

B4M36ESW: Efficient software

Lecture 5: Scalable synchronization

Michal Sojka

`michal.sojka@cvut.cz`



July 3, 2021

Synchronization

- **Multi-core** CPUs are today's norm, many-core CPUs will come tomorrow
- To take the advantage of such a hardware, **parallel (multi-threaded) programs** must be run on them
- Programs are not useful when their threads are completely independent, i.e. threads have to **communicate (synchronize)**
- Basic forms of synchronization:
 - Mutual exclusion (e.g. access to shared data)
 - Producer-consumer (e.g. a database waits for requests)
 - ...

Outline

1 Naive synchronization

■ Problems

- Non-atomic data manipulation
- Unwanted compiler optimizations
- Reordering at hardware level
- Deadlock
- Busy waiting (spinning)

2 Real-world synchronization (locking)

- Kernel semaphores
- Futex
- Cost of atomic operations and barriers

3 Read-mostly workload

4 Read-Copy-Update (RCU)

- RCU implementations

Naive synchronization

Mutual exclusion

Data should be modified at most by one thread at a time:

```
bool locked;
```

```
void func() {  
    while (locked == true)  
        /* busy wait */;  
    locked = true;  
    data++;  
    locked = false;  
}
```

Terminology: code in the “locked” region is called *critical section*

Problems:

- 1 Checking and setting the lock is not atomic
- 2 Compiler can optimize out all accesses to *locked*
- 3 Compiler can move access to data out of critical section
- 4 Hardware can reorder memory accesses even if compiler does not
- 5 Can easily deadlock
- 6 Busy waiting wastes energy

Quote

*Implementing a correct synchronization primitive is like committing the perfect crime. There are at least 50 things that can go wrong, and if you are a highly experienced genius, you **might** be able to anticipate and handle 25 of them.*

— Paul McKenney

<https://lwn.net/ml/linux-kernel/20190608160620.GH28207@linux.ibm.com/>

Outline

1 Naive synchronization

■ Problems

- Non-atomic data manipulation
- Unwanted compiler optimizations
- Reordering at hardware level
- Deadlock
- Busy waiting (spinning)

2 Real-world synchronization (locking)

- Kernel semaphores
- Futex
- Cost of atomic operations and barriers

3 Read-mostly workload

4 Read-Copy-Update (RCU)

- RCU implementations

Atomic operations

■ Example of non-atomic increment:

- C expression: `data++`;
- Assembler (x86): `inc ($data)` – atomic (uninterruptible) on a single CPU
- Hardware (as seen from other CPUs): memory bus read, ALU, memory bus write

CPU0	CPU1	data
bus read		0
ALU	bus read	0
bus write	ALU	1
	bus write	1

■ Atomic operations ensure that the operation (typically read-modify-write) is atomic (uninterruptible) even at the hardware (bus) level.

- compare-and-swap/CAS instruction (x86: `cmpxchg`)
- In C: `TYPE __atomic_exchange_n(TYPE *ptr, TYPE val, int memorder)`
Atomically set `*ptr` to `val` and return old value of `*ptr`.

```

void lock() {
    while (locked == true)
        /* busy wait */;
    locked = true;
}

```

⇒

```

void lock() {
    while (__atomic_exchange_n(&locked, true,
                               [..]) == true)
        /* busy wait */;
}

```

Atomic operations in C and C++

- For long time, atomic operations were not standardized in C/C++
 - Solution: Incompatible compiler extensions, inline assembler
- C11, C++11 introduced thread-aware memory model and defined platform independent atomic operations
- C11: `#include <stdatomic.h>`, `atomic_*` functions
- C++11
 - `#include <atomic>`
 - `std::atomic` template
 - Example:

```
std::atomic<int> x;  
x++; // atomic increment
```
- Atomic operations are more expensive than non-atomic ones

Compiler optimizations

```

bool locked;

while (locked) {}
locked = true;
data++;
locked = false;

```

⇒

```

#define barrier() \
    asm volatile("" : : : "memory")
volatile bool locked;

while (locked) {}
locked = true;
barrier();
data++;
barrier();
locked = false;

```

- Compiler expects the memory is only modified by the program being compiled
- Locked seems to be useless ⇒ optimize out
 - prevented by `volatile`
- Compiler is free to reorder operations as long as the result of single-threaded computation is the same
 - `barrier()` prevents the compiler from reordering memory operations before and after the barrier.

Compiler optimizations cont.

- Defining the variable `volatile` makes all accesses “volatile” i.e. slow.
- Sometimes, we need only certain accesses to have volatile semantics and the rest can be optimized:

```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
#define LOAD_SHARED(p) ACCESS_ONCE(p)
#define STORE_SHARED(x, v) ({ ACCESS_ONCE(x) = (v); })
```

```
#define barrier() asm volatile("" : : : "memory")
```

- See examples in later slides (RCU implementation).
- The macro `barrier` is only a **compiler barrier**, not hardware barrier, i.e., the compiler will not reorder the generated instructions, but CPU can still reorder them during execution.

