RCU
Theory and Practice

LSE
Security System
Laboratory of Epita

Marwan Burelle - LSE Summer Week 2015

Overview

- RCU concepts

  Short overview of how RCU may solves your problems

- Wait for readers

  Userland implementations for real

- Pseudo RCU

  Implementing RCU concepts with non-RCU tools

… it’s all about procrastination …

Postponing operations can solve your synchronization problem ...

… time is very relative …

LSE Security System
Laboratory of Epita

Changes append when you see them !

Read - Copy - Update

What for ?

- Concurrent shared data
- Kind of non-blocking
- Read intensive context
- Few updates
- Minimizing readers overhead

RCU is not only about when to free old pointers ...

The keystone of read-copy update is the ability to determine when all threads have passed through a quiescent state since a particular point in time.

READ-COPY UPDATE: USING EXECUTION HISTORY TO SOLVE CONCURRENCY PROBLEMS
PAUL E. MCKENNEY & JOHN D. SLINGWINE

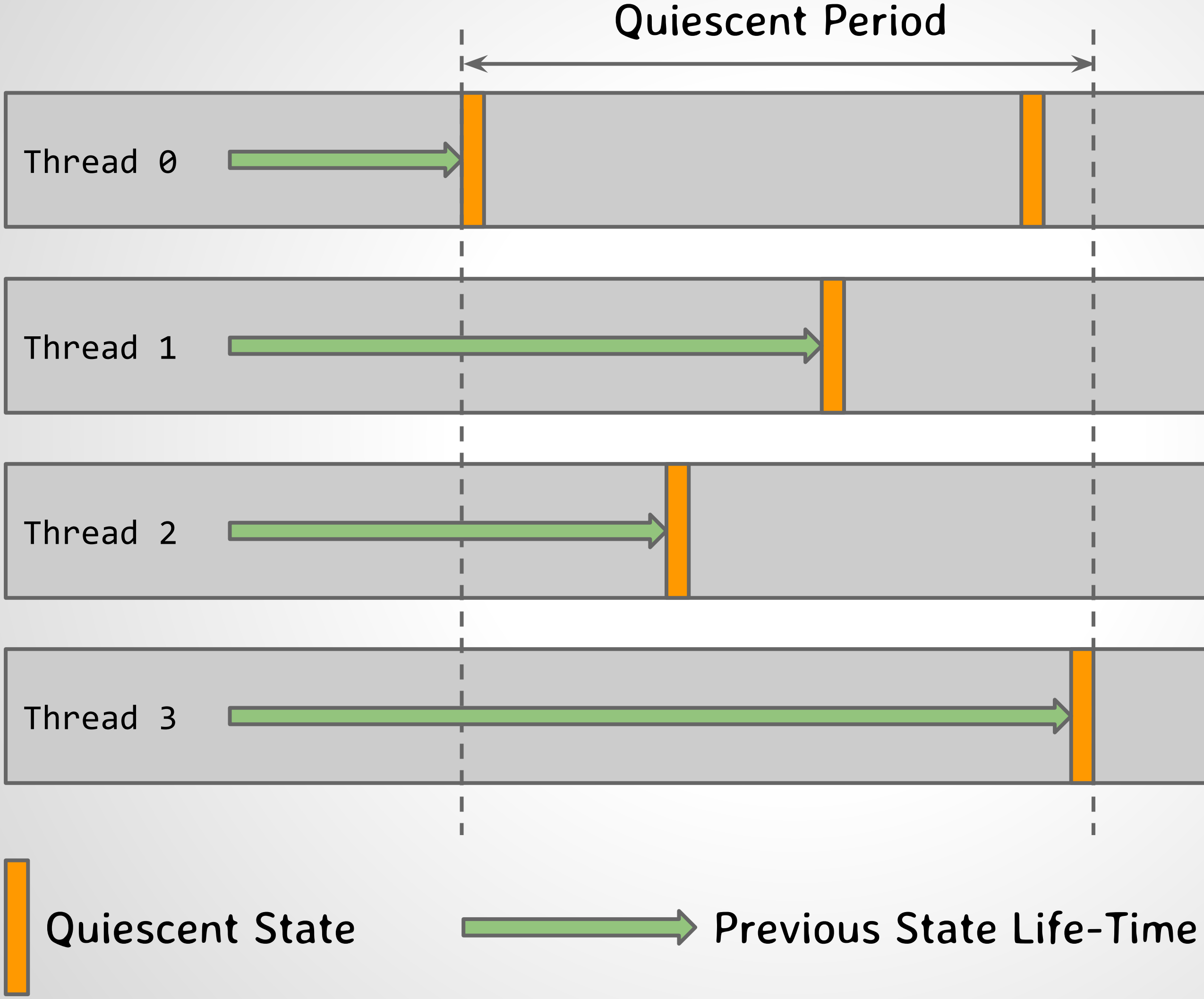
Quiescent State:

when a thread no longer care about shared and protected data structures.

