read-side critical section */
void rcu_read_unlock(void);

/* Assign to RCU-protected pointer: p = v
 * @p: pointer to assign to
 * @v: value to assign (publish) */
#define rcu_assign_pointer(p, v) ..

/* Fetch RCU-protected pointer for dereferencing
 * @p: The pointer to read, prior to dereferencing */
#define rcu_dereference(p) ...

/* Queue an RCU callback for invocation after a grace period.
 * @head: structure to be used for queueing the RCU updates.
 * @func: actual callback function to be invoked after the grace period */
void call_rcu(struct rcu_head *head, rcu_callback_t func);

/* Wait until quiescent states */
void synchronize_rcu(void);
```

Replace rwlock by RCU

```c
/* RWLock */
1 struct el {
2   struct list_head lp;
3   long key;
4   int data;
5   /* Other data fields */
6 };
7 DEFINE_RWLOCK(listlock);
8 LIST_HEAD(head);
```

```c
/* RCU */
1 struct el {
2   struct list_head lp;
3   long key;
4   int data;
5   /* Other data fields */
6 };
7 DEFINE_SPINLOCK(listlock);
8 LIST_HEAD(head);
```

Replace rwlock by RCU

```c
/* RWLock */
int search(long key, int *result)
{
  struct el *p;

  read_lock(&listlock);
  list_for_each_entry(p,&head,lp) {
    if (p->key == key) {
      *result = p->data;
      read_unlock(&listlock);
      return 1;
    }
  }
  read_unlock(&listlock);
  return 0;
}
```

```c
/* RCU */
int search(long key, int *result)
{
  struct el *p;

  rcu_read_lock();
  list_for_each_entry_rcu(p,&head,lp) {
    if (p->key == key) {
      *result = p->data;
      rcu_read_unlock();
      return 1;
    }
  }
  rcu_read_unlock();
  return 0;
}
```

Replace rwlock by RCU

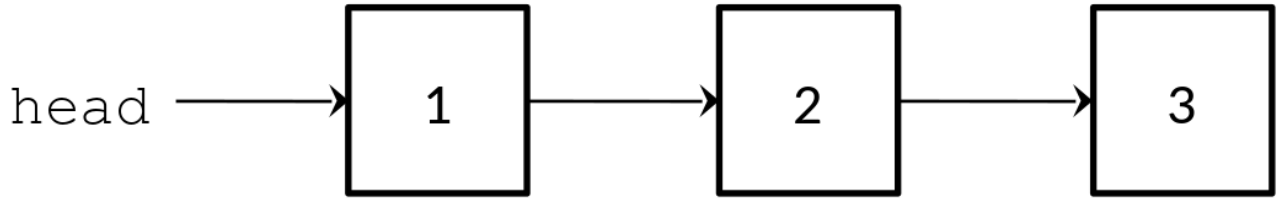
```
/* RWLock */
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   write_lock(&listlock);
 6   list_for_each_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       list_del(&p->lp);
 9       write_unlock(&listlock);

10       kfree(p);
11       return 1;
12     }
13   }
14   write_unlock(&listlock);
15   return 0;
16 }
```

```
/* RCU */
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   spin_lock(&listlock);
 6   list_for_each_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       list_del_rcu(&p->lp);
 9       spin_unlock(&listlock);
10       synchronize_rcu();
11       kfree(p);
12       return 1;
13     }
14   }
15   spin_unlock(&listlock);
16   return 0;
17 }
```

RCU Primer

Lock-free reads + Single pointer update + Delayed free

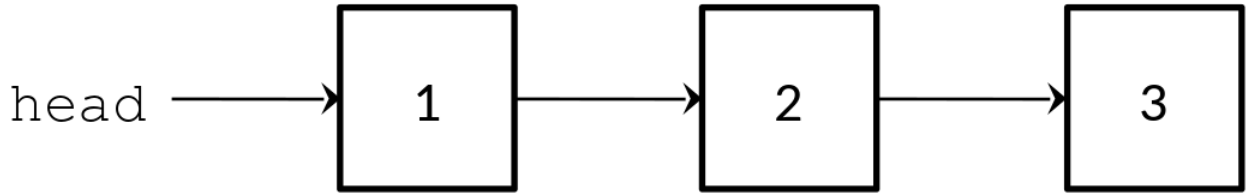



```
length() {
  rcu_read_lock(); {
    p=rcu_dereference(head); //p=head
    for(i=0;p;p=p->next,i++) ;
  } rcu_read_unlock();
  return i;
}
```

