CS 134
Operating Systems

April 17, 2019

Read-Copy-Update

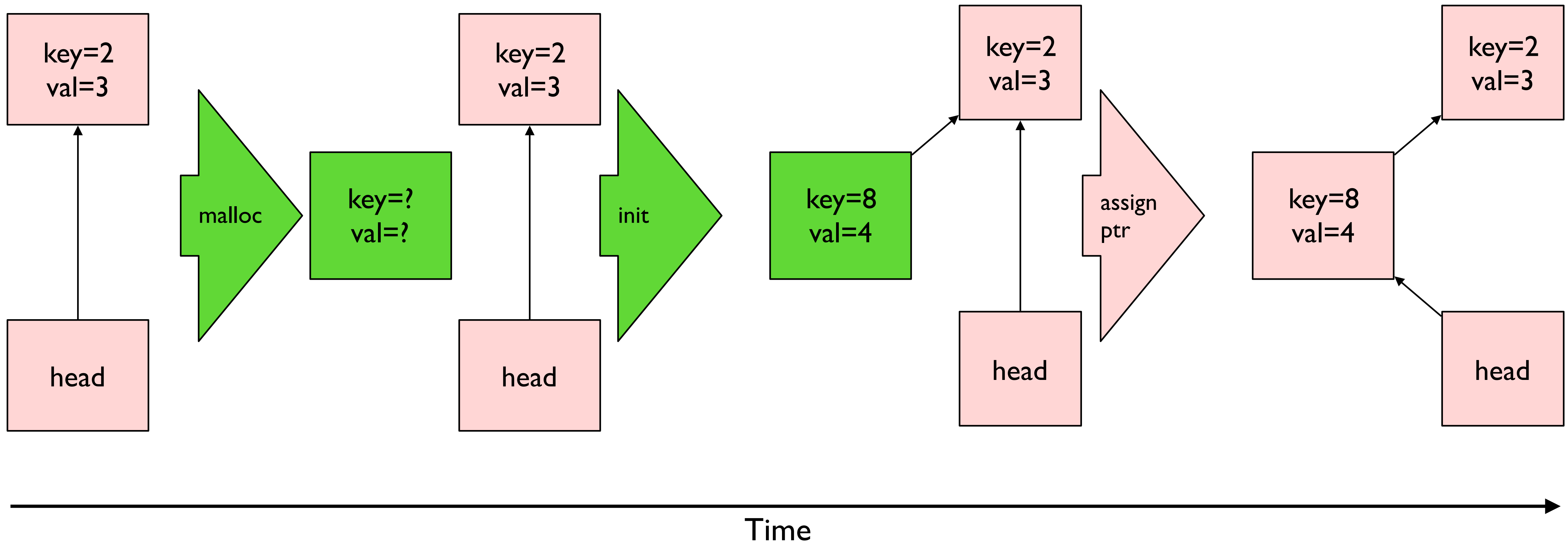
Outline

- Motivation
- What is RCU?
- How used in Linux?
- Summary
- Questions

Motivation

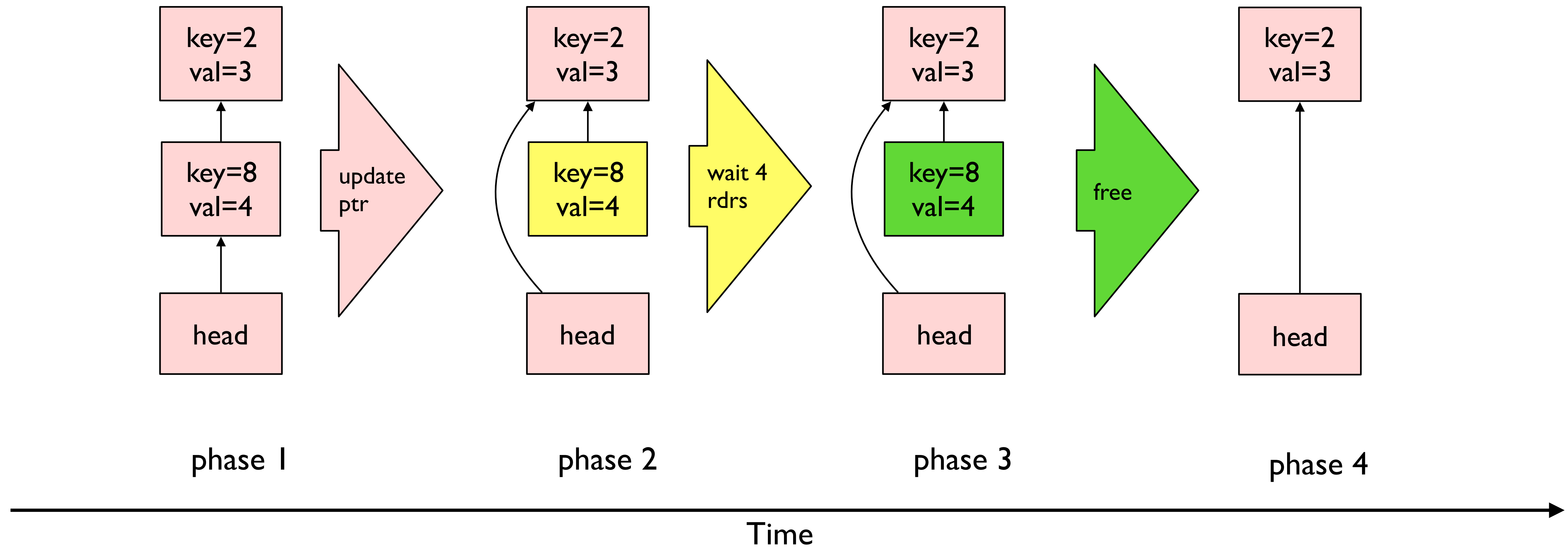
- Remember back to HW 6: threads: `put` and `get` in a hash table.
- Hash table with a lock for each bucket
