Is Parallel Programming Hard, And If So, What Can You Do About It?

Paul E. McKenney
IBM Distinguished Engineer & CTO Linux
Linux Technology Center

April 28, 2011

Who is Paul and How Did He Get This Way?

Who is Paul and How Did He Get This Way?

- Grew up in rural Oregon, USA
- First use of computer in high school (72-76)
  - IBM mainframe: punched cards and FORTRAN
  - Later ASR-33 TTY and BASIC
- BSME & BSCS, Oregon State University (76-81)
  - Tuition provided by FORTRAN and COBOL
- Contract Programming and Consulting (81-85)
  - Building control system (Pascal on z80)
  - Security card-access system (Pascal on PDP-11)
  - Dining hall system (Pascal on PDP-11)
  - Acoustic navigation system (C on PDP-11)

Who is Paul and How Did He Get This Way?

- SRI International (85-90)
  - UNIX systems administration
  - Packet-radio research
  - Internet protocol research
  - MSCS Computer Science (88)
- Sequent Computer Systems (90-00)
  - Communications performance
  - Parallel programming: memory allocators, RCU, ...
- IBM LTC (00-present)
  - NUMA-aware and brlock-like locking primitive in AIX
  - RCU maintainer for Linux kernel
  - Ph.D. Computer Science and Engineering (04)

Who is Paul and How Did He Get This Way?

- SRI International (85-90)
  - UNIX systems administration
  - Packet-radio research
  - Internet protocol research
  - MSCS Computer Science (88)
- Sequent Computer Systems (90-00)
  - Communications performance
  - Parallel programming: memory allocators, RCU, ...
- IBM LTC (00-present)
  - NUMA-aware and brlock-like locking primitive in AIX
  - RCU maintainer for Linux kernel
  - Ph.D. Computer Science and Engineering (04)
- Paul has been doing parallel programming for 20 years

Who is Paul and How Did He Get This Way?

- SRI International (85-90)
  - UNIX systems administration
  - Packet-radio research
  - Internet protocol research
  - MSCS Computer Science (88)
- Sequent Computer Systems (90-00)
  - Communications performance
  - Parallel programming: memory allocators, RCU, ...
- IBM LTC (00-present)
  - NUMA-aware and brlock-like locking primitive in AIX
  - RCU maintainer for Linux kernel
  - Ph.D. Computer Science and Engineering (04)
- Paul has been doing parallel programming for 20 years
  - What is so hard about parallel programming, then???

Why Has Parallel Programming Been Hard?

