Last Time: Synchronization

• Last time, discussed a variety of multithreading issues
  • Frequently have shared state manipulated by multiple threads
  • Usually solve this problem using some kind of mutual-exclusion mechanism, e.g. disabling interrupts, mutexes, semaphores, etc.
• Many examples of shared state within the OS kernel
  • Scheduler ready-queue, other queues (accessed concurrently on multicore systems)
  • Filesystem cache (shared across all processes on the system)
  • Virtual memory mapping (used by fault handlers and trap handlers)
• Frequently managed in linked lists (although other more sophisticated structures are often used)
• Frequently this state is read much more than it’s written
Example: `vm_area_struct` Lists

- Example: `vm_area_struct` list used for process memory

- Nodes contain many values describing memory regions
- Mostly used to resolve page faults and protection faults
- Also modified by trap handler, e.g. `mmap()`, `sbrk()` functions
Example Problem: Linked Lists

- How would we implement a linked list that supports concurrent access from multiple kernel control paths?

- Consider a simplified list type:
  - Each element contains several important fields, and a pointer to next node in the list

```c
typedef struct list_node {
    int a;
    int b;
    struct list_node *next;
} list_node;

list_node *head;
```

- Example list contents:
Example Problem: Linked Lists (2)

- Operations on our linked list:
  - Iterate over the list nodes, examining each one
    - e.g. to find relevant data, or to find a node that needs modified
  - Insert a node into the linked list
  - Modify a node in the linked list
  - Remove a node from the linked list

- All of these operations are straightforward to implement
  - Can imagine other similar operations, variants of the above

```c
typedef struct list_node {
    int a;
    int b;
    struct list_node *next;
} list_node;

list_node *head;
```
Linked List and Concurrent Access

- Should be obvious that our linked list will be corrupted if manipulated concurrently by different threads

- Example:
  - One thread is traversing the list, searching for the node with $a = 12$, so it can retrieve the current value of $b$
  - Another thread is inserting a new node into the list
Linked List and Concurrent Access (2)

- This scenario can fail in many different ways
- Writer-thread T₂ must perform several operations:
  ```c
  list_node *new = malloc(sizeof(list_node));
  new->a = 51;
  new->b = 24;
  new->next = p->next;
  p->next = new;
  ```
- We can try to specify a reasonable order...
- Really have no guarantees about how the compiler will order this. Or the CPU, for that matter.
Linked List and Concurrent Access (3)

- Operations that writer-thread T₂ must perform:
  
  ```c
  list_node *new = malloc(sizeof(list_node));
  new->a = 51;
  new->b = 24;
  new->next = p->next;
  p->next = new;
  ```

- These operations form a critical section in our code: must enforce exclusive access to the affected nodes during these operations.
Fixing Our Linked List

How do we avoid concurrency bugs in our linked list implementation?

An easy solution: use a single lock to guard the entire list

- Any thread that needs to read or modify the list must acquire the lock before accessing head

Design this solution to work from multiple kernel control paths, e.g.

- On a single-core system, trap handler and interrupt handlers simply disable interrupts while accessing the list
- On a multi-core system, use a combination of spin-locks and disabling interrupts to protect access to the list

typedef struct list_node {
    int a;
    int b;
    struct list_node *next;
} list_node;

list_node *head;
lock_t list_lock;
Fixing Our Linked List (2)

• How do we avoid concurrency bugs in our linked list implementation?
  • An easy solution: use a single lock to guard the entire list
    • Any thread that needs to read or modify the list must acquire the lock before accessing head
  
```
typedef struct list_node {
    int a;
    int b;
    struct list_node *next;
} list_node;

list_node *head;
lock_t list_lock;
```

• Why must readers also acquire the lock before reading??
  • Only way for the writer to ensure that readers won’t access the list concurrently, while it’s being modified 😞
Linked List: A Single Lock