Hardware reordering

- Different CPU architectures implement different memory consistency models
- **Problem:** Some operations can be reordered with respect to other operations

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	z/Architecture
Loads → loads	Y	Y	Y	Y	Y	N	N	N	Y	N	Y	N
Loads → stores	Y	Y	Y	Y	Y	N	N	N	Y	N	Y	N
Stores → stores	Y	Y	Y	Y	Y	Y	N	N	Y	N	Y	N
Stores → loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic → loads	Y	Y	N	Y	Y	N	N	N	N	N	Y	N
Atomic → stores	Y	Y	N	Y	Y	Y	N	N	N	N	Y	N
Dependent loads	Y	N	N	N	N	N	N	N	N	N	N	N
Incoherent inst. cache pipeline	Y	Y	N	Y	Y	Y	Y	Y	Y	N	Y	

Source: Wikipedia

- x86 can reorder stores after loads, e.g. data variable from slide 9 can be read before other CPUs see `locked` set to true!
 - Why? Stores may have to wait for cache-line ownership. Not waiting with subsequent reads improves **performance**.
- **Solution:** Insert memory barrier instructions.

Barrier (fence) instructions

- Ensure that all memory operations issued in front of the barrier are completed before issuing any memory operation after the barrier
- Some architectures provide more barrier types:
 - Full barriers prevents reordering of all memory operations,
 - weak barriers prevents reordering of either loads or stores only.
- Examples: `mfence` (full barrier), `lfence` (weak barrier) instructions (on x86), `dmb` (on ARM).
- RCU implementations at the end of this lecture use `smp_mb()` function, which is a platform independent wrapper of these instructions.

Specifying memory ordering requirements in C/C++

```
std::atomic<int> x;  
x.load(order);  
w.store(0, order);
```

- `order` specifies how regular, non-atomic memory accesses are to be ordered around an atomic operation
 - relaxed: no overhead, no order guarantee
 - consume
 - acquire
 - release
 - `acq_rel`,
 - `seq_cst`: high overhead, sequential consistency
- Depending on the CPU architecture, different `orders` cause the compiler to generate barrier instructions (e.g., `lfence` on x86)

Deadlock

- Example:
 - Single-core system
 - Spinning lock
 - Two threads low- and high-priority

```

LP_thread      HP_thread
-----
lock();
data++;
      →      ←      →
                lock(); // deadlock

```

- **Solution:** When the lock is not available, ask the OS scheduler to put your thread to sleep and wake you up after the lock is available
 - Problem: atomicity of checking the lock and going to sleep
 - Requires implementation in the OS kernel

Spinlocks

- The lock (from our initial example) that does not involve OS scheduler and waits by spinning (looping, busy waiting) is called a **spin lock**.
- Applications almost never need to use spin locks.
 - It wastes energy, CPU time and can lead to deadlocks (see above).
- Spin locks are, however, used in operating system kernels in contexts, where it is not possible to sleep.
 - Only threads (tasks, processes) can sleep.
 - Interrupt handlers cannot sleep.
 - Hence **device drivers** often use spin locks.
 - OS scheduler must also use spin locks, because it cannot put itself to sleep.

Outline

- 1 Naive synchronization
 - Problems
 - Non-atomic data manipulation
 - Unwanted compiler optimizations
 - Reordering at hardware level
 - Deadlock
 - Busy waiting (spinning)
- 2 Real-world synchronization (locking)
 - Kernel semaphores
 - Futex
 - Cost of atomic operations and barriers
- 3 Read-mostly workload
- 4 Read-Copy-Update (RCU)
 - RCU implementations

Mutex

- Most-common synchronization primitive
- Simple API (lock/unlock)
- Can be implemented in different ways
 - Kernel semaphore
 - Futex
 - ...
- All implementations (at least on multi-core systems) involve atomic instructions.
- Sleeping is implemented by interacting with the OS scheduler.

Outline

- 1 Naive synchronization
 - Problems
 - Non-atomic data manipulation
 - Unwanted compiler optimizations
 - Reordering at hardware level
 - Deadlock
 - Busy waiting (spinning)
- 2 Real-world synchronization (locking)
 - Kernel semaphores
 - Futex
 - Cost of atomic operations and barriers
- 3 Read-mostly workload
- 4 Read-Copy-Update (RCU)
 - RCU implementations

Kernel semaphores

- Lock (mutex) implementation inside OS kernel
- Each system call adds overhead (≈ 100 clock cycles on modern HW)
- It is preferable to use “fine-grain” locking, i.e. locks protect as little data as possible to prevent lock contention.
- If fine-grain locking is effective the lock is not contended and threads rarely have to sleep, but always pay the syscall overhead!
- That's not efficient – the solution in Linux is called **futex**.

Outline

- 1 Naive synchronization
 - Problems
 - Non-atomic data manipulation
 - Unwanted compiler optimizations
 - Reordering at hardware level
 - Deadlock
 - Busy waiting (spinning)
- 2 Real-world synchronization (locking)
 - Kernel semaphores
 - **Futex**
 - Cost of atomic operations and barriers
- 3 Read-mostly workload
- 4 Read-Copy-Update (RCU)
 - RCU implementations

Futex

Fast Userspace Mutex

- Uncontended mutex never goes to the kernel
- A futex consists of an atomic counter in user space memory and a wait queue in the kernel.
- The counter is updated with atomic instructions (executed by user space code)
 - `cmpxchg(val, expct, new) → prev`
- `futex_wait()` and `futex_wake()` are system calls that are called only in case of contention, when sleeping is needed

```
class mutex {
public:
    mutex() : val (0) { }

    void lock() {
        int c;
        if ((c = cmpxchg (val, 0, 1)) != 0) {
            if (c != 2)
                c = xchg (val, 2);
            while (c != 0) {
                futex_wait (&val, 2);
                c = xchg (val, 2);
            }
        }
    }

    void unlock() {
        if (atomic_dec (val) != 1) {
            val = 0;
            futex_wake (&val, 1);
        }
    }
private:
    int val;
};
```

U. Drepper, *Futexes Are Tricky*, 2011,
 Online: <https://www.akkadia.org/drepper/futex.pdf>

Futex uses

- Futex primitive can be used to implement the following higher-level synchronization mechanisms:
 - Mutexes
 - Semaphores
 - Conditional variables
 - Thread barriers
 - Read-write locks
- Today, `pthread_mutex_t` in Linux is implemented via futex.
- JVM's synchronization via Mark word and thin/fat locking is conceptually the same as futex (see next lectures)

Why are futexes tricky?

