B4M36ESW: Efficient software

Lecture 4: Scalable synchronization

Michal Sojka
sojkam1@fel.cvut.cz

March 13, 2017

Synchronization

- Multi-core CPUs are today's norm, many-core CPUs will come tomorrow

Synchronization

- Multi-core CPUs are today's norm, many-core CPUs will come tomorrow
- To take advantage of such a hardware, parallel (multi-threaded) programs must be run on them

Synchronization

- Multi-core CPUs are today's norm, many-core CPUs will come tomorrow
- To take advantage of such a hardware, parallel (multi-threaded) programs must be run on them
- It is not useful when the threads are completely independent, i.e. threads have to communicate (synchronize)

Synchronization

- Multi-core CPUs are today's norm, many-core CPUs will come tomorrow
- To take advantage of such a hardware, parallel (multi-threaded) programs must be run on them
- It is not useful when the threads are completely independent, i.e. threads have to communicate (synchronize)
- Basic forms of synchronization:

Synchronization

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

Synchronization

- Multi-core CPUs are today's norm, many-core CPUs will come tomorrow
- To take advantage of such a hardware, parallel (multi-threaded) programs must be run on them
- It is not useful when the 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. database waits for requests)

Synchronization

- Multi-core CPUs are today's norm, many-core CPUs will come tomorrow
- To take advantage of such a hardware, parallel (multi-threaded) programs must be run on them
- It is not useful when the 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. database waits for requests)
  - ...

Outline

1 Naive synchronization
- Problems

2 Semaphores

3 Futex

4 Real-mostly workload

5 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;
}
```

Naive synchronization

Mutual exclusion

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

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;
}
```

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

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*

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

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

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

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

Atomic operations

- C: `data++;`
- Assembler (x86): `inc ($data)` – uninterruptible
- Hardware: 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

- C: `data++;`
- Assembler (x86): `inc ($data)` – uninterruptible
- Hardware: 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)

Atomic operations

- C: `data++;`
- Assembler (x86): `inc ($data)` – uninterruptible
- Hardware: 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)

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

⇒

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

Compiler optimizations

```
bool locked;

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

- Compiler expects the memory is only modified by the program being compiled
- Locked seems to be useless ⇒ optimize out

Compiler optimizations

```
bool locked;

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

- Compiler expects the memory is only modified by the program being compiled
- Locked seems to be useless ⇒ optimize out
- Compiler is free to reorder operations as long as the result of the computation is the same

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
- Compiler is free to reorder operations as long as the result of the computation is the same

Compiler optimizations cont.