```
pop_n(n) {
  for(p=head;p&&n;p=p->next,n--)
    call_rcu(free, p);
  rcu_assign_pointer(head,p); //head=p
}
```

RCU Primer

Lock-free reads + **Single pointer update** + **Delayed free**



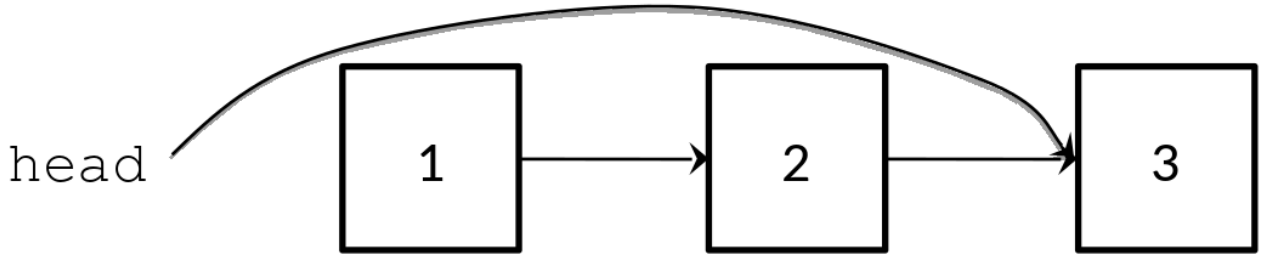

```
length() {
   rcu_read_lock(); {
     p=rcu_dereference(head); //p=head
     for(i=0;p;p=p->next,i++) ;
   } rcu_read_unlock();
   return i;
}
```

```
pop_n(n) {
   for(p=head;p&&n;p=p->next,n--)
     call_rcu(free, p);
   rcu_assign_pointer(head,p); //head=p
}
```

- No locks, no barriers
- `rcu_read_lock()` just sets the status of a thread "reading" RCU data.

RCU Primer

Lock-free reads + **Single pointer update** + **Delayed free**




```
length() {
  rcu_read_lock(); {
    p=rcu_dereference(head); //p=head
    for(i=0;p;p=p->next,i++) ;
  } rcu_read_unlock();
  return i;
}
```

```
pop_n(n) {
  for(p=head;p&&n;p=p->next,n--)
    call_rcu(free, p);
  rcu_assign_pointer(head,p); //head=p
}
```

- No locks, no barriers
- `rcu_read_lock()` just sets the status of a thread "reading" RCU data.
- Update exactly one pointer, which is atomic.

RCU Primer

Lock-free reads + **Single pointer update** + **Delayed free**


```
length() {
  rcu_read_lock(); {
    p=rcu_dereference(head); //p=head
    for(i=0;p;p=p->next,i++) ;
  } rcu_read_unlock();
  return i;
}
```

```
pop_n(n) {
  for(p=head;p&&n;p=p->next,n--)
    call_rcu(free, p);
  rcu_assign_pointer(head,p); //head=p
}
```

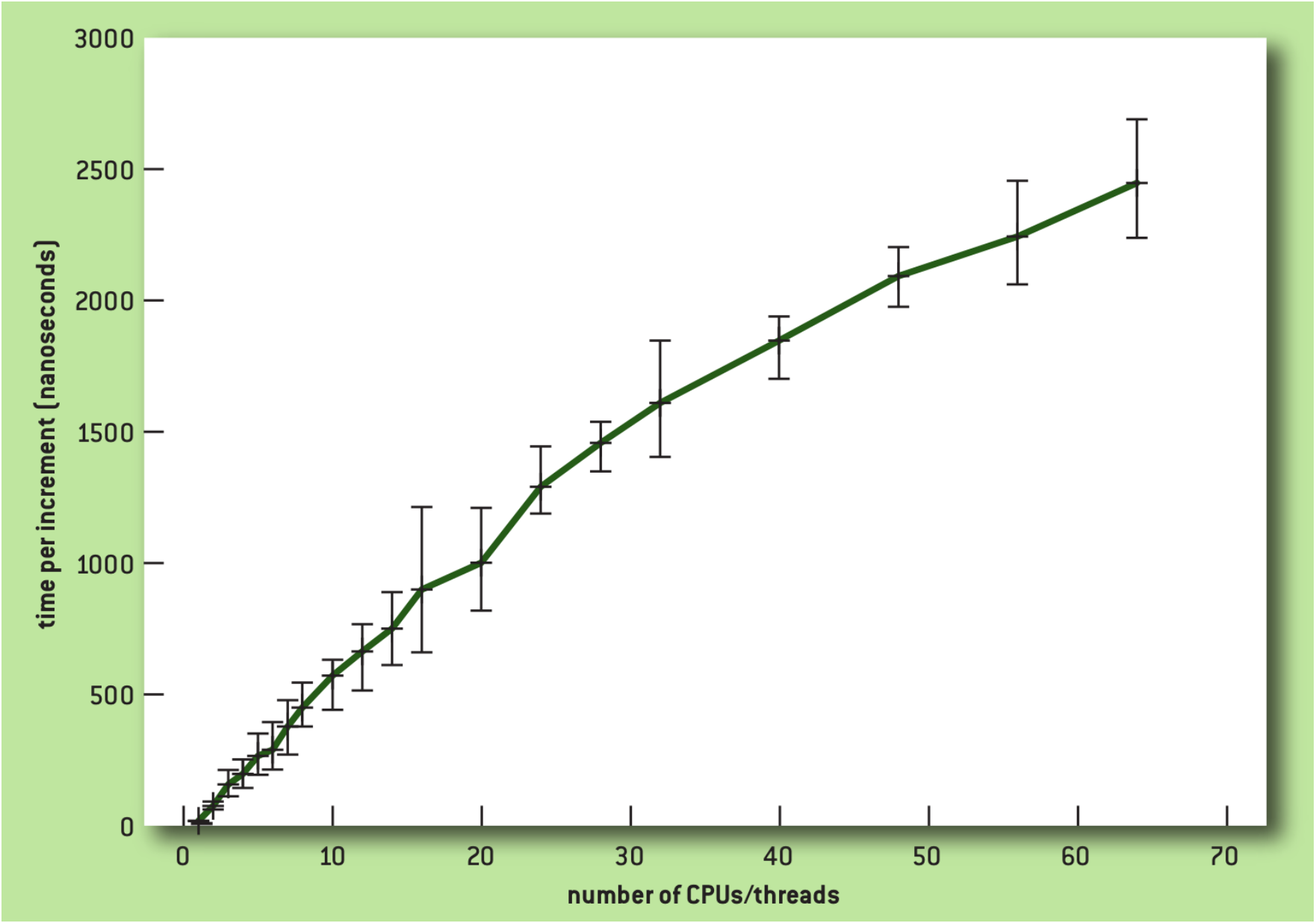
- No locks, no barriers
- `rcu_read_lock()` just sets the status of a thread "reading" RCU data.

- Update exactly one pointer, which is atomic.
- Free delayed until all readers return (e.g., by waiting for all CPU's to schedule)

Delayed Free

- Grace period, quiescent state
- Efficient and scalable grace period detection is a key challenge
  - Some obvious solutions, such as reference counting, won't work ...

Atomic Increment does Not Scale



Toy RCU Implementation

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

static inline void rcu_read_unlock(void)
{
    preempt_enable();
}

#define rcu_assign_pointer(p, v)    ({ \
    smp_wmb(); \
    ACCESS_ONCE(p) = (v); \
})

#define rcu_dereference(p)      ({ \
    typeof(p) _value = ACCESS_ONCE(p); \
    smp_read_barrier_depends(); /* nop on most architectures */ \
    (_value); \
})
```

