Linux Kernel Training. Lecture 13

Concurrency and race conditions. Part 2

Oleksandr Redchuk
<oleksandr.redchuk@gmail.com>

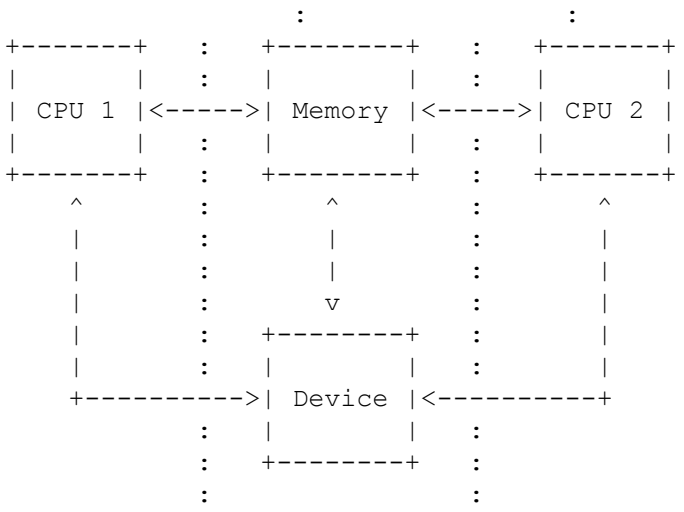
May 4, 2020. GlobalLogic

Kernel memory-ordering model

- In a perfect world one might assume that order in which their code accesses data from memory always in the same order as in source code. This is partially true for UP systems, but only partially due to ordering considerations when interacting with hardware via memory subsystem (e.g. DMA, MMIO, etc).
- On SMP things get changed dramatically where same memory region might be accessed from multiple execution flows simultaneously for non-atomic operations. Things get difficulty even more and more with improvement to CPU implementations such as out-of-order execution, cache subsystem etc.
- On the other hand compilers not stay behind from process of in-memory data access optimization. They often too smart to do worse things.
- In may situations reordering from either compiler or target system isn’t problem, but there are many cases where it is:
  - Implementing synchronization primitives or working directly with shared data on SMP.
  - Accessing device-mapped memory (e.g. DMA, MMIO, etc)

Kernel memory-ordering model (cont.)

Consider abstract memory access model



Each CPU executes a program that generates memory access operations.

In the abstract CPU, memory operation ordering is very relaxed, and a CPU may actually perform the memory operations in any order it likes, provided program causality appears to be maintained.

Similarly, the compiler may also arrange the instructions it emits in any order it likes, provided it doesn't affect the apparent operation of the program.

In general **there is no guarantee that code you wrote to access data in memory would be compiled and executed in order you expect**: either compiler by optimizing things or execution on target system easily could get things reordered finally changing expected behaviour from source code point to unpredictable.

Kernel memory-ordering model (cont.)

Reordering due to cache miss

`*r0 = r4;`	`STR R4, [R0]  @ Access 1`	Access **1** goes into write buffer
`r5 = *r1++;`	`LDR R5, [R1], #4 @ Access 2`	Access **2** causes a cache lookup which misses
`r6 = r2[2];`	`LDR R6, [R2, #8] @ Access 3`	Access **3** causes a cache lookup which hits
		Access **3** returns data into ARM register (*hit-under-miss behavior*)
		Cache linefill triggered by Access **2** returns data
		Memory store triggered by Access **1** is performed

Source: ARM® Cortex-A Series. Version 4.0. Programmer's Guide. Chapter 10. Memory Ordering

Kernel memory-ordering model (cont.)

ARMv7-A: Three mutually exclusive memory types are defined. All regions of memory are configured as one of these three types:

- Strongly-ordered
- Device
- Normal

Access A1 occurs before A2 in program code, but writes can be issued out of order.

A1 \ A2	Normal	Device	Strongly-ordered
Normal	No order enforced	No order enforced	No order enforced
Device	No order enforced	Issued in program order	Issued in program order
Strongly-ordered	No order enforced	Issued in program order	Issued in program order

Thus, the stores committed by a CPU to the memory system may not be perceived by the loads made by another CPU in the same order as the stores were committed.

Source: ARM® Cortex-A Series. Version 4.0. Programmer's Guide. Chapter 10. Memory Ordering

Kernel memory-ordering model (cont.)

- Consider the following sequence of events:

```
CPU 1                          CPU 2
===============                ===============
{ A == 1; B == 2 }
A = 3;                         x = B;
B = 4;                         y = A;
```

- The set of accesses as seen by the memory system in the middle can be arranged in 24 different combinations:

```
STORE A=3,   STORE B=4,   y=LOAD A->3,   x=LOAD B->4
STORE A=3,   y=LOAD A->3, STORE B=4,     x=LOAD B->4
STORE B=4, ...
...
```

  - Obviously this can thus result in four different combinations of values:
    `x == 2, y == 1 ; x == 2, y == 3 ; x == 4, y == 1 ; x == 4, y == 3`
  - That's why STORE A=3 on CPU1 does not mean that after y=LOAD A returns 3. It is obviously can be any value stored in A.