- Defining the variable volatile makes all accesses "volatile" i.e. slow.
- Sometime, we need volatile only certain accesses 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")
```

Hardware reordering

- Different CPU architectures implement different memory consistency models
- 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	

- x86 can reorder stores after loads, i.e. `data` can be read before other CPUs see `locked` set to true!

Hardware reordering

- Different CPU architectures implement different memory consistency models
- 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	

- x86 can reorder stores after loads, i.e. `data` 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.
  - e.g. mfence on x86
  - C11/C++11 atomics allow to specify which ordering has to be maintained

Cost of atomic operations & barriers

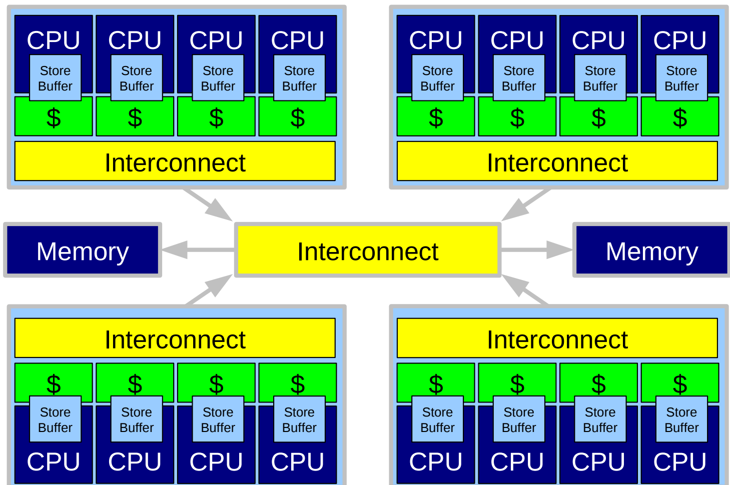
16-CPU 2.8GHz Intel X5550 (Nehalem) System

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

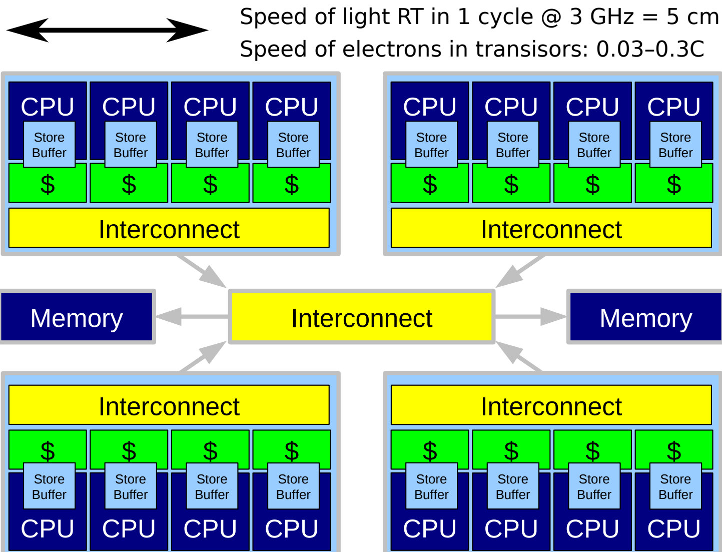
Source: Paul E. McKenney, IBM

- Barriers are typically cheaper (weak barriers more that full barriers)

Cost of atomic operations & laws of physics

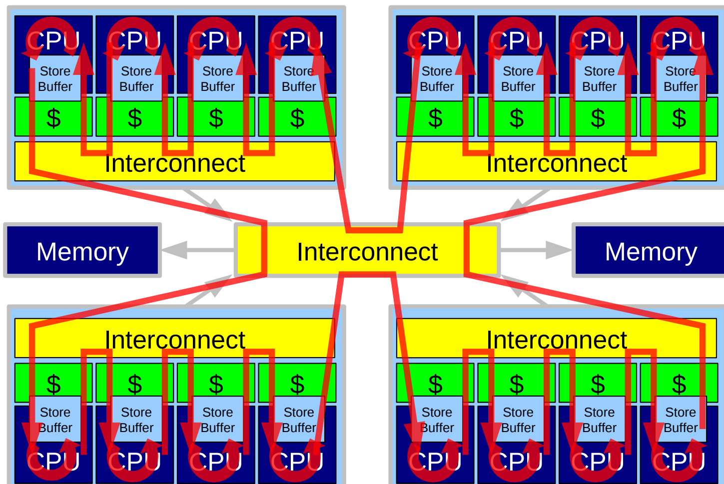


Cost of atomic operations & laws of physics



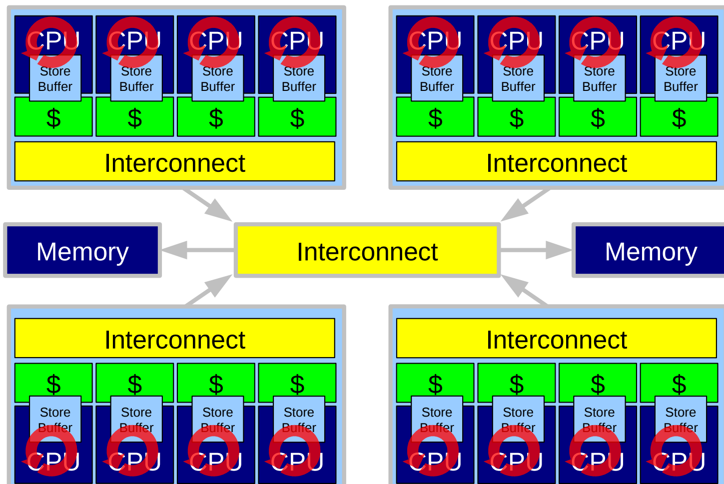
Cost of atomic operations & laws of physics

All CPUs executing atomic increment of global variable

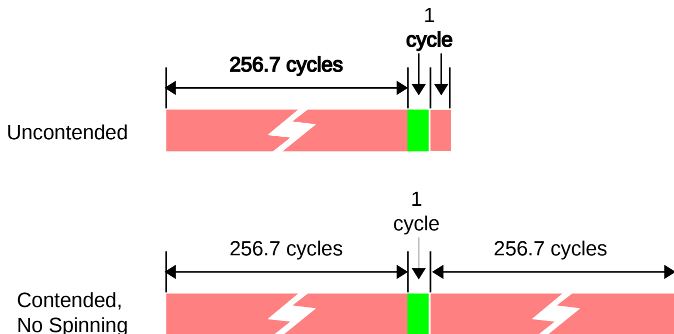


Cost of atomic operations & laws of physics

All CPUs executing atomic increment of per-cpu variable



Locking overhead



Deadlock

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

