

# Advanced Operating Systems (CS 202)

## Read Copy Update (RCU)

*(some slides from Dan Porter)*

# Linux Synch. Primitives

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction	All
Memory barrier	Avoid instruction re-ordering	Local CPU
Spin lock	Lock with busy wait	All
Semaphore	Lock with blocking wait (sleep)	All
Seqlocks	Lock based on access counter	All
Local interrupt disabling	Forbid interrupt on a single CPU	Local
Local softirq disabling	Forbid deferrable function on a single CPU	Local
Read-copy-update (RCU)	Lock-free access to shared data through pointers	All

Also Read-write locks

# Why are we reading this paper?

- Example of a synchronization primitive that is:
  - Lock free (mostly/for reads)
  - Tuned to a common access pattern
  - Making the common case fast
- What is this common pattern?
  - A lot of reads
  - Writes are rare
    - Prioritize writes
  - Stale copies are short lived – time heals all wounds
  - Ok to read a slightly stale copy
    - But that can be fixed too

# Readers/Writers (review)

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}
```

```
reader {
    wait(mutex);    // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex); // unlock readcount
    Read;
    wait(mutex);    // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex); // unlock readcount
}
```

# Lock free data structures

- Do not require locks
- Good if contention is rare
- But difficult to create and error prone
- RCU is a mixture
  - Concurrent changes to pointers a challenge for lock-free
  - RCU serializes writers using locks
  - Win if most of our accesses are reads

# Example of lock free synchronization

```
int retry = 0;
while (true)
{
    // DO NOT worry about writer access for now - this is for demonstrating atomic operation only

    int prev_readers = _readers;           // current count
    int new_readers = prev_reader + 1;     // new count - note this is using the local value prev_readers
                                           // in case _readers has changed in between

    if (_readers.compare_exchange_weak(prev_readers, new_readers))
    {
        // we've won the race
        break;
    }