Kernel memory-ordering model (cont.)

- There are some minimal guarantees that may be expected of a CPU:
  - On any given CPU, dependent memory accesses will be issued in order, with respect to itself. This means that the CPU will issue the **load** operations always in that order:

```
Q = LOAD P, D = LOAD *Q
```

  - Overlapping loads and stores within a particular CPU will appear to be ordered within that CPU. This means that the CPU will issue **load/store** in exact order:

```
a = LOAD *X, STORE *X = b
STORE *X = c, d = LOAD *X
```

- Please note that this ordering guaranteed only within particular CPU executing operations: another CPU still might see out-of-order data.

Kernel memory-ordering model (cont.)

- There are things that **must** or **must not** be assumed as guarantee:
  - It **must not** be assumed that the compiler will do what you want with memory references.
  - It **must not** be assumed that independent loads and stores will be issued in the order given:

```
X = *A; Y = *B; *D = Z;

X = LOAD *A,  Y = LOAD *B,  STORE *D = Z
Y = LOAD *B,  STORE *D = Z, X = LOAD *A
STORE *D = Z, X = LOAD *A,  Y = LOAD *B
...
```

  - It **must** be assumed that overlapping memory accesses may be merged or discarded.

```
X = *A; Y = *(A + 4);

X = LOAD *A; Y = LOAD *(A + 4);
{X, Y} = LOAD {*A, *(A + 4) };
...
```

Kernel memory-ordering model (cont.)

- It should be noted: about bitfields and guarantees we describe above:
  - These guarantees do not apply to bitfields, because compilers often generate code to modify these using non-atomic read-modify-write sequences. Do not attempt to use bitfields to synchronize parallel algorithms.
  - Even in cases where bitfields are protected by locks, all fields in a given bitfield must be protected by one lock. If two fields in a given bitfield are protected by different locks, the compiler's non-atomic read-modify-write sequences can cause an update to one field to corrupt the value of an adjacent field.
  - These guarantees apply only to properly aligned and sized scalar variables. "Properly sized" currently means variables that are the same size as "char", "short", "int" and "long". "Properly aligned" means the natural alignment, thus no constraints for "char", two-byte alignment for "short", four-byte alignment for "int", and either four-byte or eight-byte alignment for "long", on 32-bit and 64-bit systems, respectively.

Kernel memory-ordering model (cont.)

- As we said above memory access might be reordered at runtime.
- This is also possible at compile time while compiler performs optimization.
- Compiler might do correct optimization for single-threaded case but fatal for multi-threaded one.
- It might reorder load/stores, optimize variable reloads from memory by using register to store variable, optimize branching possibly storing out of order to memory, optimize loops without accessing memory holding variable(s) involved in conditionals.
- There however means to force compiler to avoid doing such optimizations. One of most important is **volatile** keyword.

Control reordering: memory barriers

- As we see above operations involving memory access can be performed in random order. This causes problems for CPU to CPU and for CPU to I/O device interaction.
- We can intervene to memory access ordering by using memory barriers!
- They impose a perceived **partial** ordering over the memory operations on either side of the barrier.
- There are following variations of memory barriers:
  - *Write* (or store) memory barriers
  - *Data* dependency barriers
  - *Read* (or load) memory barriers
  - *General* memory barriers

Control reordering: memory barriers (cont.)

- *Write* (or store) memory barriers:
  - A write memory barrier gives a guarantee that all the STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other components of the system.
  - A write barrier is a **partial** ordering on stores only; it is not required to have any effect on loads.
  - A CPU can be viewed as committing a sequence of store operations to the memory system as time progresses. All stores before a write barrier will occur in the sequence **before** all the stores after the write barrier.
- Note that write barriers should normally be paired with *read* or *data dependency* barriers:

```
        CPU1                    CPU2

STORE A                 LOAD B
WRITE BARRIER           READ BARRIER
STORE B                 LOAD A
```

Control reordering: memory barriers (cont.)

- *Data dependency* barriers:
  - A data dependency barrier is a weaker form of read barrier. In the case where two loads are performed such that the second depends on the result of the first (eg: the first load retrieves the address to which the second load will be directed), a data dependency barrier would be required to make sure that the target of the second load is updated before the address obtained by the first load is accessed.
  - A data dependency barrier is a partial ordering on interdependent loads only; it is not required to have any effect on stores, independent loads or overlapping loads.
  - A data dependency barrier issued by the CPU under consideration guarantees that for any load preceding it, if that load touches one of a sequence of stores from another CPU, then by the time the barrier completes, the effects of all the stores prior to that touched by the load will be perceptible to any loads issued after the data dependency barrier.
- Note that data dependency barriers should be paired with write barriers.

Control reordering: memory barriers (cont.)

- *Read* (or load) memory barriers:
  - A read barrier is a data dependency barrier plus a guarantee that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system.
  - A read barrier is a partial ordering on loads only; it is not required to have any effect on stores.
  - Read memory barriers imply data dependency barriers, and so can substitute for them.
- Note that read barriers should normally be paired with write barriers.
- *General* memory barriers:
  - A general memory barrier gives a guarantee that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other components of the system.
  - General memory barriers **imply both read and write** memory barriers, and so can substitute for either.