- **What’s the obvious issue with this approach?**
- Readers shouldn’t ever block other readers!
  - (we know the list will mostly be accessed by readers anyway…)
  - It’s okay if writers hold exclusive access to the list while modifying it
    - (it would be better if multiple writers could concurrently modify independent sections of the list)

- This approach has very high **lock contention**
  - Threads spend a lot of time waiting to acquire the lock so they can access the shared resource
  - No concurrent access is allowed to the shared resource

```c
typedef struct list_node {
    int a;
    int b;
    struct list_node *next;
} list_node;

list_node *head;
lock_t list_lock;
```
Linked List: Improving Concurrency

- Ideally, readers should **never** block other readers
  - (we will accept the behavior that writers block everybody, for now)

- How can we achieve this?
- Can use a **read/write lock** instead of our simple lock
  - Multiple readers can acquire **shared** access to the lock: readers can access the shared resource concurrently without any issues
  - Writers can acquire **exclusive** access to the lock
- Two lock-request operations:
  - `read_lock(rwlock_t *lock)` – used by readers
  - `write_lock(rwlock_t *lock)` – used by writers
Using a read/write lock greatly increases concurrency and reduces lock contention.

Still a few annoying issues:

- Readers still must acquire a lock every time they access the shared resource
  - All threads incur a certain amount of **lock overhead** when they acquire the lock (in this case, CPU cycles)
  - And, it turns out this overhead can be **hundreds** of CPU cycles, even for efficiently implemented read/write locks!
Using a read/write lock greatly increases concurrency and reduces lock contention.

Still a few annoying issues:
Also, writers still block everybody.

Can we come up with a way to manipulate this linked list that doesn’t require writers to acquire exclusive access?

```c
typedef struct list_node {
    int a;
    int b;
    struct list_node *next;
} list_node;

list_node *head;
rwlock_t list_lock;
```
Linked List: Multiple Locks

• One approach for reducing lock contention is to decrease the **granularity** of the lock
  • i.e. how much data is the lock protecting?

• **Idea:** Introduce more locks, each of which governs a smaller region of data

• For our linked list, could put a read/write lock in each node
  • Threads must acquire many more locks to work with the list, which means that the locking overhead goes way up 😞
  • But, writers can lock only the parts of the list they are changing, which means we can reduce lock contention/increase concurrency

```c
typedef struct list_node {
    rwlock_t node_lock;
    int a;
    int b;
    struct list_node *next;
} list_node;

list_node *head;
```
Linked List: Multiple Locks (2)

- We need one more read/write lock, to guard the `head` pointer
  - Need to coordinate accesses and updates of `head` so that a thread doesn’t follow an invalid pointer!
  - If a thread needs to change what `head` points to, it needs to protect this with a critical section

- Now we have all the locks necessary to guard the list when it’s accessed concurrently

```c
typedef struct list_node {
    rwlock_t node_lock;
    int a;
    int b;
    struct list_node *next;
} list_node;

list_node *head;
rwlock_t head_lock;
```
Linked List: Multiple Locks (3)

- With multiple locks in our structure, must beware of the potential for deadlock...
  - Can easily avoid deadlock by requiring that all threads lock nodes in the same total order
    - Prevent “circular wait” condition
  - This is easy – it’s a singly linked list! Always lock nodes in order from head to tail.
  - This makes it a bit harder on writers
    - How does a writer know whether to acquire a read lock or a write lock on a given node?
    - Need to acquire a read lock first, examine the node, then release and reacquire a write lock if the node must be altered.

```c
typedef struct list_node {
    rwlock_t node_lock;
    int a;
    int b;
    struct list_node *next;
} list_node;

list_node *head;
rwlock_t head_lock;
```
Linked List: Multiple Locks, Example

- $T_1$ acquires a read-lock on head so it won’t change.
  - Then $T_1$ follows head to the first node, and acquires a read lock on this node so it won’t change.
  - *(and so forth)*

- This process of holding a lock on the current item, then acquiring a lock on the next item before releasing the current item’s lock, is called **crabbing**
  - As long as $T_1$ holds a read lock on the current node, and acquires read-lock on the next node before visiting it, it won’t be affected by other threads.
Linked List: Multiple Locks, Example (2)