- Used the lock for writing to the linked list
- No lock for reading from the linked list

Adding to a linked list with concurrent readers



Readers can read at any time
They'll see a list with either:
- one item in it, or
- two items in it

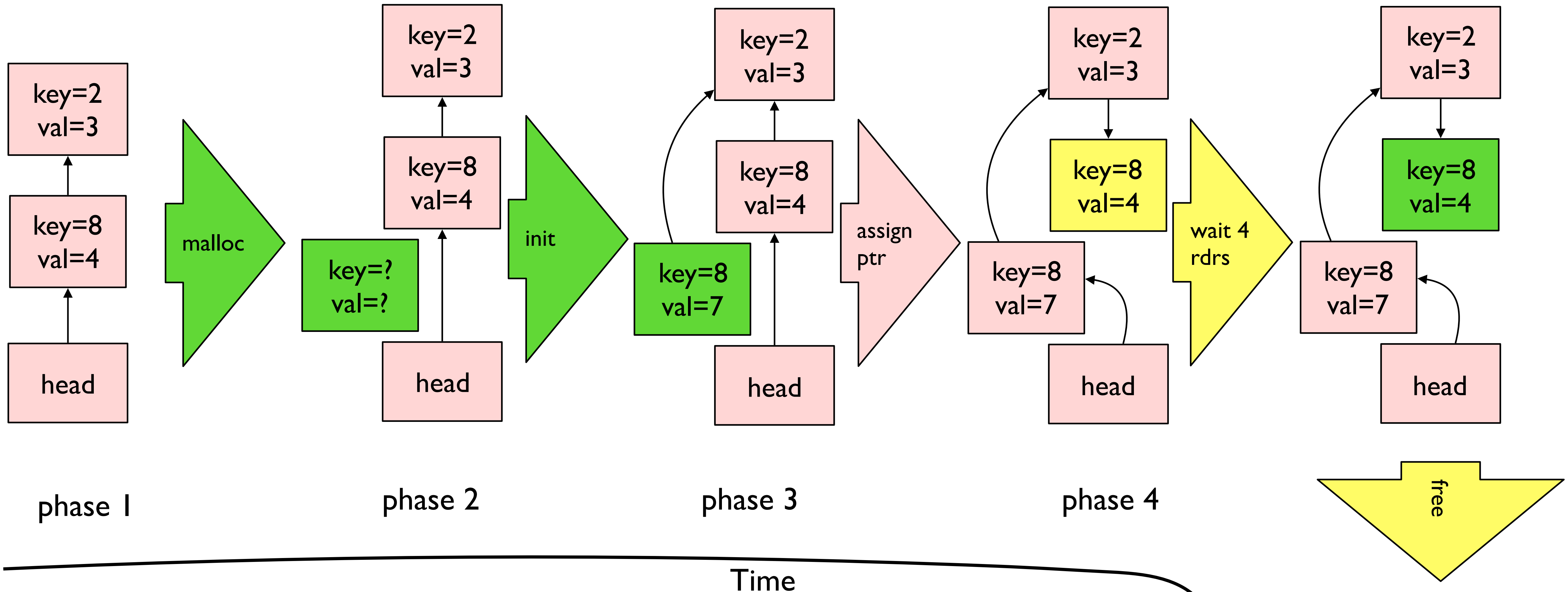
Deleting from a linked list with concurrent readers



