CS510 Concurrent Systems

Jonathan Walpole

Portland State University

RCU Usage in Linux

History of Concurrency in Linux

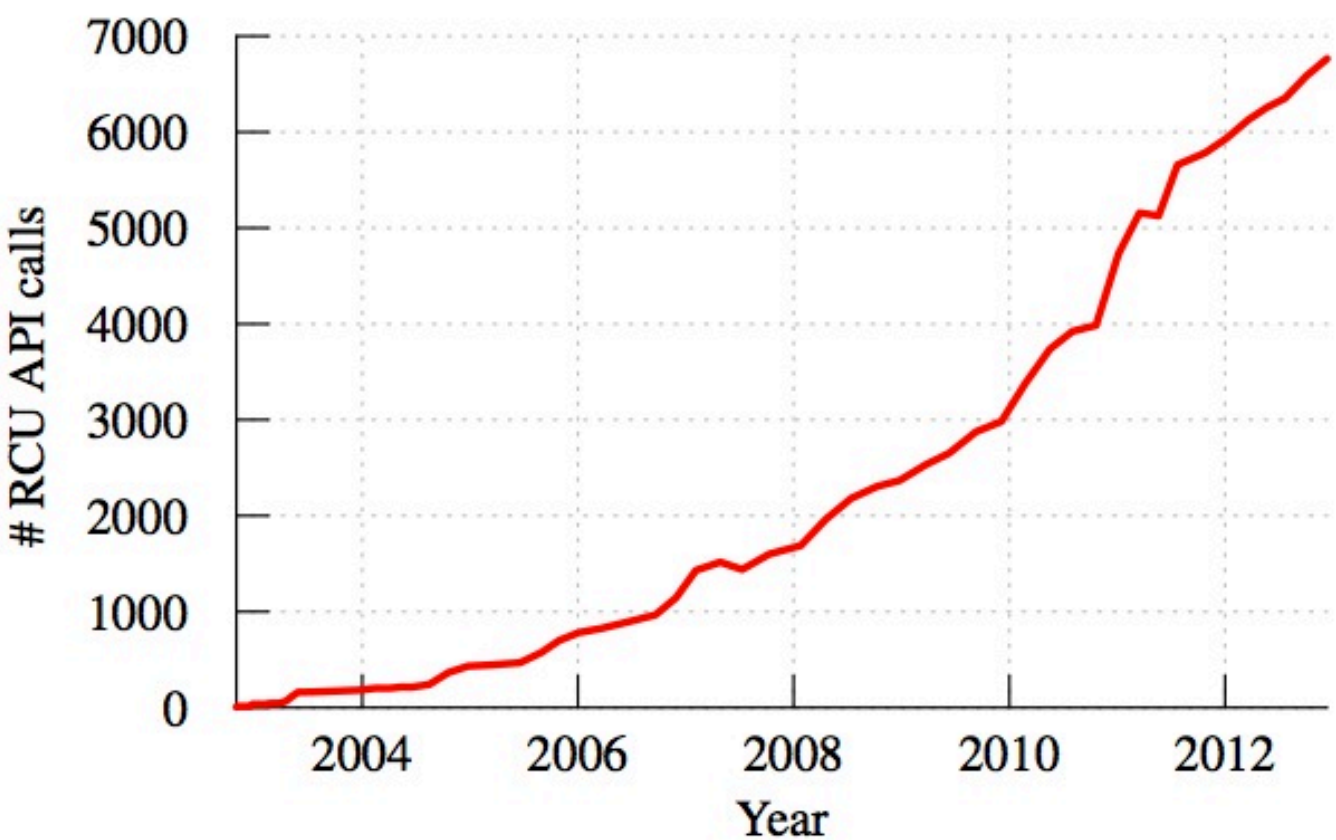
Multiprocessor support 15 years ago

- via non-preemption in kernel mode

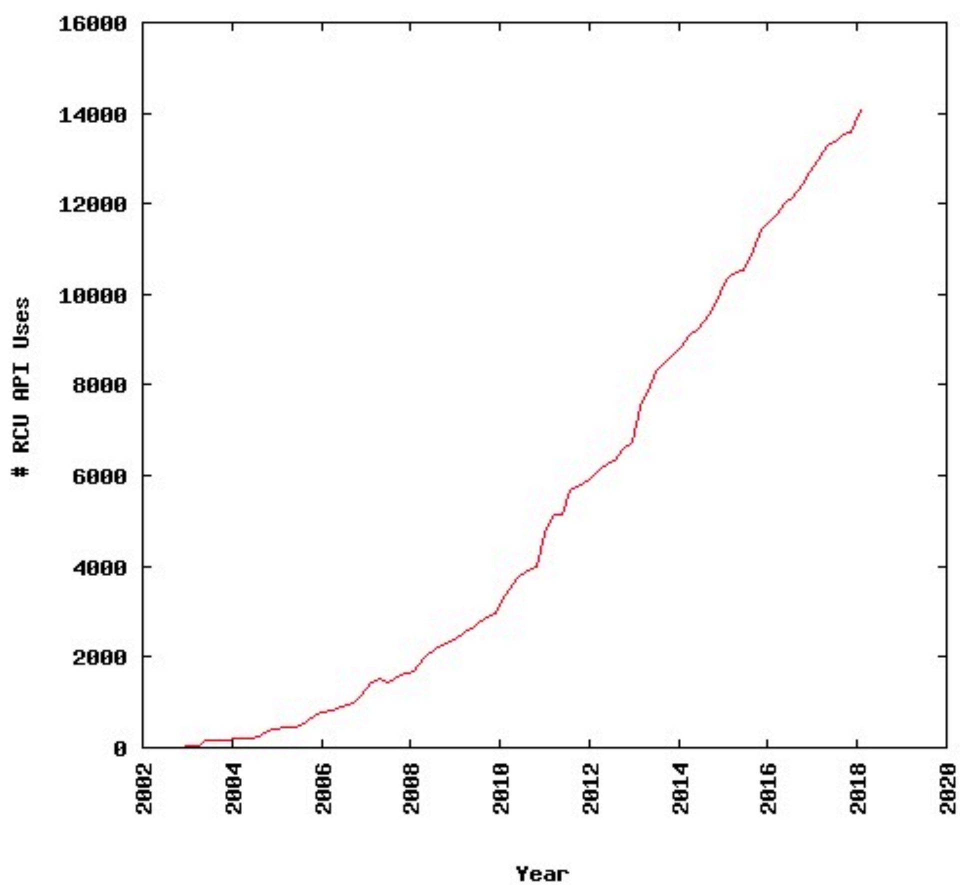
Today's Linux

- fine-grain locking
- lock-free data structures
- per-CPU data structures
- RCU

Increasing Use of RCU API



Increasing Use of RCU API



Why RCU?

Scalable concurrency

Very low overhead for readers

Concurrency between readers and writers

- writers create new versions
- reclaiming of old versions is deferred until all pre-existing readers are finished

Why RCU?

Need for concurrent reading and writing

- example: directory entry cache replacement

Low computation and storage overhead

- example: storage overhead in directory cache

Deterministic completion times

- example: non-maskable interrupt handlers in real-time systems

RCU Interface

Reader primitives

- rcu_read_lock and rcu_read_unlock
- rcu_dereference

Writer primitives

- synchronize_rcu
- call_rcu
- rcu_assign_pointer

A Simple RCU Implementation

```
void rcu_read_lock()
{
  preempt_disable[cpu_id()]++;
}

void rcu_read_unlock()
{
  preempt_disable[cpu_id()]--;
}

void synchronize_rcu(void)
{
  for_each_cpu(int cpu)
    run_on(cpu);
}
```

Practical Implementations of RCU

The Linux kernel implementations of RCU amortize reader costs

- waiting for all CPUs to context switch delays writers (collection) longer than strictly necessary
- ... but makes read-side primitives very cheap

They also batch servicing of writer delays

- polling for completion is done only once per scheduling tick or so
- thousands of writers can be serviced in a batch

RCU Usage Patterns

Wait for completion

Reference counting

Type safe memory

Publish subscribe

Reader-writer locking alternative

Wait For Completion Pattern

Waiting thread waits with

- synchronize_rcu

Waitee threads delimit their activities with

- rcu_read_lock
- rcu_read_unlock

Example: Linux NMI Handler

```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;

void handle_nmi()
{
  rcu_read_lock();
  rcu_list_for_each(&nmi_list, handler_t cb)
    cb();
  rcu_read_unlock();
}
```

