Advanced Operating Systems (CS 202)

Read Copy Update (RCU)

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

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
  - Ok to read a slightly stale copy
    - But that can be fixed too

Traditional OS locking designs

- complex
- poor concurrency
- Fail to take advantage of event-driven nature of operating systems

Motivation

- Locks have acquire and release cost
  - Each uses atomic operations which are expensive
  - Can dominate cost for short critical regions
  - Locks become the bottleneck
- Readers/writers lock is also expensive – uses atomic increment/decrement for reader count

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

Race Between Teardown and Use of Service

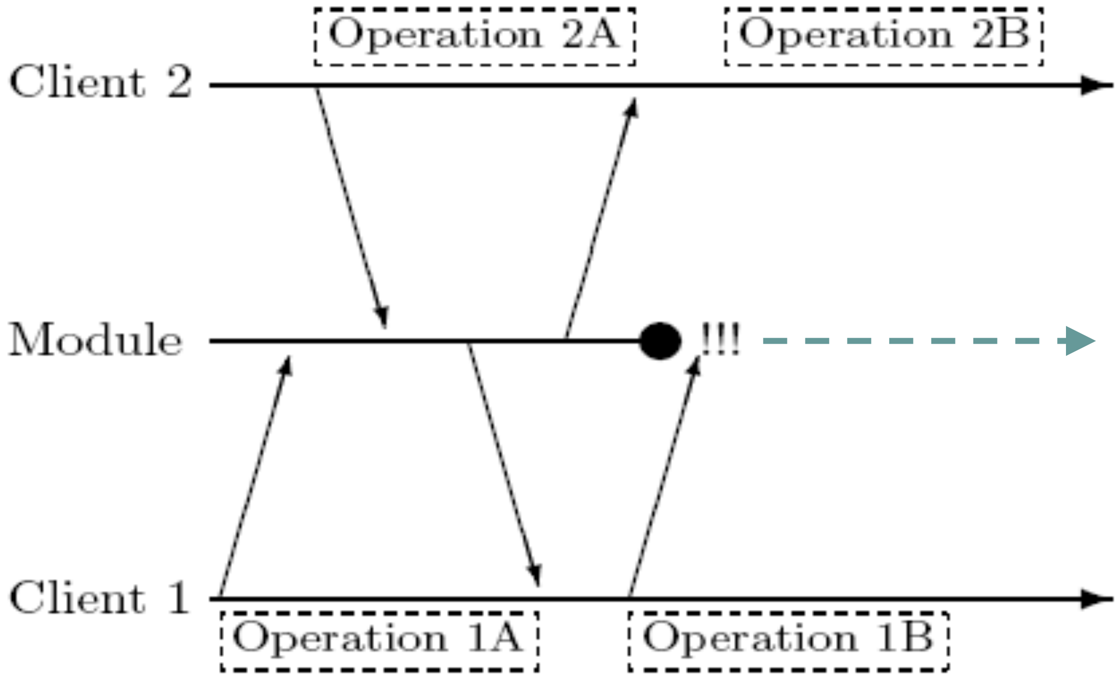


Figure 1: Race Between Teardown and Use of Service

Read-Copy Update Handling Race

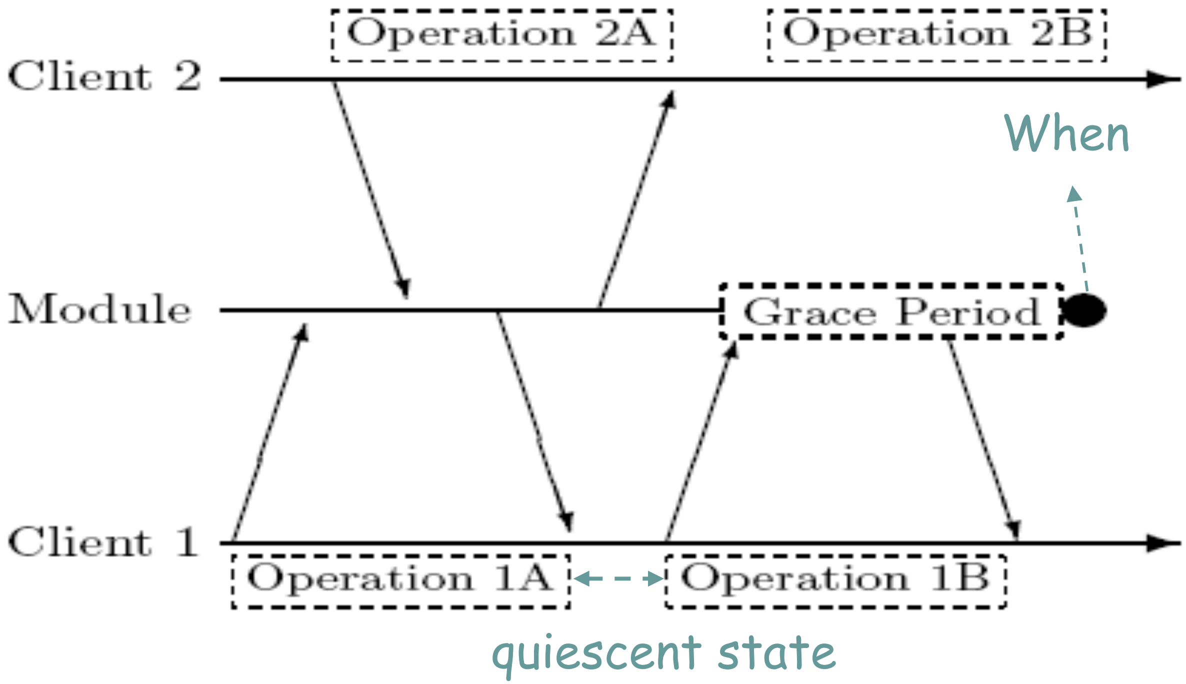


Figure 2: Read-Copy Update Handling Race

Cannot be context switched inside RCU

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 data structure, so it now may safely be reclaimed.

Read-Copy Search

```
struct el search(long addr)
{
    read_lock(&list_lock);
    p = head->next;
    while (p != head) {
        if (p->address == addr) {
            atomic_inc(&p->refcnt)
            read_unlock(&list_lock);
            return (p);
        }
        p = p->next;
    }
    read_unlock(&list_lock);
    return (NULL);
}
```

```
struct el *search(long addr)
{
    struct el *p;
    p = head->next;
    while (p != head) {
        if (p->address == addr) {
            return (p);
        }
        p = p->next;
    }
    return (NULL);
}
```

Read-Copy Deletion

```
struct el delete(struct el *p)
{
    write_lock(&list_lock);
    p->next->prev = p->prev;
    p->prev->next = p->next;
    release(p);
    write_unlock(&list_lock);
}
```

```
void delete(struct el *p)
{
    spin_lock(&list_lock);
    p->next->prev = p->prev;
    p->prev->next = p->next;
    spin_unlock(&list_lock);
    kfree_rcu(p, NULL);
}
```

Read-Copy Deletion (delete B)