    // we've failed, retry
    retry++;
    if (retry > RETRY_COUNT)
    {
        retry = 0;
        std::this_thread::yield();
    }
}
```

- Compare `_readers` with `prev_readers`
- if equal swap with `new_readers`
- Return 1 if swap successful, 0 if not
- All Atomic

# Concurrent access of linked list (without synchronization)

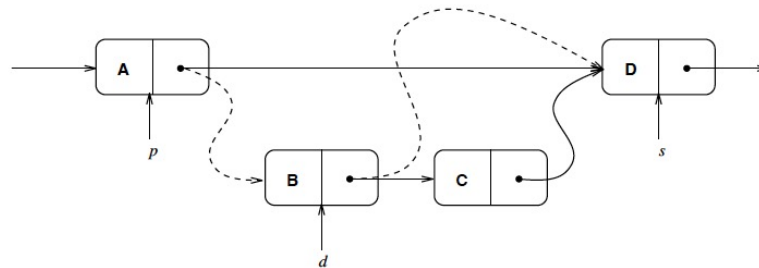


Figure 2: Deletion of B concurrent with insertion of C.

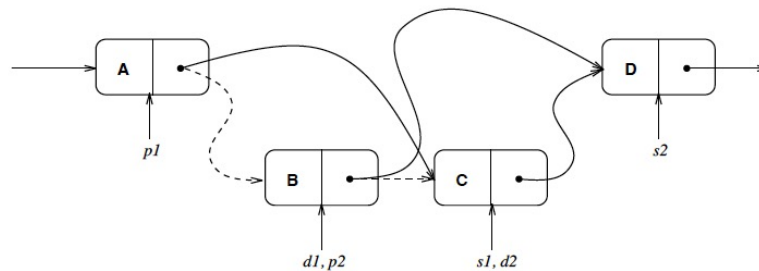


Figure 3: Concurrent deletion of B and C; second is undone.

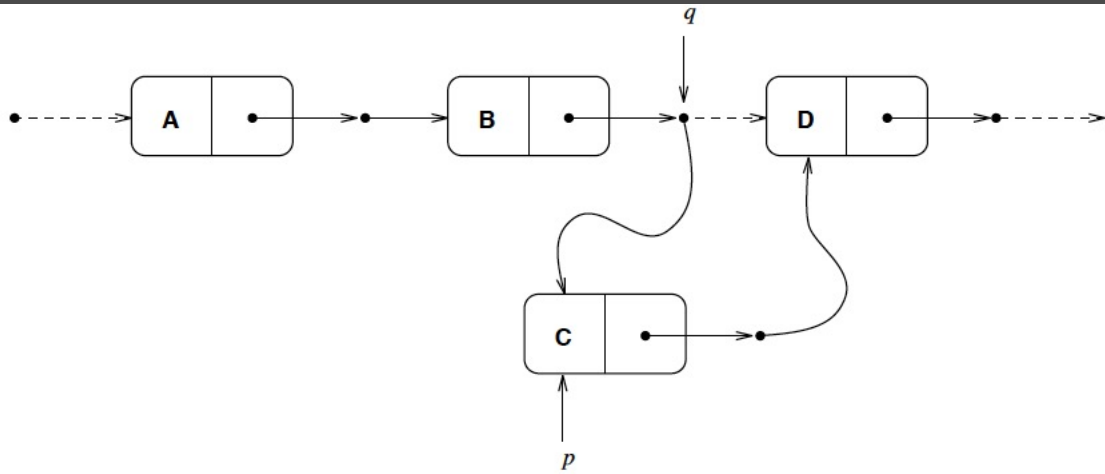


Figure 8: Inserting a new cell and auxiliary node.

---

**TRYINSERT**( $c$  : cursor,  $q$  : cell<sup>^</sup>,  $a$  : aux. node<sup>^</sup>)  
 returns boolean

- 1     **WRITE**( $q^{\wedge}.next, a$ )
  - 2     **WRITE**( $a^{\wedge}.next, c^{\wedge}.target$ )
  - 3      $r \leftarrow$  **CSW**( $c^{\wedge}.pre\_aux, c^{\wedge}.target, q$ )
  - 4     **return**  $r$
-



# Traditional OS locking designs

- poor concurrency
  - Especially if mostly reads
- Fail to take advantage of event-driven nature of operating systems
- Locks have acquire and release cost
  - Use atomic operations which are expensive
  - Can dominate cost for short critical regions
