Homework

Hook on the scheduler, should not sleep (avoid recursive scheduler invocation)

- `kmalloc(n, GFP_ATOMIC);`

`stack_trace_save_user()` is not exported

- `kallsyms_lookupname()` to look up the function address

hw7: find user stack trace

I realized the save_stack_trace_user doesn't exist in recent kernels.

Once solution is to use stack_trace_save_user to retrieve the user stack trace. You can get a pointer of this un-exported function using:

(void*)kallsyms_lookup_name("stack_trace_save_user");

Homework

"store the stack trace instead of PID as the key"

- `jhash()` - allows hash a buffer to a key
- `stack_trace_save()` and `stack_trace_save_user()` return a buffer

hw7: Explanation of "store the stack trace instead of PID as the key" COLLAPSE

stack_trace_save and stack_trace_save_user allow you to obtain a kernel stack trace and store it in an array. (the first parameter unsigned long *store)

And use jhash to get a hash value from the stack trace array (buffer).

Move hw7 deadline to March 1st? hw8 also due March 1st

Final project

Have you started yet?

Start early and have more discussion!

Recap: synchronization primitives

Protect shared data from concurrent access

Non-sleeping (non-blocking) synchronization primitives

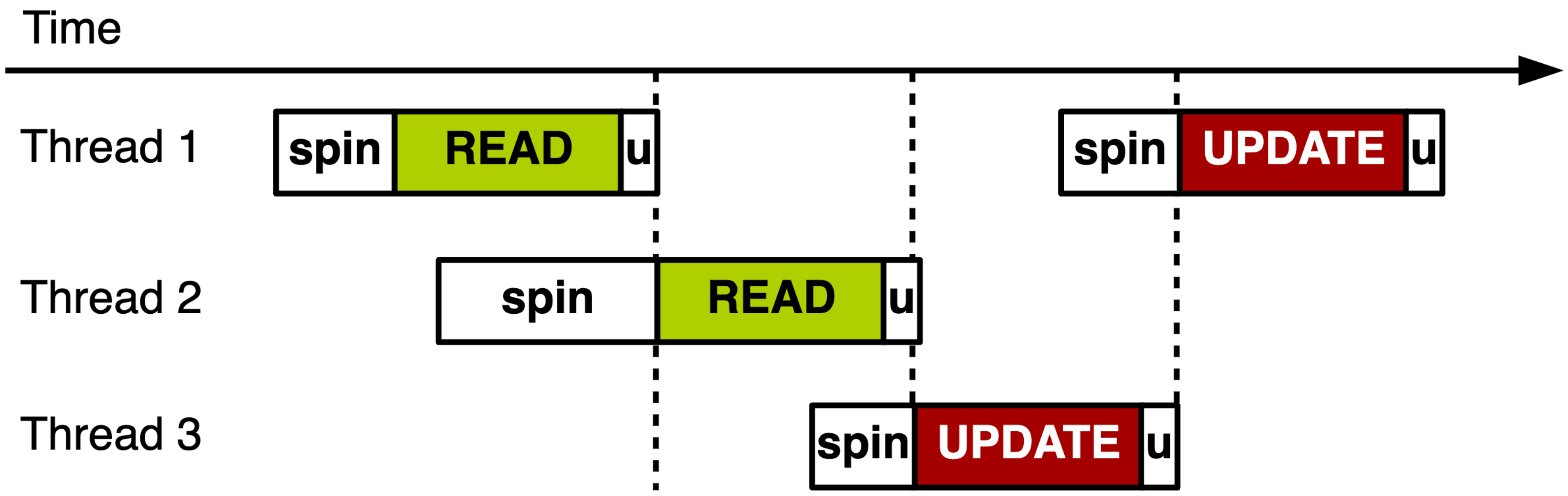
- atomic operation, spinlock, reader-write lock (`rwlock`), sequential lock (`seqlock`)

