Kernel Synchronization III

Dongyoon Lee – Lecture notes from Dr. Min @ VT
Summary of last lectures

• Tools: building, exploring, and debugging Linux kernel

• Core kernel infrastructure
  • syscall, module, kernel data structures

• Process management & scheduling

• Interrupt & interrupt handler

• Kernel synchronization: concepts and key APIs
Today: kernel synchronization III

- Memory ordering and memory barriers
- Read-copy-update (RCU)
Ordering and barriers

• Memory reads (load) and write (store) operations can be reordered for performance reasons
  • by the compiler at compile time: compiler optimization
  • by the CPU at run time: TSO (total store ordering), PSO (partial store ordering)
/* Following code can be reordered
* by a compiler (optimization) or a processor (out-of-order execution)
*
* Your code            Compiled code
* ============         ============= */
a = 4;
b = 5;
a = 4;
b = 5;

/* Following code will never be reordered because
* there is a dependency between a and b.
*
* Your code            Compiled code
* ============         ============= */
a = 1;
b = a;
a = 1;
b = a;
Memory barriers

- Instruct a compiler or a processor not to reorder instructions around a given point
- `barrier()` (a.k.a., compiler barrier)
  - Prevents the compiler from reordering stores or loads across the barrier

```c
/* Compiler does not reorder store operations of a and b
 * However, a processor may reorder the store operations for performance */
a = 4;
barrier();
b = 5;
```
Memory barrier instructions

- \texttt{rmb()} : prevents loads from being reordered across the barrier
- \texttt{wmb()} : prevents stores from being reordered across the barrier
- \texttt{mb()} : prevents loads and stores from being reordered across the barrier
- \texttt{read\_barrier\_depends()} : prevent data-dependent loads to be reordered across the barrier
  - On some architectures, it is much faster than \texttt{rmb()} because it is not needed and is, thus, a \texttt{noop}
Memory barrier example

• Initial value: \( a = 1 \) and \( b = 2 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 3; )</td>
<td>( c = b; )</td>
</tr>
<tr>
<td>( \text{mb}(); )</td>
<td>( \text{rmb}(); )</td>
</tr>
<tr>
<td>( b = 4; )</td>
<td>( d = a; )</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
</tr>
</tbody>
</table>

• \( \text{mb}() \) ensures that \( a \) is written before \( b \)
• \( \text{rmb}() \) ensures that \( b \) is read before \( a \)
Memory barrier example

• Initial value: \( a = 1, b = 2 \), and \( p = &b \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 3; )</td>
<td>—</td>
</tr>
<tr>
<td>( mb(); )</td>
<td>—</td>
</tr>
<tr>
<td>( p = &amp;a; )</td>
<td>( pp = p; )</td>
</tr>
<tr>
<td>—</td>
<td>( read_barrier_depends(); )</td>
</tr>
<tr>
<td>—</td>
<td>( b = *pp; )</td>
</tr>
</tbody>
</table>

• \( mb() \) ensures that \( a \) is written before \( p \)

• \( read\_barrier\_depends() \) is sufficient because the load of \( *pp \) depends on the load of \( p \)
## Memory barrier API for multi-processor

<table>
<thead>
<tr>
<th>Barrier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>smp_rmb()</code></td>
<td>Provides an <code>rmb()</code> on SMP, and on UP provides a <code>barrier()</code></td>
</tr>
<tr>
<td><code>smp_read_barrier_depends()</code></td>
<td>Provides a <code>read_barrier_depends()</code> on SMP, and provides a <code>barrier()</code> on UP</td>
</tr>
<tr>
<td><code>smp_wmb()</code></td>
<td>Provides a <code>wmb()</code> on SMP, and provides a <code>barrier()</code> on UP</td>
</tr>
<tr>
<td><code>smp_mb()</code></td>
<td>Provides an <code>mb()</code> on SMP, and provides a <code>barrier()</code> on UP</td>
</tr>
<tr>
<td><code>barrier()</code></td>
<td>Prevents the compiler from optimizing stores or loads across the barrier</td>
</tr>
</tbody>
</table>

- On SMP kernel they are defined as the usual memory barriers
- On UP kernel they are defined only as a compiler barrier
Recap: Synchronization primitives

- Protect shared data from concurrent access
- Non-sleeping (non-blocking) synchronization primitives
  - atomic operation, spinlock, reader-write lock (rwlock), sequential lock (seqlock)
- Sleeping (blocking) synchronization primitives
  - semaphore, mutex, completion variable, wait queue
Recap: spinlock

- Implement by mutual exclusion
Recap: rwlock

- Allow multiple readers
- Mutual exclusion between readers and a writer
- Linux rwlock is a reader-preferred algorithm
Recap: seqlock

- Consistent mechanism without starving writers

![Diagram showing seqlock](image-url)

- First trial
- Reader with seq = 0
- Writer with seq = 1
- Time
  - seq = 0
  - seq = 2
  - seq = 2
- Same seq with start point
- Start with even seq
- retry
- UPDATE
Read-Copy-Update (RCU) design principle