Control reordering: memory barriers (cont.)

- Where are memory barriers needed?
- Under normal operation, memory operation reordering is generally not going to be a problem as a single-threaded linear piece of code will still appear to work correctly, even if it's in an SMP kernel.
- There are, however, four circumstances in which reordering definitely **could** be a problem:
  - Interprocessor interaction
  - Atomic operations
  - Accessing devices
  - Interrupts

Control reordering: memory barriers (cont.)

- What may not be assumed about memory barriers:
  - There is no guarantee that any of the memory accesses specified before a memory barrier will be **complete** by the completion of a memory barrier instruction; the barrier can be considered to draw a line in that CPU's access queue that accesses of the appropriate type may not cross.
  - There is no guarantee that issuing a memory barrier on one CPU will have any direct effect on another CPU or any other hardware in the system. The indirect effect will be the order in which the second CPU sees the effects of the first CPU's accesses occur, but see the next point:
    - There is no guarantee that a CPU will see the correct order of effects from a second CPU's accesses, even **if** the second CPU uses a memory barrier, unless the first CPU **also** uses a matching memory barrier.
    - There is no guarantee that some intervening piece of off-the-CPU hardware will not reorder the memory accesses.

Control reordering: memory barriers (cont.)

- The Linux kernel has a variety of different barriers that act at different levels:
  - Compiler barrier
    - Used to prevent compiler from reordering during compile time. Mostly by using **volatile** keyword when performing memory access operations.
  - CPU memory barriers
    - Prevent from reordering at runtime by the CPU. This either **mandatory** to order access to device's memory or **conditional** to order access on SMP (if kernel is build for UP these barriers simplified to compiler barriers thus they marked as conditional.
- Although kernel provides primitives for explicitly describe memory barriers in the code there are implicit memory barriers posed by various locking primitives like spinlocks, read-write locks, semaphores and mutexes.
- We will describe explicit memory barriers of them next.

Control reordering: memory barriers (cont.)

- The Linux kernel has an explicit **compiler barrier** function that prevents the compiler from moving the memory accesses either side of it to the other side (this is a general barrier -- there are no read-read or write-write variants):

```
barrier();
```

- The `barrier()` function has the following effects:
  - Prevents the compiler from reordering accesses following the `barrier()` to precede any accesses preceding the `barrier()`. One example use for this property is to ease communication between interrupt-handler code and the code that was interrupted.
  - Within a loop, forces the compiler to load the variables used in that loop's conditional on each pass through that loop

Control reordering: memory barriers (cont.)

- `READ_ONCE()` and `WRITE_ONCE()` can be thought of as weak forms of `barrier()` that affect only the specific accesses flagged by the `READ_ONCE()` or `WRITE_ONCE()`.
- The `READ_ONCE()` and `WRITE_ONCE()` functions can prevent any number of optimizations that, while perfectly safe in single-threaded code, can be fatal in concurrent code.
- There is deprecated `ACCESS_ONCE()` that can be the only alternative in older kernels.

Control reordering: memory barriers (cont.)

- The Linux kernel has eight basic CPU memory barriers:

TYPE	MANDATORY	SMP CONDITIONAL
GENERAL	mb()	smp_mb()
WRITE	wmb()	smp_wmb()
READ	rmb()	smp_rmb()
DATA DEPENDENCY	read_barrier_depends()	smp_read_barrier_depends()

- All memory barriers except the data dependency barriers imply a compiler barrier. Data dependencies do not impose any additional compiler ordering.
- SMP memory barriers **must** be used to control the ordering of references to shared memory on SMP systems, but the use of locking instead is sufficient.

Control reordering: memory barriers (cont.)

- There are some more advanced barrier functions:
  - `smp_store_mb(var, value)`
    - This assigns the value to the variable and then inserts a full memory barrier after it. It isn't guaranteed to insert anything more than a compiler barrier in a UP compilation.
  - `smp_mb__before_atomic()` and `smp_mb__after_atomic()`
    - These are for use with atomic (such as add, subtract, increment and decrement) functions that don't return a value, especially when used for reference counting. Such functions do not imply memory barriers.
    - This makes sure that the death mark on the object is perceived to be set **before** the reference counter is decremented.

```
obj->dead = 1;
smp_mb__before_atomic();
atomic_dec(&obj->ref_count);
```

Control reordering: memory barriers (cont.)

- There are some more advanced barrier functions (cont.):
  - `dma_wmb()` and `dma_rmb()`
    - These are for use with consistent memory to guarantee the ordering of writes or reads of shared memory accessible to both the CPU and a DMA capable device.
    - The `dma_rmb()` allows us guarantee the device has released ownership before we read the data from the descriptor, and the `dma_wmb()` allows us to guarantee the data is written to the descriptor before the device can see it now has ownership.