- What happens when a futex is shared between processes and one **process unexpectedly crashes**, when it has locked the mutex?
- Deadlock! The user space counter stays in the locked state, because the only one who can unlock it has crashed. Since the kernel is not aware about the locking operation (in the uncontended case), it cannot help here.
- Linux solves this problem with so called “**Robust futex**” implementation. The locations of futex counters are registered with the kernel at initialization time. When the process crashes, the kernel scans its user space memory for locked mutexes and unlocks them.

Outline

- 1 Naive synchronization
 - Problems
 - Non-atomic data manipulation
 - Unwanted compiler optimizations
 - Reordering at hardware level
 - Deadlock
 - Busy waiting (spinning)
- 2 Real-world synchronization (locking)
 - Kernel semaphores
 - Futex
 - **Cost of atomic operations and barriers**
- 3 Read-mostly workload
- 4 Read-Copy-Update (RCU)
 - RCU implementations

Cost of atomic operations & barriers

16-CPU 2.8GHz Intel X5550 (Nehalem) System

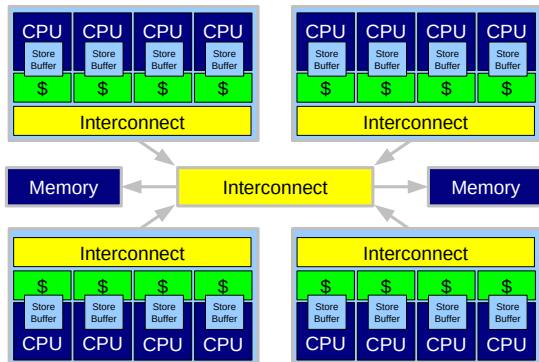
Operation	Cost (ns)	Ratio
Clock period	0.4	1.0
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss (off-core)	31.2	86.6
CAS cache miss (off-core)	31.2	86.5
Single cache miss (off-socket)	92.4	256.7
CAS cache miss (off-socket)	95.9	266.4

Source: Paul E. McKenney, IBM

- **Atomic operations are costly** (here 19–266 times slower than non-atomic operations)
 - Why? Atomic operations require exclusive access (in terms of cache coherency protocol) to the memory location, which, in the worst case, requires **broadcast** over CPU interconnect bus.
- **Barriers are typically cheaper** (weak barriers more that full barriers)

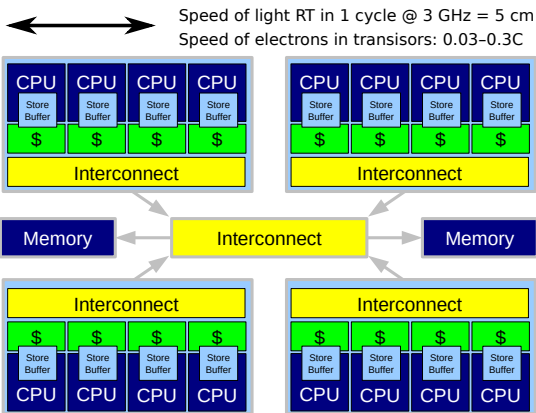
Cost of atomic operations & laws of physics

System with 4 sockets, each with 4 CPU cores



Cost of atomic operations & laws of physics

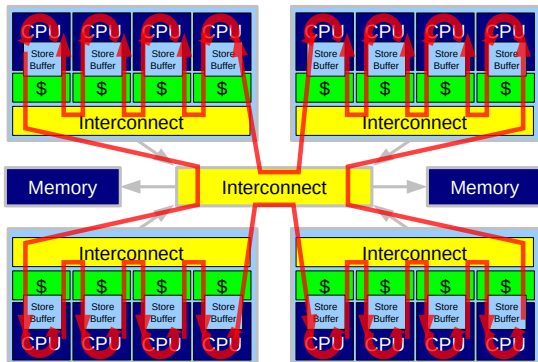
System with 4 sockets, each with 4 CPU cores



Cost of atomic operations & laws of physics

System with 4 sockets, each with 4 CPU cores

All CPUs executing atomic increment of global variable

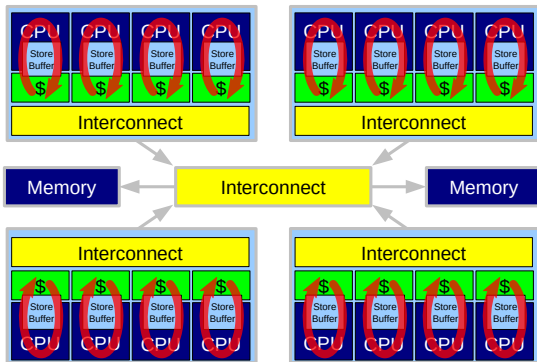


Every CPU experiences a cache miss, because other CPUs access the variable as well
 ⇒ Sequential execution