Sleeping (blocking) synchronization primitives

- semaphore, mutex, completion variable, wait queue

Recap: spinlock

Implement mutual exclusion

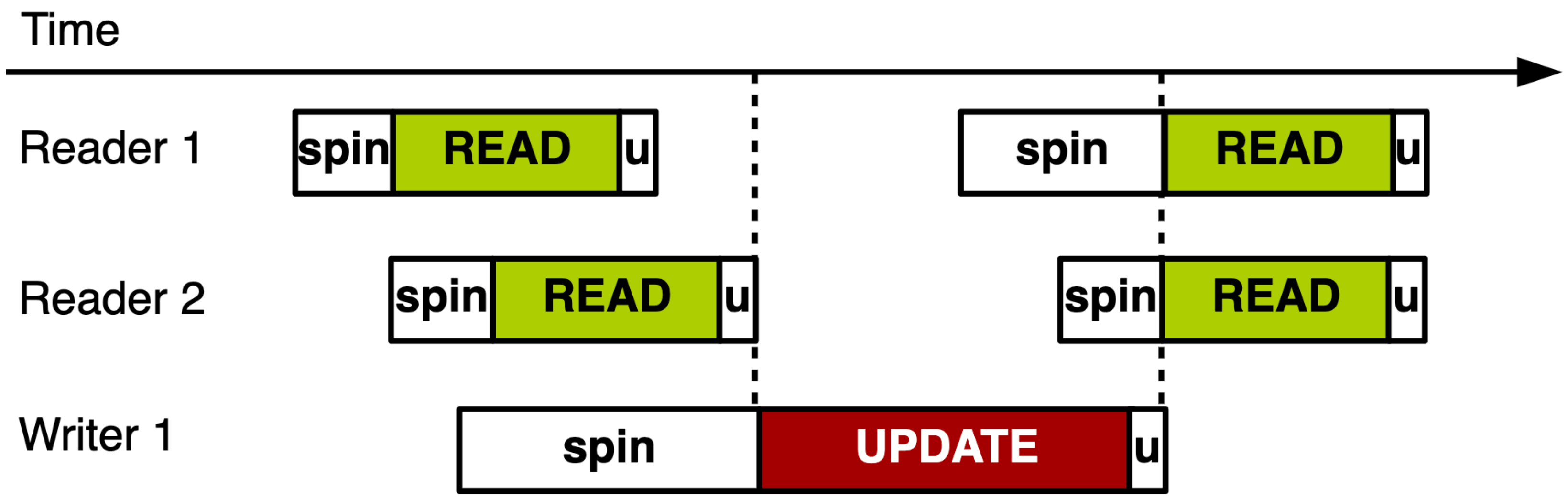


Recap: rwlock

Allow multiple readers

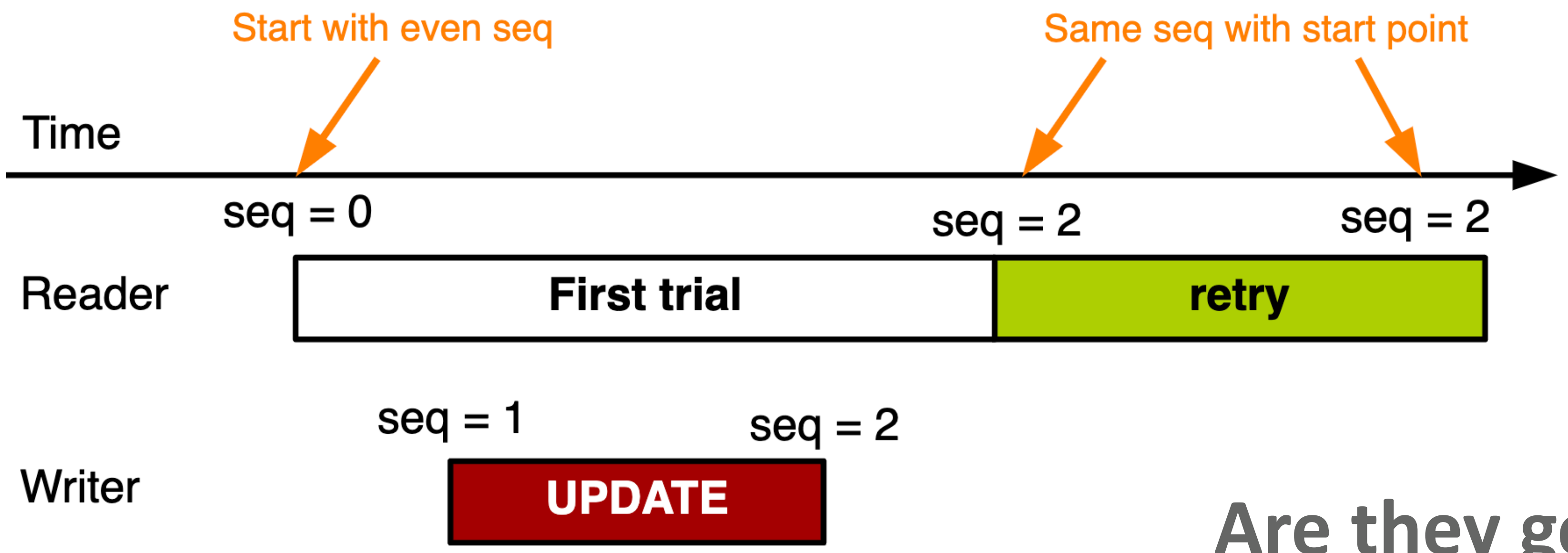
Mutual exclusion between readers and a writer

Linux `rwlock` is a reader-preferred algorithm



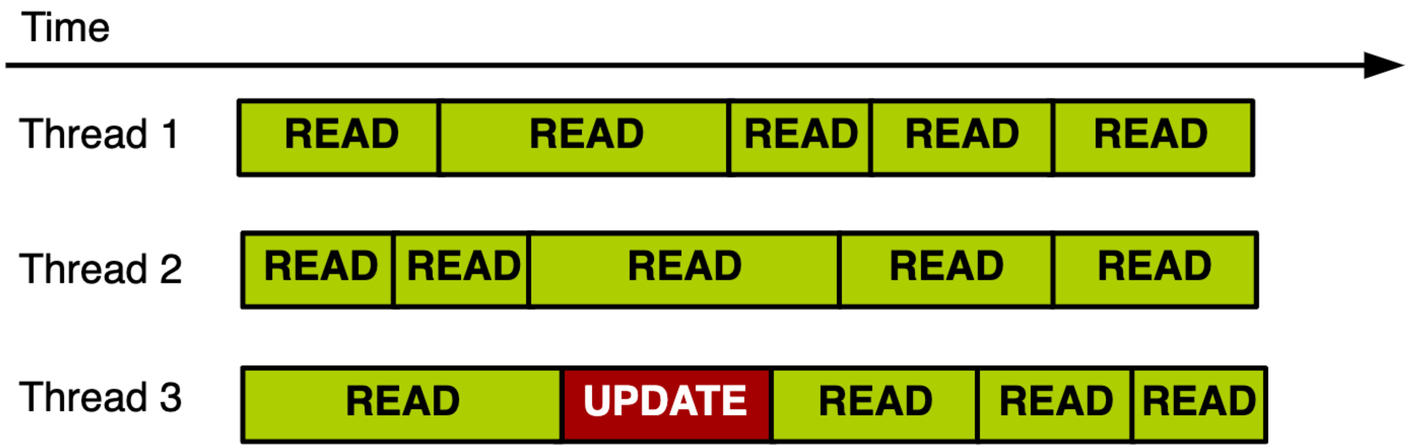
Recap: seqlock

Consistent mechanism without starving writers



Are they good enough?

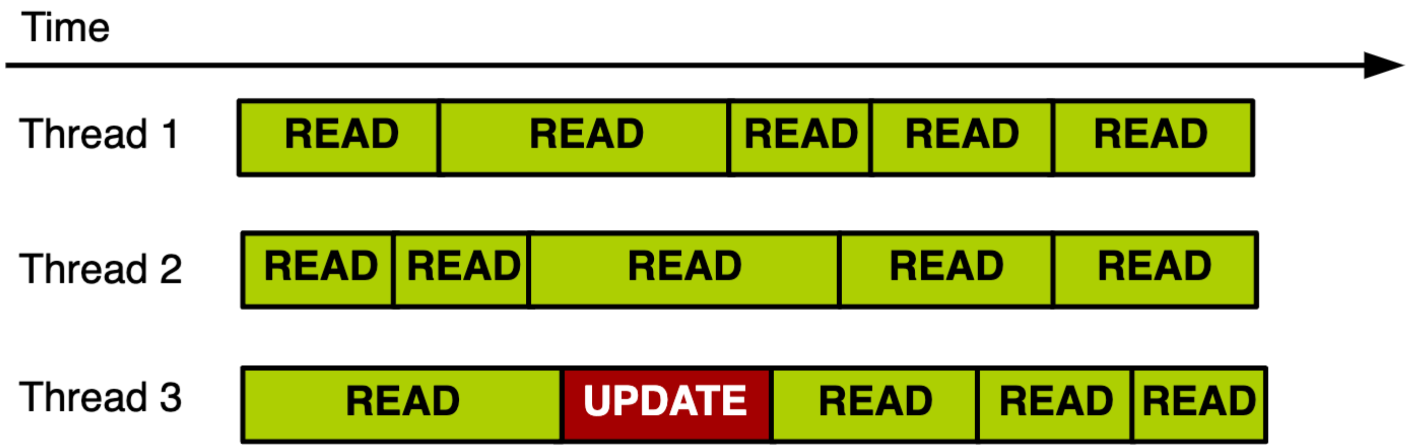
There is no stop, or busy loop when a writer update a list!