Kernel memory-ordering model and control

- Where all this stuff declared? In short: it is architecture specific.
- Most of parts related to CPU memory barriers (e.g. `mb()`, `rmb()` and `wmb()`) declared in
  - include/asm-generic/barrier.h - for generic implementation
  - arch/$ARCH/include/asm/barrier.h - for architecture specific implementations
- Compiler barrier related stuff declared in
  - include/linux/compiler.h (e.g. `READ_ONCE()`, `WRITE_ONCE()`, `ACCESS_ONCE()` etc.)
  - include/linux/compiler-gcc.h
  - include/linux/compiler-clang.h
  - include/linux/compiler-intel.h

Kernel memory-ordering model and control (cont.)

- More information about memory-ordering models and control in Linux Kernel:
  - Memory-ordering model in Linux Kernel
    - A formal kernel memory-ordering model (part 1) April 2017
    - A formal kernel memory-ordering model (part 2) April 2017
    - A Strong Formal Model of Linux-Kernel Memory Ordering
    - A Weak Formal Model of Linux-Kernel Memory Ordering
    - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0124r1.html
  - Documentation in Linux Kernel source tree
    - Documentation/memory-barrier.txt
    - Documentation/atomic_ops.txt
    - Documentation/cachetlb.txt
    - Documentation/DMA-API.txt

Sequential locks

Reader/writer consistent mechanism without starving writers (<linux/seqlock.h>). This type of lock for data where the reader wants a consistent set of information and is willing to retry if the information changes. There are two types of readers:

- Sequence readers which never block a writer but they may have to retry if a writer is in progress by detecting change in sequence number. Writers do not wait for a sequence reader.

```
unsigned read_seqbegin(const seqlock_t *sl);
unsigned read_seqretry(const seqlock_t *sl, unsigned start);
```

- Locking readers which will wait if a writer or another locking reader is in progress. A locking reader in progress will also block a writer from going forward. Unlike the regular rwlock, the read lock here is exclusive so that only one locking reader can get it.

```
void read_seqlock_excl(seqlock_t *sl);
void read_sequnlock_excl(seqlock_t *sl);
```

- Writers always lock access

```
void write_seqlock(seqlock_t *sl);
void write_sequnlock(seqlock_t *sl);
```

Sequential locks (cont.)

Usage template:

```
/* Data and and sequential lock structure */
struct some_data data;
seqlock_t data_seqlock;
```

```
/* Non-blocking reader usage */
unsigned seq;

do {
        seq = read_seqbegin(&data_seqlock);
        /*
         * Read-side access
         */
} while (read_seqretry(&data_seqlock, seq));
```

```
/* Writer usage */

write_seqlock(&data_seqlock);
/*
 * Write-side access
 */
write_sequnlock(&data_seqlock);
```

Sequential locks (cont.)

How it works:

(all functions are static inline)

```
typedef struct {
          struct seqcount seqcount;
          spinlock_t lock;
} seqlock_t;
```

```
void write_seqlock(seqlock_t *sl)
{
    spin_lock(&sl->lock);
    sl->seqcount.sequence++;
    smp_wmb();
}
```

```
void write_sequnlock(seqlock_t *sl)
{
    smp_wmb();
    sl->seqcount.sequence++;
    spin_unlock(&sl->lock);
}
```

```
unsigned read_seqbegin(const seqlock_t *sl)
{
    unsigned ret;
repeat:
    ret = READ_ONCE(sl->seqcount.sequence);
    if (unlikely(ret & 1))
        goto repeat;
    smp_rmb();
    return ret;
}
```

```
unsigned read_seqretry(const seqlock_t *sl,
                       unsigned start)
{
    smp_rmb();
    return  unlikely(sl->seqcount.sequence != start);
}
```

Per-CPU variables

- Previously we said that in general synchronization is worse thing for performance reasons and problems they might introduce (e.g. deadlocks, contention, starvation, etc).
- We told that it should be avoided whenever possible and one of the things that can help with this is to making algorithms working with data structures local to each CPU (i.e. per-CPU).
- In Linux Kernel access to Per-CPU variables requires (almost) no locking.
  - The only case where some kind of locking is required is when kernel build as PREEMPT
- Another performance benefit from using Per-CPU variables is better processor cache sharing.
- See `<linux/percpu.h>`, `<linux/percpu-defs.h>`, `<arch/*/include/asm/percpu.h>`

Per-CPU variables (cont.)

- Let's take a look at real example (net/rds/rds.h)

```
/* struct rds_statistics has a lot of uint64_t counters */

DECLARE_PER_CPU_SHARED_ALIGNED(struct rds_statistics, rds_stats);

#define rds_stats_inc_which(which, member) do {                         \
                    per_cpu(which, get_cpu()).member++;                 \
                    put_cpu();                                          \
          } while (0)

#define rds_stats_inc(member) rds_stats_inc_which(rds_stats, member)

#define rds_stats_add_which(which, member, count) do {                  \
                    per_cpu(which, get_cpu()).member += count;          \
                    put_cpu();                                          \
          } while (0)

#define rds_stats_add(member, count) rds_stats_add_which(rds_stats, member, count)
```

Per-CPU variables (cont.)