- RCU supports concurrency between multiple readers and a single writer.
  - A writer does not block readers!
  - Allow multiple readers with almost zero overhead
  - Optimize for reader performance

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>READ</td>
<td>READ</td>
<td>READ</td>
</tr>
<tr>
<td>READ</td>
<td>READ</td>
<td>READ</td>
<td>READ</td>
</tr>
<tr>
<td>READ</td>
<td>READ</td>
<td>READ</td>
<td>READ</td>
</tr>
<tr>
<td>READ</td>
<td>READ</td>
<td>READ</td>
<td>READ</td>
</tr>
<tr>
<td>READ</td>
<td>UPDATE</td>
<td>READ</td>
<td>READ</td>
</tr>
</tbody>
</table>
Read-Copy-Update (RCU)

- Only require locks for writes; carefully update data structures so readers see consistent views of data all the time.
- RCU ensures that reads are coherent by maintaining multiple version of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete.
- Widely-used for read-mostly data structures
  - Directory entry cache, DNS name database, etc.
Who developed RCU?

- Paul McKenney @ IBM
How much is RCU used in the Linux kernel?

- Source: [RCU Linux Usage](#)
RWLock-based linked list

- Even using a scalable rwlock, readers and a writer cannot concurrently access the list.
RCU-based linked list

- Allow concurrent access of readers
RCU-based linked list

- A writer copies an element first
RCU-based linked list

• And then it updates the element
RCU-based linked list

- And then it makes its change public by updating the next pointer of its previous. New readers will traverse 100 instead of 4.
RCU-based linked list

- Do not free the old node, 4, until any reader accesses it.
RCU-based linked list

- When it is guaranteed that there is no reader accessing the old node, free the old node.
RCU API

/* linux/include/linux/rcupdate.h */

/* Mark the beginning of an RCU read-side critical section */
void rcu_read_lock(void);

/* Mark the end of an RCU read-side critical section */
void rcu_read_unlock(void);

/* Assign to RCU-protected pointer: p = v */
/* @p: pointer to assign to */
/* @v: value to assign (publish) */
#define rcu_assign_pointer(p, v) ..

/* Fetch RCU-protected pointer for dereferencing */
/* @p: The pointer to read, prior to dereferencing */
#define rcu_dereference(p) ..

/* Queue an RCU callback for invocation after a grace period. */
/* @head: structure to be used for queueing the RCU updates. */
/* @func: actual callback function to be invoked after the grace period */
void call_rcu(struct rcu_head *head, rcu_callback_t func);

/* Wait until quiescent states */
void synchronize_rcu(void);
Replace rwlock by RCU

/* RWLock */
1 struct el {
2   struct list_head lp;
3   long key;
4   int data;
5   /* Other data fields */
6 };
7 DEFINE_RWLOCK(listlock);
8 LIST_HEAD(head);

/* RCU */
1 struct el {
2   struct list_head lp;
3   long key;
4   int data;
5   /* Other data fields */
6 };
7 DEFINE_SPINLOCK(listlock);
8 LIST_HEAD(head);
Replace rwlock by RCU

/* RWLock */
1 int search(long key, int *result) {
  struct el *p;
  read_lock(&listlock);
  list_for_each_entry(p,&head,lp) {
    if (p->key == key) {
      *result = p->data;
      read_unlock(&listlock);
      return 1;
    }
  }
  read_unlock(&listlock);
  return 0;
}

/* RCU */
1 int search(long key, int *result) {
  struct el *p;
  rcu_read_lock();
  list_for_each_entry_rcu(p,&head,lp) {
    if (p->key == key) {
      *result = p->data;
      rcu_read_unlock();
      return 1;
    }
  }
  rcu_read_unlock();
  return 0;
}
Replace rwlock by RCU

/* RWLock */
1 int delete(long key)
2 {
3   struct el *p;
4
5   write_lock(&listlock);
6   list_for_each_entry(p, &head, lp) {
7     if (p->key == key) {
8       list_del(&p->lp);
9       write_unlock(&listlock);
10       kfree(p);
11       return 1;
12     }
13   }
14   write_unlock(&listlock);
15   return 0;
16 }

/* RCU */
1 int delete(long key)
2 {
3   struct el *p;
4
5   spin_lock(&listlock);
6   list_for_each_entry(p, &head, lp) {
7     if (p->key == key) {
8       list_del_rcu(&p->lp);
9       spin_unlock(&listlock);
10       synchronize_rcu();
11       kfree(p);
12       return 1;
13     }
14   }
15   spin_unlock(&listlock);
16   return 0;
17 }
RCU Primer