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



Read-Copy-Update (RCU)

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**.

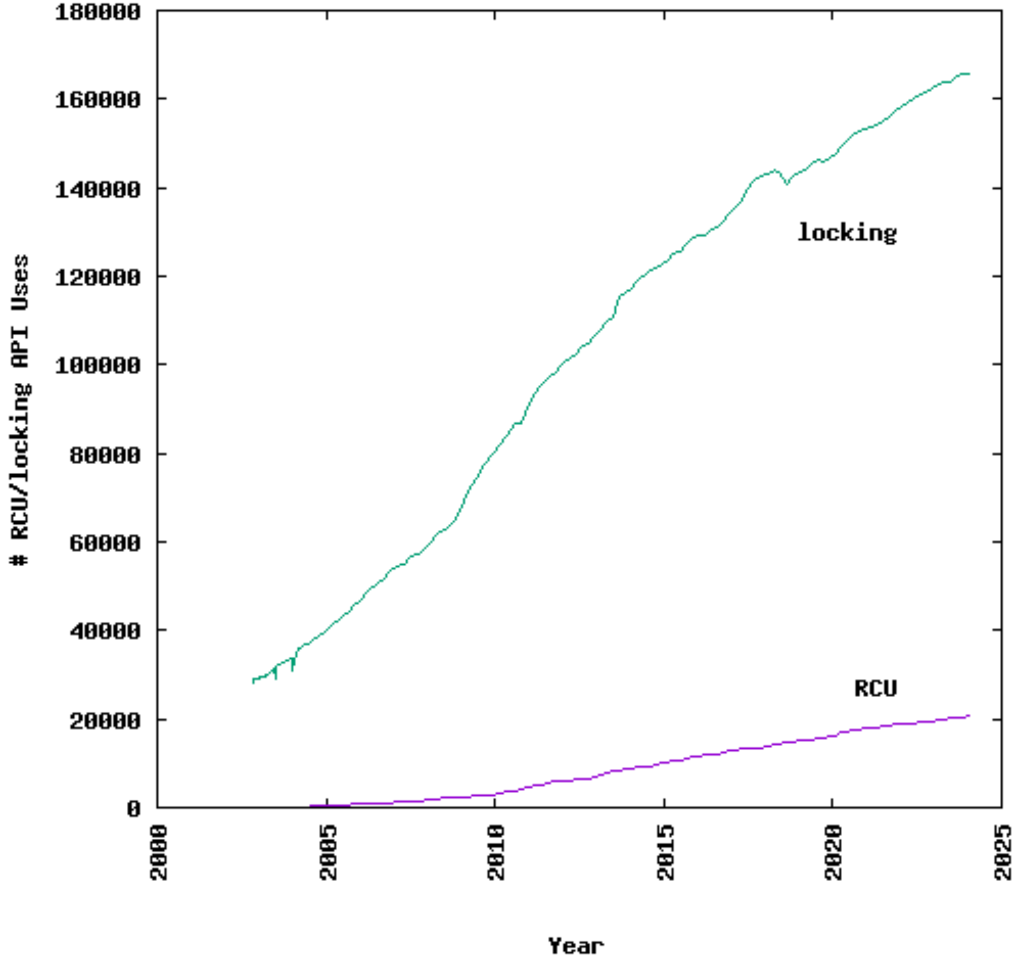
Widely-used for read-mostly data structures

- Directory entry cache, DNS name database, etc.

Who developed RCU?