  - Locks become the bottleneck

# Why RCU?

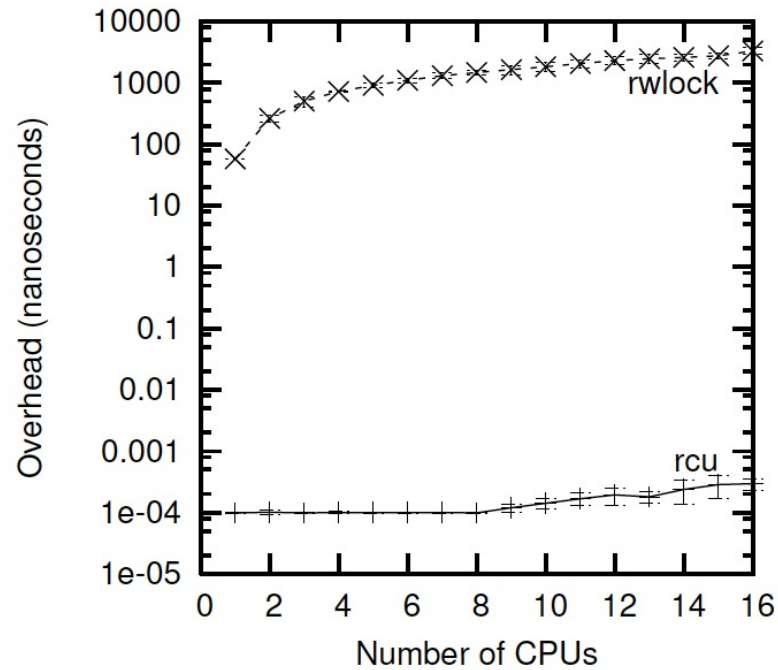
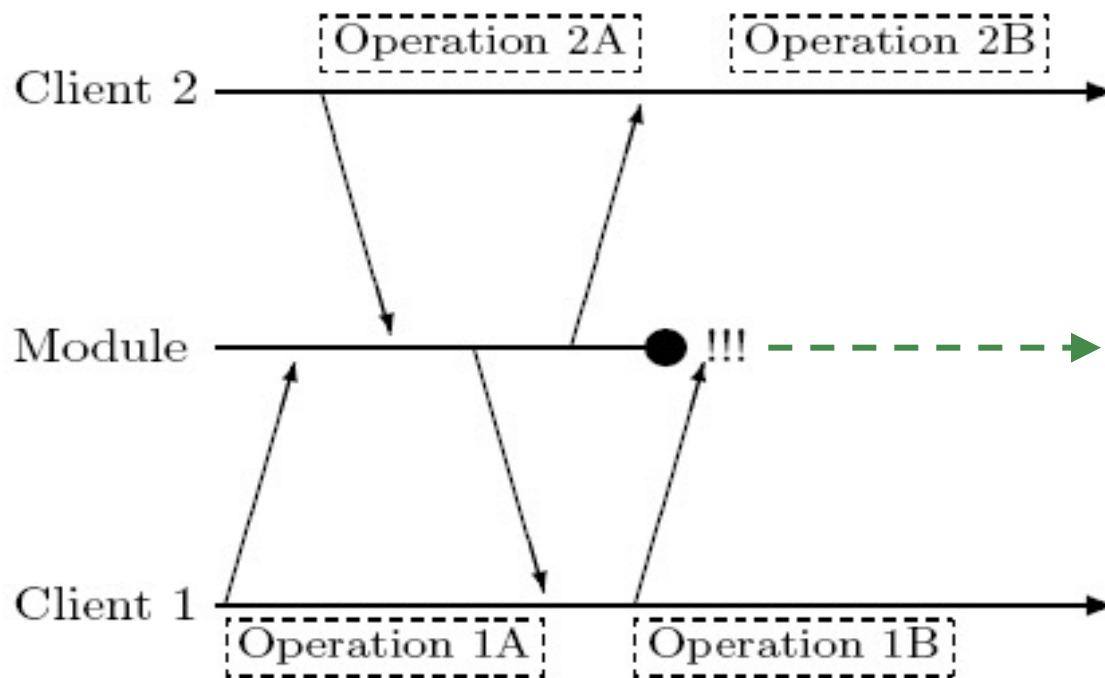


Figure 8: The overhead of entering and completing an RCU critical section, and acquiring and releasing a read-write lock.

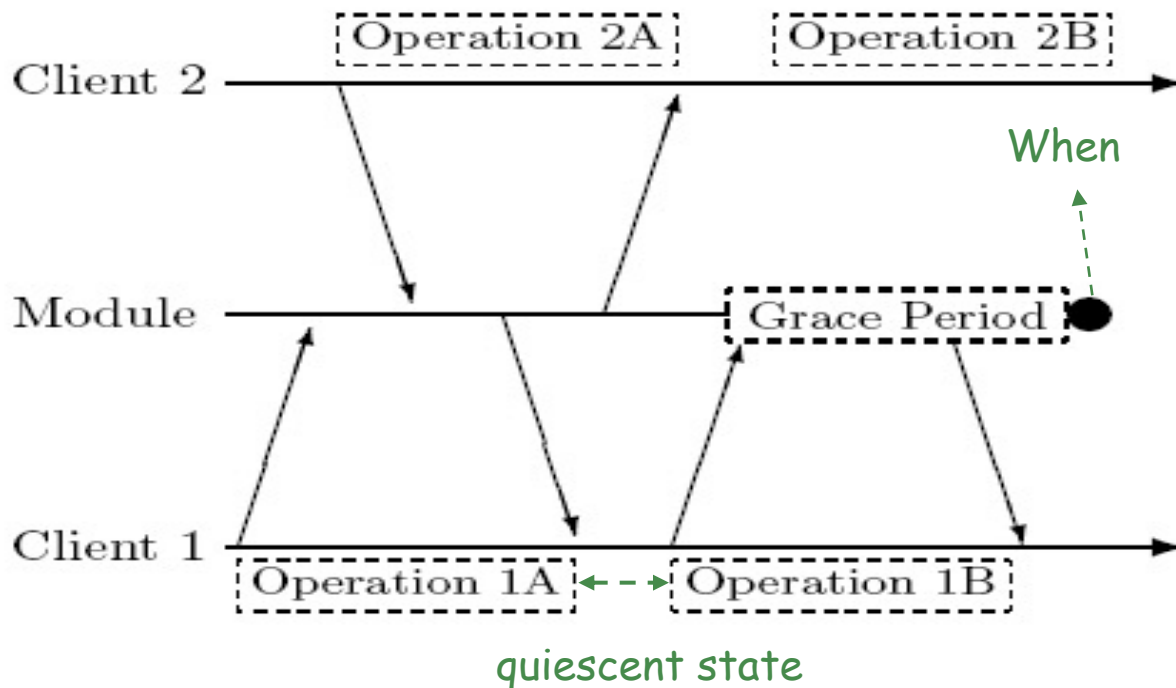
# Race Between Teardown and Use of Service



Can fix with locking, but we have to use the lock every operation

Figure 1: Race Between Teardown and Use of Service

# Read-Copy Update Handling Race



Cannot be context switched inside RCU

Figure 2: Read-Copy Update Handling Race

# Read-copy update works when

- divide an update into two phases
- proceed on stale data for common-case operations (e.g. continuing to handle operations by a module being unloaded)
- destructive updates are very infrequent.
- Often used to update linked lists
  - Which are used all over kernels

# Typical RCU update sequence

- Replace pointers to a data structure with pointers to a new version
  - Is this replacement atomic?
- Wait for all previous reader to complete their RCU read-side critical sections.
- at this point, there cannot be any readers who hold reference to the old data structure, so it now may safely be reclaimed.

# Delete implementation