```
LP_thread     HP_thread
~~~~~~~~~     ~~~~~~~~~

lock();
data++;
    →   preemption   →
                      deadlock();
```

Deadlock

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

```
LP_thread     HP_thread
~~~~~~~~~     ~~~~~~~~~

lock();
data++;
    →   preemption  →
              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

Outline

1 Naive synchronization
- Problems

2 Semaphores

3 Futex

4 Real-mostly workload

5 Read-Copy-Update (RCU)
- RCU implementations

Kernel semaphores

- Each system call adds overhead ($\approx$ 100 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

2 Semaphores

3 Futex

4 Real-mostly workload

5 Read-Copy-Update (RCU)
- RCU implementations

Futex

Fast Userspace Mutex

- Uncontended mutex never goes to kernel
- It uses atomic instruction `cmpxchg(val, expct, new)` → `prev`
- `futex_wait()` and `futex_wake()` are system calls

```
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;
};
```

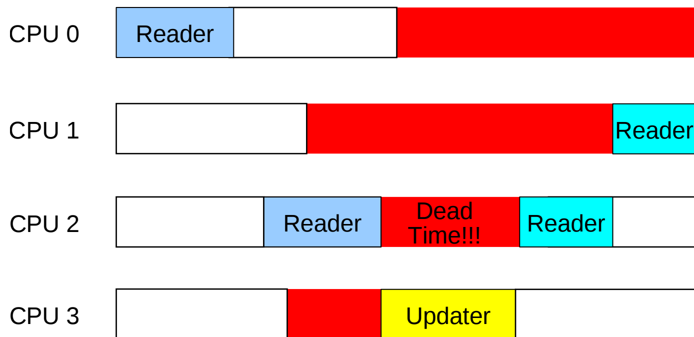
U. Drepper, *Futexes Are Tricky*, 2011, Online: https://www.akkadia.org/drepper/futex.pdf

Futex uses

- Mutexes
- Semaphores
- Conditional variables
- Thread barriers
- Read-write locks

The problem of mutex

Mutual exclusion in read-mostly workload



Outline

1 Naive synchronization
- Problems

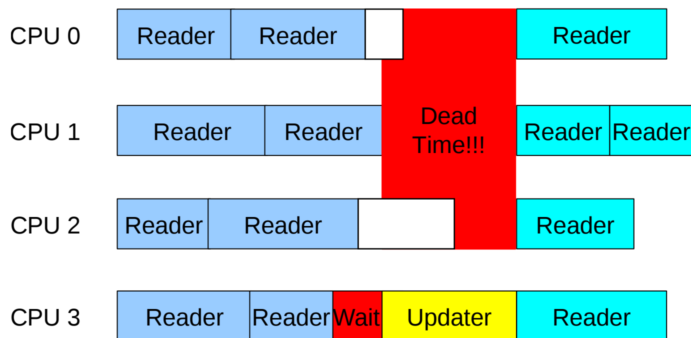
2 Semaphores

3 Futex

4 Real-mostly workload

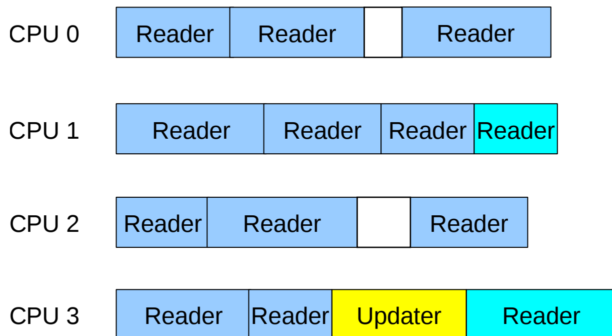
5 Read-Copy-Update (RCU)
- RCU implementations

Read-Write lock



- Update blocks readers
- Can be implemented on top of mutex(es)

We want this



- Updater does not block readers
- Is that possible?