Cost of atomic operations & laws of physics

System with 4 sockets, each with 4 CPU cores

All CPUs executing atomic increment of per-cpu variable



No cache miss \Rightarrow much faster & parallel

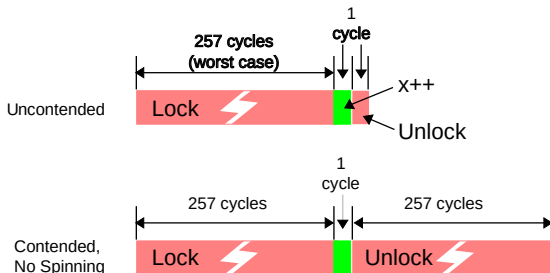
Locking overhead

- Classical locks are typically implemented with atomic instructions and ensure that lock manipulation is not reordered with critical section content.

```
pthread_mutex_lock(mutex);
x++;
pthread_mutex_unlock(mutex);
```

- Uncontended case: during lock(), mutex is not in the cache, during unlock() it is
- Contended case: mutex is not in the cache even during unlock, because there is (probably) another CPU trying to lock the mutex and thus “stealing” the lock from mutex-owner’s cache

Single-instruction critical sections protected by a lock



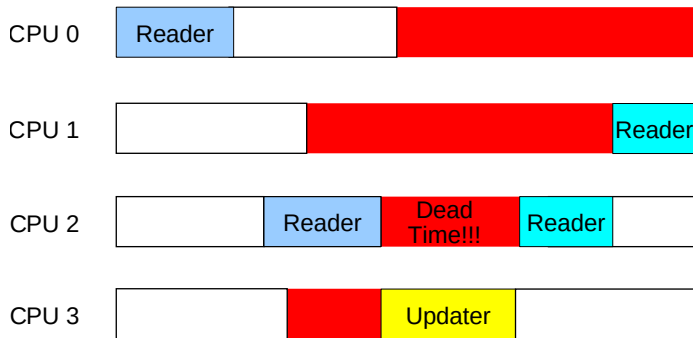
Outline

- 1 Naive synchronization
 - Problems
 - Non-atomic data manipulation
 - Unwanted compiler optimizations
 - Reordering at hardware level
 - Deadlock
 - Busy waiting (spinning)
- 2 Real-world synchronization (locking)
 - Kernel semaphores
 - Futex
 - Cost of atomic operations and barriers
- 3 Read-mostly workload
- 4 Read-Copy-Update (RCU)
 - RCU implementations

The problem of mutex

Mutual exclusion in massively parallel **read-mostly workload**

- 1 Lock/unlock overhead
- 2 Dead time during updates

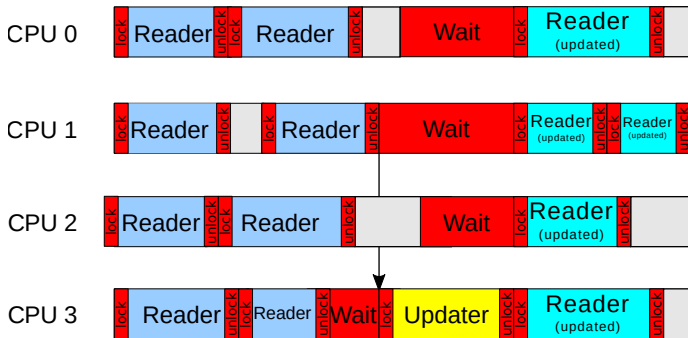


Read-mostly workload

- Many real-world workloads are “read-mostly”
- Caching – Most users only read the data. The cache is updated only from time to time (new article is published, etc.)
 - Web site cache (template rendering)
 - Content delivery network (CDN)
 - File system access
 - Domain Name System (DNS)
 - ...
- Maintaining consistency while updating – synchronization between readers and updaters (writers)

Read-Write lock

- Classical solution – read-write lock
- Multiple readers can read simultaneously
- Update blocks all readers



- Can be implemented on top of mutex(es)
- **Scales badly** due to atomic instructions in lock/unlock.

Simple RW-lock implementation

Readers have priority over writers

```
struct rwlock {
    semaphore_t sem;
    mutex_t readers_lock;
    unsigned readers_active;
};

void write_lock(rwlock &rwl) {
    rwl.sem.wait();
}

void write_unlock(rwlock &rwl) {
    rwl.sem.post();
}
```

```
void read_lock(rwlock &rwl) {
    rwl.readers_lock.lock();
    rwl.readers_active++;
    if (rwl.readers_active == 1)
        rwl.sem.wait();
    rwl.readers_lock.unlock();
}

void read_unlock(rwlock &rwl) {
    rwl.readers_lock.lock();
    rwl.readers_active--;
    if (rwl.readers_active == 0)
        rwl.sem.post();
    rwl.readers_lock.unlock();
}
```

Simple RW-lock implementation

Writers have priority over readers

```

struct rwlock {
    mutex_t lock;
    cond_t cond;
    unsigned rd_active;
    unsigned wr_waiting;
    bool wr_active;
};

void write_lock(rwlock &rw1) {
    rw1.lock.wait();
    rw1.wr_waiting++;
    while (rw1.rd_active || rw1.wr_active)
        cond_wait(rw1.cond, rw1.lock);
    rw1.wr_waiting--;
    rw1.wr_active = true;
    rw1.lock.unlock();
}

void write_unlock(rwlock &rw1) {
    rw1.lock.wait();
    rw1.wr_active = false;
    cond_broadcast(rw1.cond);
    rw1.lock.unlock();
}