Paul McKenney @ Meta

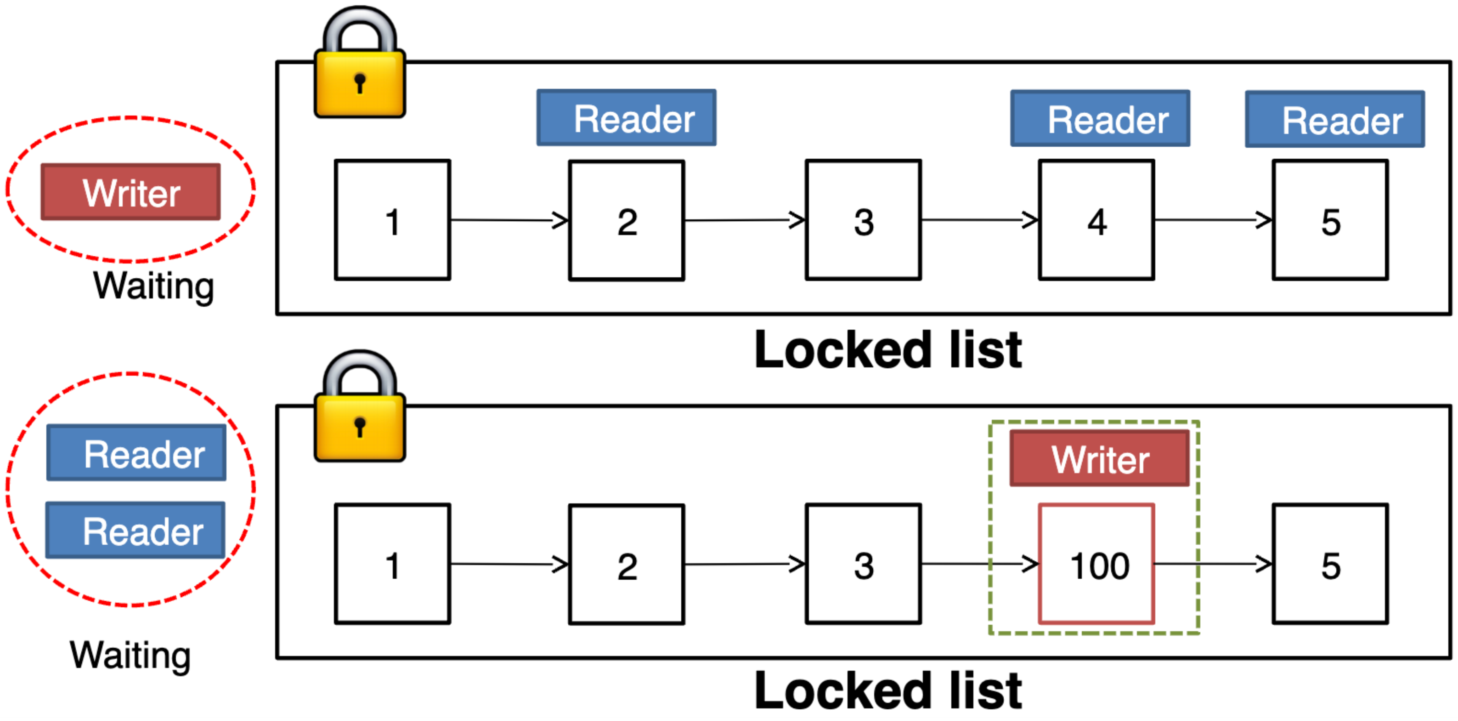
RCU usage in Linux kernel



Source: [RCU Linux Usage]