• $T_2$ behaves in a similar manner:
  • $T_2$ acquires a read-lock on head so it won’t change.
  • Then $T_2$ follows head to the first node, and acquires a read lock on this node so it won’t change.
  • When $T_2$ sees that the new node must go after the first node, it can acquire a write-lock on the first node
    • Ensures its changes won’t become visible to other threads until lock is released
  • After $T_2$ inserts the new node, it can release its locks to allow other threads to see the changes
A critical question: **How long should each thread hold on to the locks it previously acquired in the list?**

If a thread releases locks on nodes after it leaves them, then other threads might change those nodes
- Does the thread need to be aware of values written by other threads, that appear earlier in the list?
- What if another thread completely changes the list earlier on?

If these scenarios are acceptable, then threads can release locks as soon as they leave a node
- (Often, it’s acceptable!)
Linked List: Holding Earlier Locks (2)

• A critical question: **How long should each thread hold on to the locks it previously acquired in the list?**
• If such scenarios are unacceptable, threads can simply hold on to all locks until they are finished with the list
  • Ensures that each thread will see a completely consistent snapshot of the list until the thread is finished with its task

• Even simple changes in how locks are managed can have significant implications…
Lock-Based Mutual Exclusion

- **Lock-based approaches have a lot of problems**
- Have to make design decisions about what granularity of locking to use
  - Coarse-granularity locking = lower lock overhead, but writers block everyone
  - Fine-granularity locking = much higher lock overhead, but can achieve more concurrency with infrequent writers in the mix
- More locks means more potential for deadlock to occur
- Locks make us prone to other issues like priority inversion (more on this in a few lectures)
- Can’t use locks in interrupt context anyway, unless we are very careful in how they are used!
Mutual Exclusion

- What is the fundamental issue we are trying to prevent?
  - Different threads seeing (or creating) inconsistent or invalid state
- Earlier example: writer-thread $T_2$ inserting a node
  ```
  list_node *new = malloc(sizeof(list_node));
  new->a = 51;
  new->b = 24;
  new->next = p->next;
  p->next = new;
  ```
- A big part of the problem is that we can’t guarantee the order or interleaving of these operations
  - Locks help us to sidestep this issue by guarding all the operations
Order of Operations

• What if we could impose a more intelligent ordering?
• When \( T_2 \) inserts a node:
  - **Step 1**: Prepare the new node, but **don’t** insert it into the list yet
    ```
    list_node *new = malloc(sizeof(list_node));
    new->a = 51;
    new->b = 24;
    new->next = p->next;
    ```
  - Last three operations can occur in any order. No one cares, because they aren’t visible to anyone.
• \( T_1 \) can go merrily along; \( T_2 \) hasn’t made any visible changes yet.
What if we could impose a more intelligent ordering?

When $T_2$ inserts a node:

- Step 2: Atomically change the list to include the new node
  
  ```c
  p->next = new;
  ```

  This is a single-word write. If the CPU can perform this atomically, then threads will either see the old version of the list, or the new version.

Result: Reader threads will never see an invalid version of the list!

For this to work, we must ensure these operations happen in the correct order.
Read-Copy-Update

- This mechanism is called **Read-Copy-Update** (RCU)
  - A lock-free mechanism for providing a kind of mutual exclusion
- All changes to shared data structures are made in such a way that concurrent readers *never* see intermediate state
  - They either see the old version of the structure, or they see the new version.
- Changes are broken into two phases:
  - If necessary, a copy is made of specific parts of the data structure. Changes take place on the copy; readers cannot observe them.
  - Once changes are complete, they are made visible to readers in a single atomic operation.
- In RCU, this atomic operation is always changing a pointer from one value to another value
  - e.g. \(T_2\) performs \(p->\text{next} = \text{new}\), and change becomes visible
Publish and Subscribe