Example: Linux NMI Handler

```
void register_nmi_handler(handler_t cb)
{
  spin_lock(&nmi_list_lock);
  rcu_list_add(&nmi_list, cb);
  spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb)
{
  spin_lock(&nmi_list_lock);
  rcu_list_remove(cb);
  spin_unlock(&nmi_list_lock);
  synchronize_rcu();
}
```

Advantages

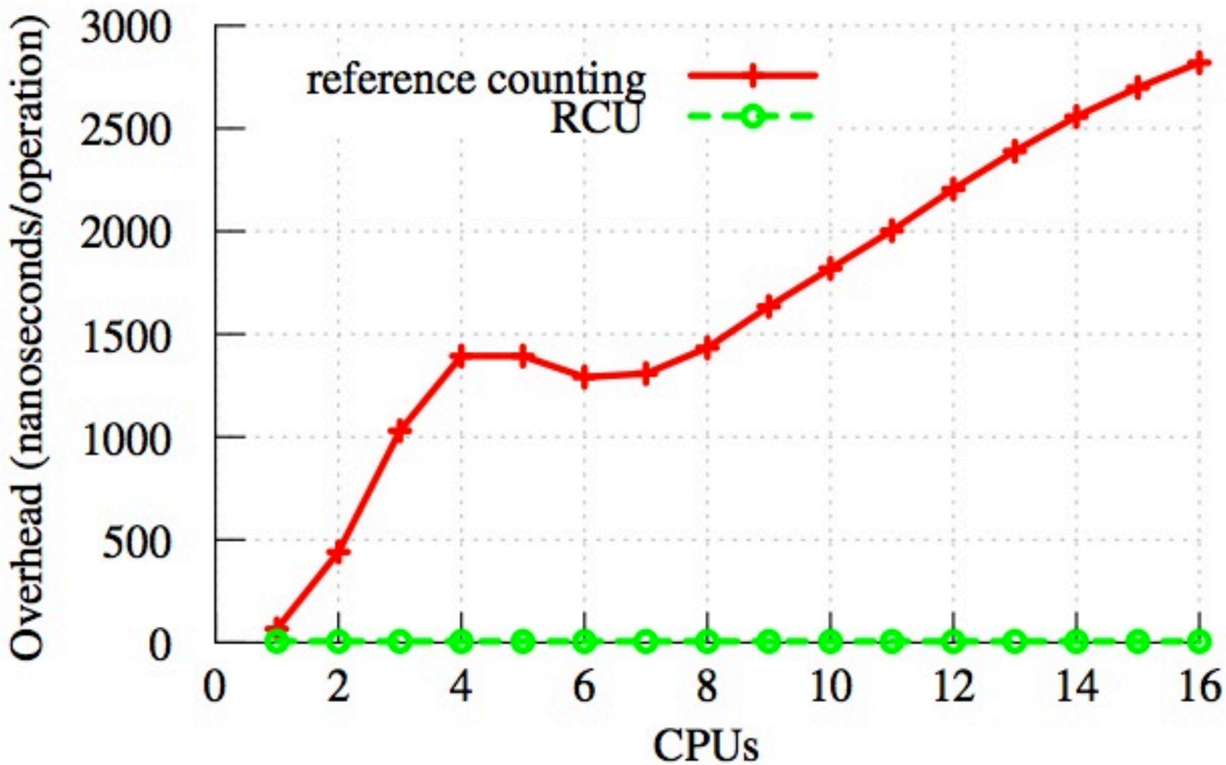
Allows dynamic replacement of NMI handlers
Has deterministic execution time
No need for reference counts

Reference Counting Pattern

Instead of counting references (which requires expensive synchronization among CPUs) simply have users of a resource execute inside RCU read-side sections

No updates, memory barriers or atomic instructions are required!