- Let's take a look at real example (net/rds/stats.c)

```
DEFINE_PER_CPU_SHARED_ALIGNED(struct rds_statistics, rds_stats);
EXPORT_PER_CPU_SYMBOL_GPL(rds_stats);

static void rds_stats_info(struct socket *sock, unsigned int len,
                           struct rds_info_iterator *iter,
                           struct rds_info_lengths *lens)
{
        struct rds_statistics stats = {0, };
        uint64_t *src,  *sum;
        int cpu;
        /* . . . */
        for_each_online_cpu(cpu) {
                src = (uint64_t *)&(per_cpu(rds_stats, cpu));
                sum = (uint64_t *)&stats;
                for (i = 0; i < sizeof(stats) / sizeof(uint64_t); i++)
                        *(sum++) += *(src++);
        }
        /* . . . */
}
```

Per-CPU variables (cont.)

- There are primitives available on SMP to deal with preemption:

```
#define get_cpu()   ({ preempt_disable(); smp_processor_id(); })
#define put_cpu()   preempt_enable()
```

- Here are PREEMPT safe variants that disable/enable preemption on get/put Per-CPU variables. In case of preemption it is possible that on return we continue execution on CPU different from one we run before preempt:

```
#define get_cpu_var(var)                \
        (*({                            \
                preempt_disable();      \
                this_cpu_ptr(&var);     \
        }))

#define put_cpu_var(var)                \
        do {                            \
                (void)&(var);           \
                preempt_enable();       \
        } while (0)
```

Per-CPU variables (cont.)

- Here are bit more APIs to work Per-CPU:

```
/* Operations with implied preemption/interrupt protection.  These
 * operations can be used without worrying about preemption or interrupt.
 */
#define this_cpu_read(pcp)              __pcpu_size_call_return(this_cpu_read_, pcp)
#define this_cpu_write(pcp, val)        __pcpu_size_call(this_cpu_write_, pcp, val)
#define this_cpu_add(pcp, val)          __pcpu_size_call(this_cpu_add_, pcp, val)
#define this_cpu_and(pcp, val)          __pcpu_size_call(this_cpu_and_, pcp, val)
#define this_cpu_or(pcp, val)           __pcpu_size_call(this_cpu_or_, pcp, val)

#define this_cpu_xchg(pcp, nval)        __pcpu_size_call_return2(this_cpu_xchg_, pcp, nval)

#define this_cpu_sub(pcp, val)          this_cpu_add(pcp, -(typeof(pcp))(val))
#define this_cpu_inc(pcp)               this_cpu_add(pcp, 1)
#define this_cpu_dec(pcp)               this_cpu_sub(pcp, 1)

#define this_cpu_add_return(pcp, val)   \
                                __pcpu_size_call_return2(this_cpu_add_return_, pcp, val)
```

Read-Copy-Update (RCU) synchronization

- Most advanced synchronization mechanism available in Linux Kernel:
  - Does not block readers even when data structure being updated (that's however presents space-time tradeoff and increased memory usage).
  - Read critical section behaves as there is no synchronization primitives: no overhead (or very small on preemptible kernels) on enter/exit to RCU critical section.
  - It is possible for readers to run concurrently with updaters (writers)
  - Most of deadlock or livelock conditions isn't applicable to it due to it's nature
- It is patented by IBM and specially licensed to Linux Kernel by it
- In Linux Kernel since 2002
- There is userspace shared library licensed under LGPL2.1 liburcu available to applications wishing to benefit from RCU in userspace applications.

Read-Copy-Update (RCU) synchronization (cont.)

- However not all workloads can benefit from RCU semantics.
- Similar to rwlocks and read-write semaphores only patterns when read access is prevailed over write access will benefit from this.
- In general updates (writes) to data structures protected by RCU are very complicated and **must** be mutually exclusive.
- There are two implementations of RCU in kernel currently available for choice at kernel build time via Kconfig symbols:
  - CONFIG_TREE_RCU - this is default and most mature implementation in kernel
  - CONFIG_TINY_RCU - simple and tiny in code terms implementation, suitable for really small embedded systems.
- There are lots of information about RCU available in Documentation/RCU

Read-Copy-Update (RCU) synchronization (cont.)

- Nearly zero overhead on performance for read access achieved when following two conditions isn't true:
  - Kernel build with PREEMPT support. In that case preemption need to be disabled while entering to RCU read critical section.
  - Build for really weak memory-ordered architecture such as DEC Alpha where access to RCU protected data implies true `smp_read_depends_on()`
- Main requirement for RCU implementation is atomicity operations on native words on particular architecture. For RCU atomicity of operations on shared memory area of pointer size is vital. Of course both compile time and CPU memory barriers should be implied (and they are using SMP memory barrier primitives that falls to compile time `barrier()` on UP builds).

Read-Copy-Update (RCU) synchronization (cont.)

- As said earlier RCU implementation fully depends on atomicity of operations on pointer sized shared memory areas with strict access ordering guaranteed by SMP ordering primitives.
- There might be two access patterns for RCU pointers:
  - read - from RCU read critical section (that accomplished with `lockless_dereference()`)
  - write - from updater (that accomplished via `smp_wmb()` barrier))
- Therefore RCU fully implements read/write barrier semantics for SMP case as described earlier in controlling ordered access to memory.
- It is critically important to protect RCU data structures from multiple updaters running concurrently to avoid data corruption. That’s mostly accomplished with one of the Linux Kernel mutual exclusion primitives (e.g. spinlocks)

Read-Copy-Update (RCU) synchronization (cont.)

