CS 477
Advanced Operating System

Lecture 04: Advanced Synchronization Primitives

Focus of today's lecture

- Classic synchronization primitives
  - Spinlock, mutexes
- Reader-writer patterns
  - rwlock vs. rwsem vs. seqlock
- Read-copy-update

Lock type: Spinlocks

- Busy-wait lock (no sleeping) → wastes CPU cycles if held long
- Disables task preemption while spinning (lock/wait)
  → Task cannot be scheduled on another core
- Safe in interrupt context
- Interrupt handling
  - If used inside interrupt handlers → must disable local interrupts
- Deadlock risk (**double acquire)**
  - If a process context holds a lock and is interrupted → interrupt handler spins on the same lock → **deadlock**
  - **Fix:** disable interrupts when acquiring the lock in process context

Spinlock issue: working with interrupts

```
/* include/linux/spinlock.h */
DEFINE_SPINLOCK(my_lock);

spin_lock(&my_lock);
/* critical region */
spin_unlock(&my_lock);
```

- **`spin_lock()/spin_unlock()`**
  - Acquire/release the lock
  - Temporarily **disable kernel preemption**
- **`spin_lock_irqsave()/spin_unlock_irqrestore()`**
  - Disable/enable interrupts in an interrupt context
- On uniprocessor systems
  - Lock operations are compiled away (no actual contention)
  - Still need **preemption control APIs** to avoid interleaving

What is the issue with this code?

```
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* access global hashtbl */
    spin_unlock(&hashtbl_lock);
}

int foo(void)
{
    /* A function running in process context */
    spin_lock(&hashtbl_lock);
    /* What happens if an interrupt occurs
    * while a task executing here? -> Deadlock */
    spin_unlock(&hashtbl_lock);
}
```

Bugfix for the deadlock case: Enable/disable interrupts

```c
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* It is okay NOT to disable interrupt here
     * because this is the only interrupt handler
     * access the shared data and this particular
     * interrupt is already disabled.
     */
    spin_unlock(&hashtbl_lock);
}

int foo(void)
{
    /* A function running in process context */
    spin_lock_irqsave(&hashtbl_lock);
    /* What happens if an interrupt occurs
    * while a task executing here? -> Deadlock */
    spin_unlock_irqrestore(&hashtbl_lock);
}
```

Lock design: Mutexes (the sleeping lock)

- Mutex = sleeping lock
  - Only one thread can hold the mutex at a time
  - Non-recursive
  - A **thread cannot exit (or be killed)** while holding the mutex
  - **Not usable in interrupt context** (since interrupts cannot sleep)
  - A thread holding a mutex **may sleep**

- Other threads waiting for the mutex
  - Placed on a **wait queue** and put to **sleep**
  - When the holder releases the mutex → **one waiter is woken up**

Spinlock vs. mutex: The tradeoff

Requirement	Recommended lock
Low overhead locking	spinlock is preferred
Short lock hold time	spinlock is preferred
Long lock hold time	mutex is preferred
Need to lock from the interrupt context	spinlock is required
Fine with sleeping when holding the lock	mutex is required

Locking golden rule

- **Rule #1:** Protect **data**, not code
- **Rule #2:** Never sleep while holding a **spinlock**
- **Rule #3:** If interrupts are involved, use **`spin_lock_irqsave()/spin_unlock_irqrestore`**
- **Rule #4:** Choose the simplest synchronization primitive that works first (**correctness first, performance second**).

Questionnaire for locking

Question	Implication	Locking needed?
Is the data global or shared?	Can be accessed from multiple contexts	Yes
Can another thread or CPU access it?	Possible concurrent access	Yes
Is data accessed from both process context and interrupt context?	Locks supporting interrupt context	spinlock
Is data shared between interrupt handlers?	Must handle concurrent interrupt execution	spinlock + interrupts disable/enable
If a process is preempted while using the data, can another process access it?	Ensure data consistency across context switches	Mutex
If the current process sleeps while holding data, what state is the data left in?	If data is shared, it will be inconsistent; need to use locks	Mutex
What if a function is called concurrently on another CPU?	Need to ensure data consistency if 2 CPUs work on the same data	Yes

Readers-writer spinlock (rwlock)

- Allows concurrency
  - **Multiple readers** can enter simultaneously
  - **Only one writer** can hold the lock (exclusive access)
- Example: list operations
  - **Write (update):** no other reader or writer allowed
  - **Read (search):** multiple readers allowed in parallel
- Benefits
  - Safe parallel reads (no data corruption)
  - Higher throughput compared to exclusive locks

Readers-writer semaphore: rwsem

- Mutex variant with readers-writer semantics
- Unlike spinlocks/rwlock → **threads may sleep while waiting**
- Useful when critical sections are long (not suitable for spinning ones)

```
#include <linux/rwsem.h>

/* declare reader-writer semaphore */
static DECLARE_RWSEM(mr_rwsem); /* or use init_rwsem(struct rw_semaphore *) */

/* attempt to acquire the semaphore for reading ... */
down_read(&mr_rwsem);
/* critical region (read only) ... */
/* release the semaphore */
up_read(&mr_rwsem);

/* ... */

/* attempt to acquire the semaphore for writing ... */
down_write(&mr_rwsem);
/* critical region (read and write) ... */
/* release the semaphore */
up_write(&mr_sem);
```