Cost of RCU vs Reference Counting



A Use of Reference Counting Pattern for Efficient Sending of UDP Packets

```
void udp_sendmsg(sock_t *sock, msg_t *msg)
{
  ip_options_t *opts;
  char packet[];

  copy_msg(packet, msg);
  rcu_read_lock();
  opts = rcu_dereference(sock->opts);
  if (opts != NULL)
    copy_opts(packet, opts);
  rcu_read_unlock();

  queue_packet(packet);
}
```

Use of Reference Counting Pattern for Dynamic Update of IP Options

```
void setsockopt(sock_t *sock, int opt,
                void *arg)
{
  if (opt == IP_OPTIONS) {
      ip_options_t *old = sock->opts;
      ip_options_t *new = arg;

      rcu_assign_pointer(&sock->opts, new);
      if (old != NULL)
        call_rcu(kfree, old);
      return;
  }

  /* Handle other opt values */
}
```

Type Safe Memory Pattern

Type safe memory is used by lock-free algorithms to ensure completion of optimistic concurrency control loops even in the presence of memory recycling

RCU removes the need for this by making memory reclamation and dereferencing safe

... but sometimes RCU can not be used directly

- e.g. in situations where the thread might block

Using RCU for Type Safe Memory

Linux slab allocator uses RCU to provide *type safe memory*

Linux memory allocator provides slabs of memory to type-specific allocators

SLAB_DESTROY_BY_RCU ensures that a slab is not returned to the memory allocator (for potential use by a different type-specific allocator) until all readers of the memory have finished

Publish Subscribe Pattern

Common pattern involves initializing new data then making a pointer to it visible by updating a global variable

Must ensure that compiler or CPU does not re-order the writers or readers operations

- initialize -> pointer update
- dereference pointer -> read data

rcu_assign_pointer and rcu_dereference ensure this!

Example Use of Publish-Subscribe for Dynamic System Call Replacement

```
syscall_t *table;
spinlock_t table_lock;

int invoke_syscall(int number, void *args...)
{
  syscall_t *local_table;
  int r = -1;

  rcu_read_lock();
  local_table = rcu_deference(table);
  if (local_table != NULL)
    r = local_table[number](args);
  rcu_read_unlock();

  return r;
}
```

Example Use of Publish-Subscribe for Dynamic System Call Replacement

```
void retract_table()
{
  syscall_t *local_table;

  spin_lock(&table_lock);
  local_table = table;
  rcu_assign_pointer(&table, NULL);
  spin_unlock(&table_lock);

  synchronize_rcu();
  kfree(local_table);
}
```

Reader-Writer Locking Pattern

RCU is used instead of reader-writer locking
- it allows concurrency among readers
- but it also allows concurrency among readers and writers!

Its performance is much better

But it has different semantics that may affect the application
- must be careful

Why Are R/W Locks Expensive?

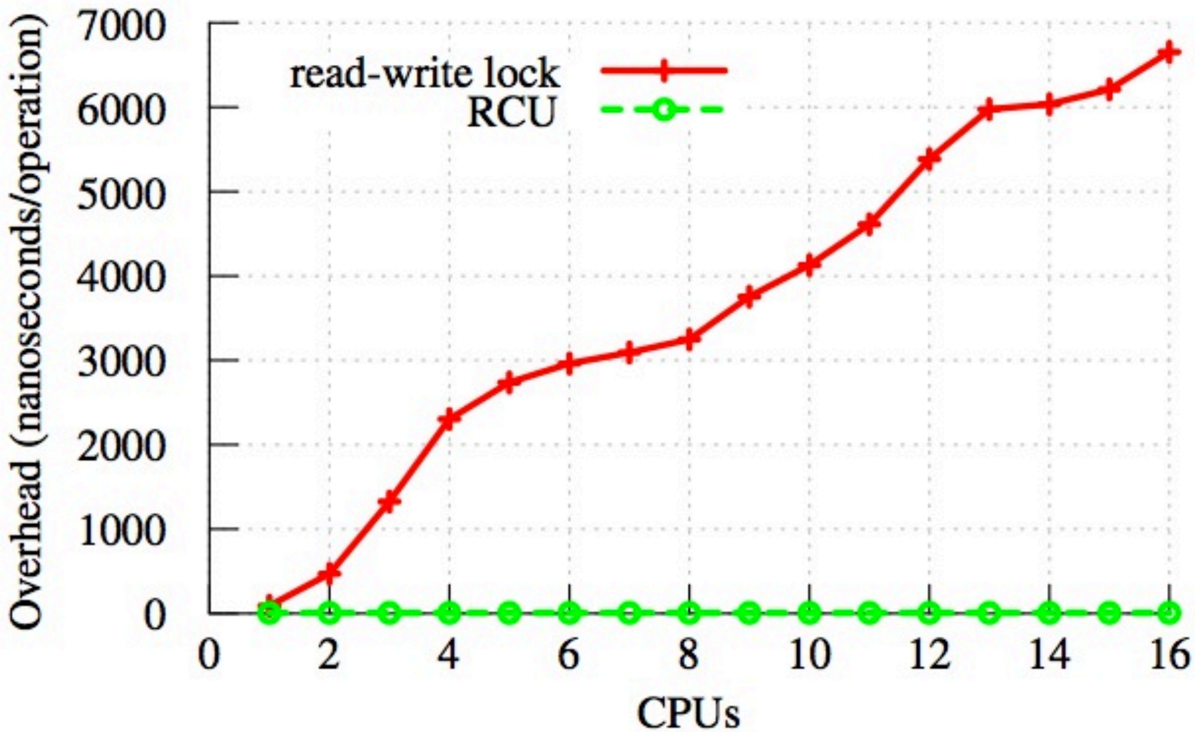
A reader-writer lock keeps track of how many readers are present