Quiescent Period:

a time interval during which each thread passes through at least one quiescent state.

Quiescent Period



RCU

RCU framework:

- ➢ Wait for readers (WFR)
- ➢ Respect of quiescent period
- ➢ Defines API/Constraints
- ➢ Underlying mechanism

RCU-based algo:

- ➢ **wait-free** readers
- ➢ Use WFR for sync
- ➢ Respect API/Constraints
- ➢ Copy-based updates

Wait For Readers

Key points: writer's operation terminates when all readers have leaved the update region

- Writer offers a *grace* period
- Readers won’t continue longer than this period
- **Quiescent Period** will be our grace period

Simple Buffer Example

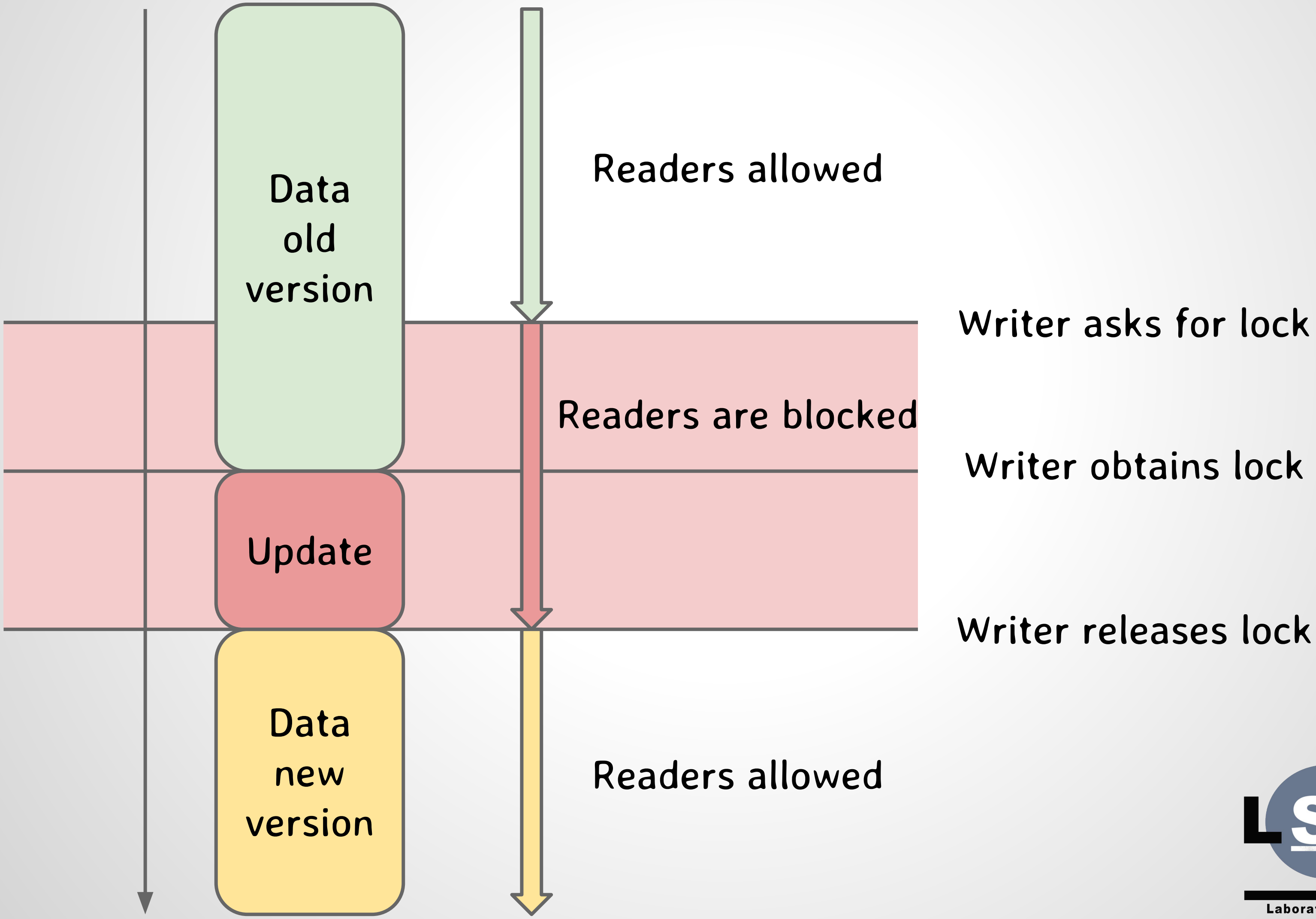
Writer:

- Copy data in new buffer
- Update new buffer
- Replace buffer pointer
- Wait for readers
- Free old buffer

Readers:

- Arrived before update:
  - See old buffer
- Arrived after update
  - See new buffer