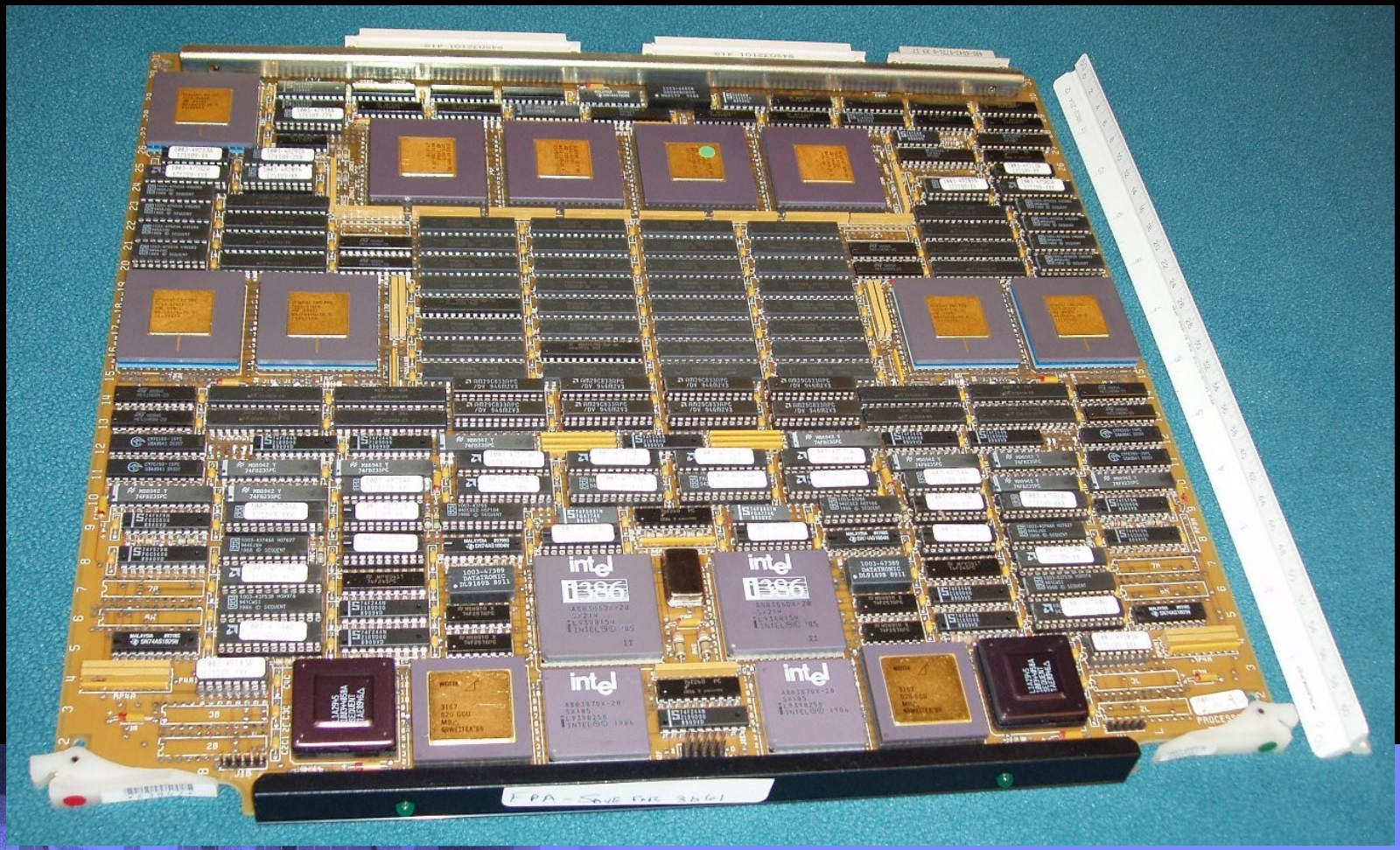
Why Has Parallel Programming Been Hard?

- Parallel systems were rare and expensive
- Very few developers had access to parallel systems: little ROI for parallel languages & tools
- Almost no publicly accessible parallel code
- Parallel-programming engineering discipline confined to a few proprietary projects
- Technological obstacles:
  - Synchronization overhead
  - Deadlock
  - Data races