Sequence lock (seqlock)

- Purpose: simple mechanism for synchronizing readers and writers
- **Why this type of lock**
  - Readers must not block writers
    - Readers retry instead of delaying writers
  - Ideal for read-mostly workloads
    - Eg, system time, counters, CPU stats
  - Lightweight synchronization
    - Readers don’t take heavy locks, unlike the readers-writer lock

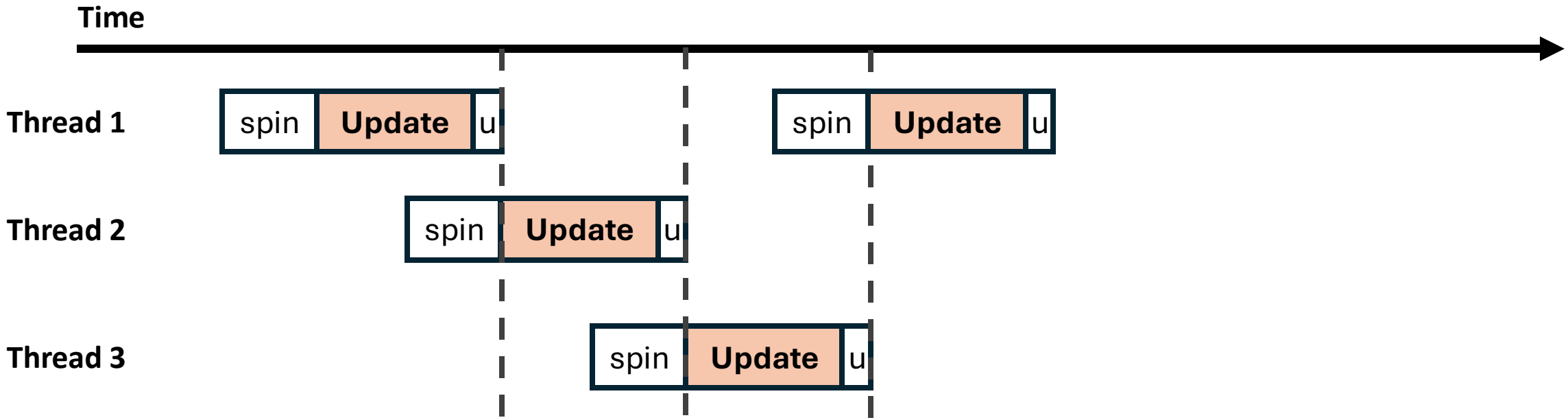
Sequence lock (seqlock)

- Writer side:
  - Acquire lock → increment sequence counter (to odd value)
  - Perform update
  - Release lock → increment sequence counter (back to an even value)
- Reader side:
  - Read sequence counter (before read)
  - Read the data
  - Read sequence counter again (after read)
  - If counters match and are even → read is valid
  - Otherwise → retry (data was modified during read)
- Cons: Need to redesign the code to use seqlock

```
/* define a seq lock */
seqlock_t my_seq_lock = DEFINE_SEQLOCK(my_seq_lock);

/* Write path */
write_seqlock(&my_seq_lock);
/* critical (write) region */
write_sequnlock(&my_seq_lock);

/* Read path */
unsigned long seq;
do {
    seq = read_seqbegin(&my_seq_lock);
    /* read data here ... */
} while(read_seqretry(&my_seq_lock, seq));
```

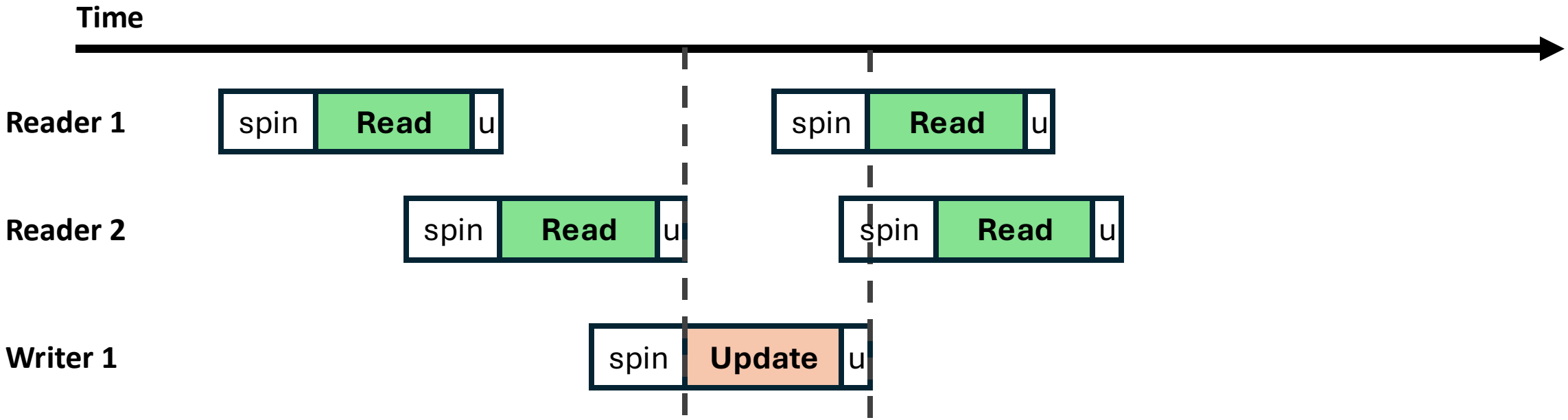
Visualizing progress of locking primitives