Reader/Writer Lock Solution



WASTE OF TIME.
quickmeme.com

Simple Buffer Example

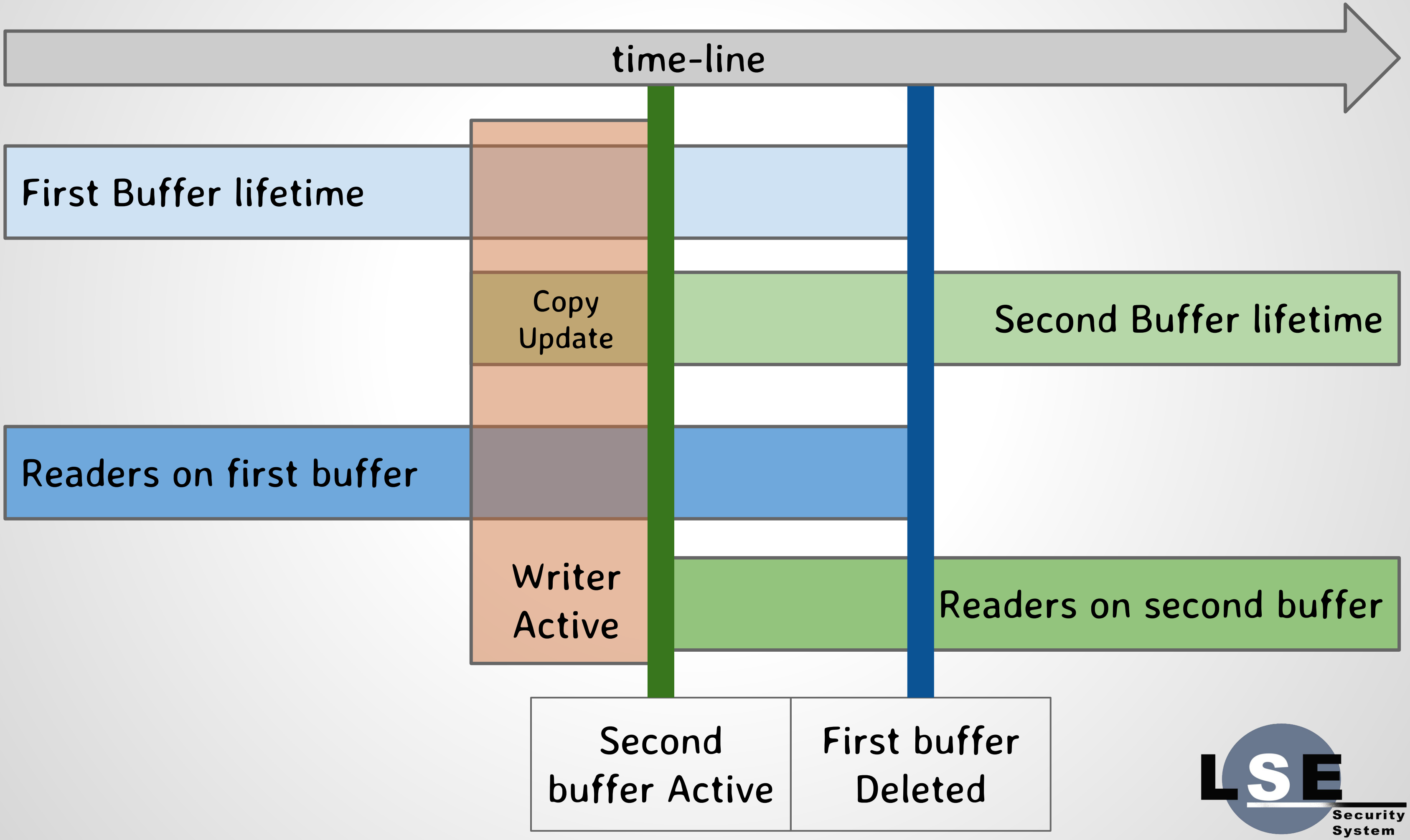
Quiescent State:

- When reader leave buffer’s critical section

Quiescent Period:

- All readers have leaved critical section

Simple Buffer Example



LSE Security System

Laboratory of Epita

Simple Buffer Pseudo Code

Writer:

```
// Sync with other writers
char *old = rcu_dereference(buf)
char *new = malloc( enough )
memcpy(new, old)        // copy
update_content(new)     // update
rcu_assign_pointer(buf, new)
synchronize_rcu()
free(old)
```

Reader:

```
rcu_read_lock()
char *b = rcu_dereference(buf)
read_content(b)           // read
rcu_read_unlock()
```

RCU Linked List

Readers Traversing Loop

```
cur = list-entry-point;
while (cur != NULL) {
  // Your job here
  cur = cur->next;
}
```

Constraints:

➢ Minimize traversing cost

➢ Wait-free (no lock, no spin)