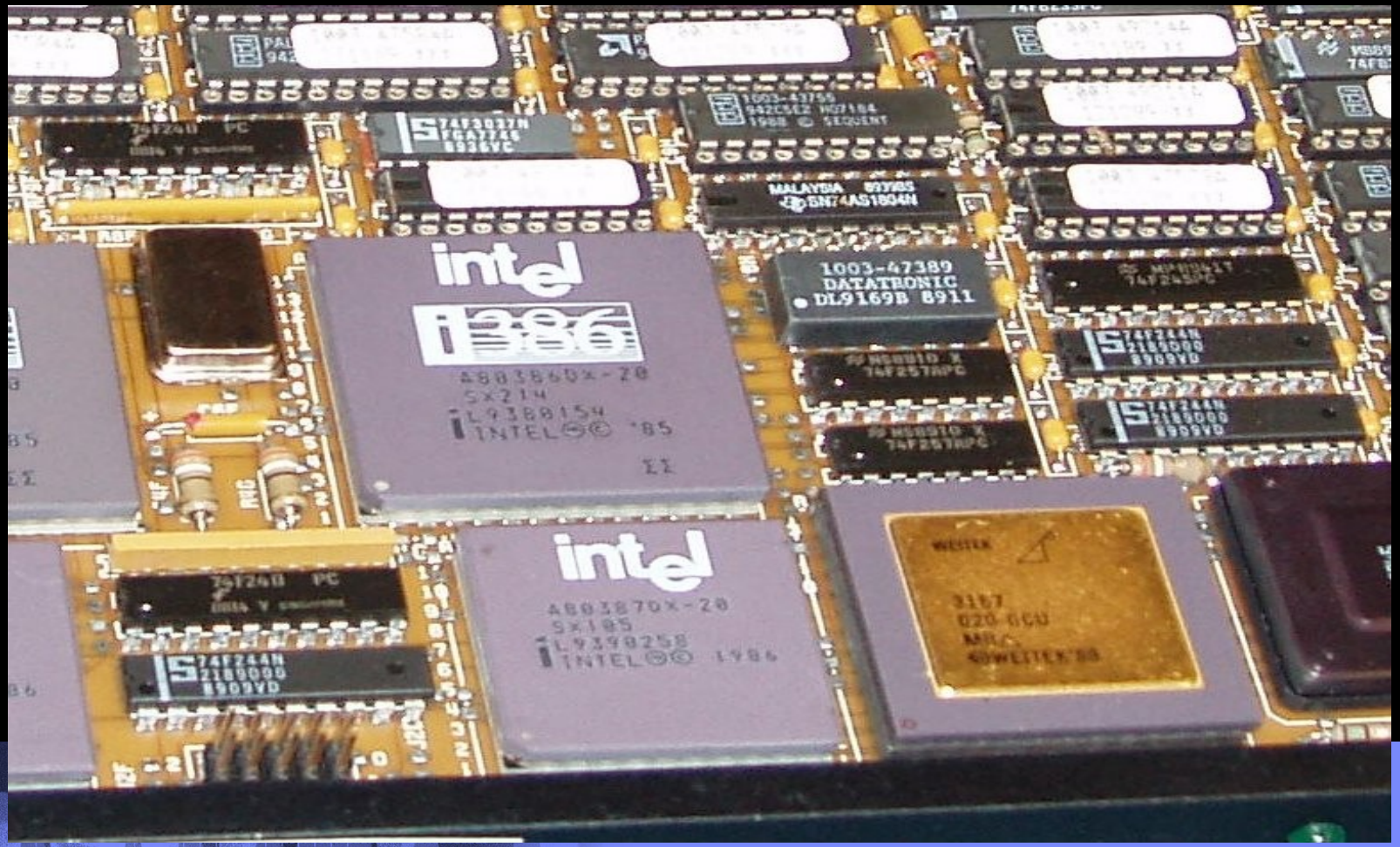
Parallel Programming: What Has Changed?

- Parallel systems were rare and expensive
- About $1,000,000 each in 1990
- About $100 in 2011
  - Four orders of magnitude decrease in 21 years
  - From many times the price of a house to a small fraction of the cost of a bicycle
- In 2006, masters student I was collaborating with bought a dual-core laptop
  - And just to be able to present a parallel-programming talk using a parallel programmer

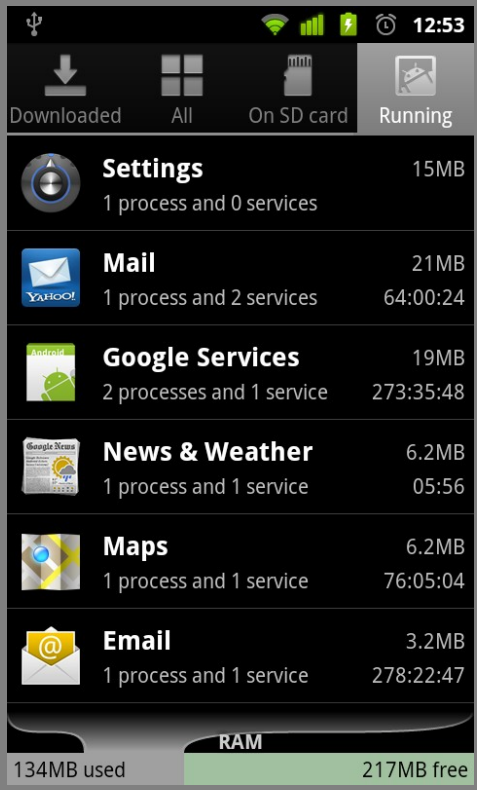
Parallel Programming: Rare and Expensive



Parallel Programming: Rare and Expensive



Parallel Programming: Cheap and Plentiful



Parallel Programming: What Has Changed?