Readers can read at any time.
During:

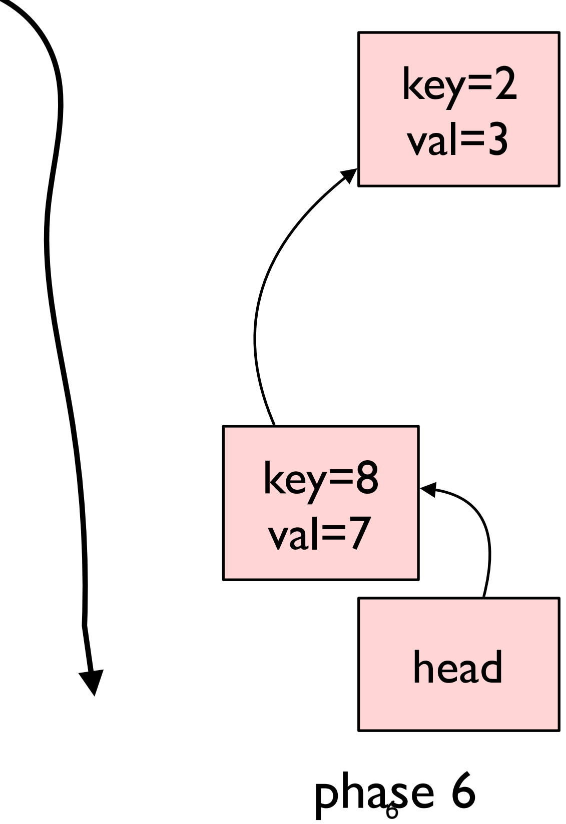
- phase 1, they see 2 items
- phase 2:
  - new readers see 1 item
  - old readers *may* see 2 items
- phase 3: they see 1 item (old readers are gone)
- phase 4: they see 1 item

Modifying an item in linked list with concurrent readers



During a phase, readers see:

- phase 1, 2, and 3: they see val=4
- phase 4:
  - new readers see val=7
  - old readers *may* see val=4
- phase 5: they see val=7 (old readers are gone)
- phase 6: they see val=7



What is RCU?

- A philosophy for updating data structures:
  - **R**eaders
  - **U**pdaters who make **C**opies, maintaining both old and new data structures until old is no longer needed
- An API:
  - For readers:
    - `rcu_read_lock()`
    - `rcu_read_unlock()`
    - `rcu_dereference()`
  - For updaters:
    - `synchronize_rcu()`
    - `rcu_assign_pointer()`

How does RCU work for readers?

- Requirements:
  - `rcu_read_lock`/`rcu_read_unlock` specify a *read-side critical section*
  - Reader not allowed to block during a read-side critical section
  - Reader has access to RCU-protected data structure only during its critical section
  - `rcu_dereference` used to dereference a pointer. Pointed-to-object exists throughout the critical section

```
rcu_read_lock();
p = rcu_dereference(pointer_to_data_structure);
do_something_with(p);
rcu_read_unlock();
```

How does RCU work for updaters?