- Spinlock/mutex: Allow mutual exclusion, i.e., one thread at a time updates shared data



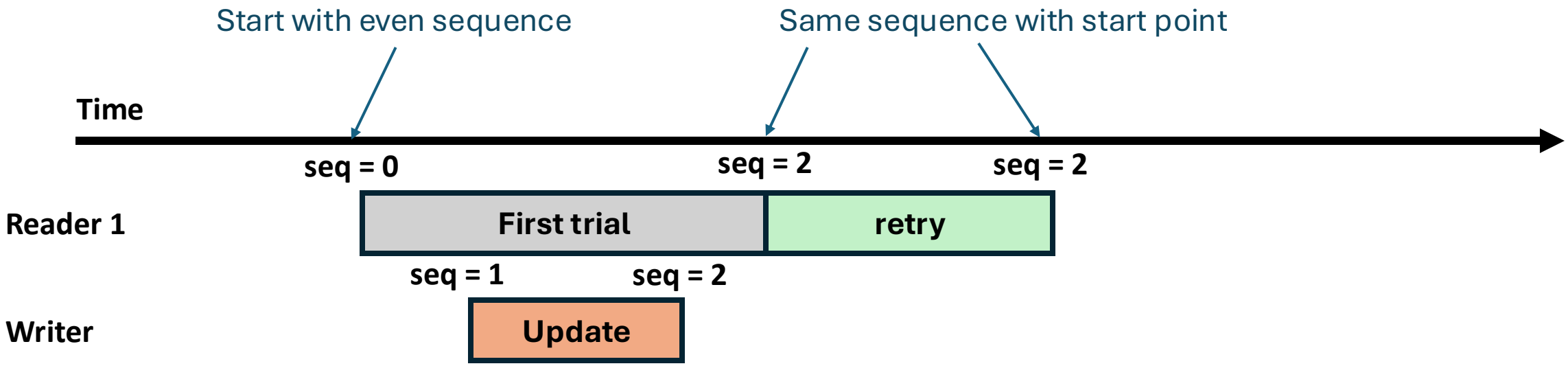
Visualizing progress of locking primitives

- Readers-writer lock: Allow multiple readers
  - Mutual exclusion between readers and writer



Visualizing progress of locking primitives

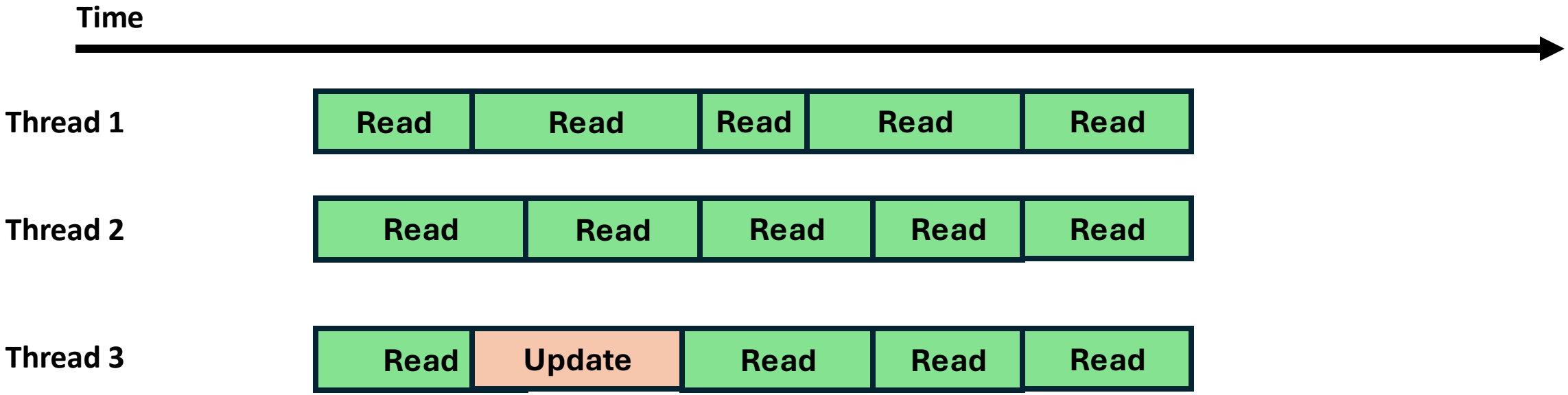
- Seqlock: Reading consistent data without starving writers
- **A writer blocks readers**