➢ *Independent* iterations

➢ cur must be valid in body

➢ cur can be detached

➢ cur->next must be valid

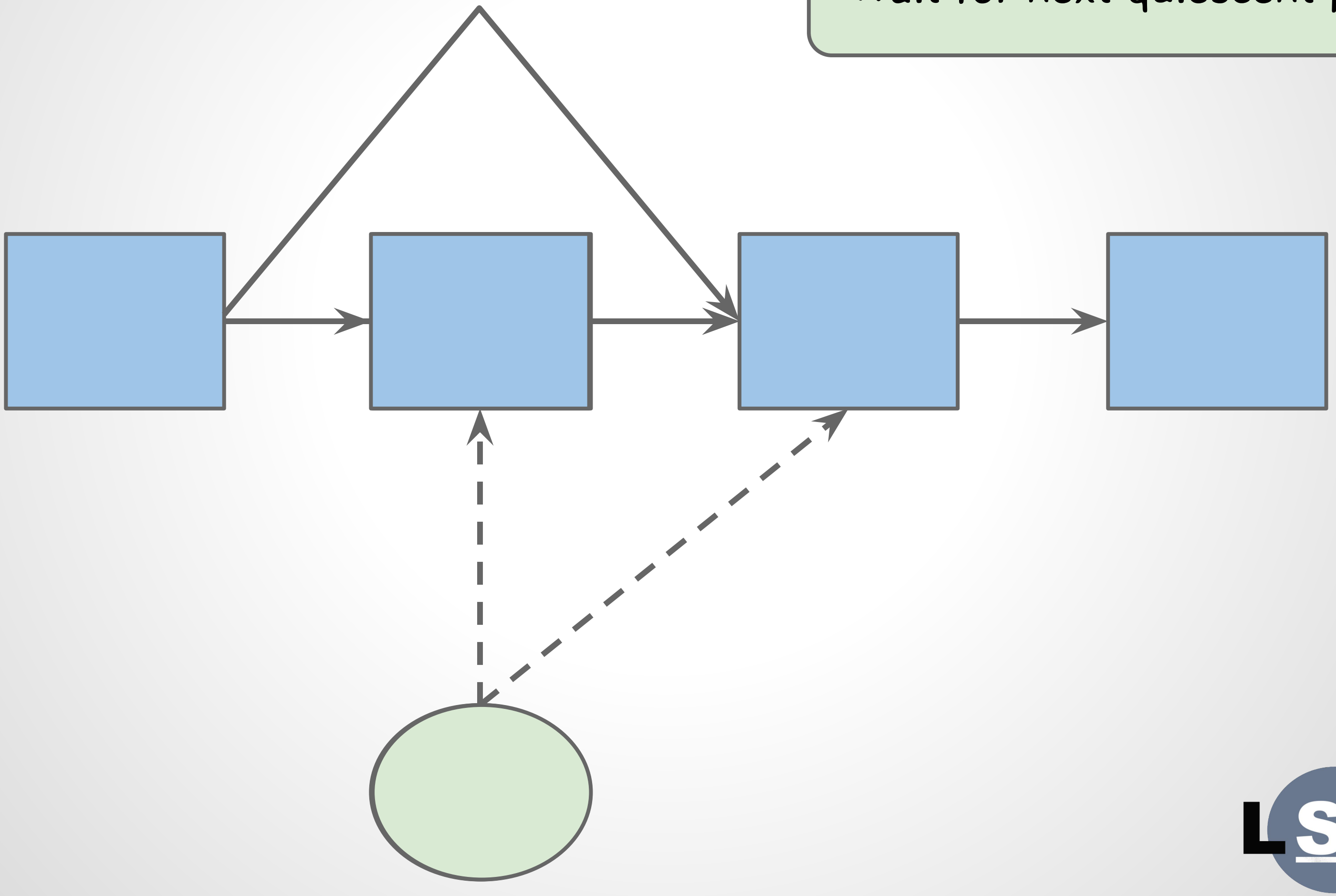
Classical Solutions

- Coarse Grain lock: *enough said …*
- Fine Grain lock: not wait free (chain locking)
- Optimistic lock:
- Lazy lock
- Lock-Free