- Overview
  - Make changes not seen by readers at will
  - Use `rcu_assign_pointer` to atomically change a pointer (readers can now see)
  - Call `synchronize_rcu` to wait for all existing readers to leave their critical section
    - New readers may go enter critical section; no wait
  - Clean up. At this point, any objects only referenced from old pointer value aren't accessible by readers

```
p = (Node *) malloc(sizeof(Node));
old = head;
p->val = 3; p->key = 8; p->next = old->next;
rcu_assign_pointer(head, p)
synchronize_cpus(); // grace period for existing readers
free(old)
```

Goals of RCU

- Allow concurrent reads
  - Concurrent with other readers
  - Concurrent with updaters, too
  - Low computation and space overhead
  - Deterministic completion times for reads

Nothing is faster than nothing

- Let’s say we’re using a non-preemptive kernel (like JOS, old Linux (2.4))

```
#define rcu_read_lock()

#define rcu_read_unlock()

#define rcu_dereference(p) ({ \
   typeof(p) ______p1 = (*(volatile typeof(p)*) &p);\
   read_barrier_depends(); // defined by arch \
   ______p1; // “returns” this value
 })
```

But how does `synchronize_rcu` work?

- Reader not allowed to block during a read-side critical section
- Reader has access to RCU-protected data structure only during its critical section
- So, if a reader yields, then it must not be in the critical section

- Simple idea: if every other CPU has called scheduler since `synchronize_cpu` was issued, then, all old readers must be done

But how does `synchronize_rcu` work?

- Have each CPU keep a count of how many times scheduler has been called

- Have synchronize_rcu read those counts when it starts, it returns when each count becomes larger
  - Many tricks to make this quicker, and to amortize multiple read-side critical sections

What if you have a preemptive kernel

- Postpone preemption while in read-side critical section

- In Linux, if kernel thread `preempt_count` is non-zero, thread can't be preempted (used for spin-locks and RCU)

```
#define rcu_read_lock() current_thread_info()->preempt_count++

#define rcu_read_unlock() current_thread_info()->preempt_count--
```

What if you have multiple writers?

- You'll still need a write lock to prevent multiple simultaneous writers

Limitations of RCU

- Data structures requiring an update that can't be captured in a single atomic pointer update
  - E.g., doubly-linked lists
  - Although Linux still allows, but doesn't allow RCU-readers to traverse backwards
- Special mechanisms necessary if stale data isn't OK
- Best if ratio of readers to writers is very high

Use of RCU in Linux

- Introduced to Linux in 2002 (by Paul McKenney)
  - He references this work in his 2004 Ph.D. thesis, all about RCU
- Now has >10K uses in the kernel
  - Especially in file system and networking
  - Must synchronize millions of kernel objects (direntries, for example)

Summary

- Understand intuition of RCU
- Understand how to insert/delete/modify a list node in RCU
- Pros/cons of RCU

Questions

- If a single integer overhead for a read/write lock is sometimes unacceptable, does that means RCUs have no storage overhead?
  - Yes
- Confused about examples of type-safe memory
- "Without proper care, a reader accessing a data item an updater concurrently initialized and inserted could observe that item's pre-initialized state". Why can't it prevent this by giving the updater a lock while it's updating?

```
head=NULL
…
p->a = 6
head = p
```

thread 1

```
if (head)
  head->a not necessarily 6!
```

thread 2

Questions

- Why on Linux, synchronize_cpu uses context switches rather than scheduling a thread on each CPU?
  - “The Linux RCU implementation essentially batches reader-to-writer communication by waiting for context switches. When possible, writers can use an asynchronous version of synchronize_rcu, call_rcu, that will asynchronously invokes a specified callback after all CPUs have passed through at least one context switch.”
- Deterministic completion time for read operations? When could it be non-deterministic?
  - Waiting on a spin-lock, for example