```

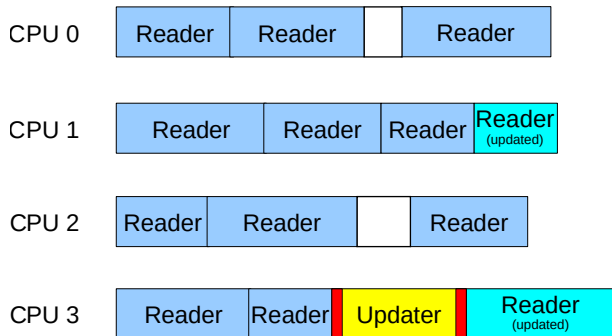
```

void read_lock(rwlock &rw1) {
    rw1.lock.lock();
    while (rw1.wr_waiting || rw1.wr_active)
        cond_wait(rw1.cond, rw1.lock);
    rw1.rd_active++;
    rw1.lock.unlock();
}

void read_unlock(rwlock &rw1) {
    rw1.lock.lock();
    rw1.rd_active--;
    if (rw1.rd_active == 0)
        cond_broadcast(rw1.cond);
    rw1.lock.unlock();
}

```

We want this

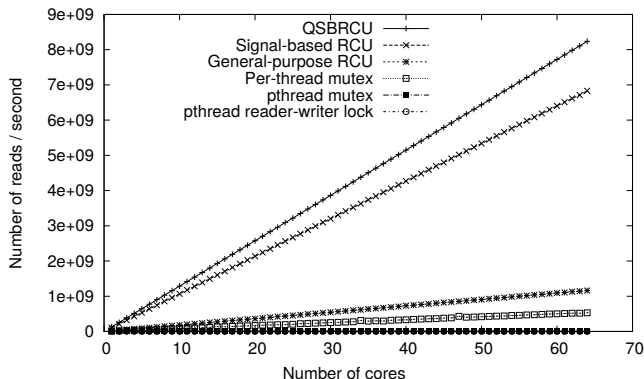


- Readers have no overhead
- Updater does not block readers
- Is that possible? **Yes**
- For what cost? **See next slides**

Outline

- 1 Naive synchronization
 - Problems
 - Non-atomic data manipulation
 - Unwanted compiler optimizations
 - Reordering at hardware level
 - Deadlock
 - Busy waiting (spinning)
- 2 Real-world synchronization (locking)
 - Kernel semaphores
 - Futex
 - Cost of atomic operations and barriers
- 3 Read-mostly workload
- 4 Read-Copy-Update (RCU)
 - RCU implementations

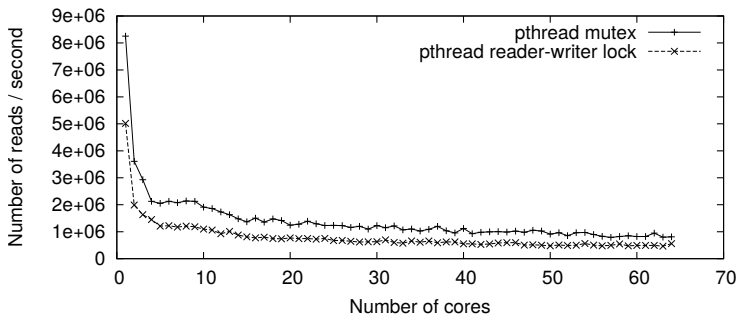
Read-side scalability of synchronization primitives



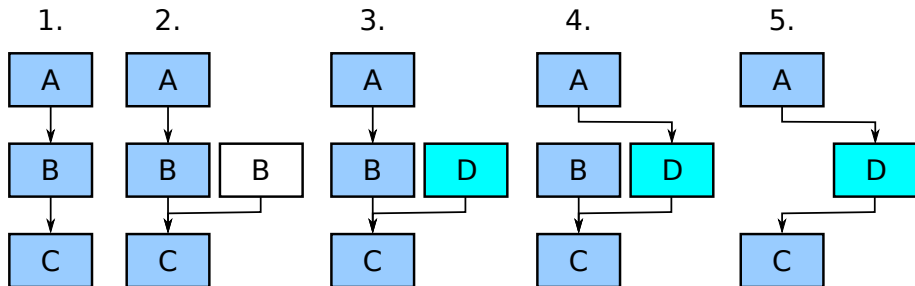
- RCU is **scalable** – typically it scales almost linearly up to hundreds or thousands of CPUs
 - This is possible due to RCU not using **atomic instructions** on the read side.
 - Atomic instructions are not needed because no two threads write to the same variable.
- Locking does **not scale** (see next slide)

Locking scalability

■ Zoomed in version of the previous graph



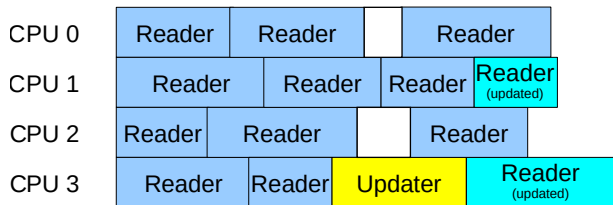
Updating an RCU-protected list



- 1 Original list
- 2 Copy B
- 3 Update B to D
- 4 Make the updated element visible to readers
- 5 Wait after all readers stop accessing B and free it

Steps 1–4 are trivial, making step 5 efficient is why RCU is needed!

How does RCU know when no reader access old data?