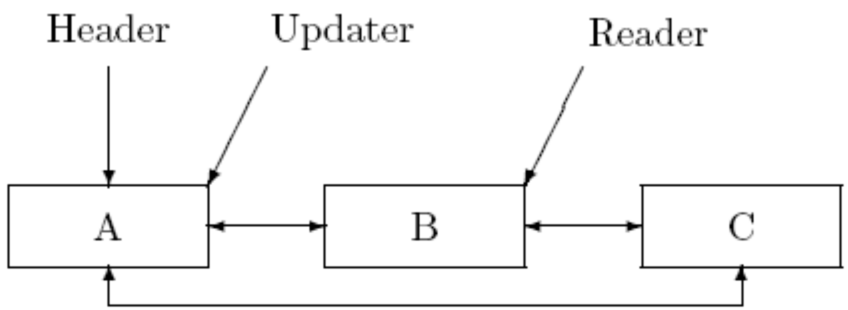


Figure 11: List Initial State

the first phase of the update

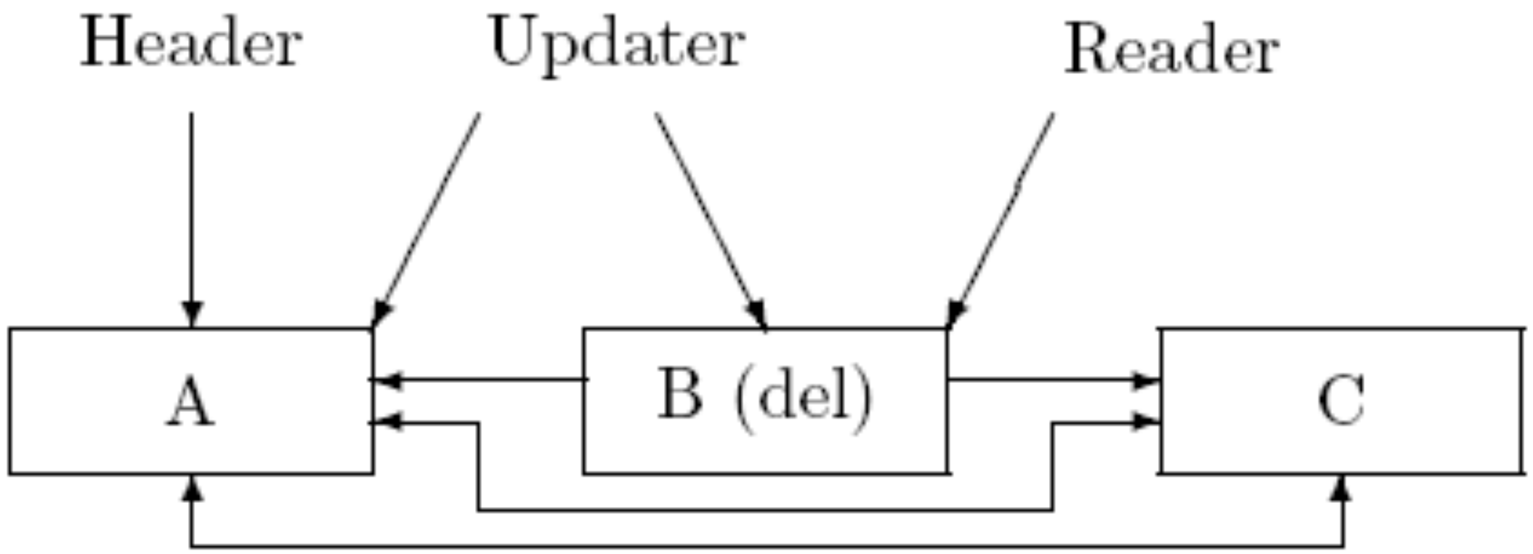


Figure 12: Element B Unlinked From List

Read-Copy Deletion

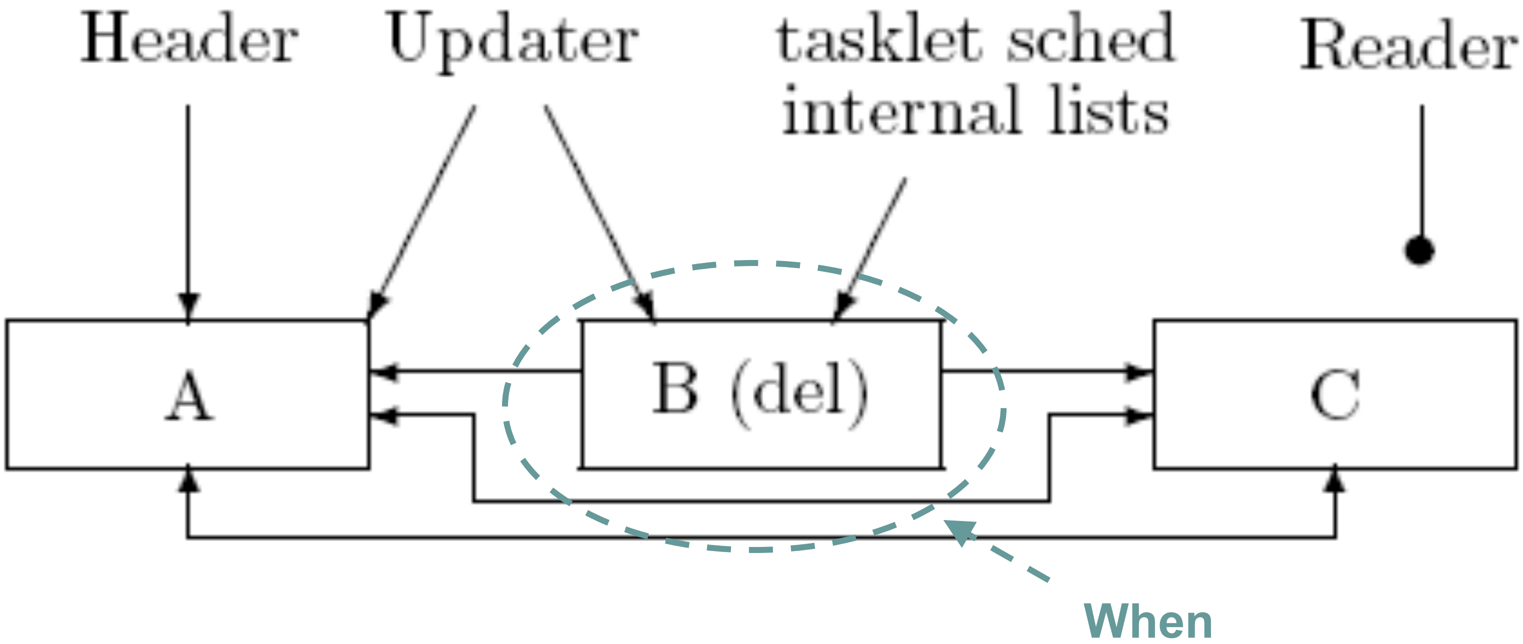


Figure 13: List After Grace Period

Read-Copy Deletion

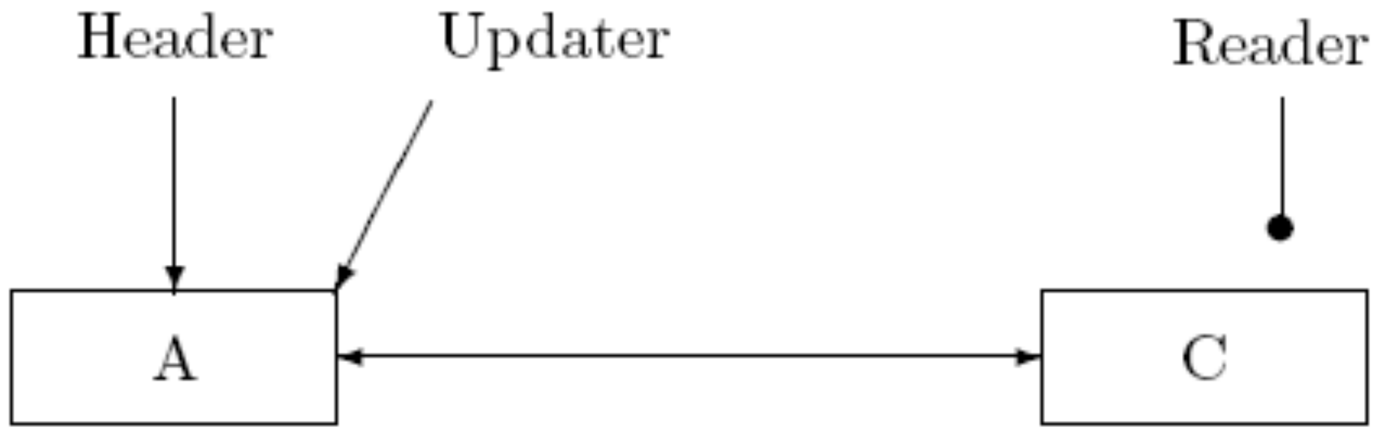