- Back then: very few developers had access to parallel systems
  - Maybe 1,000 developers with parallel systems
  - Suppose that a tool could save 10% per developer
    - Then it would not make sense to invest 100 developers
    - Even assuming compatible environments: 'Fraid not!!!
- Now: almost anyone can have a parallel system
  - More than 100,000 developers with parallel systems
    - Easily makes sense to invest 100 developers for 10% gain
    - Even assuming multiple incompatible environments
- We can expect many more parallel tools
  - Linux kernel lockdep, lock profiling, ...

Why Has Parallel Programming Been Hard?

- Almost no publicly accessible parallel code
- Parallel-programming engineering discipline confined to a few proprietary projects
  - Linux kernel: 13 million lines of parallel code
  - MariaDB (the database formerly known as MySQL)
  - memcached
  - Apache web server
  - Hadoop
  - BOINC
  - OpenMP
- Lots of parallel code for study and copying

Why Has Parallel Programming Been Hard?

- Technological obstacles:
  - Synchronization overhead
  - Deadlock
  - Data races

Why Has Parallel Programming Been Hard?

- Technological obstacles:
  - Synchronization overhead
  - Deadlock
  - Data races

- **If these obstacles are impossible to overcome, why are there more than 13 million lines of parallel Linux kernel code?**

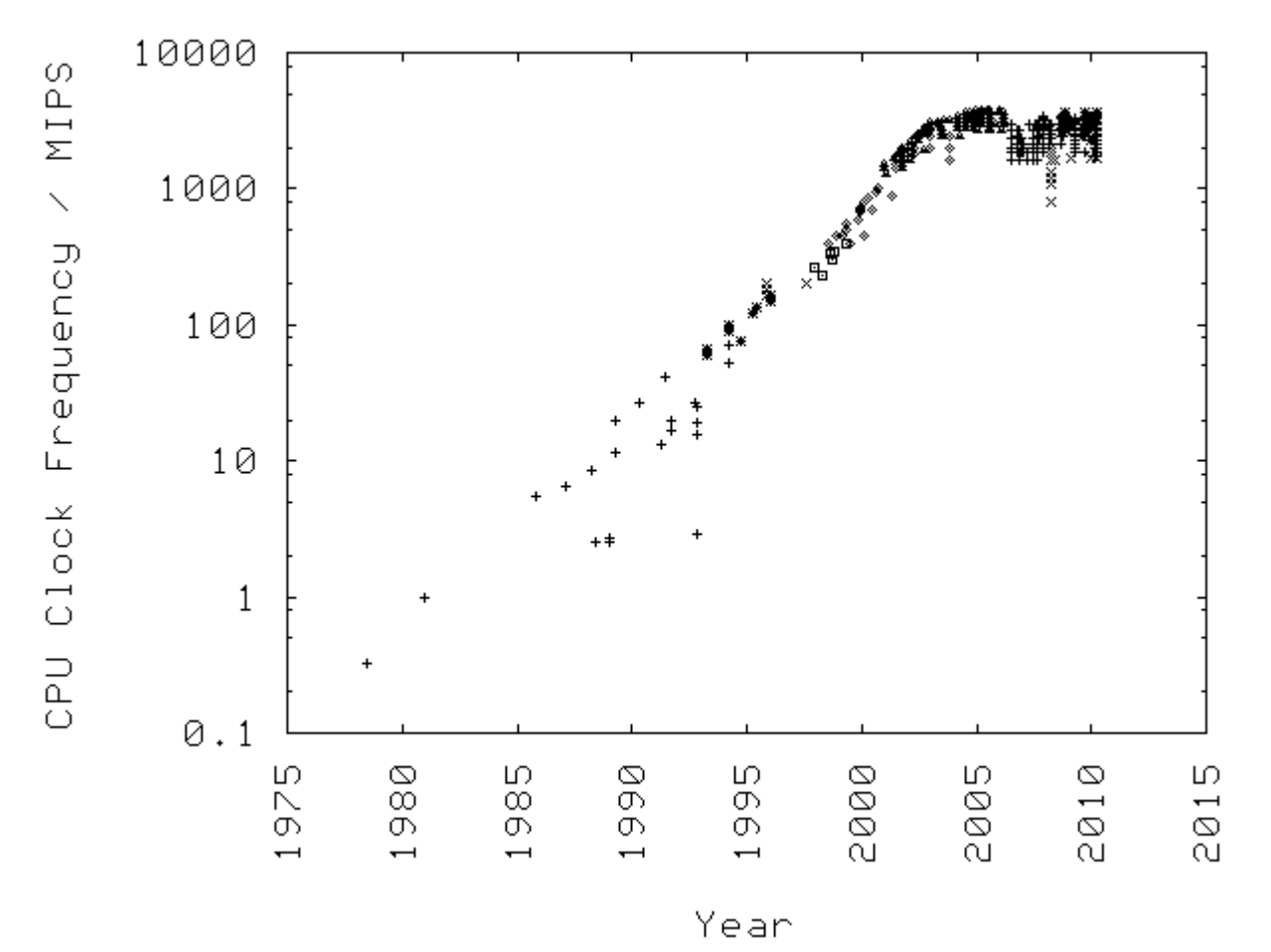
Why Has Parallel Programming Been Hard?

