Read-Copy Update

Paul E. McKenney, IBM Beaverton
Jonathan Appavoo, U. of Toronto
Andi Kleen, SuSE
Orran Krieger, IBM T.J. Watson
Rusty Russell, RustCorp
Dipankar Sarma, IBM India Software Lab
Maneesh Soni, IBM India Software Lab
Goal

• Simple, high-performance and -scaling algorithms for read-mostly situations
  - Readers must not be required to acquire locks, execute atomic operations, or disable interrupts
    • Read-side code same as UP user-level implementation
    • Want performance to scale with CPU core clock rate, not with memory latency
  - OK if writers have to do a little more work
• Focus on non-preemptive OS kernel
Overview

- Why is this goal important?
- How the $#@#!! can readers safely access a changing data structure without locking???
  - Without writers needing a gazillion cycles to perform an update?
- Does this really help in real-world code?
Current CPU-Memory Architectures

CPU 0
- L1 Cache
- L2 Cache
- CPU 0

CPU 1
- L1 Cache
- L2 Cache
- CPU 1

CPU 2
- L1 Cache
- L2 Cache
- CPU 2

CPU 3
- L1 Cache
- L2 Cache
- CPU 3

I/O

Memory

Interconnect
Current CPU-Memory Architectures

Shared Variable in Red
All Reads Local (Fast)

I/O
Memory
Interconnect
L2 Cache
L1 Cache
CPU 0
L2 Cache
L1 Cache
CPU 1
L2 Cache
L1 Cache
CPU 2
L2 Cache
L1 Cache
CPU 3
Current CPU-Memory Architectures

Shared Variable in Red: CPU 0 Updates it (Slow)

Cacheline Invalidations

I/O

Memory

L2 Cache
L1 Cache
CPU 0

L2 Cache
L1 Cache
CPU 1

L2 Cache
L1 Cache
CPU 2

L2 Cache
L1 Cache
CPU 3
Current CPU-Memory Architectures

Shared Variable in Red:
CPU 3 Reads (Slow)
Long-Term Architectural Trends

Data from Sequent/IBM NUMA-Q Machines

Paul E. McKenney et al.
Recent Architectural Trends

Global Locks and Reference Counts: >100-to-1 Hit!!!
Architectural-Trend Consequences

- Global locks becoming increasingly expensive
- Globally-used reference counters also becoming increasingly expensive
- Would like some way for read-only accesses to read-mostly data structures to avoid locks and reference-count manipulation...
  - Take advantage of periodic nature of the Linux kernel
Overview

• Why is this goal important?
• How the $#@#!! can readers safely access a changing data structure without locking???
  – Without writers needing a gazillion cycles to perform an update?
• Does this really help in real-world code?
Periodic Nature of Linux Kernel

- Execution divided into relatively short-duration “operations” with respect to shared data structure
- No “covert channels” between operations
What is an “Operation’’???

Can Begin or End Anyplace Where No Locks Are Held

• For non-preemptible kernels:
  - begin and end at a context switch
• For preemptible Linux kernels:
  - begin and end at a voluntary context switch
• For K42 research OS:
  - duration of K42’s analog of syscall
• For this talk, focus on non-preemptible kernels
void really_bad_idea(void) {
    struct foo *fp;
    spin_lock(&my_lock);
    fp = list_head;
    spin_unlock(&my_lock);
    /* Do some other stuff… */
    spin_lock(&my_lock);
    fp = fp->next;
    spin_unlock(&my_lock);
}

• “Covert channels” bad for locks
  - Just as bad for read-copy update
  - Drop locks, and whole world can change!
Example Problem

- Race between use and deletion of list element

Diagram:

Client 1

Linked List

Client 2

Find 1A
Delete
Find 1B

Find 2A
Find 2B
Global Locks to the Rescue...

But rather slowly...

Client 1

Find 1A

Delete

Find 2A

Find 2B

Linked List

Client 2

Find 1A

Delete

Find 2A

Find 2B
Reference Counts to the Rescue...