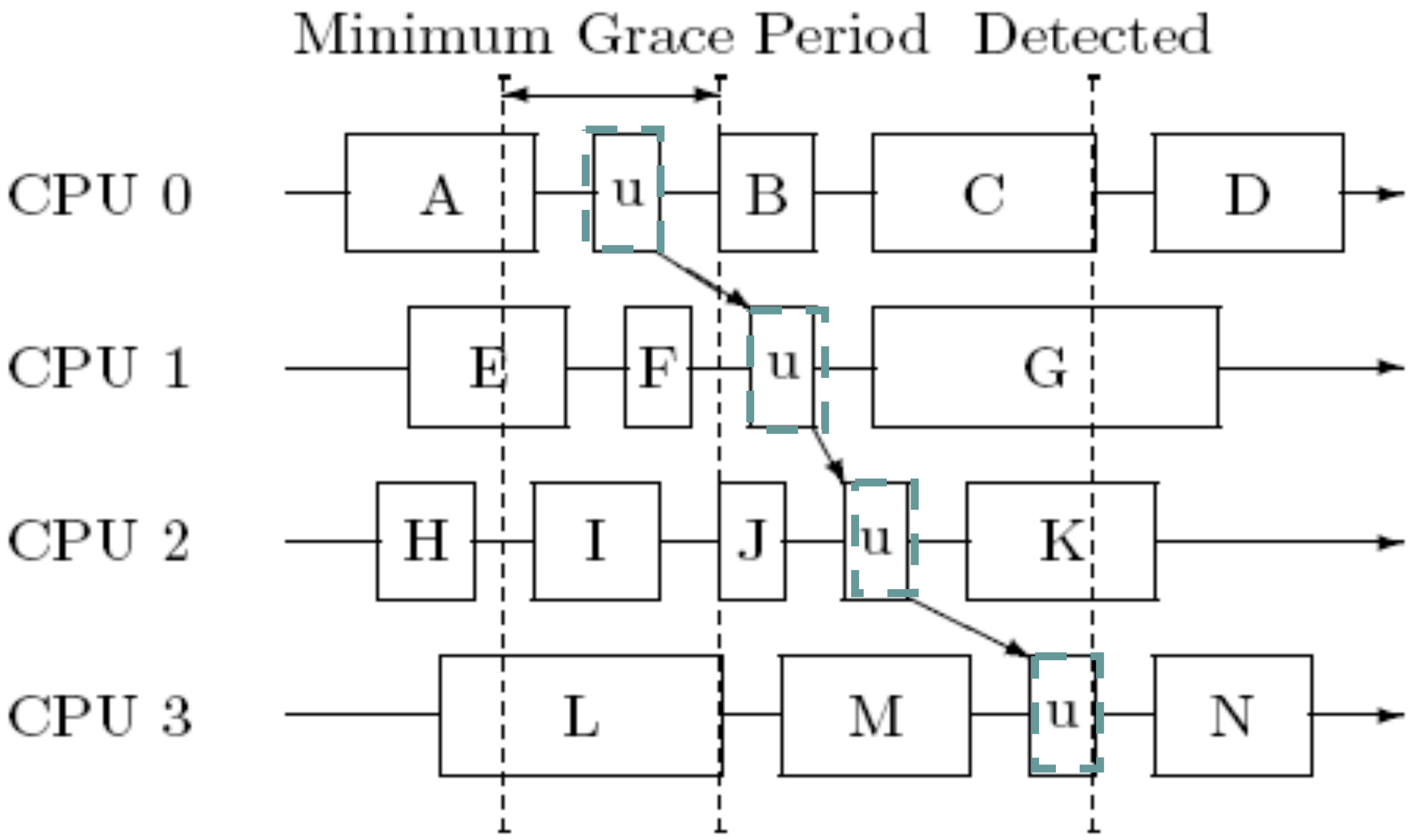


Figure 14: List After Element B Returned to Freelist

Simple Grace-Period Detection



wait_for_rcu() I

```
void wait_for_rcu(void)
{
    unsigned long cpus_allowed;
    unsigned long policy;
    unsigned long rt_priority;
    /* Save current state */
    cpus_allowed = current->cpus_allowed;
    policy = current->policy;
    rt_priority = current->rt_priority;
    /* Create an unreal time task. */
    current->policy = SCHED_FIFO;
    current->rt_priority = 1001 +
    sys_sched_get_priority_max(SCHED_FIFO);
    /* Make us schedulable on all CPUs. */
    current->cpus_allowed =
                (1UL<<smp_num_cpus)-1;