- Technological obstacles:
  - Synchronization overhead
  - Deadlock
  - Data races

- **If these obstacles are impossible to overcome, why are there more than 13 million lines of parallel Linux kernel code?**
  - **But parallel programming *is* harder than sequential programming...**

So Why Parallel Programming?

Why Parallel Programming? (Party Line)



Why Parallel Programming? (Reality)

- Parallelism can increase performance
  - Assuming...
- Parallelism may be important battery-life extender
  - Run two CPUs at half frequency
  - 25% power per CPU, 50% overall with same throughput assuming perfect scaling
    - And assuming CPU consumes most of the power
    - Which is not the case on many embedded systems
- But parallelism is but one way to optimize performance and battery lifetime
  - Hashing, search trees, parsers, cordic algorithms, ...

Why Parallel Programming? (Reality)

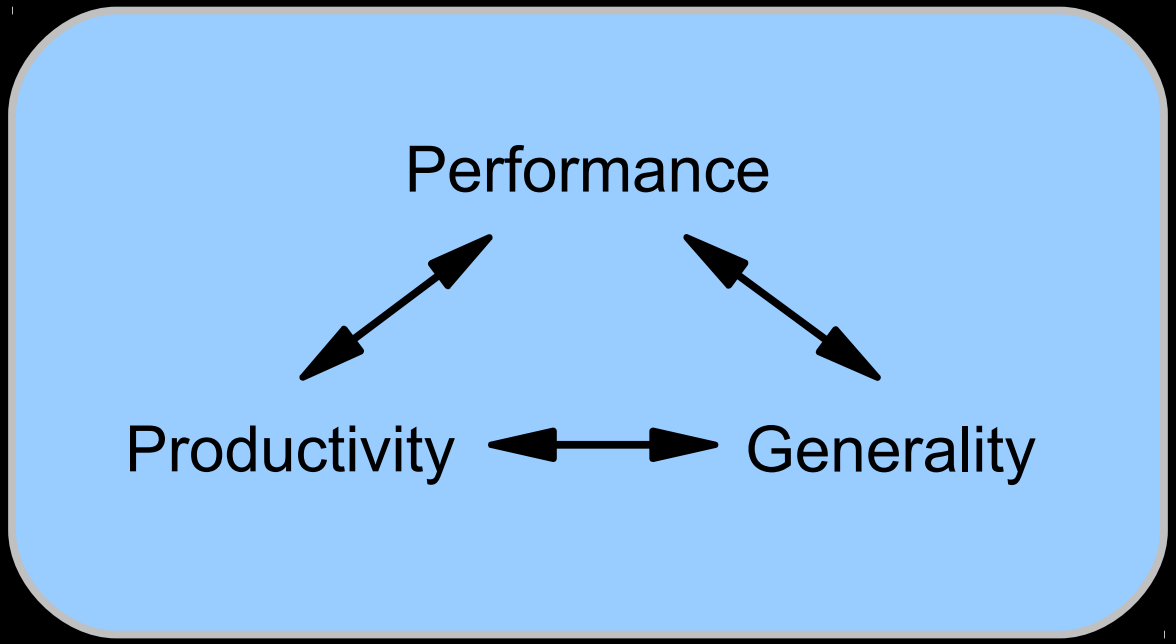
If your code runs well enough single threaded, then leave it be single threaded.