- Remember the architectural trends!!!
- Incrementing a shared reference count is \~100 times more expensive than a typical instruction!!!
- And the relative expense has been getting steadily worse over the past two decades.
- Sometimes must write to shared storage
  - But let’s not do it gratuitously…
Read-Copy Update: Grace Period

Client 1

Find 1A

Delete

Find 1B

Linked List

ref

ref

ref

Grace Period

no ref

Client 2

Find 2A

Find 2B
struct el *find(long key) 
{
    struct el *p;
    p = head->next;
    while (p != head) {
        if (p->key == key) {
            return (p);
        }
        p = p->next;
    }
    return (NULL);
}

- Identical to single-threaded code
- No cache thrashing or spinning
  - Scale with core CPU clock rate
Read-Copy Update

```c
void delete(struct el *p) {
    struct el *p;

    lock(&list_updater_lock);
    p->next->prev = p->prev;
    p->prev->next = p->next;
    unlock(&list_updater_lock);

    wait_for_rcu();

    kfree(p);
}
```

- Simple, straightline code
- Locking OK if read-mostly access
  - Faster designs available

Phase 1
Grace Period
Phase 2
Read-Copy Update Animation

• Initial List State
  - Doubly linked list

```
Header

A

B

C

Updater (CPU 0)  Reader (CPU 1)
```
Read-Copy Update Animation

- First Phase Complete:

```c
lock(&list_updater_lock);
p = prev->next;
prev->next = p->next;
unlock(&list_updater_lock);
```

![Diagram](attachment:read_copy_update_animation.png)
Read-Copy Update Animation

- End of Grace Period

```c
wait_for_rcu();
```

Diagram:

- Header (CPU 0)
- Updater (CPU 0)
- Reader (CPU 1)
- Node A
- Node B (del)
- Node C

Connections:
- A to B
- B to C
- C to A
Read-Copy Update Animation

- Second Phase Complete

```c
kfree(p);
```

Diagram:
- Header
- Updater (CPU 0)
- Reader (CPU 1)
- A
- C

Connections:
- A to Header
- A to Updater (CPU 0)
- A to Reader (CPU 1)
- C to Reader (CPU 1)
What is a Grace Period?

- Time after which all pre-existing operations have completed.

\[ \phi_1 \quad \phi_2 \]

CPU 0

A → B → C → D

CPU 1

E → F → G

CPU 2

H → I → J → K

CPU 3

L → M → N
Detecting End of Grace Period

CPU 0
A \rightarrow u \rightarrow B \rightarrow C \rightarrow D

CPU 1
E \rightarrow F \rightarrow u \rightarrow G

CPU 2
H \rightarrow I \rightarrow J \rightarrow u \rightarrow K

CPU 3
L \rightarrow M \rightarrow N \rightarrow u

\phi_1 \quad \phi_2 \quad \phi_2 \text{ detected}
void wait_for_rcu()
{
    for (i = 0; i < n; i++) {
        run_on(i);
    }
}

- This simple implementation has some shortcomings: blocks caller, so:
  - Cannot be invoked from interrupt, with spinlock held, or with interrupts disabled
  - Sloooooow:
    - Cannot “batch” requests for grace periods.
    - Multiple context switches per grace period: high overhead
    - Can be stalled indefinitely by real-time tasks (see paper)
    - Can throttle update rate
Overview

• Why is this goal important?
• How the $#@#!! can readers safely access a changing data structure without locking???
  – Without writers needing a gazillion cycles to perform an update?
• Does this really help in real-world code?
One Grace Period for All...

\[
\phi_1 \quad \phi_2 \quad \phi_2 \text{ detected}
\]

CPU 0

\[\begin{align*}
A & \quad u & \quad B & \quad C & \quad D \\
E & \quad F & \quad u & \quad G \\
H & \quad I & \quad J & \quad u & \quad K \\
L & \quad M & \quad N & \quad u
\end{align*}\]
Amortized Grace Period

- Update code registers a callback
- Callbacks queued onto list
- Daemon periodically:
  - Removes all callbacks from list
  - Invokes wait_for_rcu()
  - Invokes all callbacks removed from list
- Allows batching, use from all execution contexts, and is much faster
Design to Detect Grace Period

Grace-Period Detection Requests

call_rcu(struct rcu_head *head,
    void (*func)(void *head))

List of struct rcu_head

wait_for_rcu(void)

free_pending_rcus(void)

Invoke destructor callback functions

Periodic Invocation
Code to Detect Grace Period

```c
struct rcu_head
{
    struct rcu_head *next;
    void (*func)(void *obj);
};
```

- The “next” pointer links the list together
- The “func” field is the destructor
void call_rcu(struct rcu_head *head,  
            void (*func)(void *head))
{
    unsigned long flags;
    head->destructor = func;
    spin_lock_irqsave(&rcu_lock, flags);
    head->next = rcu_list;
    rcu_list = head;
    spin_unlock_irqrestore(&rcu_lock, flags);
}