```

wait_for_rcu() II

```
    /* Eliminate current cpu, reschedule */
    while ((current->cpus_allowed &= ~(1 <<
                cpu_number_map(
                 smp_processor_id()))) != 0)
        schedule();
    /* Back to normal. */
    current->cpus_allowed = cpus_allowed;
    current->policy = policy;
    current->rt_priority = rt_priority;
}
```

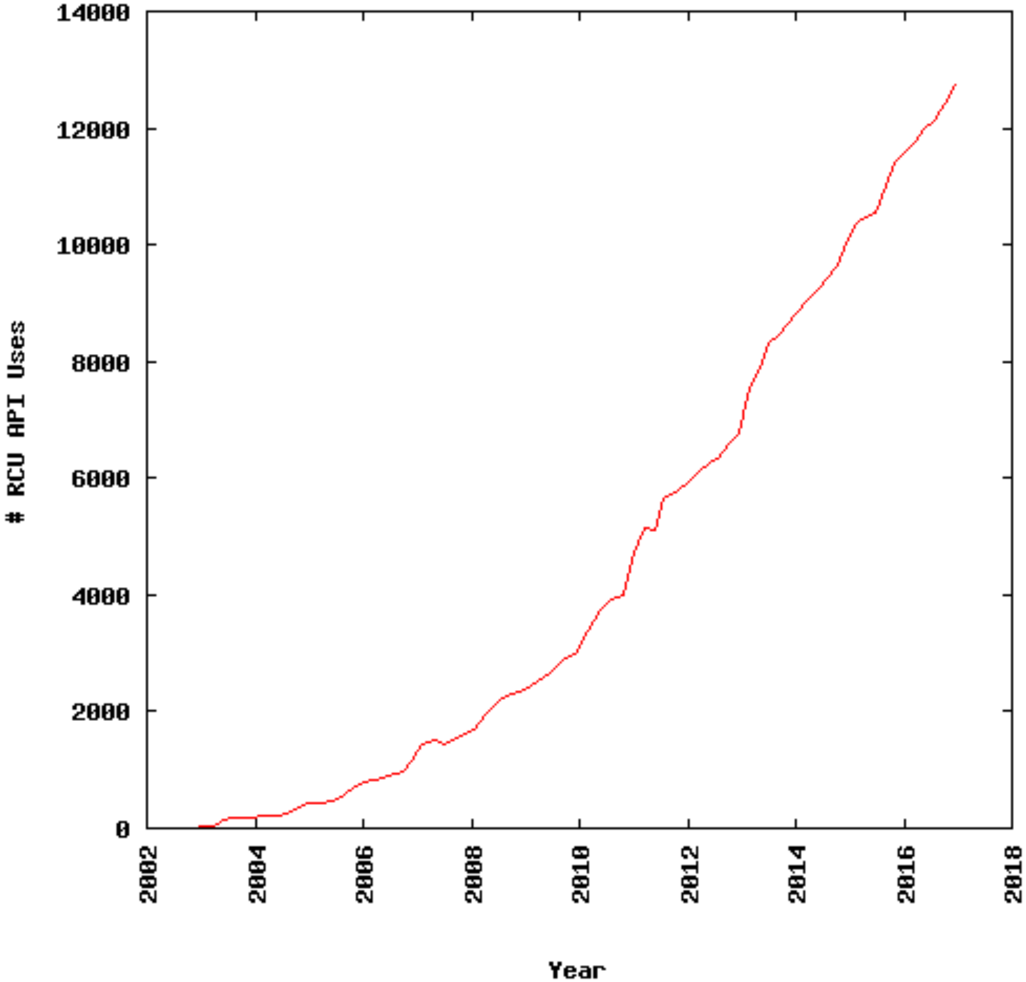
Implementations of Quiescent State

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 (option 4)

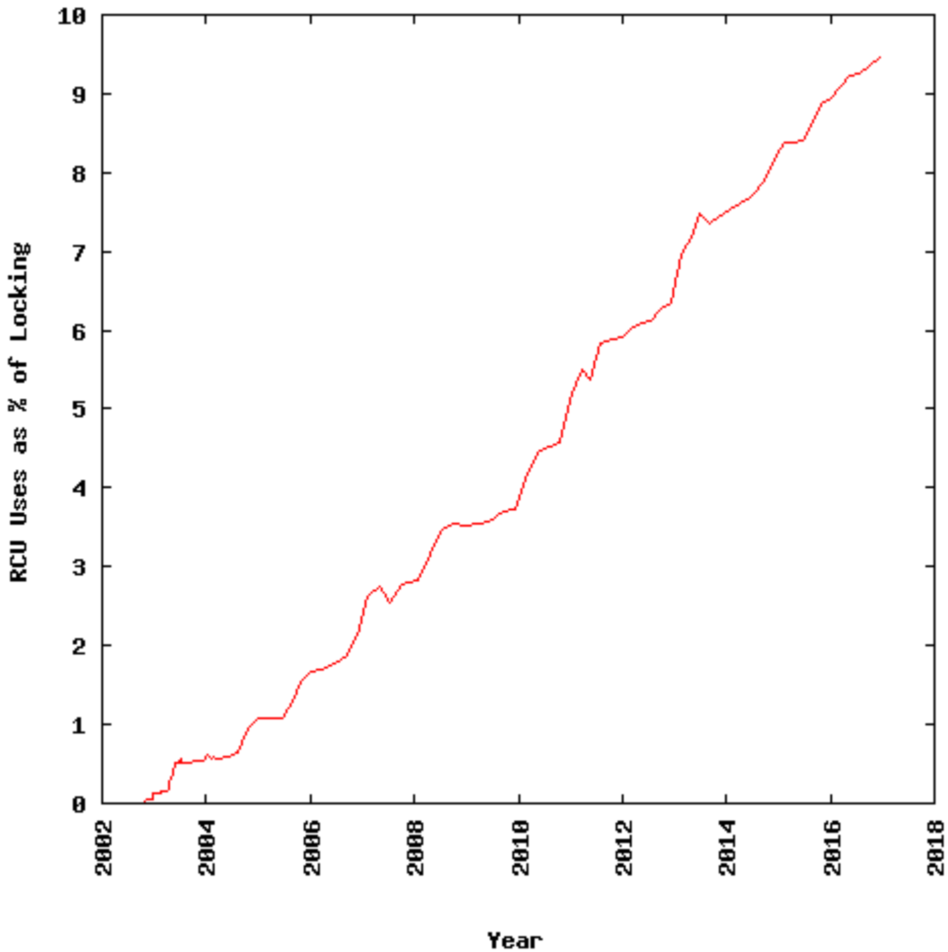
- Generation counter for each RCU region
- Generation updated on write
- Every read increments generation counter going in
  - And decrements it going out
- Quiescence = counter is zero