Readers and writers update the lock state

The required atomic instructions are expensive!

- for short read sections there is no reader-reader concurrency in practice

RCU vs Reader-Writer Locking



Example Use of RCU Instead of RWL

```
pid_table_entry_t pid_table[];

process_t *pid_lookup(int pid)
{
  process_t *p

  rcu_read_lock();
  p = pid_table[pid_hash(pid)].process;
  if (p)
    atomic_inc(&p->ref);
  rcu_read_unlock();
  return p;
}
```

Example Use of RCU Instead of RWL

```
void pid_free(process *p)
{
  if (atomic_dec(&p->ref))
    free(p);
}

void pid_remove(int pid)
{
  process_t **p;

  spin_lock(&pid_table[pid_hash(pid)].lock);
  p = &pid_table[pid_hash(pid)].process;
  rcu_assign_pointer(p, NULL);
  spin_unlock(&pid_table[pid_hash(pid)].lock);

  if (*p)
    call_rcu(pid_free, *p);
}
```

Semantic Differences

Consider the following example:

- writer thread 1 adds element A to a list
- writer thread 2 adds element B to a list
- concurrent reader thread 3 searching for A then B finds A but not B
- concurrent reader thread 4 searching for B and then A finds B but not A

This is non-linearizable, and allowed by RCU!

- Is this allowed by reader-writer locking?
- Is this correct?

Some Solutions

Insert level of indirection

Mark obsolete objects

Retry readers

Insert Level of Indirection

Does your code depend on all updates in a write-side critical section becoming visible to readers atomically?

If so, hide all the updates behind a single pointer, and update the pointer using RCU's publish-subscribe pattern

Mark Obsolete Objects/Retry Readers

Does your code depend on readers not seeing older versions?

If so, associate a flag with each object and set it when a new version of the object is produced

Readers check the flag and fail or retry if necessary

Where is RCU Used?

Subsystem	Uses	LoC	Uses / KLoC
virt	72	6,749	10.67
net	3251	740,382	4.39
ipc	35	8,306	4.21
security	251	68,494	3.66
kernel	628	198,304	3.17
mm	196	88,904	2.20
block	58	27,975	2.07
lib	70	52,235	1.34
fs	666	1,057,713	0.63
init	2	3,382	0.59
include	279	552,507	0.50
crypto	12	64,537	0.19
drivers	1061	8,530,160	0.12
arch	183	2,459,105	0.07
Total	6764	13,858,753	0.49

Which RCU Primitives Are Used Most?

Type of Usage	API Usage
RCU critical sections	3035
RCU dereference	972
RCU synchronization	696
RCU list traversal	574
RCU list update	524
RCU assign	358
Annotation of RCU-protected pointers	304
Initialization and cleanup	273
RCU lockdep assertion	28
Total	6764

Conclusions and Future Work

RCU solves real-world problems

It has significant performance, scalability and software engineering benefits

It embraces concurrency

- which opens up the possibility of non-linearizable behaviors!
- this requires the programmer to cultivate a new mindset

- Ongoing future work: relativistic programming