Q. Can we design a synchronization primitive that does not block readers?

Read-copy-update (RCU)

- Concurrency between multiple readers and a single writer
  - **A writer does not block readers**
  - Allow multiple readers with **almost zero overhead**
  - Optimized for **read performance**



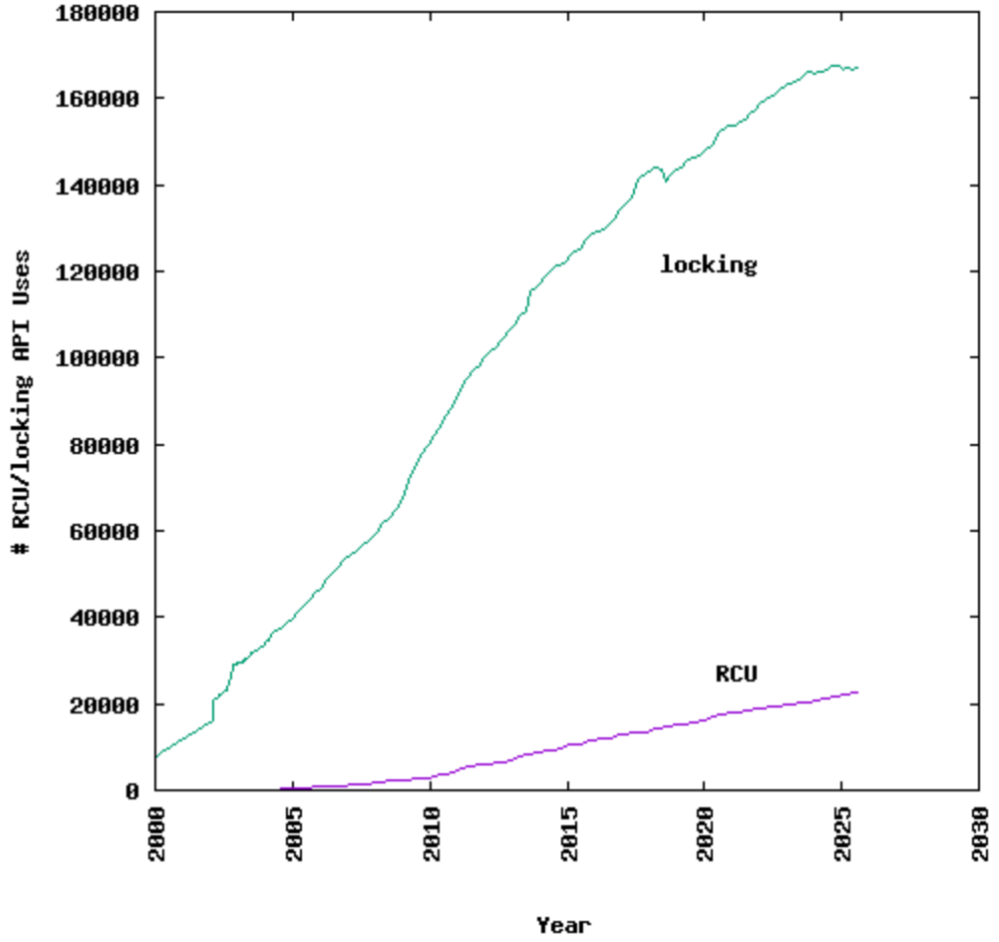
Read-copy-update (RCU)

- Readers run without locks
  - Always see **a consistent snapshot** of data
- Writers update carefully
  - Make a **new copy** of the data
  - Publish it **atomically**
  - Old version is kept **until all readers finish**
- Guarantees
  - Readers never block writers
  - Writers **never invalidate** data still in use
- Best suited for **read-mostly data**
  - Eg, directory entry cache, DNS name database

Who developed RCU?