RCU usage in Linux



Source: http://www.rdrop.com/users/paulmc/RCU/linuxusage.html

RCU as percentage of all locking in linux



Source: http://www.rdrop.com/users/paulmc/RCU/linuxusage.html

Shortcomings

- Does not work in a preemptive kernel unless preemption is suppressed in all read-side critical sections
- Cannot be called from an interrupt handler
- Should not be called while holding a spinlock or with interrupts disabled
- Relatively slow

Preemptible kernels

- Read-side critical section
  - Readers can now be preempted in their read-side critical
  - Disable preemption on entry and re-enable on exit

- Memory freed using synchronize_sched()
  - Counts scheduler preemptions

- Benefits and trade-offs
  - Allows use of RCU with preemptible kernel
  - Read-side critical section won't be preempted by RT events, negative consequences for RT responsiveness
  - Additional read-side work to disable/enable preemption

RCU – with counters

- Per-CPU counter
  - Atomic increment in rcu_read_lock()
  - Atomic decrement in rcu_read_unlock()
- Quiescent state defined as all per CPU counters down to 0

Advanced Operating Systems (CS 202)

Distributed OS– intro and discussion

Overview

- Hardware is changing, so software must too
  - Multicores are here to stay
  - Architectures are heterogeneous
  - Applications are unpredictable unlike specialized systems
- How do operating systems scale?
- Do we need new OS architectures?

Landscape/motivation

- Systems are diverse
  - different implementations require different tradeoffs
    - Some nice examples
- Cores are increasingly diverse
  - Different general-purpose cores
  - Accelerators and specialized processors
  - Typically cannot share an OS with such differences
- Interconnects matter: within cores and across cores

What has gone on before?

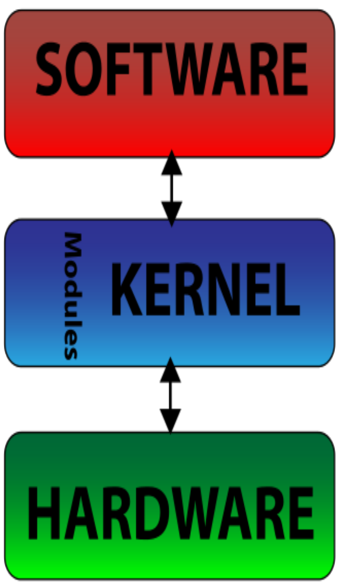
- Early on, locks were not so expensive
  - Just use them
- Hardware evolved, memory expensive
  - Large caches
  - Cache coherence
  - NUMA machines
  - Increasing gap between memory and processor
  - Shared memory expensive!

The Multikernel: A New OS Architecture for Scalable Multicore Systems

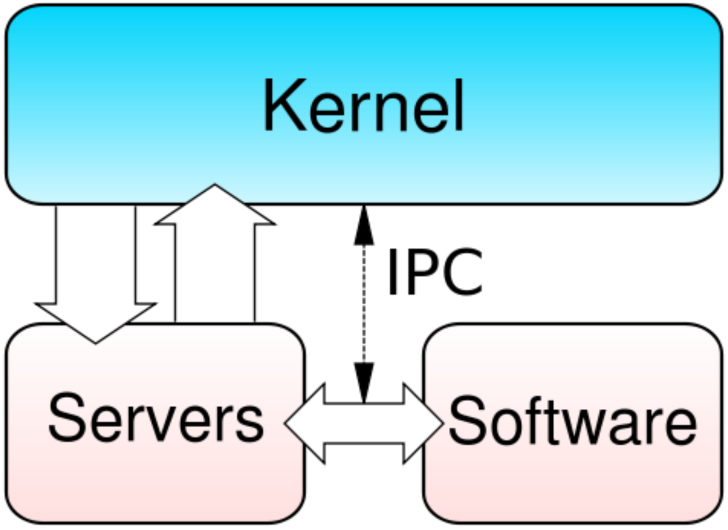
By (last names): Baumann, Barham, Dagand, Harris, Isaacs, Peter, Roscoe, Schupbach, Singhania