Outline

1 Naive synchronization
- Problems

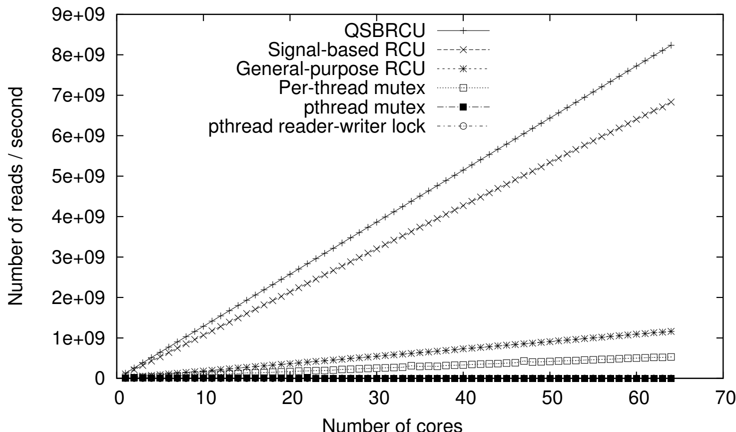
2 Semaphores

3 Futex

4 Real-mostly workload

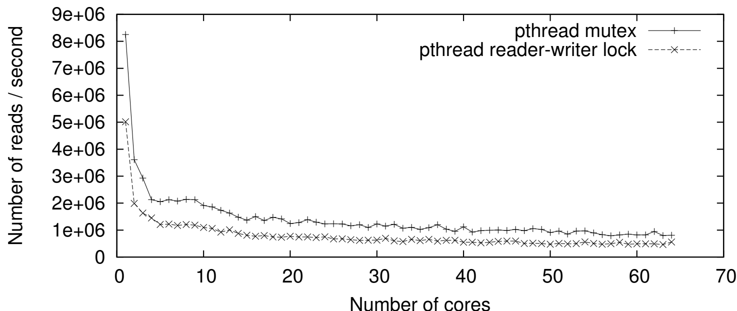
5 Read-Copy-Update (RCU)
- RCU implementations

Read-Copy-Update (RCU)



- Read-side scalability of various synchronization primitives
- RCU is scalable – typically up to hundreds or thousands of CPUs

Read-Copy-Update (RCU)

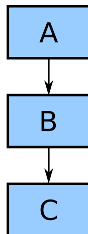


Zoomed in

- Read-side scalability of various synchronization primitives
- RCU is scalable – typically up to hundreds or thousands of CPUs
- Locking does not scale

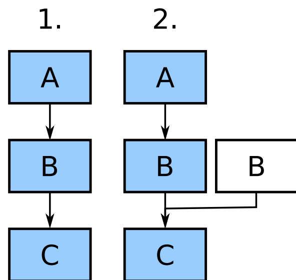
Updating RCU-based list

1.



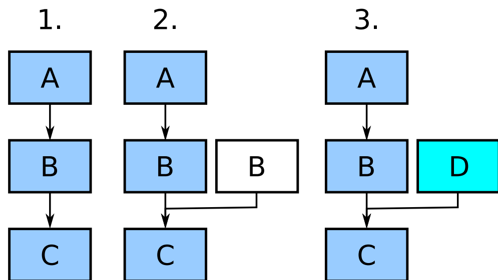
1. Original list

Updating RCU-based list



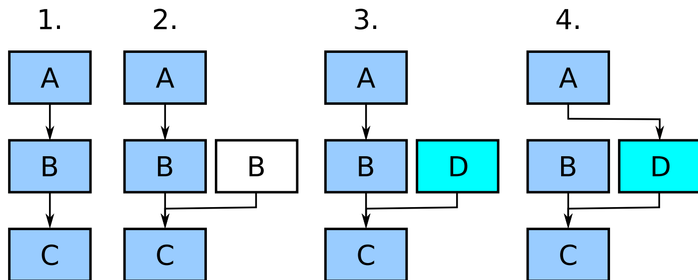
1. Original list
2. Copy B

Updating RCU-based list



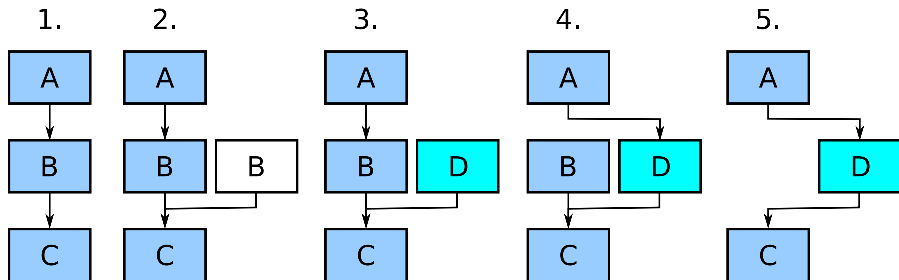
1. Original list
2. Copy B
3. Update B to D