wait free but with overhead

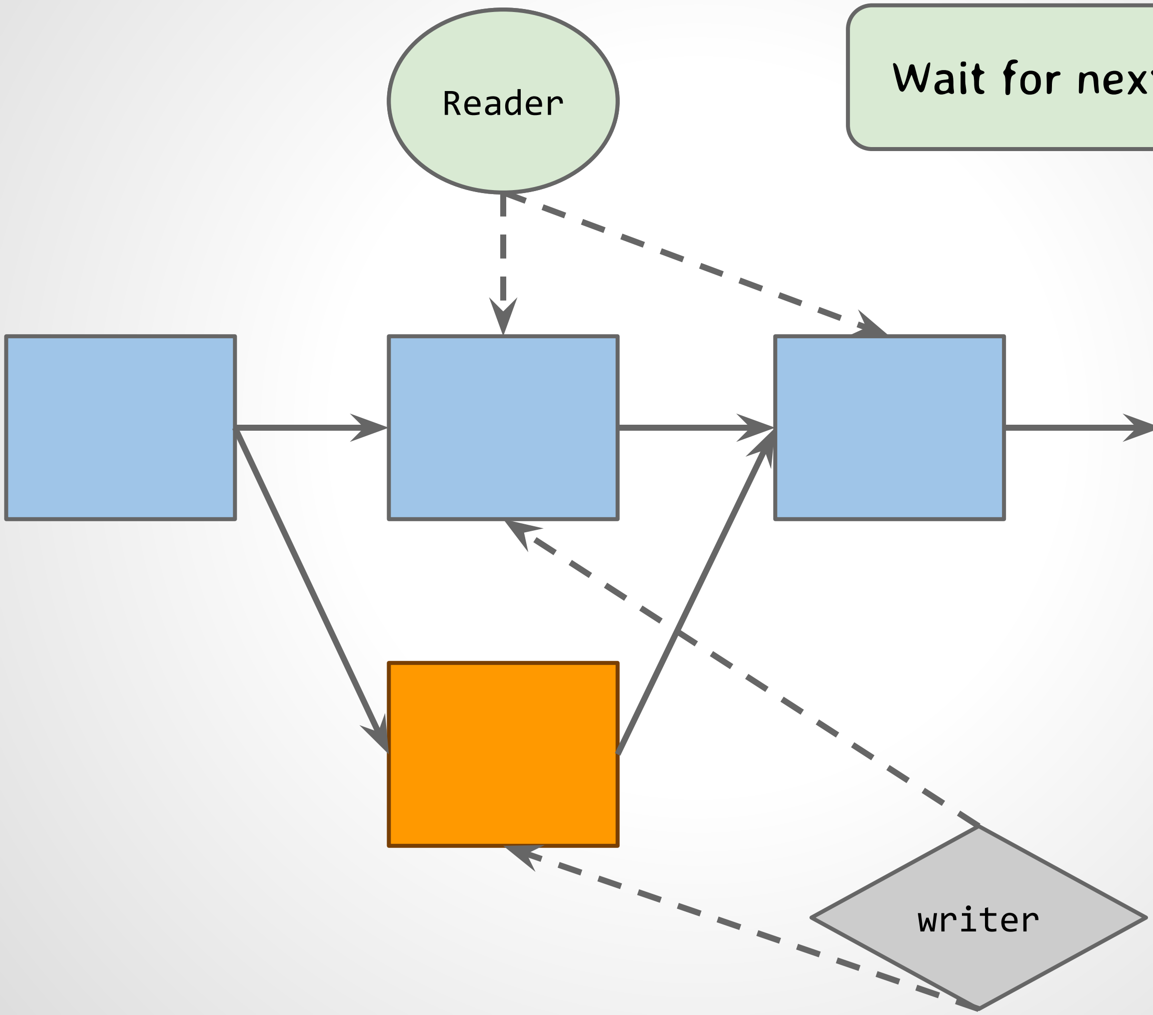
RCU List Delete Element

Wait for next quiescent period.



RCU Update List Element

Wait for next quiescent period.



Writer strategy

Update:

- ➢ Copy the element and update the copy
- ➢ Replace access pointer when ready

Delete:

- ➢ Replace pointer

Both:

- ➢ Wait for readers before deleting old element

Waiting For Readers ?

I'm waiting, I got a lot of time ...

Non-preemptive Kernel

- Threads leave CPU only when complete
- Available CPU means one or more threads gone through quiescent state
- All CPU available means end of quiescent period

And in Userland ?

- Previous strategy is irrealistic
- We need more stuff
- We still want:
  - wait-free readers
  - minimal overhead in readers
  - keep the same *structure*

Common operations

- rcu_dereference: load/consume
- rcu_assign_pointer: store/release
- Compiler barrier

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

- Memory barrier (full sequential consistency)

Common data

Per thread meta-info:

- TLS or like
- Threads need to register
- Readable from writer

Strategies ?

- Use GC/HP like mechanism
- Using RCU reader lock/unlock and barrier
- Update brain ?

Garbage Collecting

- Pretty easy when available
- Price ?
- Not sufficient for certain cases
- More suited ? Eventually Hazard Pointers
- Using Smart counters ?

See later ...

LSE Security System
Laboratory of Epita

Using RCU reader lock/unlock

- Already explicit in the code
- May break requirement of minimal overhead
- Still interesting

Issues

- Wait-free (bounded spin, non blocking ops)
- Nested read sections
- RCU properties must hold ...