- On the other hand update (writes) procedure is complicated and in general split into following stages:
  - Take copy (read) of in use data structure (hence C letter in RCU)
  - Update required value in that copy (hence U letter in RCU)
  - Replace old data structure we copy previously with new one we modify safely for readers
- After we did update we need to deal with old, replaced copy of data structure. This stage is called reclaim and this is might be a most complicated part of RCU synchronization implementation:
  - We can not release old data structure in place after we replace it with new one in readers safe manner: there might be concurrent readers that working with it while we perform update
  - We need to wait to at least one **grace period** (quiescent time) before releasing data.

Read-Copy-Update (RCU) synchronization (cont.)

- Now we known how data being updated and reclaimed with respect to concurrent readers, it is time to known when old data structure might be released finally.
- That’s fully depends on RCU read critical section semantics:
  - Quiescent state is state of executing thread outside of RCU critical section.
  - Grace period is a any time period during which each thread executes outside of RCU critical section (i.e. in quiescent state).
- It is obviously that following restrictions placed on RCU read critical section to reach reclaim state when data can be released correctly:
  - Sleeping within RCU critical section isn’t supported unless CONFIG_PREEMPT_RCU is on
  - Referencing any RCU protected data in quiescent state isn’t allowed: that may lead to use-after-free when data released after grace period.

Read-Copy-Update (RCU) synchronization (cont.)

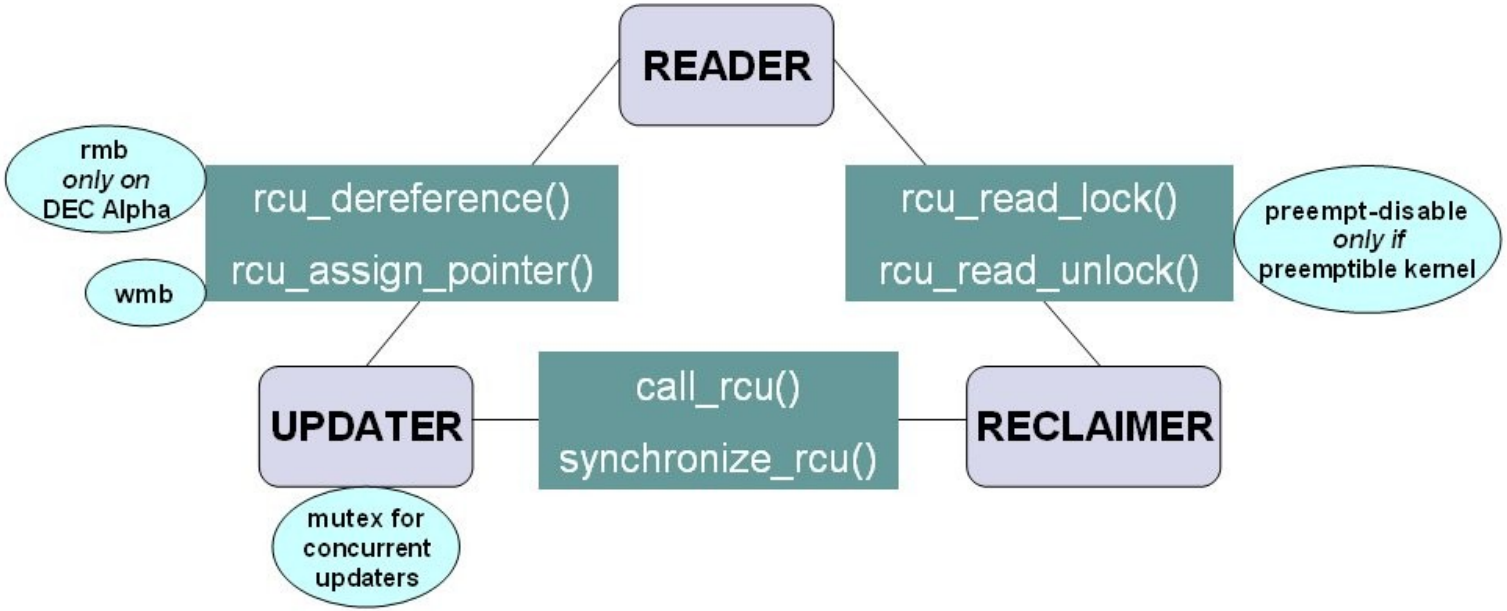
- How can the updater tell when a grace period has completed if the RCU readers give no indication when they are done?
- Just as with spinlocks, RCU readers are not permitted to block, switch to user-mode execution, or enter the idle loop.
- Therefore, as soon as a CPU is seen passing through any of these three states, we know that that CPU has exited any previous RCU read-side critical sections.
- So, if we remove an item from a linked list, and then wait until all CPUs have switched context, executed in user mode, or executed in the idle loop, we can safely free up that item.

Read-Copy-Update (RCU) synchronization (cont.)