```
1 void delete(struct el *p)
2 {
3     spin_lock(&list_lock);
4     p->next->prev = p->prev;
5     p->prev->next = p->next;
6     spin_unlock(&list_lock);
7     call_rcu(&p->my_rcu_head,
8     my_free, p);
9 }
```

Phase 1



Schedule phase 2  
after quiescence

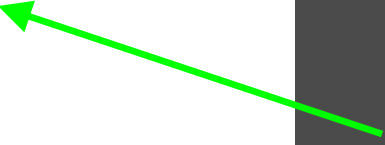


Figure 4: Read-Copy Dequeue From Doubly-Linked List

# Read-Copy Deletion (delete B)

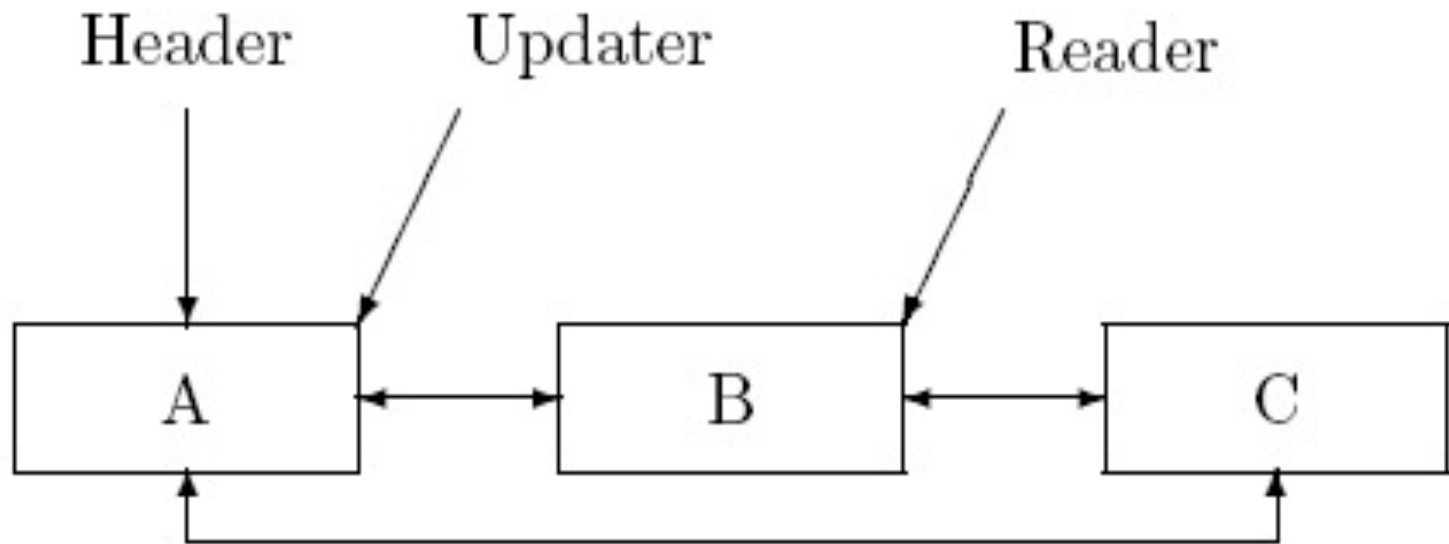


Figure 11: List Initial State



# the first phase of the update

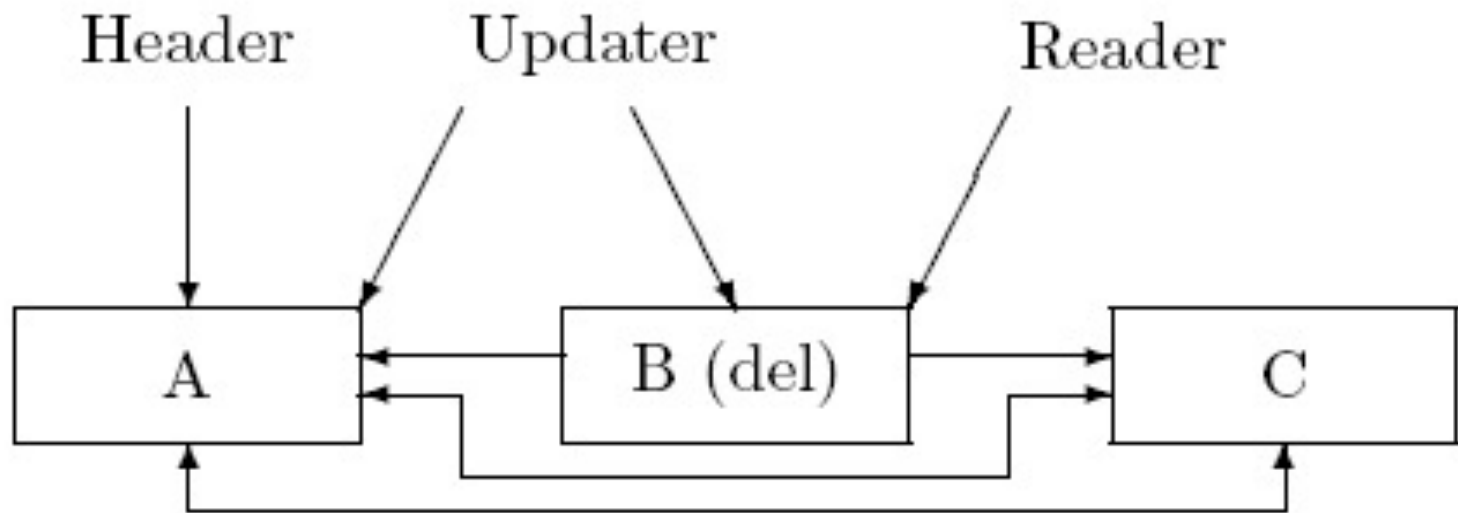


Figure 12: Element B Unlinked From List

# Read-Copy Deletion

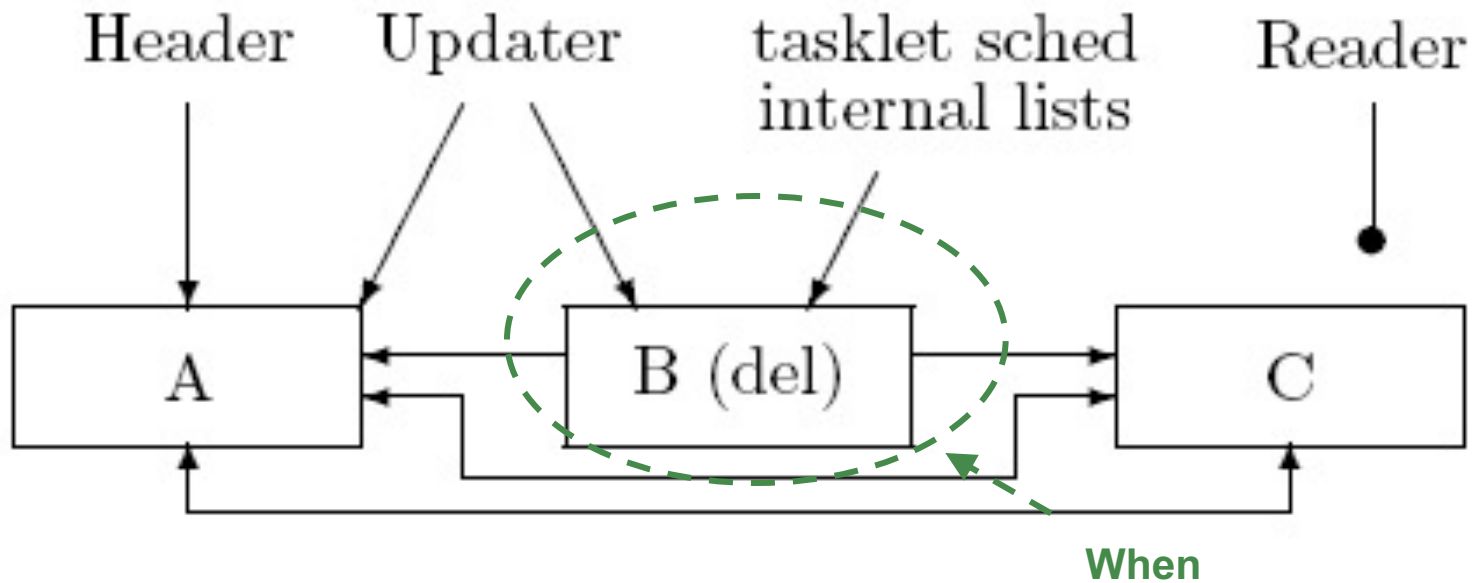


Figure 13: List After Grace Period

# Read-Copy Deletion

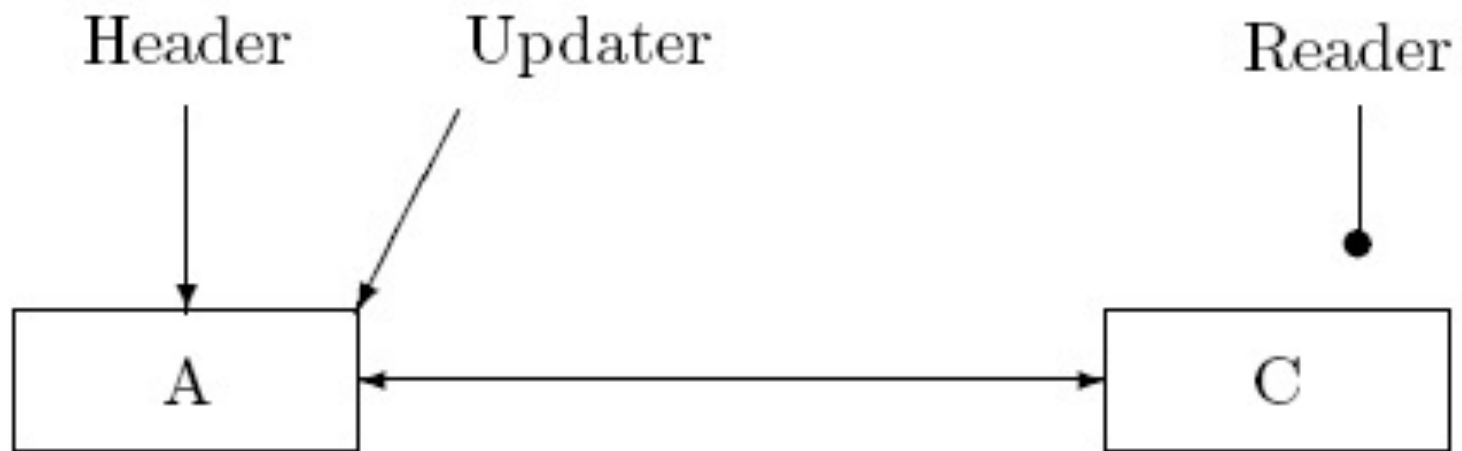


Figure 14: List After Element B Returned to Freelist