- It’s helpful to think of changing the `p->next` pointer in terms of a publish/subscribe problem
- \(T_2\) operations:
  - **Step 1**: Prepare the new node
    ```c
    list_node *new = malloc(sizeof(list_node));
    new->a = 51;
    new->b = 24;
    new->next = p->next;
    ```
  - **Step 2**: Atomically change the list to include the new node
    ```c
    p->next = new;
    ```
- Before the new node is **published** for others to access, all initialization must be completed
- We can enforce this with a write memory barrier
  - Enforce that all writes before the barrier are completed before any writes after the barrier are started
  - (Also need to impose an optimization barrier for the compiler…)
Publish and Subscribe (2)

• Implement this as a macro:

```c
/* Atomically publish a value v to pointer p. */
/* smp_wmb() also includes optimization barrier. */
#define rcu_assign_pointer(p, v) ({
    smp_wmb(); (p) = (v);
})
```

• IA32 and x86-64 ISAs both guarantee that as long as the pointer-write is properly word-aligned (or dword-aligned), it will be atomic.

• (Even on multiprocessor systems!)

• $T_2$ operations become:

```c
list_node *new = malloc(sizeof(list_node));
new->a = 51;
new->b = 24;
new->next = p->next;
/* Publish the new node! */
rcu_assign_pointer(p->next, new);
```
Publish and Subscribe (3)

• $T_1$ needs to see the “current state” of the $p\rightarrow$next pointer (whatever that value might be when it reads it)

• Example: $T_1$ is looking for node with a specific value of $a$:

```c
list_node *p = head;
int b = -1;
while (p != NULL) {
    if (p->a == value) {
        b = p->b;
        break;
    }
    p = p->next;
}
return b;
```

• When $T_1$ reads $p\rightarrow$next, it is subscribing to the most recently published value
Publish and Subscribe (4)

- Example: \( T_1 \) is looking for node with a specific value of \( a \):
  
  ```c
  list_node *p = head;
  int b = -1;
  while (p != NULL) {
    if (p->a == value) {
      b = p->b;
      break;
    }
    p = p->next;
  }
  return b;
  ```

- Must ensure that the read of \( p->\text{next} \) is completed before any accesses to \( p->a \) or \( p->b \) occur
  
  - We could use a read memory barrier, but IA32 already ensures that this occurs, automatically
  
  - (Not all CPUs ensure this… DEC ALPHA CPU, for example…)
Again, encapsulate this “subscribe” operation in a macro:

```c
#define rcu_dereference(p) ({
    typeof(p) _value = ACCESS_ONCE(p);
    smp_read_barrier_depends();
    (_value);
})
```

- On IA32, `smp_read_barrier_depends()` is a no-op
  - On DEC ALPHA, it’s an actual read barrier
- `ACCESS_ONCE(x)` is a macro that ensures `p` is read directly from memory, not a register
  - (Usually generates no additional instructions)
- Subscribing to a pointer is very inexpensive. Nice!
Publish and Subscribe (6)

• Updated version of $T_1$ code:

```c
list_node *p = rcu_dereference(head);
int b = -1;
while (p != NULL) {
    if (p->a == value) {
        b = p->b;
        break;
    }
    p = rcu_dereference(p->next);
}
return b;
```

• So far, this is an extremely inexpensive mechanism!
  • Writers must sometimes perform extra copying, and use a write memory barrier.
  • But, we expect writes to occur infrequently. And, writers don’t block anyone anymore. (!!!)
  • Usually, readers incur zero overhead from RCU. (!!!)
Modifying a List Node

• Another example: change node with $a = 19$; set $b = 15$
  • Assume pointer to node being changed is in local variable $p$
  • Assume pointer to previous node is in $prev$
  • (Also, assume $rcu\_dereference()$ was used to navigate to $p$)
• Can’t change the node in place; must make a copy of it
  
  ```c
  copy = malloc(sizeof(list_node));
  copy->a = p->a;
  copy->b = 15;
  copy->next = p->next;
  rcu_assign_pointer(prev->next, copy);
  ```
Modifying a List Node (2)