- Can be called with locks held, from interrupt handler, with interrupts suppressed
- Also much faster
  - Still subject to cacheline bouncing...
- And we still need to clean up rcu_list...
Code to Clean Up rcu_list

```c
void free_pending_rcus(void)
{
    struct rcu_head *list;
    unsigned long flags;
    spin_lock_irq(&rcu_lock, flags);
    list = rcu_list;
    rcu_list = NULL;
    spin_unlock_irq(&rcu_lock, flags);
    if (list) {
        wait_for_rcu();
        while (list) {
            struct rcu_head *next =
                list->next;
            list->destructor(list);
            list = next;
        }
    }
}
```
Read-Copy Update API

- `wait_for_rcu()`: Wait for a grace period.
- `call_rcu(struct rcu_head *head, void (*func)(void *head))`: Invoke the specified function at the end of a grace period.
- `kmalloc_rcu(size_t size, in flags)`: `kmalloc()`, with an `rcu_head` struct prepended.
- `kfree_rcu(void*obj, void(*destructor)(void*))`: `kfree()` after a grace period. Memory must be from `kmalloc_rcu()`.
Read-Copy Update API

• `vmalloc_rcu()` & `vfree_rcu()`: ditto.
• For `kmem_cache_alloc/_free()`, use `call_rcu()` specifying destructor function.

• Related APIs
  - `wmb()`: Needed to make sure that writes are seen in order by other CPUs
  • Existing API
  • Some question about the Alpha implementation: Alpha’s WMB instruction affects only the CPU that executes it...
Overview

• Why is this goal important?
• How the $#@#!! can readers safely access a changing data structure without locking???
  – Without writers needing a gazillion cycles to perform an update?
• Does this really help in real-world code?
if (i) {
    /* initialize new sets and bits */
}

nfds = xchg(&files->max_fdset, nfds);
new_openset = xchg(&files->open_fds,
                  new_openset);
new_execset = xchg(&files->close_on_exec,
                   new_execset);
write_unlock(&files->file_lock);
free_fdset(new_openset, nfds);
free_fdset(new_execset, nfds);
write_lock(&files->file_lock);

Original Code.  read_lock() used to access.
if (i) {
    /* initialize new sets and bits */
}
wmb();
files->open_fds = new_openset;
files->close_on_exec = new_execset;
wmb();
files->max_fdset = nfds;
spin_unlock(&files->file_lock);
wait_for_rcu();
spin_lock(&files->file_lock);
free_fdset(old_openset, old_nfds);
free_fdset(old_execset, old_nfds);

Read-Copy Update, Slow Version. No locks to access!!!
if (i) {
    /* initialize new sets and bits */
}

wmb();
files->open_fds = new_openset;
files->close_on_exec = new_execset;
wmb();
files->max_fdset = nfds;
free_fdset(old_openset, old_nfds);
free_fdset(old_execset, old_nfds);

kfree_rcu() instead of kfree()

Read-Copy Update, Fast Version. No locks to access!!!
“Chat” Benchmark  (M. Soni: http://lse.sourceforge.net/locking/files_struct_rcu.txt)
Other uses in Linux

- Module unloading patch for 2.4 kernel
  - Correctly handle module uses that race with unloading that module

- Hotplug CPU patch for 2.4 kernel
  - Ensure that all other CPUs are aware outgoing CPU is leaving before it is fully removed

- See paper for more details
Known Uses

- DYNIX/ptx (since 1993)
- Tornado/K42 (pervasive)
- Patches to Linux:
  - Module unloading
  - File descriptor management
  - Hotplug CPU support
Related Work

• Garbage-collector-based grace period
  - Kung & Lehman, 1980

• Timeout-based grace period
  - Jacobson, 1993

• Read-copy update with grace period left as an exercise to the reader
  - Manber & Ladner, 1984; Pugh, 1990

• Wait-free synchronization
  - Herlihy, 1993
Future Work

• Look at simple but fast implementations
• Continue investigating application to Linux
• Investigate applicability to user-level software
  – Real-time databases and systems
  – Reactive systems
• Analysis at high contention levels
• Formal description and correctness proofs
Availability

- Read-copy update may be freely used under GPL.
Summary

• Read-copy update can reduce complexity while improving performance and scaling
  - Works with rather than against Moore’s Law
• Key idea: break update into two phases:
  - Create updated version while leaving old version for ongoing operations (first phase)
  - Wait for a dynamically sized grace period
  - Do cleanup operations (second phase)
Conclusion

• Simple, high-performance and -scaling algorithms for read-mostly situations
  - Readers are not be required to acquire locks, execute atomic operations, or disable interrupts
    • Read-side code same as UP user-level implementation
    • Performance scales with CPU core clock rate, not with memory latency
  - Writers have to do a little more work
• Algorithms available for preemptive environments