The Modern Kernel(s)

Monolithic



Microkernel



The Problem with Modern Kernels

- Modern Operating systems can no longer take serious advantage of the hardware they are running on
- There exists a scalability issue in the shared memory model that many modern kernels abide by
- Cache coherence overhead restricts the ability to scale to many-cores

Solution: MultiKernel

- Treat the machine as a network of independent cores
- Make all inter-core communication explicit; use message passing
- Make OS structure hardware-neutral
- View state as replicated instead of shared

But wait! Isn't message passing slower than Shared Memory?

Not at scale

But wait! Isn't message passing slower than Shared Memory?

- At scale it has been shown that message passing has surpassed shared memory efficiency
- Shared memory at scale seems to be plagued by cache misses which cause core stalls
- Hardware is starting to resemble a message-passing network

But wait! Isn't message passing slower than Shared Memory? (cont.)

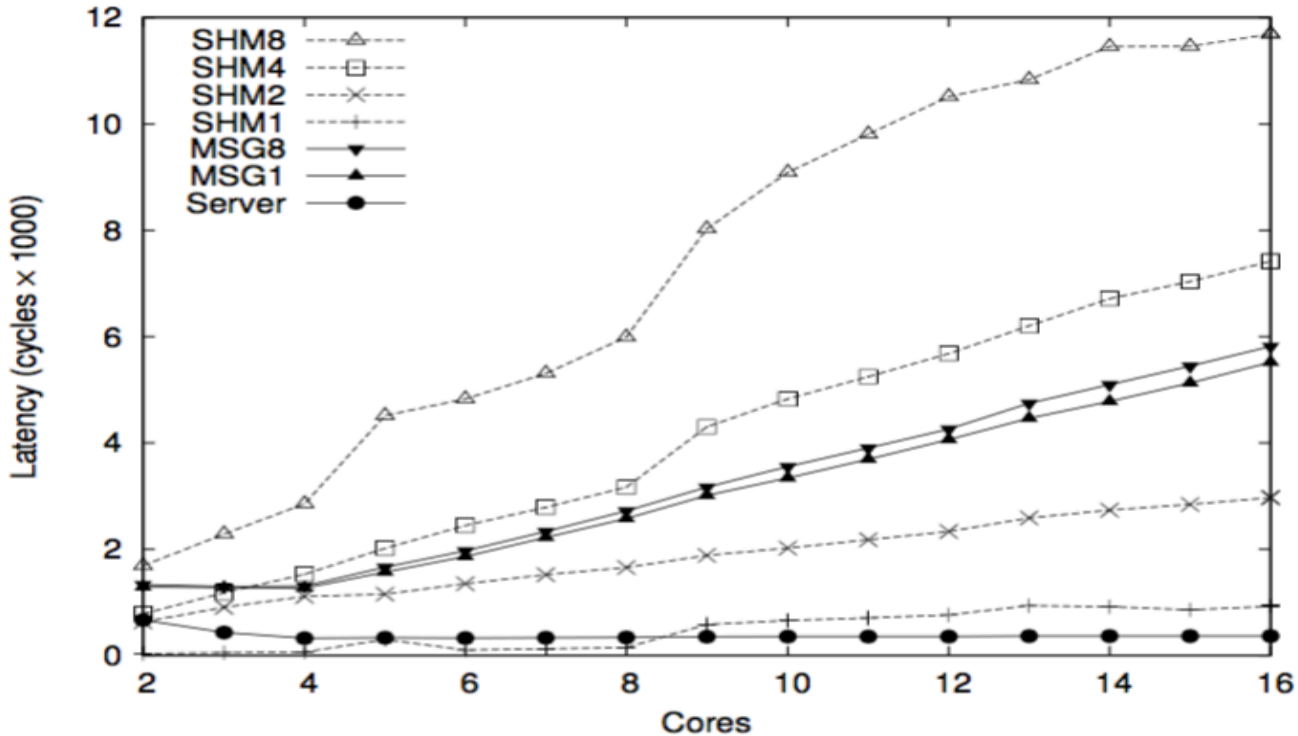


Figure 3: Comparison of the cost of updating shared state using shared memory and message passing.

But wait! Isn't message passing slower than Shared Memory? (cont.)