- Assuming all the above we can describe RCU protected data structures workflow:
  - There are multiple readers running in parallel in SMP system performing access to RCU protected data only from read critical section.
  - Remove pointer references to data structure being reclaimed (either replace with new one or delete) so no further readers can access this data structure.
  - Wait for grace period to elapse to make sure all readers finish using that data structure.
  - After that there should be no readers to data structure (if they are - your usage of RCU is buggy) and it can be reclaimed (e.g. consumed memory released).
- Since RCU works with pointers you need to either use kernel standard data structures that already RCU aware (most of them) or implement primitives to operate on your own data structures in RCU safe manner.

Read-Copy-Update (RCU) synchronization (cont.)

- To put all together using nice picture from [wikipedia page for RCU]



Read-Copy-Update (RCU) synchronization (cont.)

- There are standard kernel basic types that are updated to be RCU aware:
  - Linked lists (<linux/rculist.h> variant as well as few routines in <linux/list.h> also RCU safe)
  - Hash tables (<linux/rculist.h> variant as well as few routines in <linux/list.h> also RCU safe)
  - Red-Black trees (<linux/rbtree.h>)
- RCU core API to operate on pointers directly as well as routines to enter/leave read critical section, wait for grace period available in <linux/rcupdate.h>
- There is implementation specific parts, that should not be included directly to your source code:
  - <linux/rcutree.h> - for CONFIG_TREE_RCU
  - <linux/rcutiny.h> - for CONFIG_TINY_RCU

Read-Copy-Update (RCU) synchronization (cont.)

- Let’s describe general and low level, RCU API that used when implementing such type of synchronization:

```
/* Used by a reader to inform the reclaimer that the reader is
 * entering an RCU read-side critical section.  It is illegal
 * to block while in an RCU read-side critical section, though
 * kernels built with CONFIG_PREEMPT_RCU can preempt RCU
 * read-side critical sections.  Any RCU-protected data structure
 * accessed during an RCU read-side critical section is guaranteed to
 * remain unreclaimed for the full duration of that critical section.
 * Reference counts may be used in conjunction with RCU to maintain
 * longer-term references to data structures.
*/
void rcu_read_lock(void);
```

Read-Copy-Update (RCU) synchronization (cont.)

- Continue with RCU core API:

```
/* Used by a reader to inform the reclaimer that the reader is exiting
 * an RCU read-side critical section.
 *
 * Note that RCU read-side critical sections may be nested and/or
 * overlapping.
 */
void rcu_read_unlock(void);
```

Read-Copy-Update (RCU) synchronization (cont.)

- Continue with RCU core API:

```
/* Marks the end of updater code and the beginning of reclaimer code.
 *
 * It does this by blocking until all pre-existing RCU
 * read-side critical sections on all CPUs have completed.
 *
 * Note that synchronize_rcu() will -not- necessarily wait for
 * any subsequent RCU read-side critical sections to complete.
 * For example, consider the following sequence of events:
 */
void synchronize_rcu(void);
```

Read-Copy-Update (RCU) synchronization (cont.)

- What actually does mean “...synchronize_rcu() will -not- necessarily wait for * any subsequent RCU read-side critical sections to complete...”

```
        CPU 0                  CPU 1                     CPU 2
    ---------------- ------------------------- ---------------
1.  rcu_read_lock()
2.                   enters synchronize_rcu()
3.                                             rcu_read_lock()
4.  rcu_read_unlock()
5.                   exits synchronize_rcu()
6.                                             rcu_read_unlock()
```

Read-Copy-Update (RCU) synchronization (cont.)

- Continue with RCU core API:

```
/* The updater uses this function to assign a new value to an
 * RCU-protected pointer, in order to safely communicate the change
 * in value from the updater to the reader.  This function returns
 * the new value, and also executes any memory-barrier instructions
 * required for a given CPU architecture.
 */
typeof(p) rcu_assign_pointer(p, typeof(p) v);

/* An simplified implementation might look like following */
#define rcu_assign_pointer(p, v)     \
        ({                           \
                smp_wmb();           \
                (p) = (v);           \
        })
```

Read-Copy-Update (RCU) synchronization (cont.)

- Continue with RCU core API:

```
/* The reader uses rcu_dereference() to fetch an RCU-protected pointer,
 * which returns a value that may then be safely dereferenced. Note that
 * rcu_deference() does not actually dereference the pointer, instead,
 * it protects the pointer for later dereferencing.  It also executes
 * any needed memory-barrier instructions for a given CPU architecture.
 * Currently, only Alpha needs memory barriers within rcu_dereference()
 * -- on other CPUs, it compiles to nothing, not even a compiler directive.
 */
typeof(p) rcu_dereference(p);

#define rcu_dereference(p)                      \
        ({ typeof(p) _________p1 = p;           \
           smp_read_barrier_depends();          \
           (_________p1); })
```

Read-Copy-Update (RCU) synchronization (cont.)