- Since `rcu_assign_pointer()` atomically publishes the change, readers must fall into one of two categories:
  - Readers that saw the old value of `prev->next`, and therefore end up at the old version of the node
  - Readers that see the new value of `prev->next`, and therefore end up at the new version of the node

- All readers will see a valid version of the shared list
  - And, we achieve this with much less overhead than with locking!
  - (The writer has to work a bit harder…)

```
head
  a = 5
  b = 31
  next

prev
  a = 19
  b = 2
  next

p

copy

a = 19
b = 2
next

a = 19
b = 2
next

a = 12
b = 6
next
```
Modifying a List Node (3)

• Are we finished?
  
  ```
  copy = malloc(sizeof(list_node));
  copy->a = p->a;
  copy->b = 15;
  copy->next = p->next;
  rcu_assign_pointer(prev->next, copy);
  ```

• Thread must deallocate the old node, or else there will be a memory leak
  ```
  free(p);
  ```

• Problems?
  • If a reader saw the old version of `prev->next`, they may still be using the old node!
Reclaiming Old Data

• The hardest problem in RCU is ensuring that old data is only deleted after all readers have finished with it
• How do we tell that all readers have actually finished?

• Define the concept of a read-side critical section:
  • A reader enters a read-side critical section when it reads an RCU pointer (`rcu_dereference()`)
  • A reader leaves the read-side critical section when it is no longer using the RCU pointer

• Require that readers explicitly denote the start and end of read-side critical sections in their code:
  • `rcu_read_lock()` starts a read-side critical section
  • `rcu_read_unlock()` ends a read-side critical section
Read-Side Critical Sections

• Update $T_1$ to declare its read-side critical section:

```c
rcu_read_lock();  /* Enter read-side critical section */
list_node *p = rcu_dereference(head);
int b = -1;
while (p != NULL) {
    if (p->a == value) {
        b = p->b;
        break;
    }
    p = rcu_dereference(p->next);
}
rcu_read_unlock();  /* Leave read-side critical section */
return b;
```
Read-Side Critical Sections (2)

- A critical constraint on read-side critical sections:
  - Readers **cannot** block / sleep inside read-side critical sections!

- Should be obvious that $T_1$ follows this constraint:

```c
rcu_read_lock(); /* Start read-side critical section */
list_node *p = rcu_dereference(head);
int b = -1;
while (p != NULL) {
    if (p->a == value) {
        b = p->b;
        break;
    }
    p = rcu_dereference(p->next);
}
rcu_read_unlock(); /* End read-side critical section */
return b;
```
Read-Side Critical Sections (3)

- Can use read-side critical sections to define when old data may be reclaimed
- Each reader’s interaction with shared data structure is contained entirely within its read-side critical section
  - Each reader’s arrow starts with a call to `rcu_read_lock()`, and ends with `rcu_read_unlock()`
Read-Side Critical Sections (4)

• Writer publishes a change to the data structure with a call to `rcu_assign_pointer()`
  • Divides readers into two groups – readers that might see the old version, and readers that cannot see the old version
• What readers might see the old version of the data?
  • Any reader that called `rcu_read_lock()` before `rcu_assign_pointer` is called

```
rcu_assign_pointer()
```

```
Reader 1
Reader 2
Reader 3
Reader 4
Reader 5
Reader 6
Reader 7
```

Writer: Replace

`time`
Read-Side Critical Sections (5)

• When can the writer reclaim the old version of the data?
• After all readers that called `rcu_read_lock()` before `rcu_assign_pointer()` have also called `rcu_read_unlock()`
• This is the *earliest* that the writer may reclaim the old data; it is also allowed to wait longer (no cost except that resources are still held)
• Time between release and reclamation is called the **grace period**

```
rcu_assign_pointer()
```