Updating RCU-based list



1. Original list
2. Copy B
3. Update B to D
4. Make the updated element visible to readers

Updating RCU-based 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

Main mechanisms of RCU

1. Publishing of updates (3→4)
   - Compiler and memory barrier

Main mechanisms of RCU

1. Publishing of updates (3→4)
   - Compiler and memory barrier
2. Accessing new versions of data (how readers traverse the list)
   - Compiler and memory barrier

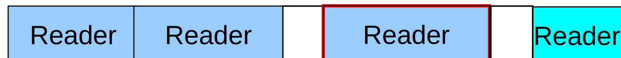
Main mechanisms of RCU

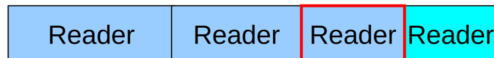
1. Publishing of updates (3$\rightarrow$4)
   - Compiler and memory barrier
2. Accessing new versions of data (how readers traverse the list)
   - Compiler and memory barrier
3. Waiting for all readers to finish

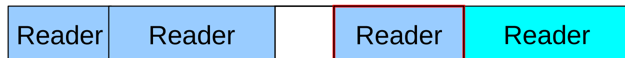
Main mechanisms of RCU

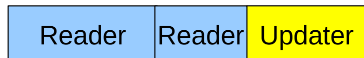
1. Publishing of updates (3→4)
    - Compiler and memory barrier
2. Accessing new versions of data (how readers traverse the list)
    - Compiler and memory barrier
3. Waiting for all readers to finish
    - The tricky part!
    - No explicit (and expensive) tracking of each reader
    - RCU uses indirect way of determining end of all read-side sections
    - In certain implementations (QSBR) this has zero overhead

RCU concepts and API





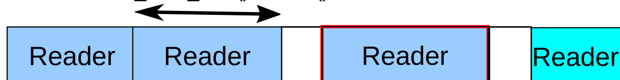


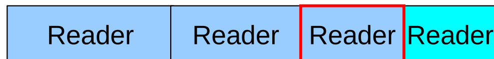


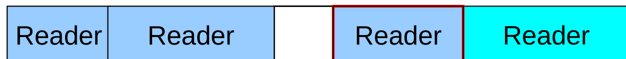
RCU concepts and API

Read-side critical section

rcu_read_lock()/unlock()



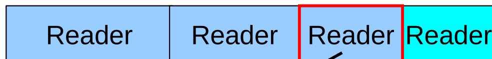




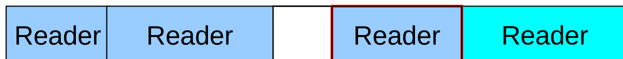
RCU concepts and API

Read-side critical section

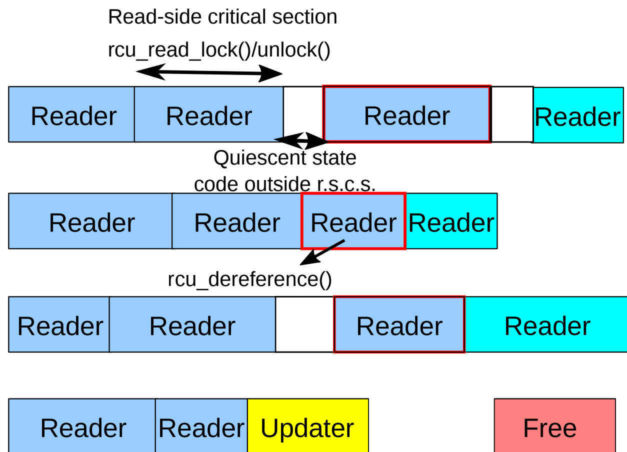
rcu_read_lock()/unlock()