And you always have the option of implementing key function in hardware.

But where it makes sense, parallelism can be quite valuable.

Parallel Programming Goals

Parallel Programming Goals



Parallel Programming Goals: Why Performance?

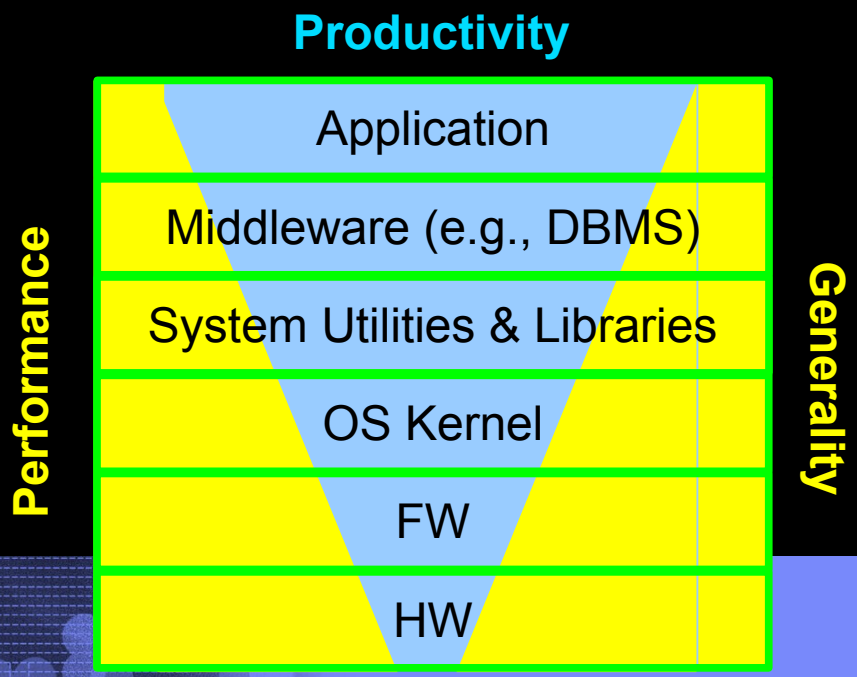
- (Performance often expressed as scalability or normalized as in performance per watt)
- If you don't care about performance, *why* are you bothering with parallelism???
  - Just run single threaded and be happy!!!
- But what about:
  - All the multi-core systems out there?
  - Efficient use of resources?
  - Everyone saying parallel programming is crucial?
- Parallel Programming: one optimization of many
- CPU: one potential bottleneck of many

Parallel Programming Goals: Why Productivity?

- 1948 CSIRAC (oldest intact computer)
  - 2,000 vacuum tubes, 768 20-bit words of memory
  - $10M AU construction price
  - 1955 technical salaries: $3-5K/year
  - Makes business sense to dedicate 10-person team to increasing performance by 10%
- 2008 z80 (popular 8-bit microprocessor)
  - 8,500 transistors, 64K 8-bit works of memory
  - $1.36 per CPU in quantity 1,000 (7 OOM decrease)
  - 2008 SW starting salaries: $50-95K/year US (1 OOM increase)
  - Need 1M CPUs to break even on a one-person-year investment to gain 10% performance!
    - Or 10% more performance must be blazingly important
    - Or you are doing this as a hobby... In which case, do what you want!

Parallel Programming Goals: Why Generality?

- The more general the solution, the more users to spread the cost over, but...



Parallel Programming Goals: Why Generality?

- The more general the solution, the more users to spread the cost over, but...
- Embedded unit volumes are so high that very specialized hardware accelerators are often the way to go
- So we might yet see GPGPUs on Android smartphones, and much else besides!!!