Toy RCU Implementation

```
void call_rcu(void (*callback) (void *), void *arg)
{
    /* add callback/arg pair to a list */
}

void synchronize_rcu(void)
{
    int cpu, ncpus = 0;

    for_each_cpu(cpu)
        schedule_current_task_to(cpu);

    for each entry in the call_rcu list
        entry->callback (entry->arg);
}
```

RCU List

```c
/* linux/include/linux/rculist.h */
/* Circular doubly-linked list */

/* Add a new entry to rcu-protected list
 * @new: new entry to be added
 * @head: list head to add it after */
void list_add_rcu(struct list_head *new, struct list_head *head);

/* Deletes entry from list without re-initialization
 * @entry: the element to delete from the list. */
void list_del_rcu(struct list_head *entry);

/* Replace old entry by new one
 * @old : the element to be replaced
 * @new : the new element to insert */
void list_replace_rcu(struct list_head *old, struct list_head *new);

/* Iterate over rcu list of given type
 * @pos:    the type * to use as a loop cursor.
 * @head:   the head for your list.
 * @member: the name of the list_head within the struct. */
#define list_for_each_entry_rcu(pos, head, member) ..
```

RCU hlist

```c
/* linux/include/linux/rculist.h */
/* Non-circular doubly-linked list */

/* Adds the specified element to the specified hlist,
 * while permitting racing traversals.
 * @n: the element to add to the hash list.
 * @h: the list to add to. */
void hlist_add_head_rcu(struct hlist_node *n, struct hlist_head *h);

/* Replace old entry by new one
 * @old : the element to be replaced
 * @new : the new element to insert */
void hlist_replace_rcu(struct hlist_node *old, struct hlist_node *new);

/* Deletes entry from hash list without re-initialization
 * @n: the element to delete from the hash list. */
void hlist_del_rcu(struct hlist_node *n);

/* Iterate over rcu list of given type
 * @pos:    the type * to use as a loop cursor.
 * @head:   the head for your list.
 * @member: the name of the hlist_node within the struct. */
#define hlist_for_each_entry_rcu(pos, head, member) ...
```

RCU Limitations

- Do not provide a mechanism to coordinate multiple writers
  - Most RCU-based algorithms end up using spinlock to prevent concurrent write operations
- All modifications should be a single-pointer-update
  - Challenging to do in many cases ...

Further Readings

- Read-log-update: al ightweight synchronization mechanism for concurrent programming, SOSP'15
- Is Parallel Programming Hard, And, If So, What Can You Do About It?
- Structured Deferral: Synchronization via Procrastination
- Introuction to RCU Concepts
- LWN: What is RCU, Fundamentally?
- Introduction to RCU
- Userspace RCU