# Applied to linux route cache update

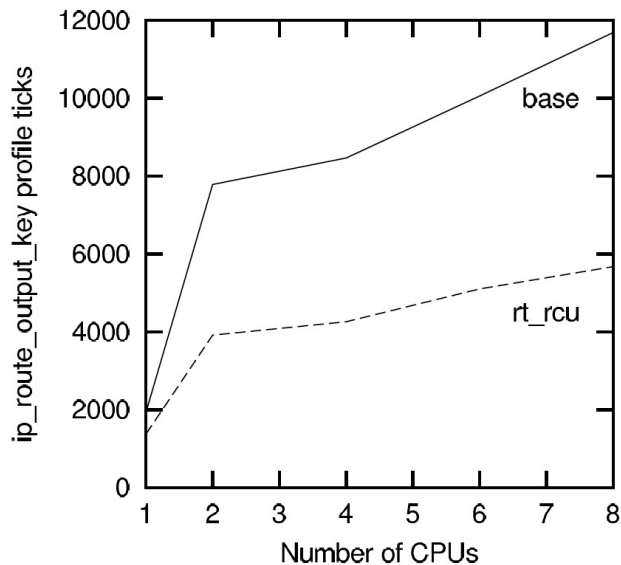


Figure 6: IP Route Cache Speedup Using rcu

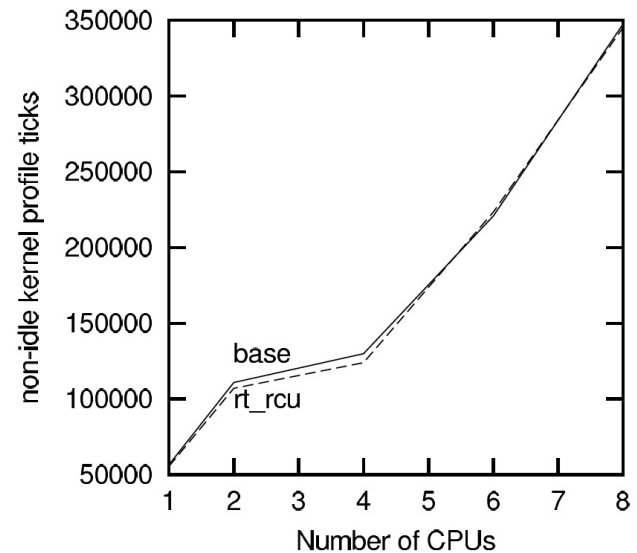


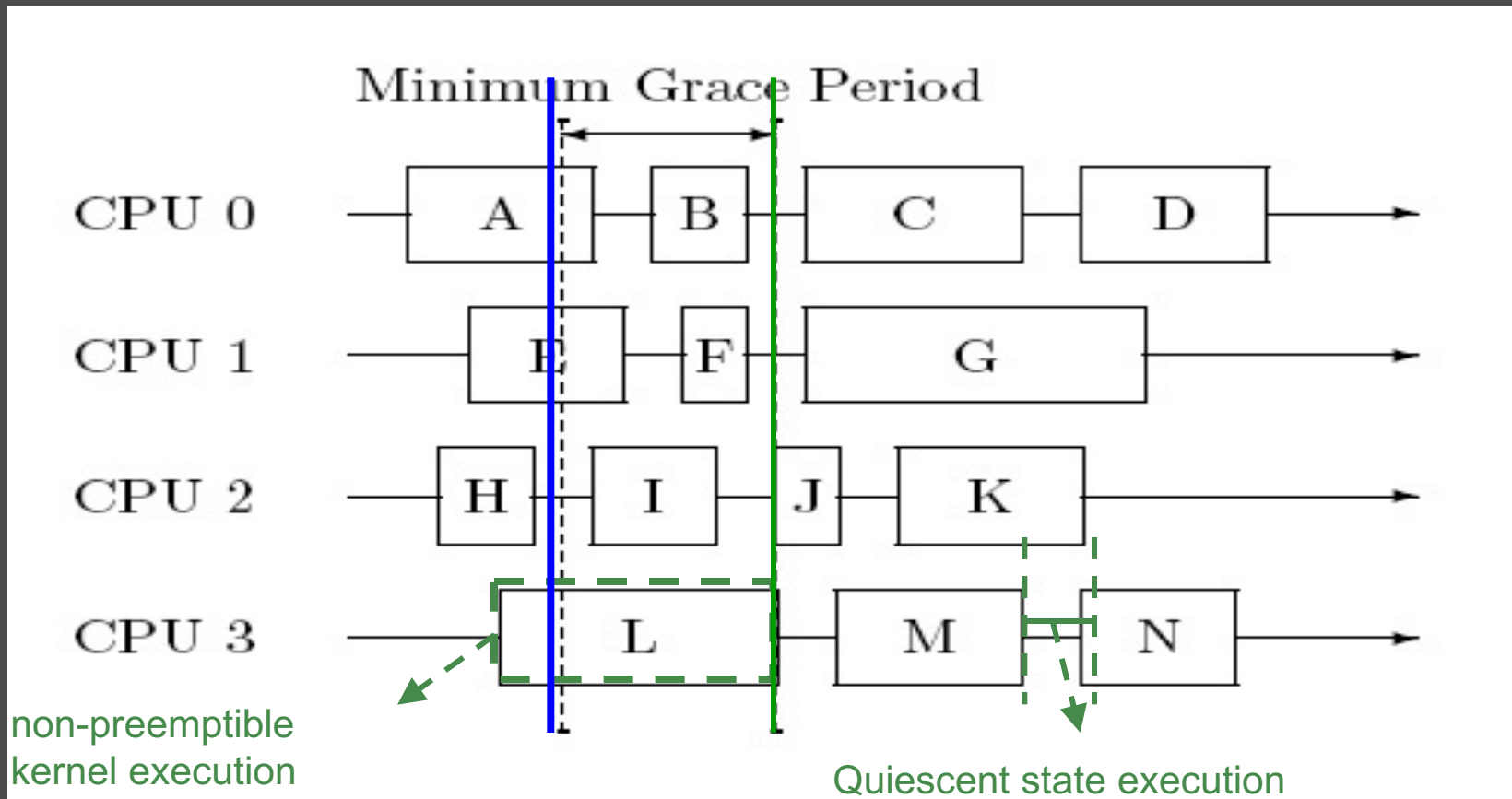
Figure 7: IP Route Cache System Performance Using rcu

Specific workload; overall speedup not that high; up to 30% speedup for other kernel functions

# How to detect quiescence?

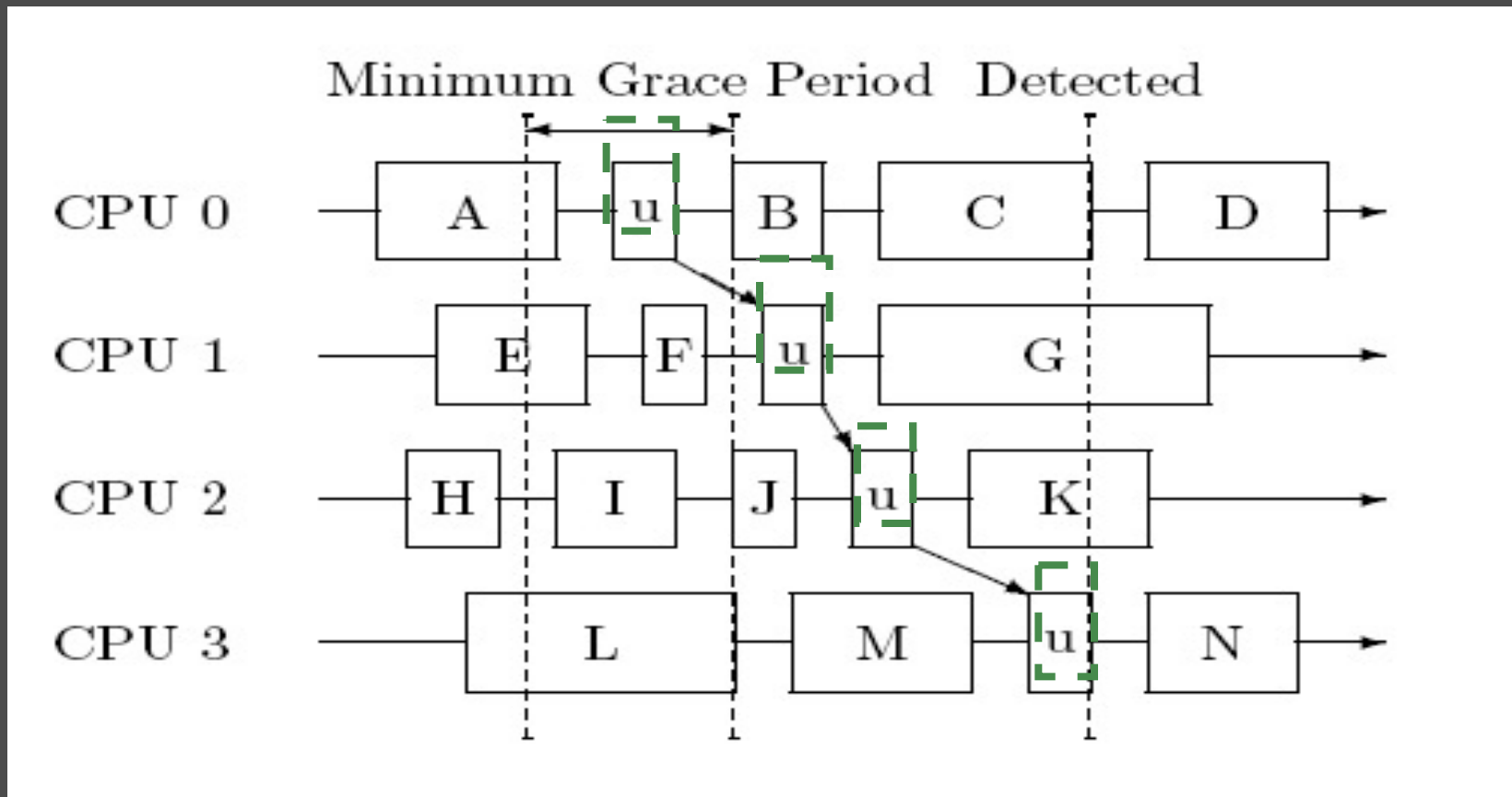
- Idea of grace periods
  - Readers of old information will eventually leave
  - Exploit context switches
    - Threads do not hold OS locks across context switches
- How do we identify?
  - Paper goes into many alternatives and evaluates them (polling; counters; ...)
  - Batching to reduce cost
  - Force context switch?
    - Expensive and some tasks are not preemptible