Parallel Programming: What Might You Need?

- Scheduler features
  - Tracing?
  - Gang scheduling (media handling)?
- Lightweight virtualization (containers)?
  - Hard limits and guarantees?
  - Process/thread groups (cgroups)?
- Lightweight CPU hotplug?
- Real-time response?
- OS jitter?

Performance of Synchronization Mechanisms

Performance of Synchronization Mechanisms

4-CPU 1.8GHz AMD Opteron 844 system

Need to be here!
(Partitioning/RCU)

Operation	Cost (ns)	Ratio
Clock period	0.6	1
Best-case CAS	37.9	63.2
Best-case lock	65.6	109.3
Single cache miss	139.5	232.5
CAS cache miss	306.0	510.0

Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot

Performance of Synchronization Mechanisms

4-CPU 1.8GHz AMD Opteron 844 system

Need to be here!
(Partitioning/RCU)

Operation	Cost (ns)	Ratio
Clock period	0.6	1
Best-case CAS	37.9	63.2
Best-case lock	65.6	109.3
Single cache miss	139.5	232.5
CAS cache miss	306.0	510.0

Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot

But this is an old system...

Performance of Synchronization Mechanisms

4-CPU 1.8GHz AMD Opteron 844 system

Need to be here!
(Partitioning/RCU)

Operation	Cost (ns)	Ratio
Clock period	0.6	1
Best-case CAS	37.9	63.2
Best-case lock	65.6	109.3
Single cache miss	139.5	232.5
CAS cache miss	306.0	510.0

Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot

But this is an old system...

And why low-level details???

Why All These Low-Level Details???

- Would you trust a bridge designed by someone who did not understand strengths of materials?
  - Or a ship designed by someone who did not understand the steel-alloy transition temperatures?
  - Or a house designed by someone who did not understand that unfinished wood rots when wet?
  - Or a car designed by someone who did not understand the corrosion properties of the metals used in the exhaust system?
  - Or a space shuttle designed by someone who did not understand the temperature limitations of O-rings?
- So why trust algorithms from someone ignorant of the properties of the underlying hardware???

Why Such An Old System???

- Because early low-power embedded CPUs are likely to have similar performance characteristics.

But Isn't Hardware Just Getting Faster?

Performance of Synchronization Mechanisms

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“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

What a difference a few years can make!!!

Performance of Synchronization Mechanisms

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“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

Not *quite* so good... But still a 6x improvement!!!

Performance of Synchronization Mechanisms

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“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

Maybe not such a big difference after all...
And these are best-case values!!! (Why?)

Performance of Synchronization Mechanisms

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Most embedded devices here

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“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

Maybe not such a big difference after all...
And these are best-case values!!! (Why?)