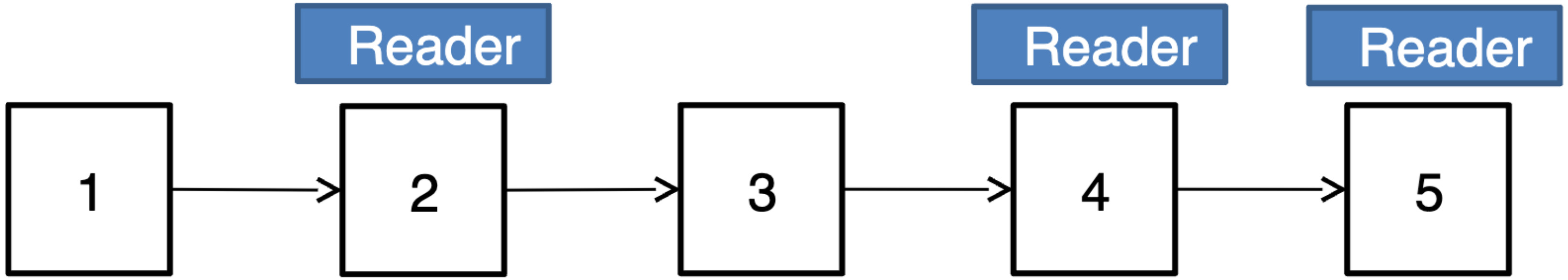
RWLock-based linked list

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



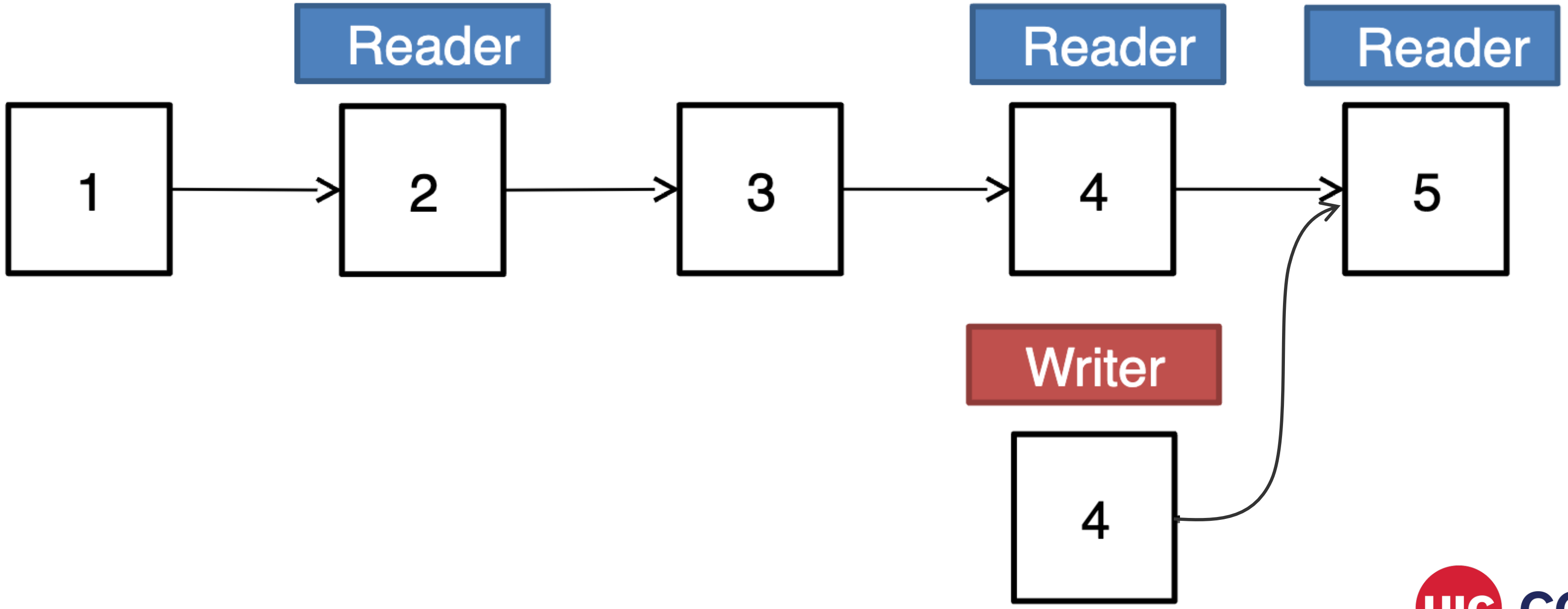
RCU-based linked list

Allow concurrent access of readers



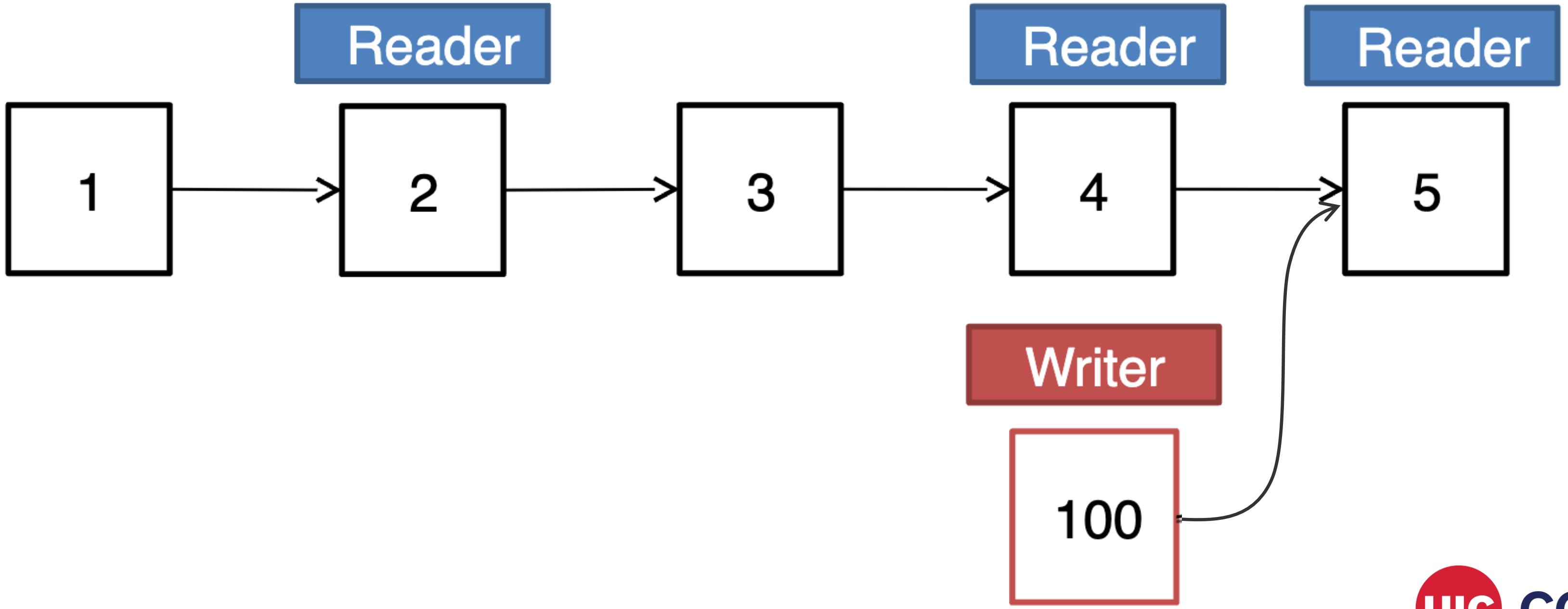
RCU-based linked list

A writer copies an element first



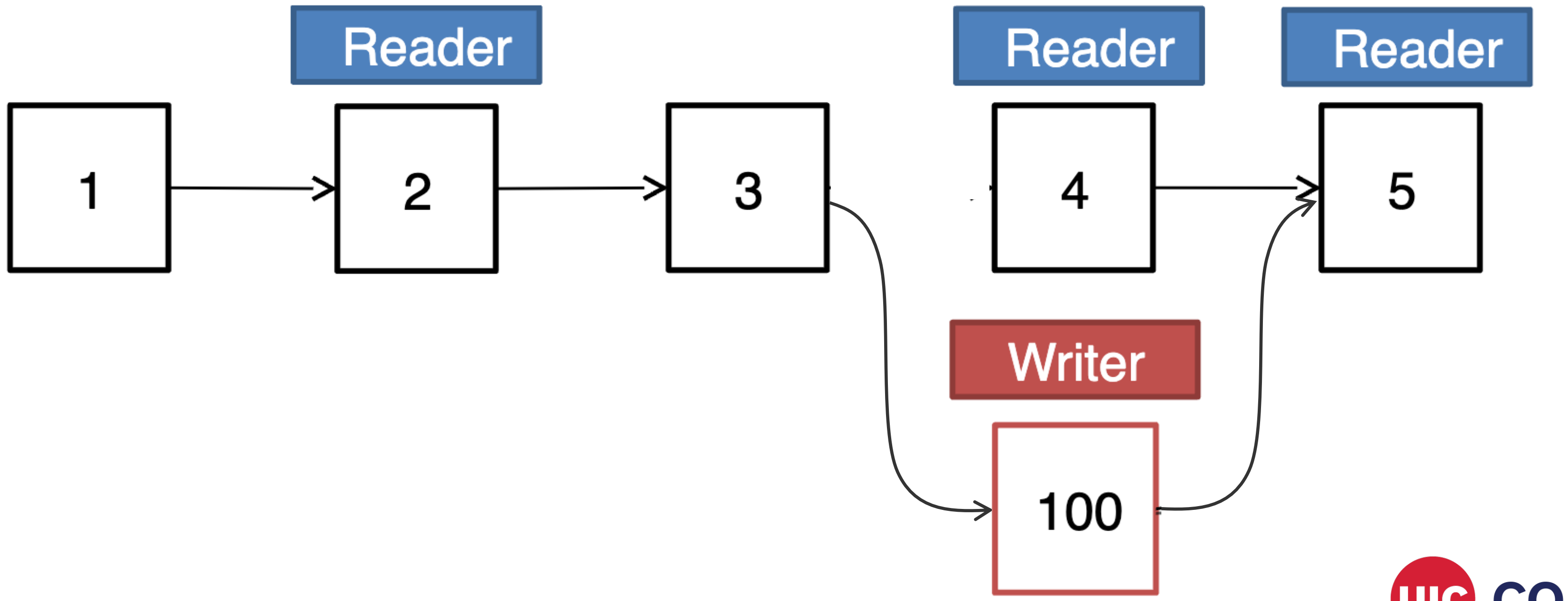
RCU-based linked list

Then it updates the element



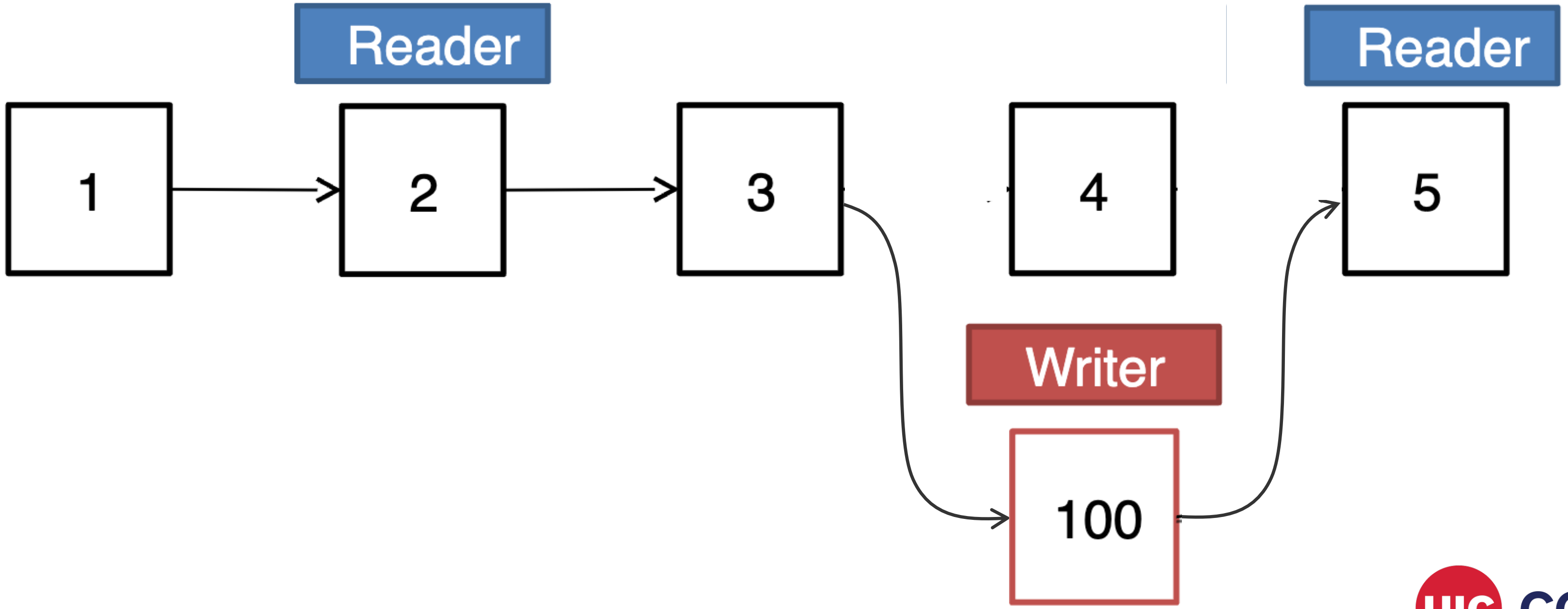
RCU-based linked list

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



RCU-based linked list

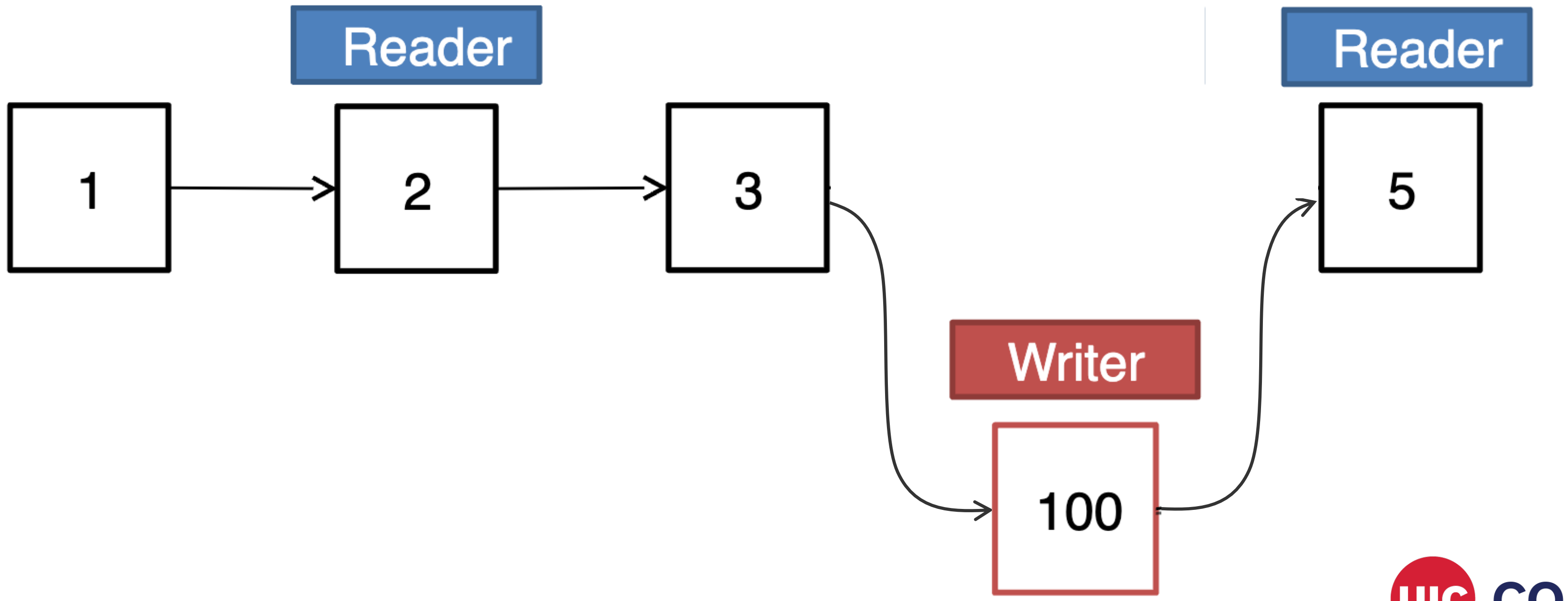
Do not free the old node, 4, until no reader accesses it.



UIC COMPUTER SCIENCE

RCU-based linked list

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



RCU API

```c
/* 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 */
struct el {
  struct list_head lp;
  long key;
  int data;
  /* Other data fields */