Principle

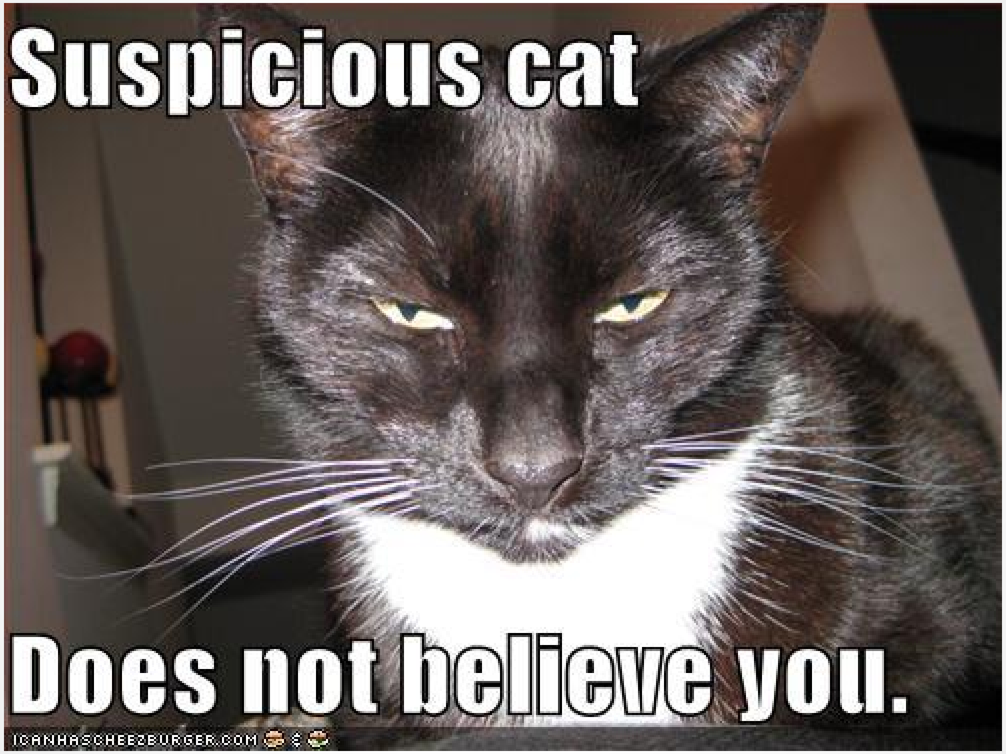
- Per reader *counter* set using memory barrier
  - High order bit: phase (global)
  - lower-order bits: nesting
- Writer does 2 grace period spin-waits
  - spin on each reader with same phase and nesting $\neq 0$
  - 2 grace periods to avoid race condition

Discussion

- Only lock/unlock (read) and synchronize (write)
- Safe and easy to use in all cases

- Single writer (that's RCU, don't care …)
- **Memory barriers are expensive !**

We can eliminate barrier cost using POSIX thread signal !


Suspicious cat
Does not believe you.
ICANHASCHEEZBURGER.COM

Avoiding unneeded barriers

Goal: avoid barriers when no update is running

- Add a *need barrier* tag to meta-info
- Writer set the tag on readers and send a signal
- Signal handler reset the tag
- Signal enforces a real memory barrier

- **Real gain** (check urcu papers)

LSE Security System
Laboratory of Epita

Can do better ?