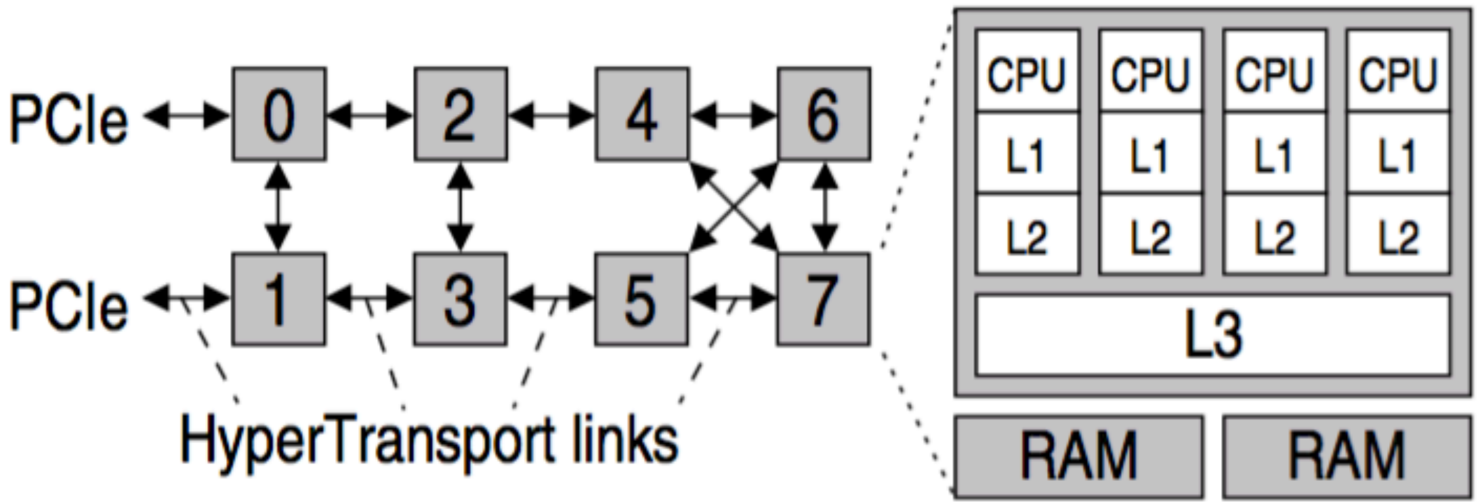


Figure 2: Node layout of an 8×4-core AMD system

The MultiKernel Model

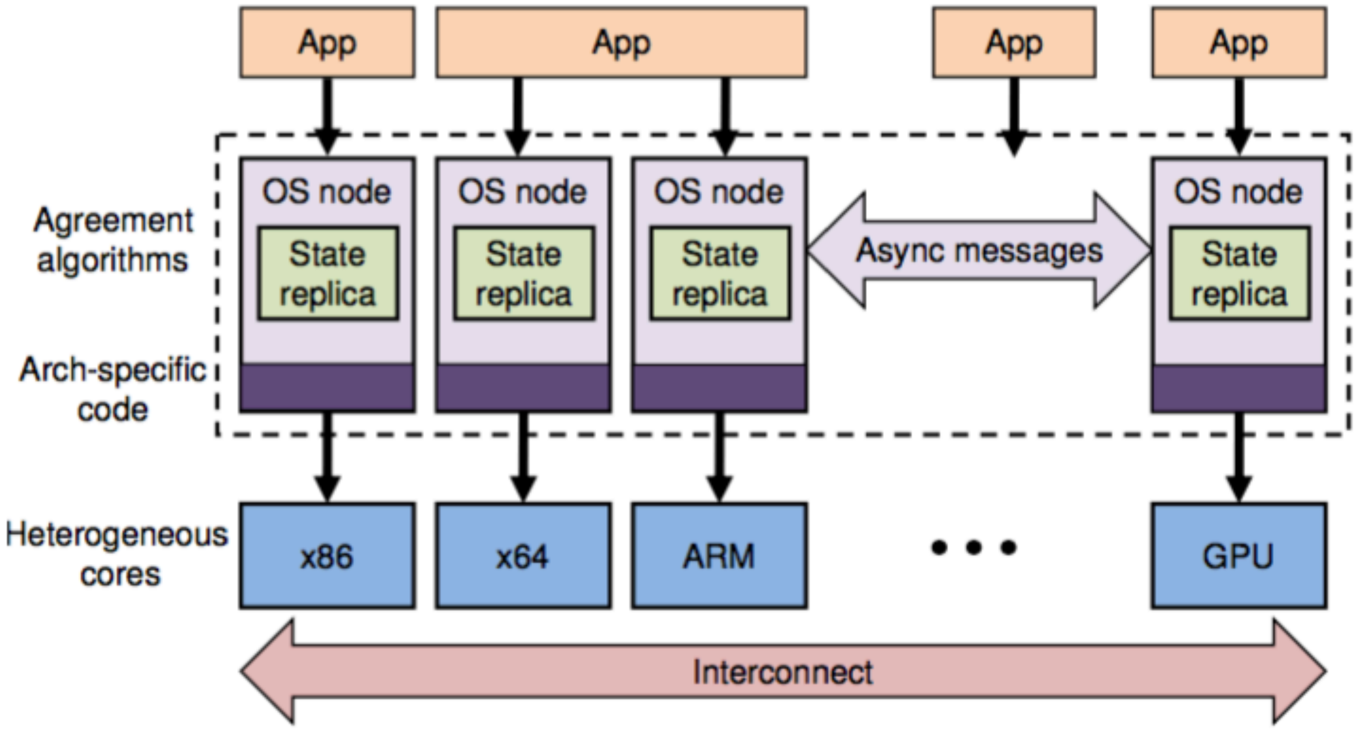


Figure 1: The multikernel model.