- Continue with RCU core API:

```
/* types of RCU callback function */
typedef void (*rcu_callback_t)(struct rcu_head *head);
typedef void (*call_rcu_func_t)(struct rcu_head *head,
                                rcu_callback_t func);

/* The call_rcu() API is a callback form of synchronize_rcu(), and is
 * described in more detail in a later section.  Instead of blocking, it
 * registers a function and argument which are invoked after all ongoing
 * RCU read-side critical sections have completed. This callback variant
 * is particularly useful in situations where it is illegal to block or
 * where update-side performance is critically important.
 */
void call_rcu(struct rcu_head *head, rcu_callback_t func);
```

- Note that struct rcu_head is declared in <linux/types.h>

Read-Copy-Update (RCU) synchronization (cont.)

- To get full picture let’s make analogy with read-write spinlocks (rwlock)

```
1 struct el {
2   struct list_head list;
3   long key;
4   spinlock_t mutex;
5   int data;
6   /* Other data fields */
7 };
8 rwlock_t listmutex;
9 struct el head;
```

```
1 struct el {
2   struct list_head list;
3   long key;
4   spinlock_t mutex;
5   int data;
6   /* Other data fields */
7 };
8 spinlock_t listmutex;
9 struct el head;
```

Read-Copy-Update (RCU) synchronization (cont.)

- To get full picture let's make analogy with read-write spinlocks (cont.)

```
 1 int search(long key, int *result)
 2 {
 3   struct list_head *lp;
 4   struct el *p;
 5
 6   read_lock(&listmutex);
 7   list_for_each_entry(p, head, lp) {
 8     if (p->key == key) {
 9       *result = p->data;
10       read_unlock(&listmutex);
11       return 1;
12     }
13   }
14   read_unlock(&listmutex);
15   return 0;
16 }
```

```
 1 int search(long key, int *result)
 2 {
 3   struct list_head *lp;
 4   struct el *p;
 5
 6   rcu_read_lock();
 7   list_for_each_entry_rcu(p, head, lp) {
 8     if (p->key == key) {
 9       *result = p->data;
10       rcu_read_unlock();
11       return 1;
12     }
13   }
14   rcu_read_unlock();
15   return 0;
16 }
```

Read-Copy-Update (RCU) synchronization (cont.)

- To get full picture let's make analogy with read-write spinlocks (cont.)

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

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

Read-Copy-Update (RCU) synchronization (cont.)

- There are cases when sleeping in `synchronize_rcu()` isn't possible. For that cases using `call_rcu()` primitive to schedule function to be called after grace period:

```
struct foo {
    int a;
    char b;
    long c;
    /* this is passed as
     * @rp to foo_reclaim()
     */
    struct rcu_head rcu;
};
```

```
void foo_reclaim(struct rcu_head *rp)
{
    /* here we get pointer to struct foo
       from rcu_head */
    struct foo *fp = container_of(rp,
                                  struct foo, rcu);

    foo_cleanup(fp->a);

    kfree(fp);
}
```

Read-Copy-Update (RCU) synchronization (cont.)

- and finally our path that could not block would look as following:

```
void foo_update_a(int new_a)
{
    struct foo *new_fp;
    struct foo *old_fp;
    new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
    BUG_ON(!new_fp);
    spin_lock(&foo_mutex);
    old_fp = rcu_dereference_protected(gbl_foo, lockdep_is_held(&foo_mutex));
    *new_fp = *old_fp;
    new_fp->a = new_a;
    rcu_assign_pointer(gbl_foo, new_fp);
    spin_unlock(&foo_mutex);
    call_rcu(&old_fp->rcu, foo_reclaim);
}
```

Read-Copy-Update (RCU) synchronization (cont.)

- There are additional variation of RCU implementation in Linux Kernel:
- Bottom Half (SoftIRQ), one with reduced grace period that indicates quiescent state after all SoftIRQ processing completes.
  - It’s main intend to be used in hard/soft irq contexts, while can be used to synchronize with user context too (but that requires some tricks like disabling hard/soft irqs).
  - Primitives implementing this variant of RCU suffixed with “_bh”:
    - rcu_read_lock_bh()
    - rcu_read_unlock_bh()
    - synchronize_rcu_bh()
    - call_rcu_bh()
  - Functionality or lock/unlock should not be confused with spin_lock_bh() and spin_unlock_bh() and others that disable SoftIRQs via local_bh_disable() and local_bh_enable() respectively.

Read-Copy-Update (RCU) synchronization (cont.)

- Refer following sources for more information about RCU implementation:
  - http://www.rdrop.com/users/paulmck/RCU
  - https://en.wikipedia.org/wiki/Read-copy-update
  - What is RCU, Fundamentally?
  - What is RCU? Part 2: Usage
  - What is RCU? Part 3: the RCU API
  - The RCU API, 2010 Edition
  - 2010 Big API Table
  - The RCU API, 2014 Edition
  - 2014 Big API Table
- Documentation in Linux Kernel source tree
  - Documentation/RCU

That’s all for now. Thank you for viewing.

Authors:

- Serhii Popovych, Software Engineer
  - Cisco-IOSXE Linux SDK, GlobalLogic Ukraine LLC
  - Email: serhii.popovych@globallogic.com
- Oleksandr Redchuk, Software Engineer
  - GlobalLogic Ukraine
  - Email: oleksandr.redchuk@gmail.com