- **No explicit (and expensive) tracking** of each reader (e.g. no reference counting, no locking in readers)
- RCU uses **indirect** way of determining the end of all read-side sections
- In certain implementations (QSBR) read-side has **zero overhead**

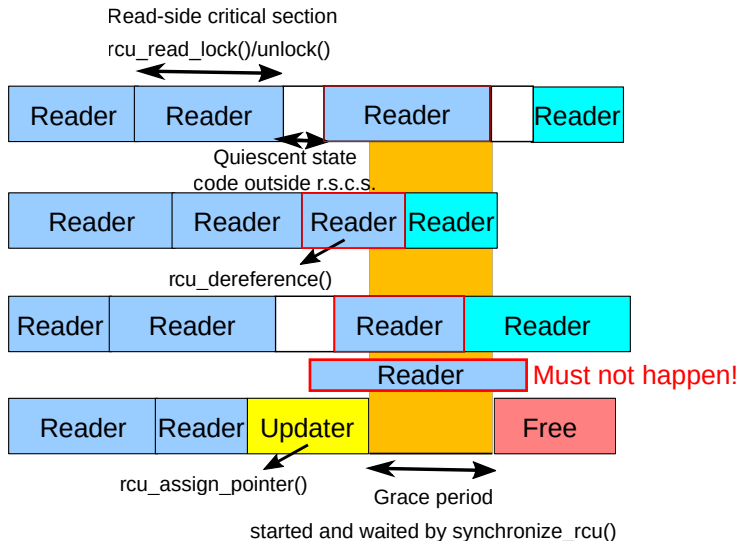
Main mechanisms of RCU

- 1 Publishing of updates (3→4)
 - Ensure that updated data reach memory before the updated pointer
 - Implemented via compiler and memory barriers (no atom. inst.)
(in `rcu_assign_pointer()`)
- 2 Accessing new versions of data (how readers traverse the list)
 - Ensure that we see all the updates made before publishing
 - Compiler and memory barrier in `rcu_dereference()`
- 3 Waiting for all readers to finish
 - **The tricky part!** ⇒ `synchronize_rcu()`

Note: RCU and Java

- Does it makes sense to use RCU in Java?
- No, because JVM decides when to free objects by using a tracing garbage collector. Garbage collectors have much higher overhead than RCU (stop-the-world, marking overhead, ...).

RCU concepts and API



Using RCU

Read-side critical section

```
rcu_read_lock(); /* Start critical section. */  
p = rcu_dereference(cptr);  
/* *p guaranteed to exist. */  
do_something_with(p);  
rcu_read_unlock(); /* End critical section. */  
/* From now on, *p might be freed by an updater!!! */
```

- `rcu_read_lock()/unlock()` and `rcu_dereference()` are cheap, sometimes empty functions (*nop*).
- Updaters are more heavy-weight.

Pitfalls in read-side access (without RCU)

See [PerfBook, 15.3.2]

Instead of:

```
p = rcu_dereference(cptr):
```

Example

```
1 data_t *p = cptr;
2 use(p->field1);
3 use(p->field2);
4 ...
5 data_t *p = cptr;
6 use(p->field1);
7 use(p->field2);
```

- 1 On DEC Alpha, line 1 can be executed after line 2.
- 2 Compiler can omit the load at line 5 and use the value from line 1.
- 3 Compiler can load `cptr` multiple times – e.g. `cptr` can be reloaded between lines 2 and 3. If `*cptr` changes its value, lines 2 and 3 will use different value of `p`.

Using RCU

Updater

```
pthread_mutex_lock(&updater_lock);
old_p = cptr;
*new_p = *old_p;           // copy if needed
new_p->... = ...;
rcu_assign_pointer(cptr, new_p); // update
pthread_mutex_unlock(&updater_lock);
synchronize_rcu();         // wait for grace period end
free(old_p);
```

- Updater can use locks, because we deal with read-mostly workload, where updates are infrequent.
- Locks are not needed if there is just one updater

Outline

- 1 Naive synchronization
 - Problems
 - Non-atomic data manipulation
 - Unwanted compiler optimizations
 - Reordering at hardware level
 - Deadlock
 - Busy waiting (spinning)
- 2 Real-world synchronization (locking)
 - Kernel semaphores
 - Futex
 - Cost of atomic operations and barriers
- 3 Read-mostly workload
- 4 Read-Copy-Update (RCU)
 - RCU implementations

How does it work?

- Many implementations possible
- Trade-off between read-side overhead and constraints of application structure
- We will look at the following implementations:
 - Quiescent-state based reclamation (QSBR)
 - General-purpose
- See <https://www.efficios.com/pub/rcu/urcu-suppl.pdf>, Appendix D for more implementations and details.

Quiescent-state based reclamation (QSBR) example

	■ Exec, □ Sleep				rcu_gp_ctr	rcu_reader.ctr			
	CPU0	CPU1	CPU2	CPU3		CPU0	CPU1	CPU2	CPU3
← Time									
Startup					1				
rcu_register_thread	■	■	■	■		1	1	1	1
rcu_quiescent_state	■								
rcu_quiescent_state		■							
synchronize_rcu start				■	3				0
rcu_quiescent_state	■			□		3			
rcu_quiescent_state			■	□			3		
rcu_quiescent_state		■		□		3			
synchronize_rcu end				■					3
:									
synchronize_rcu start		■			5	0			
rcu_quiescent_state		□		■					5
:		□							
rcu_quiescent_state	■	□				5			
rcu_quiescent_state		□	■				5		
:		□							
synchronize_rcu end		■				5			

Global `rcu_gp_ctr` counter is incremented by updaters.