![Diagram showing the timing of read and write operations.
Reader 1, Reader 2, Reader 3, Reader 4, Reader 5, Reader 6, Reader 7, Writer: Replace Grace Period Reclaim]
End of Grace Period

• Writer must somehow find out when grace period is over
  • Doesn’t have to be a precise determination; can be approximate, as long as writer can’t think it’s over before it’s actually over
• Encapsulate this in the `synchronize_rcu()` operation
  • This call blocks the writer until the grace period is over
• Updating our writer’s code:

```c
    copy = malloc(sizeof(list_node));
    copy->a = p->a;
    copy->b = 15;
    copy->next = p->next;
    rcu_assign_pointer(prev->next, copy);

    /* Wait for readers to get out of our way... */
    synchronize_rcu();
    free(p);
```
End of Grace Period (2)

- Updated diagram with call to `synchronize_rcu()`

- But how does this actually work?
End of Grace Period (3)

- Recall: readers are not allowed to block or sleep when inside a read-side critical section
- What is the maximum number of readers that can be inside read-side critical sections at any given time?
  - Same as the number of CPUs in the system
  - If a reader is inside its read-side critical section, it must also occupy a CPU

```c
rcu_assign_pointer()
```

![Diagram](image)
End of Grace Period (4)

- Recall: readers are not allowed to block or sleep when inside a read-side critical section.
- Also, require that the operating system cannot preempt a kernel thread that’s currently inside a read-side critical section.
  - Don’t allow OS to context-switch away from a thread in a read-side critical section.
  - In other words, don’t allow kernel preemption during the read-side critical section.

```plaintext
rcu_assign_pointer()

Reader 1 ➔ Reader 5

Reader 2

Reader 3 ➔ Reader 6

Reader 4

Reader 7

Writer: Replace Grace Period Synchronize_rcu() Reclaim

time
```
End of Grace Period (5)

- Recall: readers are not allowed to block or sleep when inside a read-side critical section
- If a CPU executes a context-switch, then we know the kernel-thread completed any read-side critical section it might have been in...
- Therefore, `synchronize_rcu()` can simply wait until at least one context-switch has occurred on every CPU in the system
  - Gives us an upper bound on the length of the grace period... Good enough! 😊
Completing the RCU Implementation

• Now we know enough to complete RCU implementation
• `synchronize_rcu()` waits until at least one context-switch has occurred on each CPU
  ```c
  void synchronize_rcu() {
    int cpu;
    for_each_online_cpu(cpu)
      run_on(cpu);
  }
  ```
• `run_on()` causes the kernel thread to run on a specific processor
• Can be implemented by setting kernel thread’s processor-affinity, then yielding the current CPU
• Once the kernel thread has switched to every processor, at least one context-switch has definitely occurred on every CPU (duh!)
Completing the RCU Implementation (2)

- On a single-processor system, `synchronize_rcu()` is a no-op (!!!)
  - `synchronize_rcu()` might block; therefore it cannot be called from within a read-side critical section
  - Any read-side critical section started before `synchronize_rcu()` was called, must have already ended at this point
  - Therefore, since `synchronize_rcu()` is running on the CPU, the grace period is already over, and the old data may be reclaimed
Completing the RCU Implementation (3)

- `read_lock()` and `read_unlock()` are very simple:
  - Since `synchronize_cpu()` uses context-switches to tell when grace period is over, these functions don’t actually have to do any bookkeeping (!!!)
- On a multicore system, or an OS with kernel preemption:
  - Must enforce constraint that readers cannot be switched away from while inside their read-side critical section
    ```c
    void read_lock() {
        preempt_disable(); /* Disable preemption */
    }
    void read_unlock() {
        preempt_enable(); /* Reenable preemption */
    }
    ```
  - `(preempt_disable() and preempt_enable()) simply increment or decrement preempt_count; see Lecture 9)
Completing the RCU Implementation (4)

- On a single-processor system with an OS that doesn’t allow kernel preemption:
  - (Recall: this means all context-switches will be scheduled context-switches)
  - In this case, `read_lock()` and `read_unlock()` don’t have to do anything!
    - Already have a guarantee that nothing can cause a context-switch away from the kernel thread inside its read-side critical section