};
DEFINE_RWLOCK(listlock);
LIST_HEAD(head);
```

```c
/* RCU */
struct el {
  struct list_head lp;
  long key;
  int data;
  /* Other data fields */
};
DEFINE_SPINLOCK(listlock);
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;
}
```

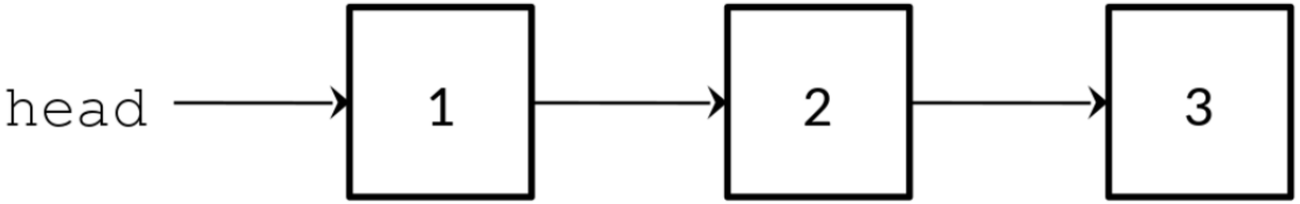
Replace rwlock by RCU

```c
/* RWLock */
int delete(long key)
{
  struct el *p;

  write_lock(&listlock);
  list_for_each_entry(p, &head, lp) {
    if (p->key == key) {
      list_del(&p->lp);
      write_unlock(&listlock);

      kfree(p);
      return 1;
    }
  }
  write_unlock(&listlock);
  return 0;
}
```

```c
/* RCU */
int delete(long key)
{
  struct el *p;

  spin_lock(&listlock);
  list_for_each_entry(p, &head, lp) {
    if (p->key == key) {
      list_del_rcu(&p->lp);
      spin_unlock(&listlock);
      synchronize_rcu();
      kfree(p);
      return 1;
    }
  }
  spin_unlock(&listlock);
  return 0;
}
```