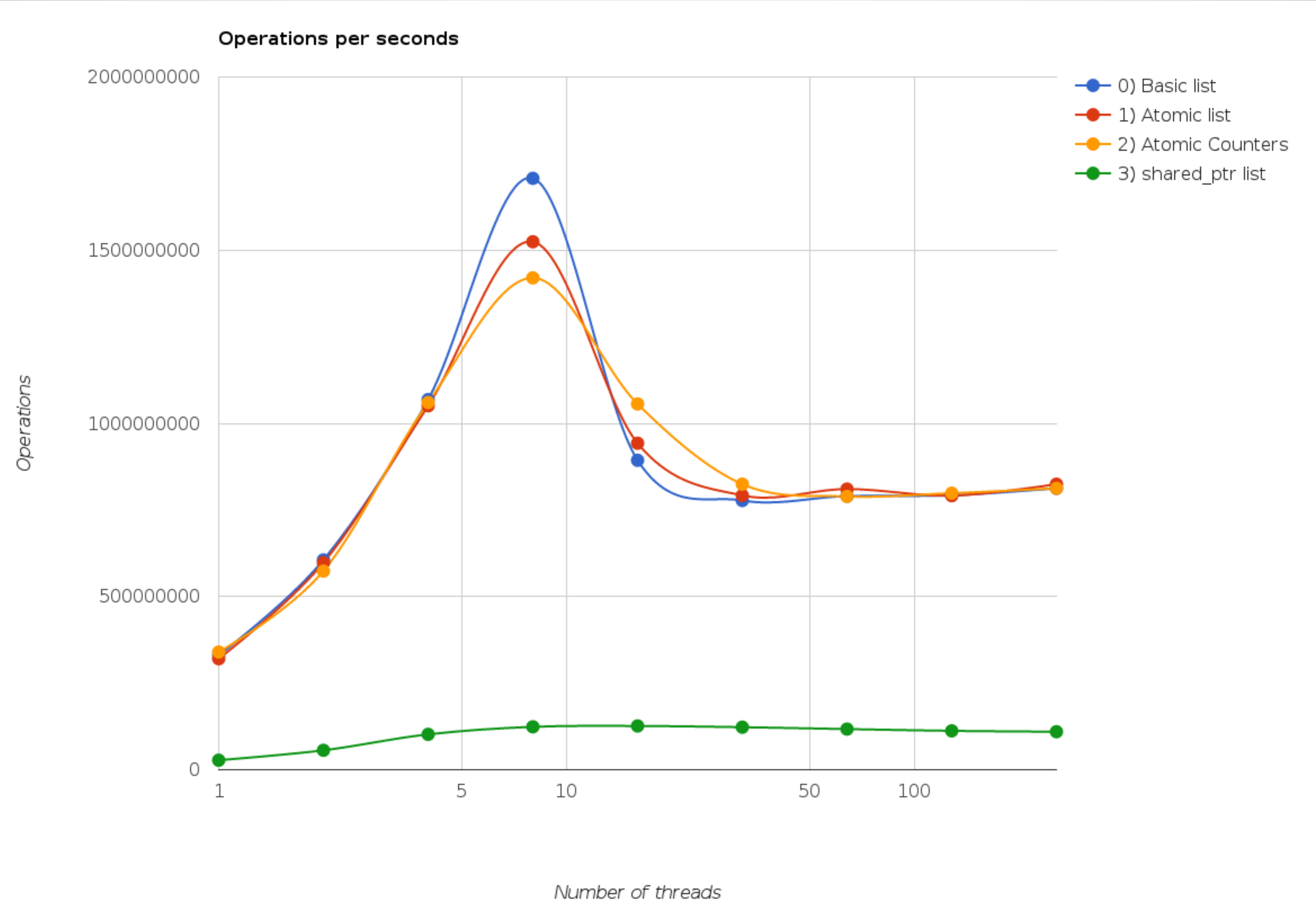
Detecting Quiescent States

- Nothing in lock/unlock
- Explicit indication of quiescent states
- More intrusive but more efficient !

Marking Quiescent State

- Snapshot current period counter in reader
  - wait free operation
- Writer wait while readers have current value
  - blocking, but that’s RCU
- Possible overflow: solved with 64bits counter
  - can be solved using double period also
- Provides also extended quiescent state
  - thread online/offline API

Was it worth the effort ?


Operations per seconds
2000000000
1500000000
1000000000
500000000
0
Operations
1
5
10
50
100
Number of threads
0) Basic list
1) Atomic list
2) Atomic Counters
3) shared_ptr list

Some bench ...

Pseudo RCU

Most RCU-like algorithms can be implemented using other pointers reclamation mechanism.

Using smart pointers

- Pretty easy to set-up
- Use C++11's `std::shared_ptr`
- Smart pointers are synchronized
- Hope ? `std::atomic_is_lock_free ?`

Using Hazard Pointers

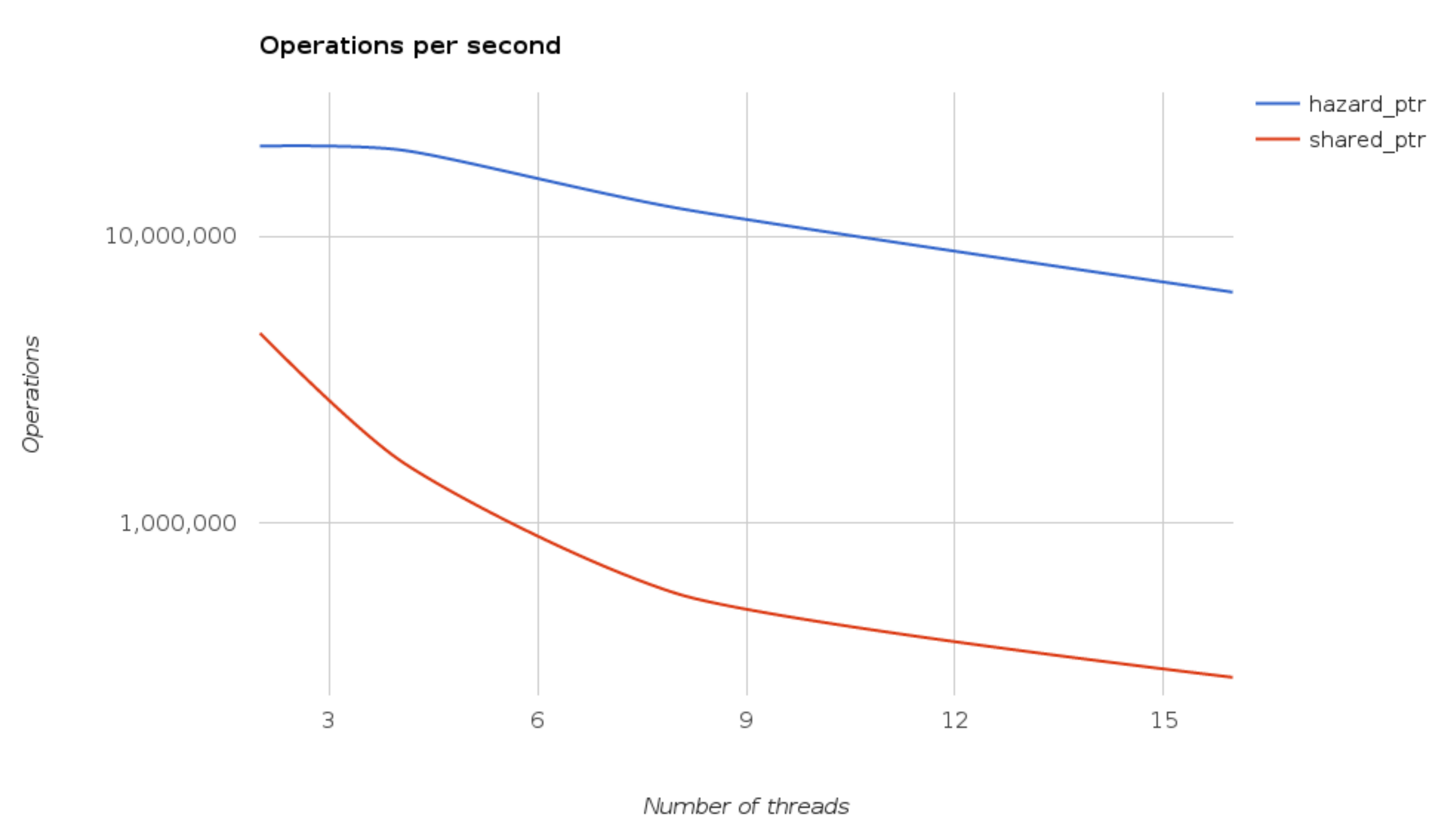
- HP are wait free
- Just need a double check when getting pointer
- HP are another kind of relativistic programming