- Paul McKenney @ Meta

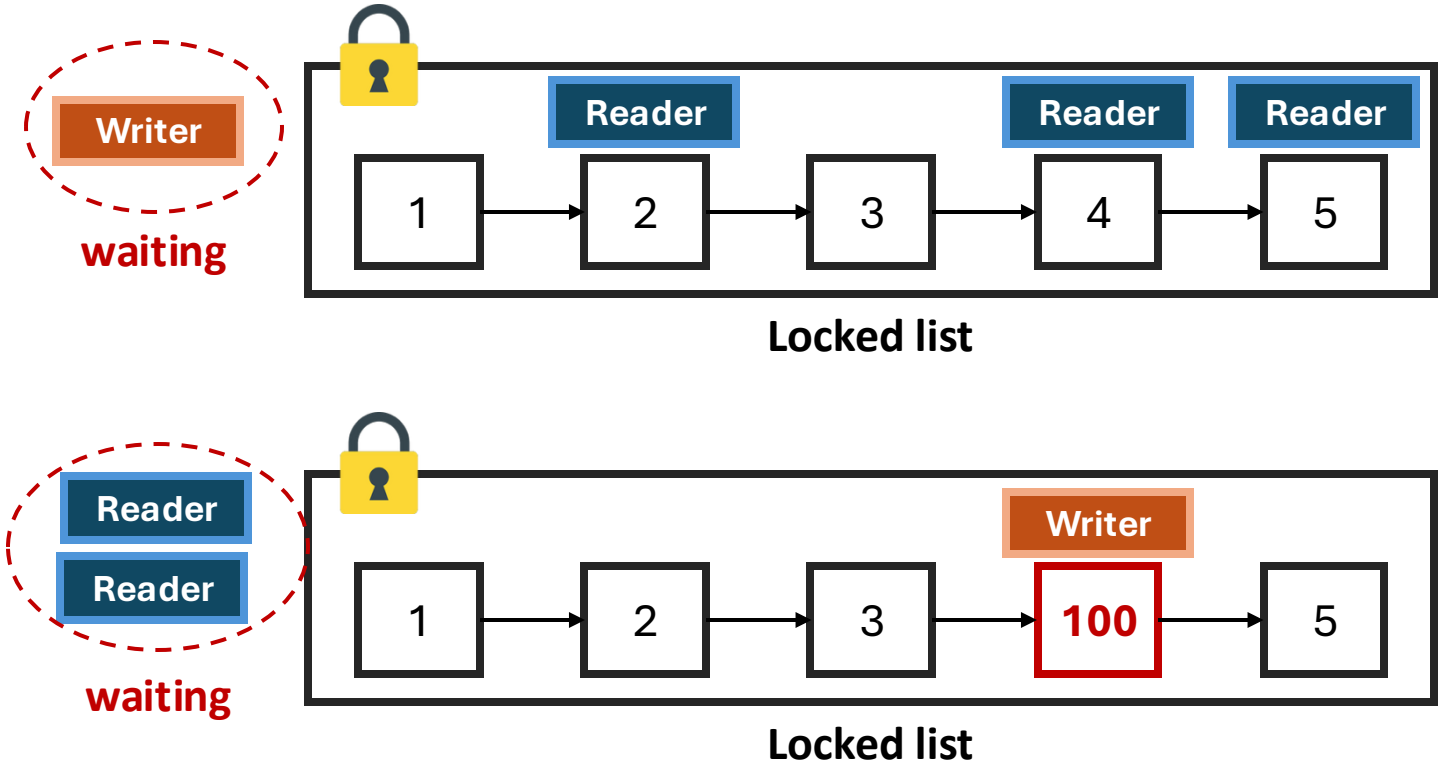
RCU usage in the kernel



- Source: RCU Linux usage

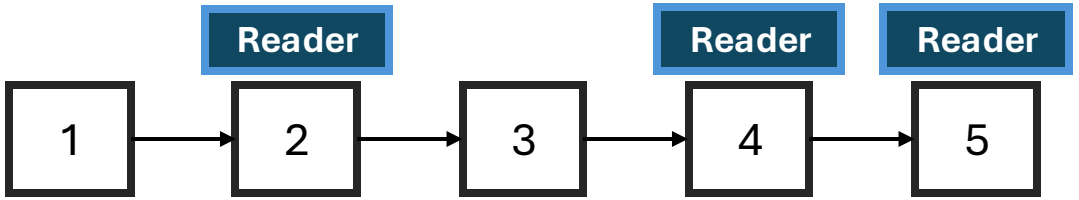
RWLock-based linked list

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



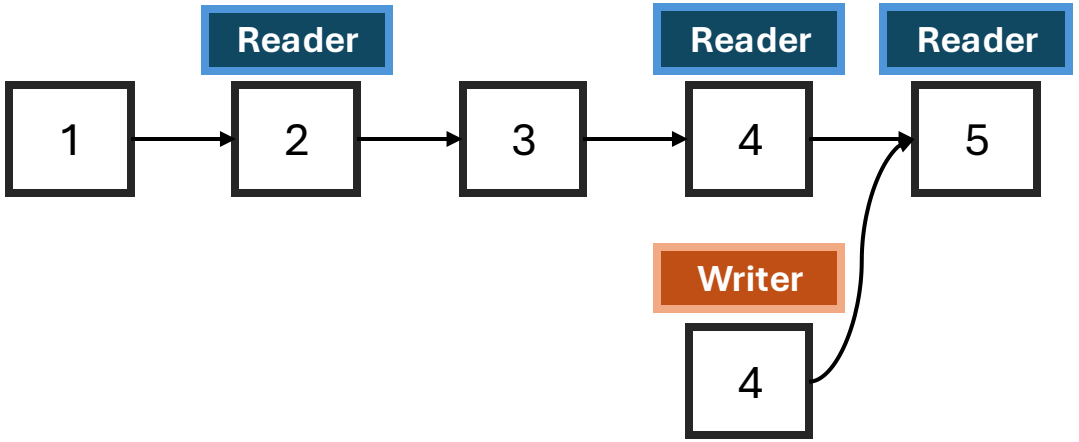
RCU-based linked list

- Allow concurrent access of readers