Per-thread `rcu_reader.ctr` counters are updated by readers in `rcu_quiescent_state`.

“■” represents execution of a RCU related operation. At other times, the CPU can execute other parts of the application.

Note: `synchronize_rcu` is not guaranteed to **end immediately** when the last CPU calls `rcu_quiescent_state`! We tolerate overhead for updaters in order to reduce overhead of readers.

Quiescent-state based reclamation (QSBR)

```

// Protects registry from concurrent accesses
pthread_mutex_t rcu_gp_lock =
    PTHREAD_MUTEX_INITIALIZER;

LIST_HEAD(registry);

struct rcu_reader {
    // bit 0 = online, bits 1-63 = counter
    unsigned long ctr;
    struct list_head node;
    pthread_t tid;
};

// per-thread variable
struct rcu_reader __thread rcu_reader;

void rcu_register_thread(void) {
    rcu_reader.tid = pthread_self();
    mutex_lock(&rcu_gp_lock);
    list_add(&rcu_reader.node, &registry);
    mutex_unlock(&rcu_gp_lock);
    rcu_thread_online();
}

void rcu_unregister_thread(void) {
    rcu_thread_offline();
    mutex_lock(&rcu_gp_lock);
    list_del(&rcu_reader.node);
    mutex_unlock(&rcu_gp_lock);
}

#define RCU_GP_ONLINE 0x1
#define RCU_GP_CTR 0x2

// global counter - note: not accessed with atomic instr.
unsigned long rcu_gp_ctr = RCU_GP_ONLINE;

static inline void rcu_read_lock(void) {}
static inline void rcu_read_unlock(void) {}

// Every thread must call this function periodically
// outside of read-side critical section.
// Note 1: There is no atomic instruction and barriers are
// cheaper than atomic instructions.
// Note 2: This is not called per read-side critical section
// only "from time to time" e.g. in the main program
// loop.
static inline void rcu_quiescent_state(void) {
    smp_mb();
    STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
    smp_mb();
}

// call before a blocking system call
static inline void rcu_thread_offline(void) {
    smp_mb();
    STORE_SHARED(rcu_reader.ctr, 0);
}

// call after return from a blocking system call
static inline void rcu_thread_online(void) {
    STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
    smp_mb();
}

```

Quiescent-state based reclamation (QSBR), cont.

```
// Updater will call these
void synchronize_rcu(void) {
    unsigned long was_online;
    was_online = rcu_reader.ctr;
    smp_mb();
    if (was_online)
        STORE_SHARED(rcu_reader.ctr, 0);
    mutex_lock(&rcu_gp_lock);
    update_counter_and_wait();
    mutex_unlock(&rcu_gp_lock);
    if (was_online)
        STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
    smp_mb();
}

static void update_counter_and_wait(void) {
    struct rcu_reader *index;
    STORE_SHARED(rcu_gp_ctr, rcu_gp_ctr + RCU_GP_CTR);
    barrier();
    list_for_each_entry(index, &registry, node) {
        while (rcu_gp_ongoing(&index->ctr))
            msleep(10);
    }
}

static inline int rcu_gp_ongoing(unsigned long *ctr)
{
    unsigned long v;
    v = LOAD_SHARED(*ctr);
    return v && (v != rcu_gp_ctr);
}
```

Properties:

- Threads must call `rcu_quiescent_state()` from time to time (e.g. in the main event loop).
- Threads must call `rcu_threads_off/online` around blocking calls (e.g. `epoll_wait()`).
 - Otherwise, grace period can never end.
- Grace periods are not shared
- Long waiting \Rightarrow higher memory consumption
- Works only on 64-bit architectures – the counter must not overflow

General-purpose RCU

```

// bit 16 = 1-bit phase counter
#define RCU_GP_CTR_PHASE 0x10000
// bits 0-15 = 16-bit nesting counter
#define RCU_NEST_MASK 0x0ffff
#define RCU_NEST_COUNT 0x1

unsigned long rcu_gp_ctr = RCU_NEST_COUNT;

static inline void rcu_read_lock(void)
{
    unsigned long tmp;
    tmp = rcu_reader.ctr;
    if (!(tmp & RCU_NEST_MASK)) {
        STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
        smp_mb(); // memory barrier (fence) instr.
    } else {
        // Nested critical section does not need a barrier
        STORE_SHARED(rcu_reader.ctr, tmp + RCU_NEST_COUNT);
    }
}

static inline void rcu_read_unlock(void)
{
    smp_mb(); // memory barrier instr.
    STORE_SHARED(rcu_reader.ctr, rcu_reader.ctr - RCU_NEST_COUNT);
}

```

Properties:

- Does not restrict application structure
 - No need to call `rcu_quiescent_state`
 - No need to call `rcu_thread_(on|off)line` around blocking syscalls
- No counter-overflow problem (different mechanism with only 1-bit counters)
- Higher read-side overhead: memory barrier (still less than typical locks, i.e., atomic instructions).
- Supports nesting of read-side critical sections