… once again, it's all about procrastination

LSE Security System

Laboratory of Epita

LAZY DALEK IS IN NO RUSH
PROCRASTINATE!!
DIYLOL.COM

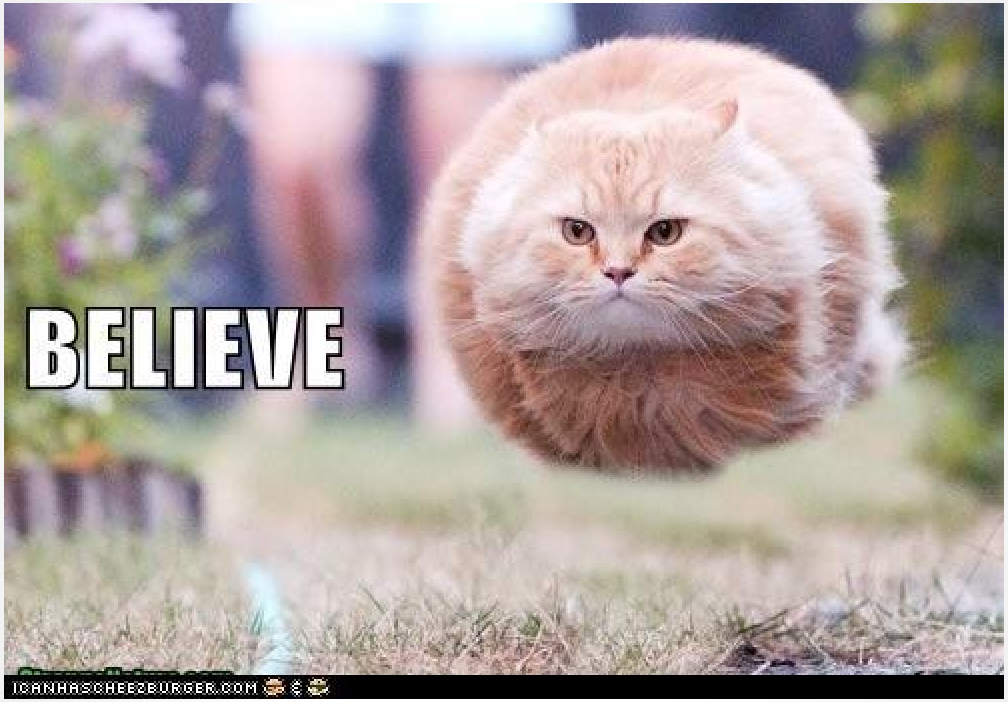


RCU-like shared buffer

Readers checking content

Occasional single writer update buffer

Lock-free shared_ptr is like ...


BELIEVE
ICANHASCHEEZBURGER.COM

I has a question...

Readings

- RCU author's page: http://www.rdrop.com/~paulmck/RCU/
  Lot of links and useful articles
- *User-Level Implementations of Read-Copy Update*
  Desnoyers, McKenney, Stern, Dagenais and Walpole
  IEEE Transaction on Parallel and Distributed Systems, 23 (2): 375-382 (2012)
- *Structured Deferral: Synchronization via Procrastination*
  Paul E. McKenney, ACM Queue 2013
- *Introduction to RCU Concepts*
  Liberal application of procrastination for accommodation of the laws of physics – for more than two decades!
  Paul E. McKenney, LinuxCon 2013