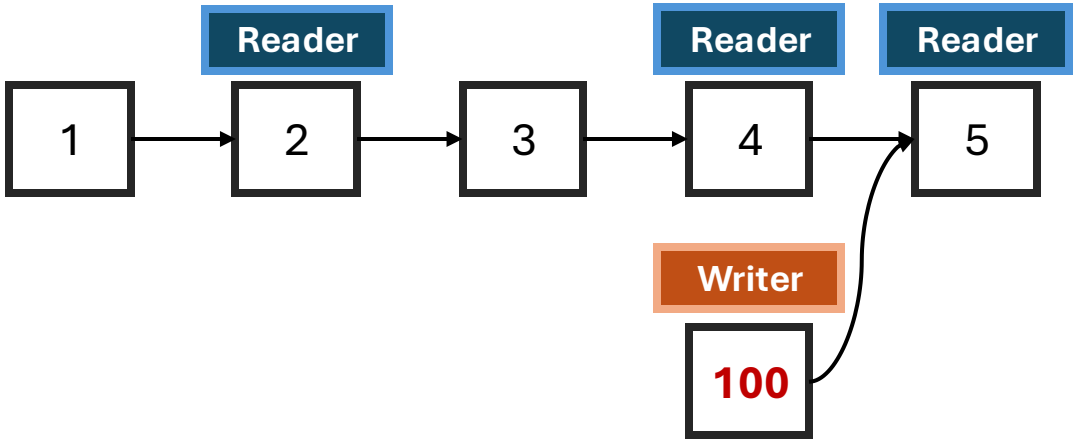
RCU-based linked list

- A writer copies an element first



RCU-based linked list

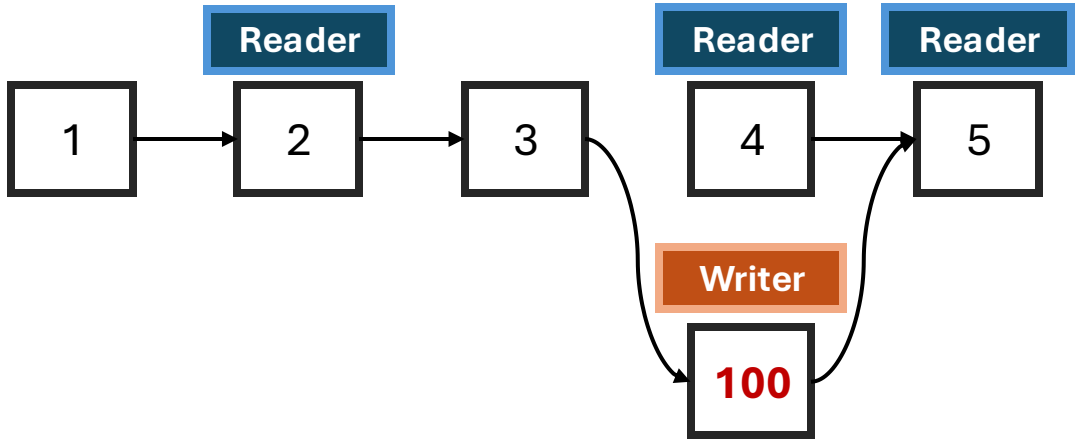
- And then it updates the element



RCU-based linked list

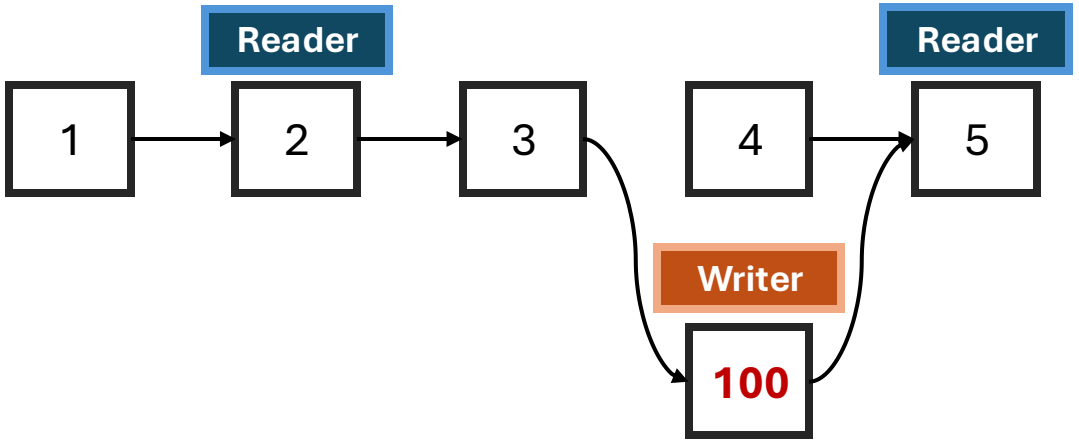
- Makes its changes public by updating the next pointer of its previous