General-purpose RCU, cont.

```

void synchronize_rcu(void)
{
    // Grace period has two phases
    smp_mb(); // memory barrier instr.
    mutex_lock(&rcu_gp_lock);
    update_counter_and_wait(); // phase 1
    barrier(); // compiler barrier
    update_counter_and_wait();
    mutex_unlock(&rcu_gp_lock); // phase 2
    smp_mb(); // memory barrier instr.
}

static void update_counter_and_wait(void)
{
    struct rcu_reader *index;
    STORE_SHARED(rcu_gp_ctr, rcu_gp_ctr | RCU_GP_CTR_PHASE); // flip the counter bit (^ = XOR)
    barrier(); // compiler barrier
    list_for_each_entry(index, &registry, node) {
        while (rcu_gp_ongoing(&index->ctr))
            msleep(10);
    }
}

static inline int rcu_gp_ongoing(unsigned long *ctr)
{
    unsigned long v;
    v = LOAD_SHARED(*ctr);
    // GP phase ongoing = nesting counter > 0 and global (1-bit) counter != local counter
    // => GP phase ends when nesting counter == 0 (quiescent state) or global (1-bit) counter == local counter
    return (v & RCU_NEST_MASK) > 0 && ((v & rcu_gp_ctr) & RCU_GP_CTR_PHASE);
}

```

General-purpose RCU example

	■ Exec, □ Sleep					rcu_reader.ctr			
← Time	CPU0	CPU1	CPU2	CPU3	rcu_gp_ctr	CPU0	CPU1	CPU2	CPU3
Startup rcu_register_thread ⋮	■	■	■	■	○1	○0	○0	○0	○0
rcu_read_lock	■					○1			
synchronize_rcu start				■	●1				
rcu_read_lock		■		□			●1		
rcu_read_unlock	■			□		○0			
rcu_read_lock			■	□				●1	
synchronize_rcu mid				■	○1				
rcu_read_unlock		■		□			●0		
rcu_read_lock			■	□				●2	
rcu_read_unlock			■	□				●1	
rcu_read_unlock			■	□				●0	
synchronize_rcu end				■					

○0=0x00000
 ○1=0x00001
 ○2=0x00002
 ●0=0x10000
 ●1=0x10001
 ●2=0x10002

Why is not a single phase sufficient?
 Synchronization would fail in one corner case (see the details at page 7 of [URCU paper supplemental material](#)).

Update benchmarks

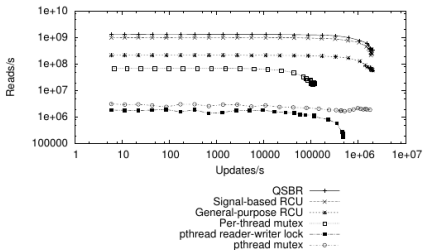


Fig. 9. Update Overhead, 8-core Intel Xeon, Logarithmic Scale

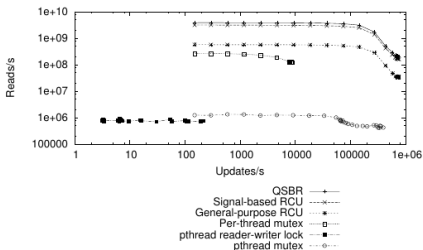


Fig. 10. Update Overhead, 64-core POWER5+, Logarithmic Scale

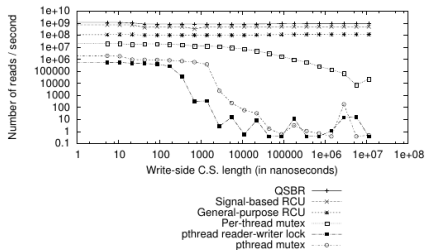


Fig. 11. Impact of Update-Side Critical Section Length on Read-Side, 8-core Intel Xeon, Logarithmic Scale

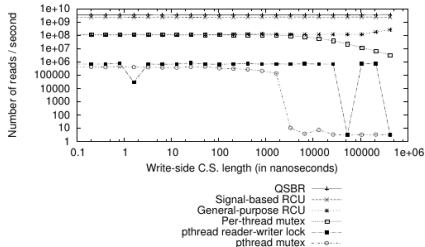


Fig. 12. Impact of Update-Side Critical Section Length on Read-Side, 64-core POWER5+, Logarithmic Scale

Conclusion

- RCU is a scalable synchronization mechanism for hundreds/thousands of CPUs and read-mostly workload
- We have seen an RCU-based implementation of single-linked list, but many other common data structures can be implemented in an RCU-compatible way

References

- [PerfBook] Paul E. McKenney et al.: Is Parallel Programming Hard, And, If So, What Can You Do About It?
<https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html>
- Desnoyers, Mathieu, McKenney, Paul. E., Stern, Alan S., Dagenais, Michel R. and Walpole, Jonathan, User-Level Implementations of Read-Copy Update. IEEE Transaction on Parallel and Distributed Systems, 23 (2): 375-382 (2012).
<https://www.efficios.com/publications>
(especially: <https://www.efficios.com/pub/rcu/urcu-main.pdf>, <https://www.efficios.com/pub/rcu/urcu-suppl.pdf>)
- Paul E. McKenney, What Is RCU? Guest Lecture for Technische Universität Dresden
<http://www2.rdrop.com/users/paulmck/RCU/RCU.2014.05.18a.TU-Dresden.pdf>

Appendix

- RCU is being proposed for inclusion into C++ standard:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1122r2.pdf>

- RCU is patented by IBM, which freely licenses the patent to copy-left software. Userspace RCU library (copyleft – LGPL) can be used in proprietary code. Most patents already expired; patents for some implementations are still valid.

<https://lwn.net/Articles/777519/>

<https://www.google.com/search?q=RCU+patents>