RCU Primer

Lock-free reads + Single pointer update + Delayed free

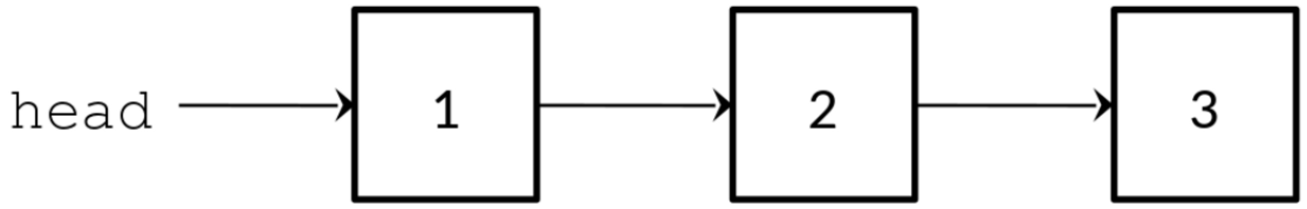



```
length() {
  rcu_read_lock(); {
    p=rcu_dereference(head); //p=head
    for(i=0;p;p=p->next,i++) ;
  } rcu_read_unlock();
  return i;
}
```

```
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



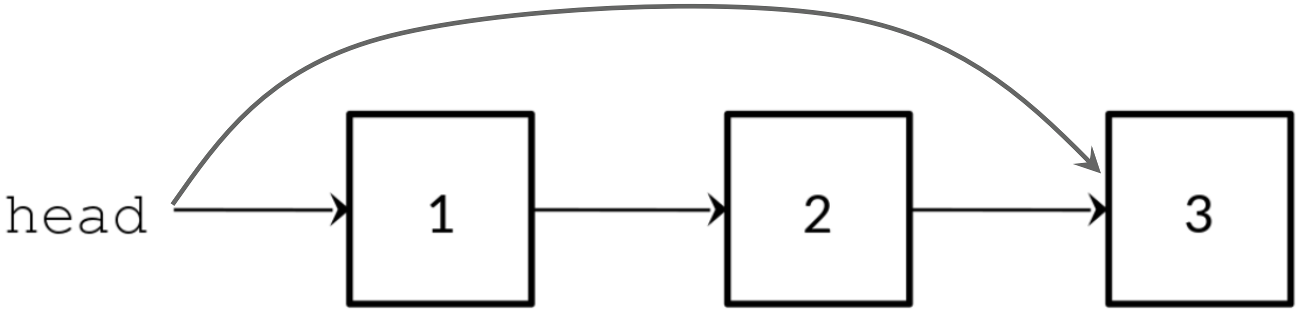

```
length() {
  rcu_read_lock(); {
    p=rcu_dereference(head); //p=head
    for(i=0;p;p=p->next,i++) ;
  } rcu_read_unlock();
  return i;
}
```

```
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 thread's status "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
}
```

- Update one pointer, which is atomic

- No locks, no barriers;
- rcu_read_lock() just sets the thread's status "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
}
```

- Update one pointer, which is atomic
- Free delayed until all readers return

- No locks, no barriers;
- rcu_read_lock() just sets the thread's status "reading" RCU data

RCU list

```c
/* 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) ..
```

UIC COMPUTER SCIENCE

RCU hlist

```
/* 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) ...
```

Limitations of RCU

Do not provide a mechanism to coordinate multiple writers

- Most RCU-based algorithms end up using `spinlock` to prevent concurrent write operations

All modification should be a single-pointer-update.

- This is challenging!

Further readings

Introduction to RCU Concepts

What is RCU, Fundamentally?

Read-log-update: a lightweight synchronization mechanism for concurrent programming, SOSP'15

Is Parallel Programming Hard, And, If So, What Can You Do About It?

Timer and Time Management

Xiaoguang Wang

Kernel notion of time

Having the notion of time passing in the kernel is essential in multiple cases:

- Perform periodic tasks (e.g., CFS time accounting)
- Delay event processing at a relative time in the future
- Give the time of the day

Kernel notion of time

Central role of the system timer

- Periodic interrupt, system timer interrupt
- Update system uptime, time of day, balance runqueues, record statistics, etc.
- Pre-programmed frequency, timer tick rate
- tick = 1/(tick rate) seconds

Set a dynamic timer to schedule an event in a relative time from now

Tick rate and jiffies

The tick rate (system timer frequency) is defined in the HZ variable

Set to `CONFIG_HZ` in `include/asm-generic/param.h`

- Kernel compile-time configuration option

Default value is per-architecture:

Architecture	Frequency (HZ)	Period (ms)
x86	1000	1
ARM	100	10
PowerPC	100	10

Tick rate: the ideal `HZ` value

High timer frequency → high precision

- Kernel timers (finer resolution)
- System call with timeout value (e.g., `poll`) → significant performance improvement for some applications
- More accurate timing measurements
- Process preemption occurs more accurately
  - low frequency allows processes to potentially get (way) more CPU time after the expiration of their timeslices

Tick rate: the ideal HZ value

High timer frequency → more timer interrupt → larger overhead

- Not a very significant issue in modern hardware

Tickless OS

Option to compile the kernel as a tickless system

- `NO_HZ` family of compilation options

The kernel dynamically reprogram the system timer according to the current timer status

Overhead reduction, Energy savings

- CPUs spend more time in low power idle states

jiffies

A global variable holds the number of timer ticks since the system booted (`unsigned long`)

Conversion between `jiffies` and seconds

- `jiffies` = seconds * `HZ`
- seconds = `jiffies` / `HZ`

```
unsigned long time_stamp = jiffies;          /* Now */
unsigned long next_tick = jiffies + 1;       /* One tick from now */
unsigned long later = jiffies + 5*HZ;        /* 5 seconds from now */
unsigned long fraction = jiffies + HZ/10;    /* 100 ms from now */
```