- The “implementation” also becomes a no-op:
  ```c
  #define read_lock()
  #define read_unlock()
  ```
Results: The Good

- RCU is a very sophisticated mechanism for supporting concurrent access to shared data structures
  - Conceptually straightforward to understand how to implement readers and writers
  - Understanding how it works is significantly more involved…
- Doesn’t involve any locks (!!!):
  - Little to no lock overhead, no potential for deadlocks, no priority-inversion issues with priority scheduling
- Extremely lightweight
  - In common scenarios, many RCU operations either reduce to a single instruction, or a no-op
  - Only requires a very small number of clocks; far fewer than acquiring a lock
Entire RCU Implementation

/** RCU READER SUPPORT FUNCTIONS **/

/* Enter read-side critical section */
void read_lock(void) {
    preempt_disable();
}

/* Leave read-side critical section */
void read_unlock(void) {
    preempt_enable();
}

/* Subscribe to pointer p's value */
/* smp_wmb() includes opt.barrier */
#define rcu_dereference(p) ({
    typeof(p) _v = ACCESS_ONCE(p);
    smp_read_barrier_depends();
    (_value); })

/** RCU WRITER SUPPORT FUNCTIONS **/

/* Publish a value v to pointer p */
/* smp_wmb() includes opt.barrier */
#define rcu_assign_pointer(p, v) ({
    smp_wmb(); (p) = (v); })

/* Wait for grace period to end */
void synchronize_rcu(void) {
    int cpu;
    for_each_online_cpu(cpu)
        run_on(cpu);
}
Results: The Bad and the Ugly

- RCU is only useful in very specific circumstances:
  - Must have many more readers than writers
  - Consistency must not be a strong requirement
    - Under RCU, readers may see a mix of old and new versions of data, or even only old data that is about to be reclaimed

- If either of these conditions isn’t met, may be much better to rely on more standard lock-based approaches
- Surprisingly, many parts of Linux satisfy the above circumstances, and RCU is becoming widely utilized
Abstract

Read-copy update (RCU) is a scalable high-performance synchronization mechanism implemented in the Linux kernel. RCU's novel properties include support for concurrent reading and writing, and highly optimized inter-CPU synchronization. Since RCU's introduction into the Linux kernel over a decade ago its usage has continued to expand. Today, most kernel subsystems use RCU. This paper discusses the requirements that drove the development of RCU, the design and API of the Linux RCU implementation, and how kernel developers apply RCU.

1 Introduction

The first Linux kernel to include multiprocessor support is just over 15 years old. This kernel provided support for concurrently running applications, but serialized all execution in the kernel using a single lock. Concurrently executing applications that frequently invoked the kernel performed poorly.

Today the single kernel lock is gone, replaced by highly concurrent kernel subsystems. Kernel intensive applications that would have performed poorly on dual-processor machines 15 years ago, now scale and perform well on multicore machines [2].

Kernel developers have used a variety of techniques to improve concurrency, including fine-grained locks, lock-free data structures, per-CPU data structures, and read-copy-update (RCU), the topic of this paper. Uses of the RCU API have increased from none in 2002 to over 6500 in 2013 (see Figure 1). Most major Linux kernel subsystems use RCU as a synchronization mechanism. Linus Torvalds characterized a recent RCU-based patch to the virtual file system "as seriously good stuff" because developers were able to use RCU to remove bottlenecks affecting common workloads [22]. RCU is not unique to Linux (see [6, 12, 17] for other examples), but Linux's wide variety of RCU usage patterns is, as far as we know, unique among the commonly used kernels. Understanding RCU is now a prerequisite for understanding the Linux implementation and its performance.

The success of RCU is, in part, due to its high performance in the presence of concurrent readers and updaters. The RCU API facilitates this with two relatively simple primitives: readers access data structures within RCU read-side critical sections, while updaters use RCU synchronization to wait for all pre-existing RCU read-side critical sections to complete. When combined, these primitives allow threads to concurrently read data structures, even while other threads are updating them.