# Read-Copy Update Grace Period



Is it important to detect grace period quickly?

# Simple Grace-Period Detection



Schedule a thread on each CPU to ensure quiescence

# Implementations of Quiescent State

1. simply execute onto each CPU in turn.
2. use context switch, execution in the idle loop, execution in user mode, system call entry, trap from user mode as the quiescent states.
3. voluntary context switch as the sole quiescent state
4. tracks beginnings and ends of operations



# Implementation (polling/counter)

- Poll to figure out when safe to delete
- Generation counter for each RCU region
  - Generation updated on write
- Track readers of each generation
  - Every read increments generation counter going in
    - And decrements it going out
  - Quiescence = counter is zero

# call\_rcu() latency

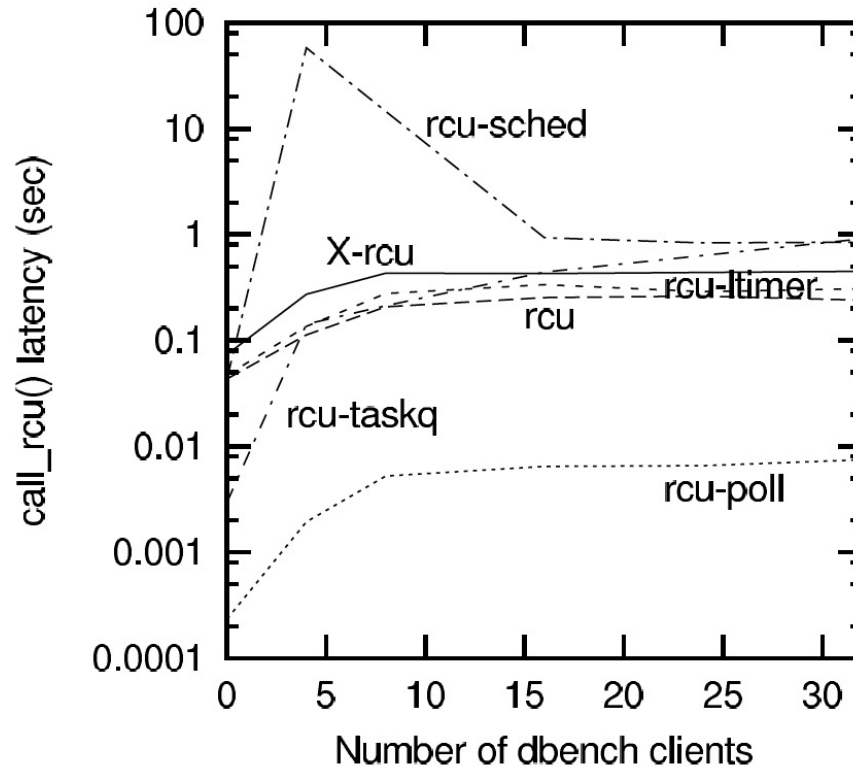