Representation of `jiffies`

`sizeof(jiffies)` is 32 bits on 32-bit architectures and 64 bits for 64-bit architectures

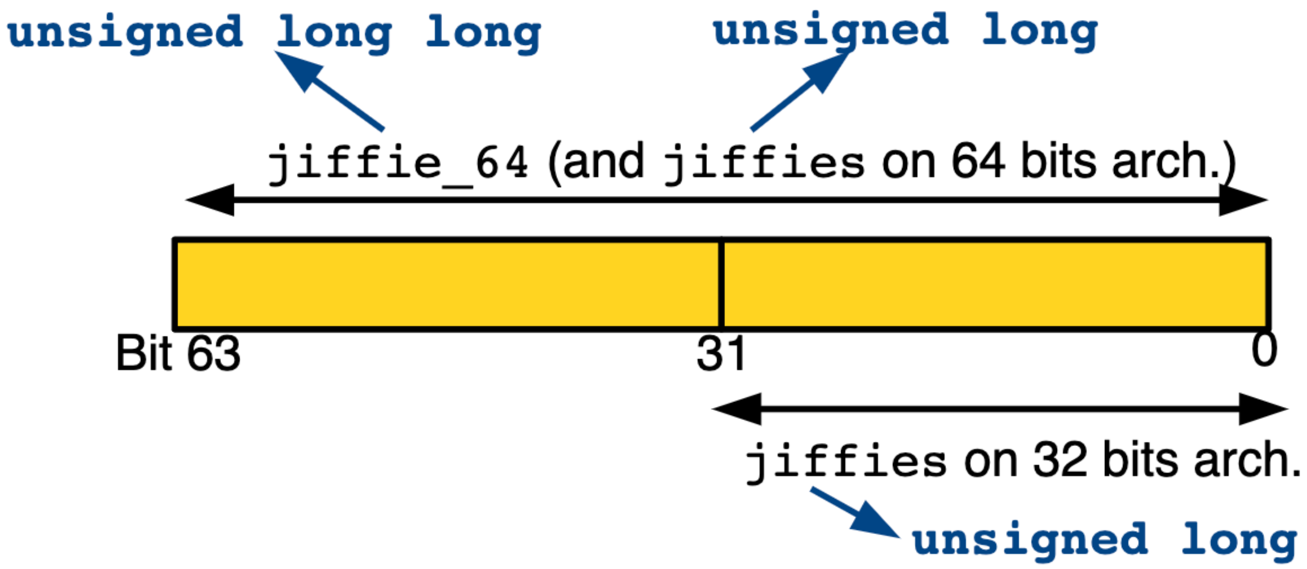
On a 32 bits variable with `HZ` == 100, overflows in 497 days

- Still on 32 bits with `HZ` == 1000, overflows in 50 days

But on a 64 bits variable, no overflow for a very long time

Representation of `jiffies`

We want access to a 64 bits variable while still maintaining an unsigned long on both architectures



`jiffies` wraparound

An unsigned integer going over its maximum value wraps around to zero

- On 32 bits, `0xFFFFFFFF + 0x1 == 0x0`

```
unsigned long timeout = jiffies + HZ/2; /* timeout in 0.5s */

/* do some work ... */

/* then see whether we took too long */
if (timeout > jiffies) {
    /* we did not time out, good ... */
} else {
    /* we timed out, error ... */
}
```

Q: Is there any issue with this code?

jiffies wraparound

```c
/* include/linux/jiffies.h */
#define time_after(a,b)
#define time_before(a,b)
#define time_after_eq(a,b)
#define time_before_eq(a,b)

/* -------------------------------- */
/* An example of using a time_*() macro */
unsigned long timeout = jiffies + HZ/2; /* timeout in 0.5s */

/* do some work ... */

/* then see whether we took too long */
if (time_before(jiffies, timeout)) { /* Use time_*() macros */
    /* we did not time out, good ... */
} else {
    /* we timed out, error ... */
}
```

Userspace and HZ

`clock(3)`

For conversion between architecture-specific `jiffies` and user-space clock tick, Linux kernel provides APIs and macros

`USER_HZ`: user-space clock tick (100 in x86)

Conversion between `jiffies` and user-space clock tick

- `clock_t jiffies_to_clock_t(unsigned long x);`
- `clock_t jiffies_64_to_clock_t(u64 x);`

Hardware clocks and timers

Real-Time Clock (RTC)

- Stores the wall-clock time (still incremented when the computer is powered off)
- Backed-up by a small battery on the motherboard

Hardware clocks and timers

System timer

- Provide a mechanism for driving an interrupt at a periodic rate regardless of architecture
- System timers in x86
  - Local APIC timer: primary timer today
  - Programmable interrupt timer (PIT): was a primary timer until 2.6.17

Hardware clocks and timers

Processor's time stamp counter (TSC)

- `rdtsc`, `rdtscp`
- most accurate (CPU clock resolution)
- invariant to clock frequency (x86 architecture)
- seconds = clocks / maximum CPU clock Hz

Timer interrupt processing

Constituted of two parts: (1) architecture-dependent and (2) architecture-independent

Architecture-dependent part is registered as the handler (top-half) for the timer interrupt

1. Acknowledge the system timer interrupt (reset if needed)
2. Save the wall clock time to the RTC
3. Call the architecture independent function (still executed as part of the top-half)

Timer interrupt processing

Architecture independent part: `tick_handle_periodic()`

1. Call `tick_periodic()`
2. Increment `jiffies64`
3. Update statistics for the currently running process and the entire system (load average)
4. Run dynamic timers
5. Run `scheduler_tick()`

Timer interrupt processing

```
/* kernel/time/tick-common.c */
static void tick_periodic(int cpu)
{
    if (tick_do_timer_cpu == cpu) {
        write_seqlock(&jiffies_lock);

        /* Keep track of the next tick event */
        tick_next_period =
            ktime_add(tick_next_period, tick_period);

        do_timer(1); /* ! */
        write_sequnlock(&jiffies_lock);
        update_wall_time(); /* ! */
    }

    update_process_times(
        user_mode(get_irq_regs())); /* ! */
    profile_tick(CPU_PROFILING);
}
```

do_timer()

```c
/* kernel/time/timekeeping.c */
void do_timer(unsigned long ticks)
{
    jiffies_64 += ticks;
    calc_global_load(ticks);
}
```