→ New readers will traverse 100 instead of 4



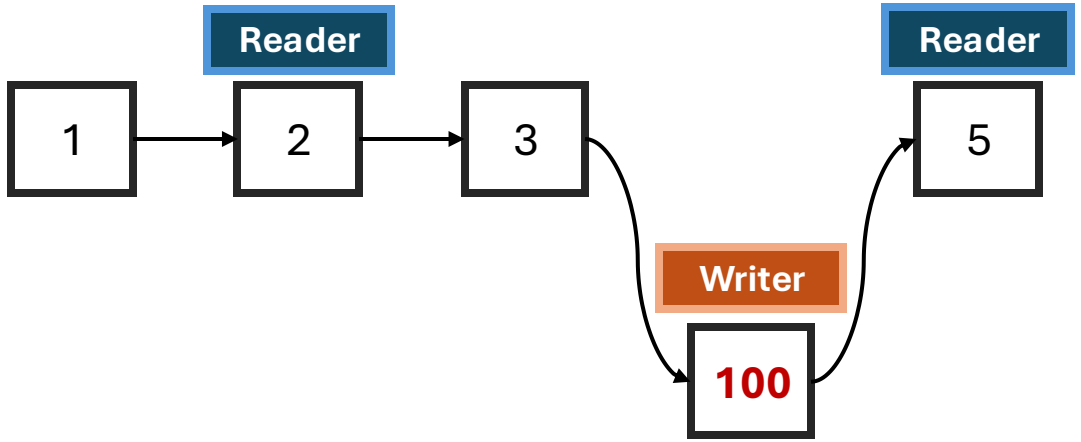
RCU-based linked list

- Do not free the old node, 4, until any reader accesses 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 */
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;
}
```

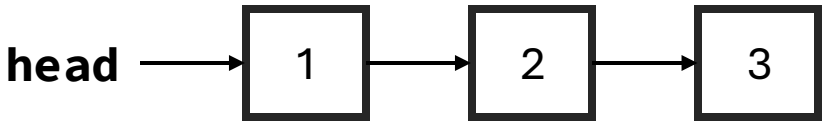
Replace rwlock by RCU

```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

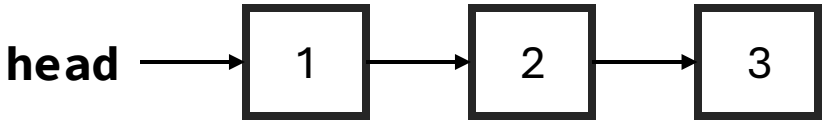



```
length() {
  rcu_read_lock(); {
    p = rcu_derference(head); // p = head
    for (i = 0; p; p = p->next; i++);
  } rcu_read_unlock();
  return i;
}
```

```
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



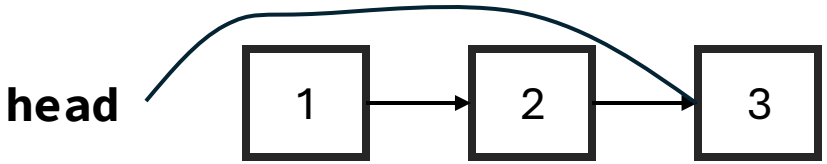

```
length() {
  rcu_read_lock(); {
    p = rcu_derference(head); // p = head
    for (i = 0; p; p = p->next; i++);
  } rcu_read_unlock();
  return i;
}
```

```
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_derference(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**

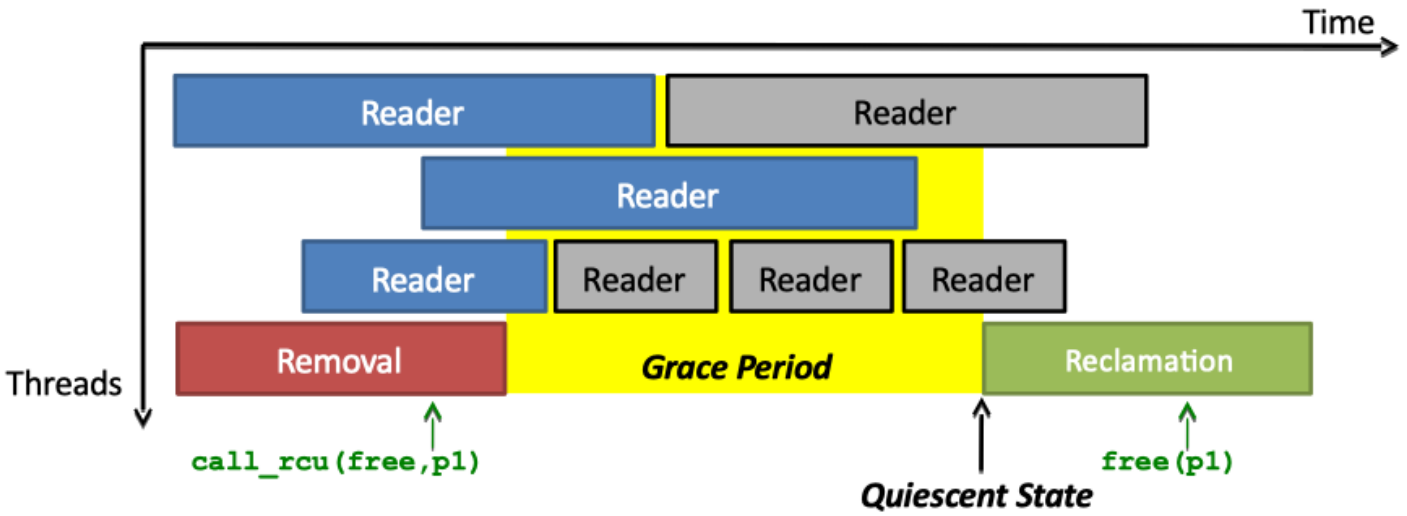

```
length() {
  rcu_read_lock(); {
    p = rcu_derference(head); // p = head
    for (i = 0; p; p = p->next; i++);
  } rcu_read_unlock();
  return i;
}
```