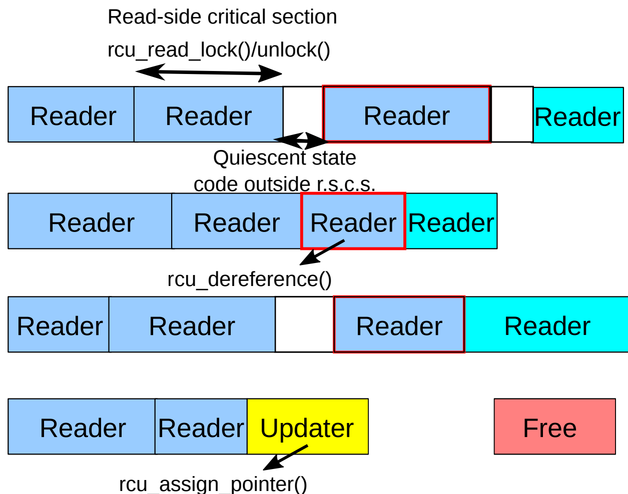
rcu_dereference()



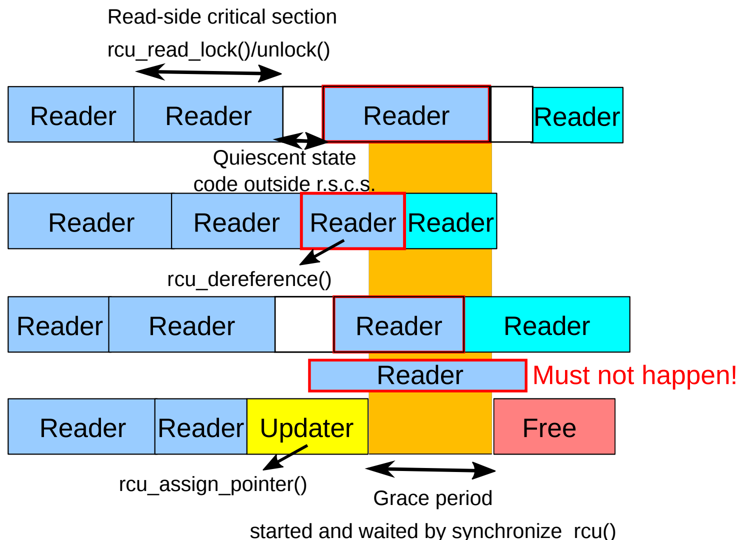
RCU concepts and API



RCU concepts and API



RCU concepts and API



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. */
/* *p might be freed!!! */
```

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

RCU updater

```
pthread_mutex_lock(&updater_lock); /* not needed if there is
                                    * just one updater */
old_p = cptr;
/* copy if needed */
rcu_assign_pointer(cptr, new_p);   /* update */
pthread_mutex_unlock(&updater_lock);
synchronize_rcu();                  /* Wait for grace period.
free(old_p);
```

How does it work?

- Many implementations possible
- Trade-off between read-side overhead and application structure

Quiescent-state based reclamation (QSBR)

```
pthread_mutex_t rcu_gp_lock =
    PTHREAD_MUTEX_INITIALIZER;

LIST_HEAD(registry);

struct rcu_reader {
  unsigned long ctr;
  char need_mb;
  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);
}
```

Quiescent-state based reclamation (QSBR)

```
pthread_mutex_t rcu_gp_lock =
    PTHREAD_MUTEX_INITIALIZER;

LIST_HEAD(registry);

struct rcu_reader {
  unsigned long ctr;
  char need_mb;
  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
unsigned long rcu_gp_ctr = RCU_GP_ONLINE;

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

Quiescent-state based reclamation (QSBR)

```c
pthread_mutex_t rcu_gp_lock =
    PTHREAD_MUTEX_INITIALIZER;

LIST_HEAD(registry);

struct rcu_reader {
  unsigned long ctr;
  char need_mb;
  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);
}
```

```c
#define RCU_GP_ONLINE 0x1
#define RCU_GP_CTR    0x2

// global counter
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.
 */
static inline void rcu_quiescent_state(void) {
  smp_mb();
  STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
  smp_mb();
}
static inline void rcu_thread_offline(void) {
  smp_mb();
  STORE_SHARED(rcu_reader.ctr, 0);
}
static inline void rcu_thread_online(void) {
  STORE_SHARED(rcu_reader.ctr,
  LOAD_SHARED(rcu_gp_ctr));
  smp_mb();
}
```

QSBR

```
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:

- Grace periods are not shared
- Long waiting $\Rightarrow$ higher memory consumption

Conclusion

References

- 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
- 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