Lock-free reads + Single pointer update + Delayed free

```
length() {
    rcu_read_lock(); {
        p=rcu_dereference(head); //p=head
        for(i=0;p;p=p->next,i++) ;
    } rcu_read_unlock();
    return i;
}
```

```
pop_n(n) {
    for(p=head;p&&n;p=p->next,n--)
        call_rcu(free, p);
    rcu_assign_pointer(head,p); //head=p
}
```
RCU Primer

Lock-free reads  +  Single pointer update  +  Delayed free

• No locks, no barriers
• `rcu_read_lock()` just sets the status of a thread “reading” RCU data.
RCU Primer

Lock-free reads + Single pointer update + Delayed free

- No locks, no barriers
- `rcu_read_lock()` just sets the status of a thread “reading” RCU data.

```c
length() {
    rcu_read_lock(); {
        p=rcu_dereference(head); //p=head
        for(i=0;p;p=p->next,i++)
    } rcu_read_unlock();
    return i;
}
```

```c
call_rcu(free, p);
rcu_assign_pointer(head, p); //head=p
```

- Update exactly one pointer, which is atomic.
RCU Primer

Lock-free reads + Single pointer update + Delayed free

head

3

- No locks, no barriers
- `rcu_read_lock()` just sets the status of a thread “reading” RCU data.
- Update exactly one pointer, which is atomic.
- Free delayed until all readers return (e.g., by waiting for all CPU’s to schedule)

```c
length() { 
    rcu_read_lock(); { 
        p=rcu_dereference(head); //p=head
        for(i=0;p;p=p->next,i++) ;
    } rcu_read_unlock();
    return i;
}
```

```c
pop_n(n) { 
    for(p=head;p&&n;p=p->next,n--)
        call_rcu(free, p);
    rcu_assign_pointer(head,p); //head=p
}
```
Efficient and scalable grace period detection is a key challenge

Some obvious solutions, such as reference counting, never work.
Atomic increment does not scale
Toy RCU implementation

```c
static inline void rcu_read_lock(void)
{
    preempt_disable();
}

static inline void rcu_read_unlock(void)
{
    preempt_enable();
}

#define rcu_assign_pointer(p, v)    ({
    smp_wmb();
    ACCESS_ONCE(p) = (v);
})

#define rcu_dereference(p)      ({
    typeof(p) _value = ACCESS_ONCE(p);
    smp_read_barrier_depends(); /* nop on most architectures */
    (_value);
})
```
Toy RCU implementation

```c
void call_rcu(void (*callback) (void *), void *arg) {
    /* add callback/arg pair to a list */
}

void synchronize_rcu(void) {
    int cpu, ncpus = 0;

    for_each_cpu(cpu)
        schedule_current_task_to(cpu);

    for each entry in the call_rcu list
        entry->callback (entry->arg);
}
```
/* linux/include/linux/rculist.h */
/* Circular doubly-linked list */

/* Add a new entry to rcu-protected list
 * @new: new entry to be added
 * @head: list head to add it after */
void list_add_rcu(struct list_head *new, struct list_head *head);

/* Deletes entry from list without re-initialization
 * @entry: the element to delete from the list. */
void list_del_rcu(struct list_head *entry);

/* Replace old entry by new one
 * @old : the element to be replaced
 * @new : the new element to insert */
void list_replace_rcu(struct list_head *old, struct list_head *new);

/* Iterate over rcu list of given type
 * @pos: the type * to use as a loop cursor.
 * @head: the head for your list.
 * @member: the name of the list_head within the struct. */
#define list_for_each_entry_rcu(pos, head, member) ..
RCU hlist

/* linux/include/linux/rculist.h */
/* Non-circular doubly-linked list */

/* Adds the specified element to the specified hlist,
 * while permitting racing traversals.
 * @n: the element to add to the hash list.
 * @h: the list to add to. */
void hlist_add_head_rcu(struct hlist_node *n, struct hlist_head *h);

/* Replace old entry by new one
 * @old : the element to be replaced
 * @new : the new element to insert */
void hlist_replace_rcu(struct hlist_node *old, struct hlist_node *new);

/* Deletes entry from hash list without re-initialization
 * @n: the element to delete from the hash list. */
void hlist_del_rcu(struct hlist_node *n);

/* Iterate over rcu list of given type
 * @pos: the type * to use as a loop cursor.
 * @head: the head for your list.
 * @member: the name of the hlist_node within the struct. */
#define hlist_for_each_entry_rcu(pos, head, member) ...
Limitations of RCU

• Do not provide a mechanism to coordinate multiple writers
  • Most RCU-based algorithms end up using spinlock to prevent concurrent write operations
• All modification should be a single-pointer-update.
  • This is challenging!
Further readings

• Read-log-update: a lightweight synchronization mechanism for concurrent programming, SOSP15
• Is Parallel Programming Hard, And, If So, What Can You Do About It?
• Structured Deferral: Synchronization via Procrastination
• Introduction to RCU Concepts
• LWN: What is RCU, Fundamentally?
• Notes on Read-Copy Update
• Tiny Little Things for Manycore Scalability: Scalable Locking and Lockless Data Structures