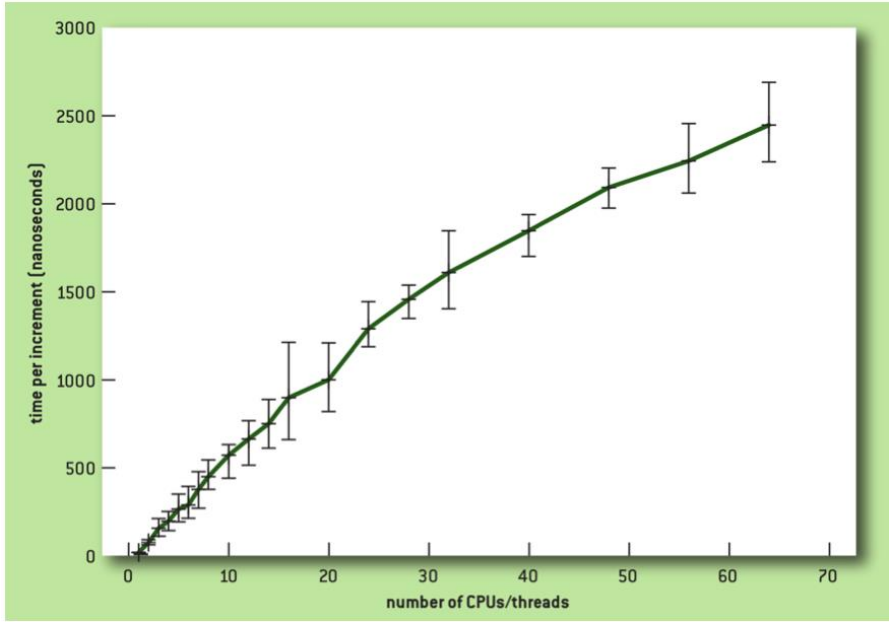
```
pop_n(n) {
  for (p=head; p&&n; p=p->next; n--)
    call_rcu(free, p);
  rcu_assign_pointer(head, p); // head = p
}
```

- 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 CPUs to schedule)

Delayed free





Atomic increment cost

- **Grace period** (time interval in which all pre-existing readers complete) **+ quiescent state** (each CPU is out of the RCU read-side CS)
- Efficient and scalable grace period detection is a key challenge
  - Some obvious solutions, like reference counting, never work

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) ..
```

Limitations of RCU

- Do not provide mechanism to coordinate multiple writers
  - Most RCU-based algorithms use spinlock to prevent concurrent write operations

- All modifications should be **a single pointer update**
  - This is challenging!

Synchronization primitives comparison

Primitive	Sleep/spin?	Readers allowed?	Writers blocked by readers?	Can be used in interrupts?	Typical use case
Spinlock	busy-wait (spin)	No	Yes (writer waits until lock is free)	✅ Yes	Short critical sections, IRQ handlers
mutex	sleep	No	Yes (writer waits if lock held)	❌ No	Long critical section in process context
rwlock	spin	✅ multiple readers	✅ Writer waits until all readers exit	✅ Yes	Reader-heavy workloads, but risk of writer starvation
rwsem	sleep	✅ multiple readers	✅ Writer waits until all readers exit	❌ No	Long reader-heavy sections in process context
seqlock	spin (retry-based)	✅ multiple readers, but must retry	❌ Writer never waits for readers	✅ Yes	Read-mostly data (e.g., system time)
RCU	lock-free reads	✅ multiple readers	❌ Writers never **block** readers (copy+publish)	✅ (read-side); writers in process context	Read-mostly data (e.g., dcache, routing table)

Further readings

- Read-log-update: a lightweight synchronization mechanism for concurrent programming, SOSP15
- Is Parallel Programming Hard, And, If So, What Can You Do About It?
- Structured Deferral: Synchronization via Procrastination
- Introduction to RCU Concepts
- LWN: What is RCU, Fundamentally?