This paper describes the performance requirements that led to the development of RCU, gives an overview of the RCU API and implementation, and examines how kernel developers have used RCU to optimize kernel performance. The primary goal is to provide an understanding of the RCU API and how to apply it.

The remainder of the paper is organized as follows. Section 2 explains the important requirements for production-quality RCU implementations. Section 3 gives an overview of the RCU API and implementation, and Section 4 examines how kernel developers have used RCU to optimize kernel performance. Section 5 concludes the paper and discusses future work.

Figure 1: The number of uses of the RCU API in Linux kernel code from 2002 to 2013.
Figure 8: The overhead of entering and completing an RCU critical section, and acquiring and releasing a read-write lock.

A key difference between RCU and read-write locking is that RCU supports concurrent reading and writing of the same data while read-write locking enforces mutual exclusion. As a result, concurrent operations on an RCU protected data structure can yield results that a read-write lock would prevent. In the example above, suppose two threads simultaneously add processes A and B to different buckets in the table. A concurrently executing reading thread searching for process A then process B, might find process A, but not process B. Another concurrently executing reader searching for process B then A, might find process B, but not process A. This outcome is valid, but could not occur if the PID table used read-write locks.

Developers considering using RCU must reason about requirements of their application to decide if the additional orderings allowed by RCU, but disallowed by read-write locks, are correct. In addition to the PID table, other important kernel subsystems, such as the directory cache, networking routing tables, the SELinux access vector cache, and the System V IPC implementation, use RCU as an alternative to read-write locks. A tentative conclusion to draw from RCU's widespread use in Linux is that many kernel subsystems are either able to tolerate additional orderings allowed by RCU or use the techniques described in the next section to avoid problematic orderings.

Figure 9: Pseudocode for the Linux PID table implemented using RCU as an alternative to read-write locks.

After calling `pid_lookup`, a thread calls `pid_free` to release its reference to the process.

**Algorithmic Transformations**

Since RCU does not force mutual exclusion between readers and updaters, mechanical substitution of RCU for reader-writer locking can change the application's semantics. Whether this change violates correctness depends on the specific correctness properties required. Experience in the Linux kernel has uncovered a few common scenarios in which the changes in semantics are problematic, but are handled by the techniques described below. The following subsections discuss three commonly used techniques, explaining why they are needed, how they are applied, and where they are used.
RCU Implementation Notes

- There are much more advanced implementations of RCU
- RCU discussed today is known as “Classic RCU”
  - Many refinements to the implementation as well, offering additional features, and improving performance and efficiency
  - Our implementation is a “toy implementation,” but it still works
  - (Also doesn’t support multiple writers accessing the same pointer; need to use locks to prevent this, so it gets much slower…)
- SRCU (Sleepable RCU) allows readers to sleep inside their read-side critical sections
  - Also preemption of kernel threads inside read-side critical sections
- Preemptible RCU also supports readers suspending within their read-side critical sections
References

• For everything you could ever want to know about RCU:
  • Paul McKenney did his PhD research on RCU, and has links to an extensive array of articles, papers and projects on the subject
  • http://www2.rdrop.com/users/paulmck/RCU/

• Most helpful/accessible resources:
  • What is RCU, Really? (3-part series of articles)
    • http://www.rdrop.com/users/paulmck/RCU/whatisRCU.html
  • What Is RCU? (PDF of lecture slides)
  • User-Level Implementations of Read-Copy Update
    • http://www.rdrop.com/users/paulmck/RCU/urcu-main-accepted.2011.08.30a.pdf (actual article)
References (2)

• Andrei Alexandrescu has also written a few good articles:
  • Lock-Free Data Structures (big overlap with many RCU concepts)
    • http://www.drdobbs.com/lock-free-data-structures/184401865
  • Lock-Free Data Structures with Hazard Pointers