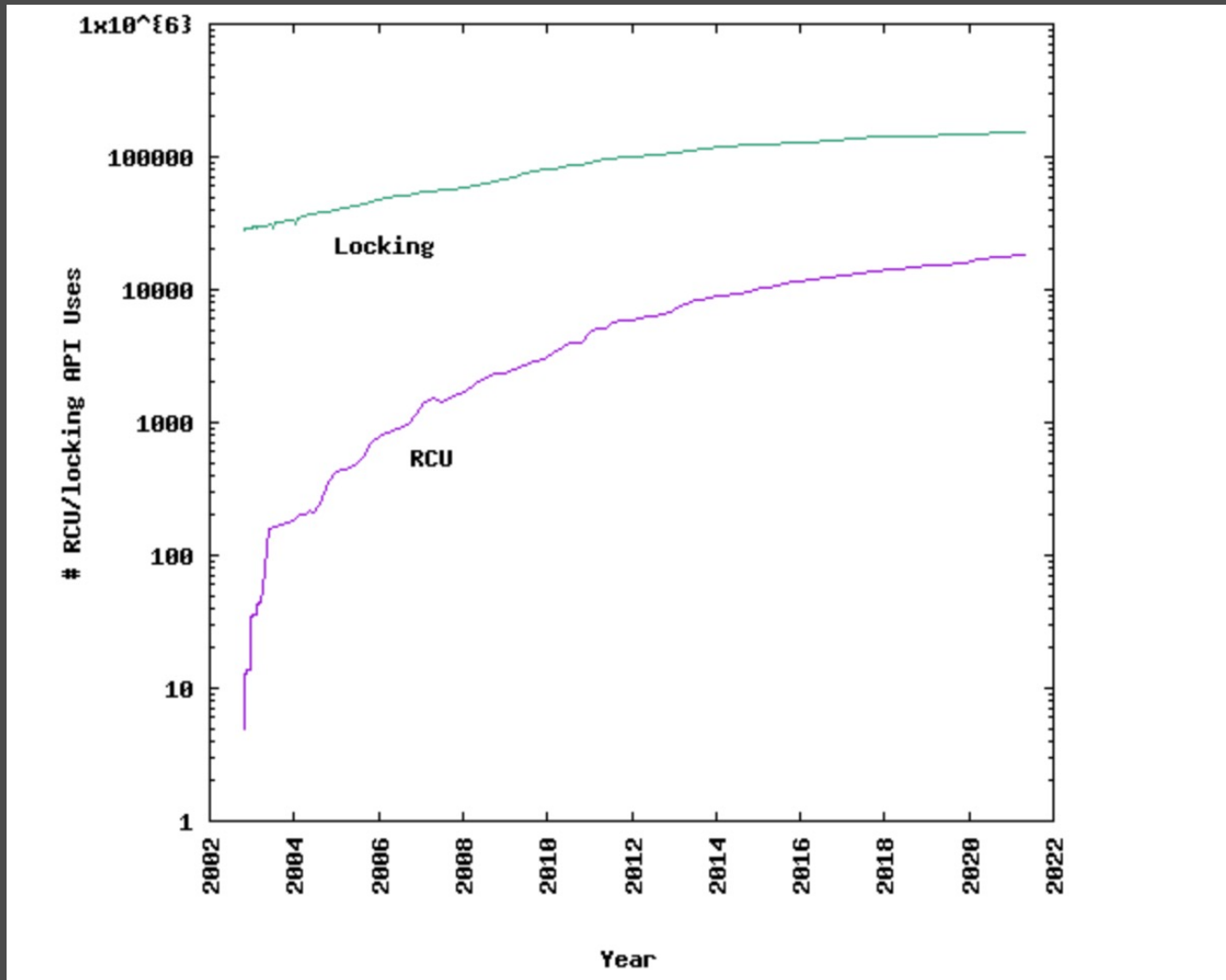
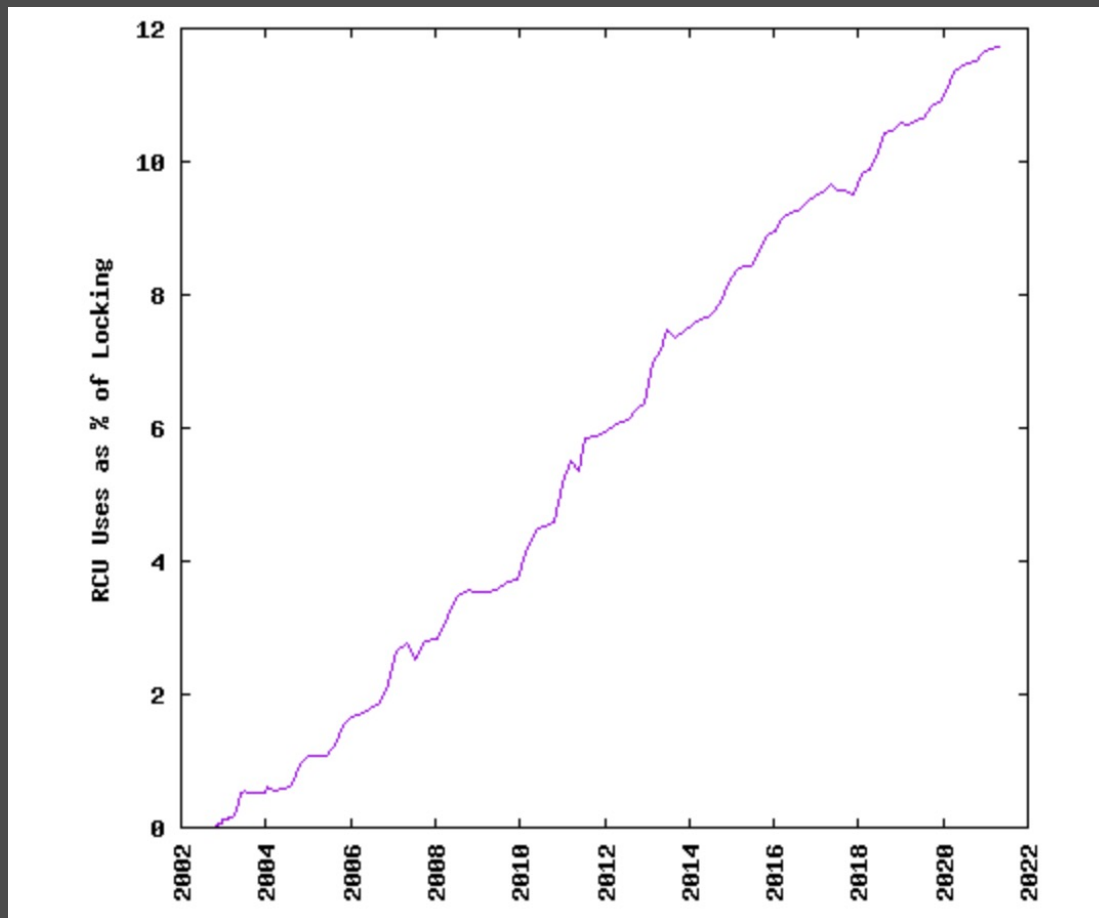


Figure 10: call\_rcu() Latency Under dbench Load (logscale)

# RCU usage in Linux



# RCU as percentage of all locking in linux



# RCU Usage

Type of Usage	API Usage
RCU critical sections	4,431
RCU dereference	1,365
RCU synchronization	855
RCU list traversal	813
RCU list update	745
RCU assign	454
Annotation of RCU-protected pointers	393
Initialization and cleanup	341
RCU lockdep assertion	37
Total	9,434

Figure 11: Linux 3.16 RCU usage by RCU API function.

Subsystem	Uses	LoC	Uses / KLoC
ipc	92	9,094	10.12
virt	82	10,037	8.17
net	4519	839,441	5.38
security	289	73,134	3.95
kernel	885	224,471	3.94
block	76	37,118	2.05
mm	204	103,612	1.97
lib	75	94,008	0.80
fs	792	1,131,589	0.70
init	2	3,616	0.55
include	331	642,722	0.51
drivers	1949	10,375,284	0.19
crypto	12	74,794	0.16
arch	249	2,494,395	0.10
tools	2	144,181	0.01
Total	9,559	16,257,496	0.59

Figure 10: Linux 3.16 RCU usage by subsystem.

# Discussion of paper

- Really challenging paper to read
  - Written by OS hackers (good thing!)
  - Mixes fundamentals and implementations
    - We have to try to step back and identify them
  - Too many ideas/alternatives
    - Better just to focus on one or two?
  - Back end of the paper is a survey
- What are your thoughts?

# SeqLock

- Another special synchronization primitive
- Goal is to avoid writer starvation in reader writer locks
- Has a lock and a sequence number
  - Lock for writers only
  - Writer increments sequence number after acquiring lock and before releasing lock
- Readers do not block
  - But can check sequence number