Performance of Synchronization Mechanisms

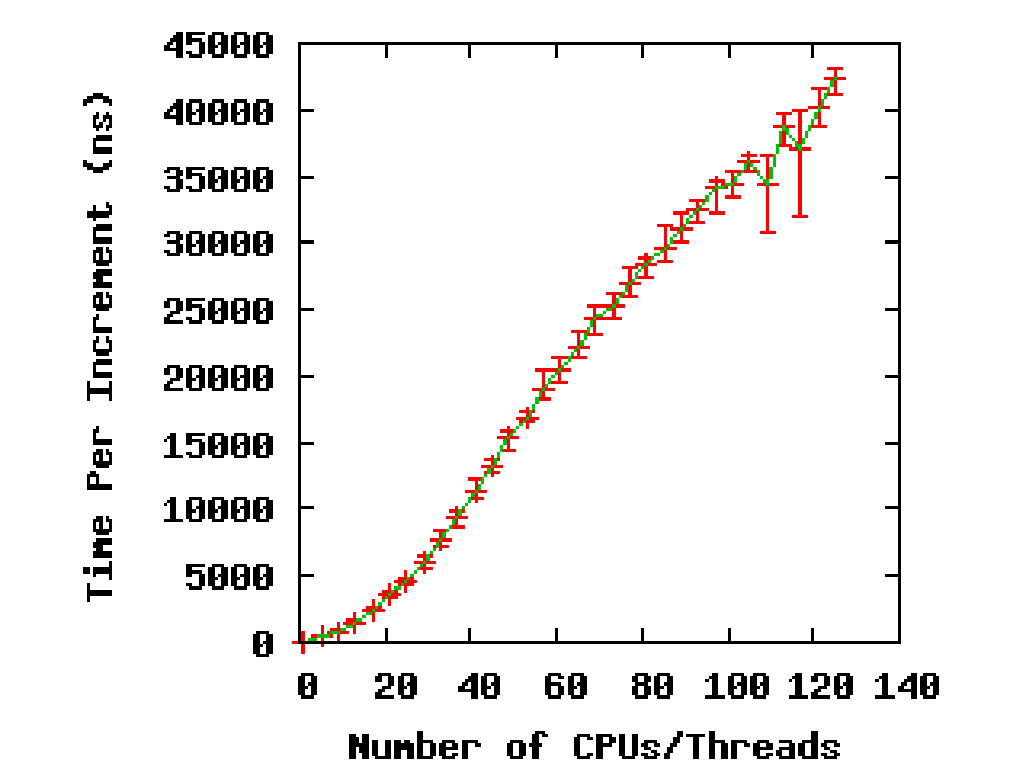
16-CPU 2.8GHz Intel X5550 (Nehalem) System

Most embedded devices here For now...

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“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

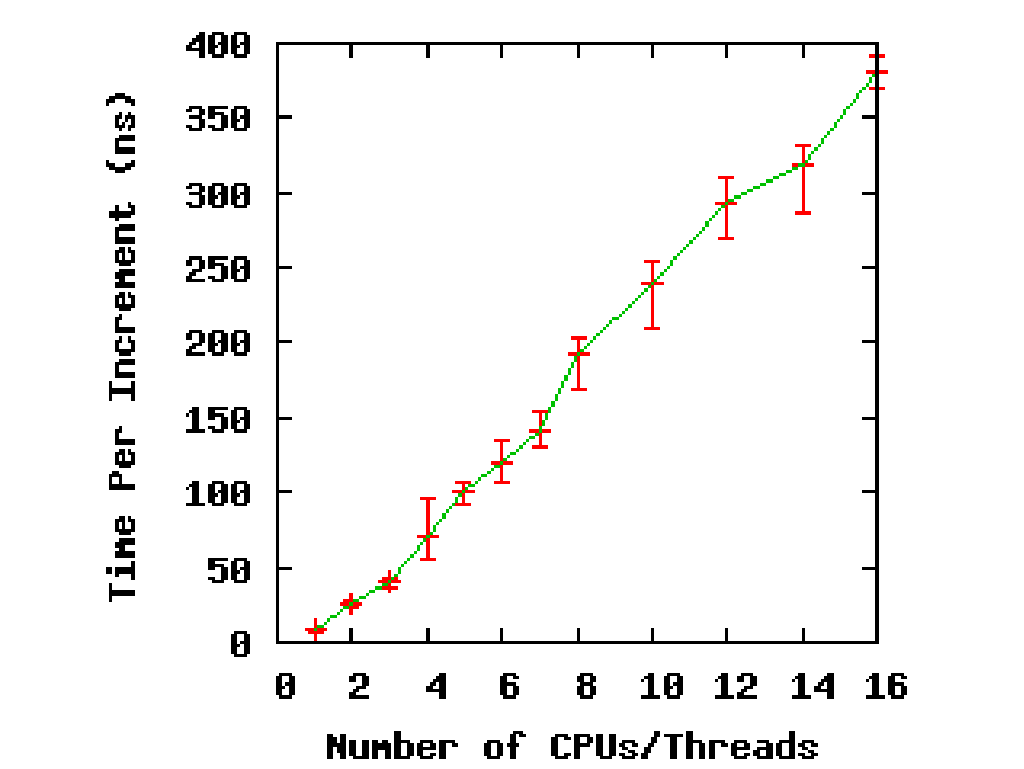
Maybe not such a big difference after all...
And these are best-case values!!! (Why?)

Performance of Synchronization Mechanisms



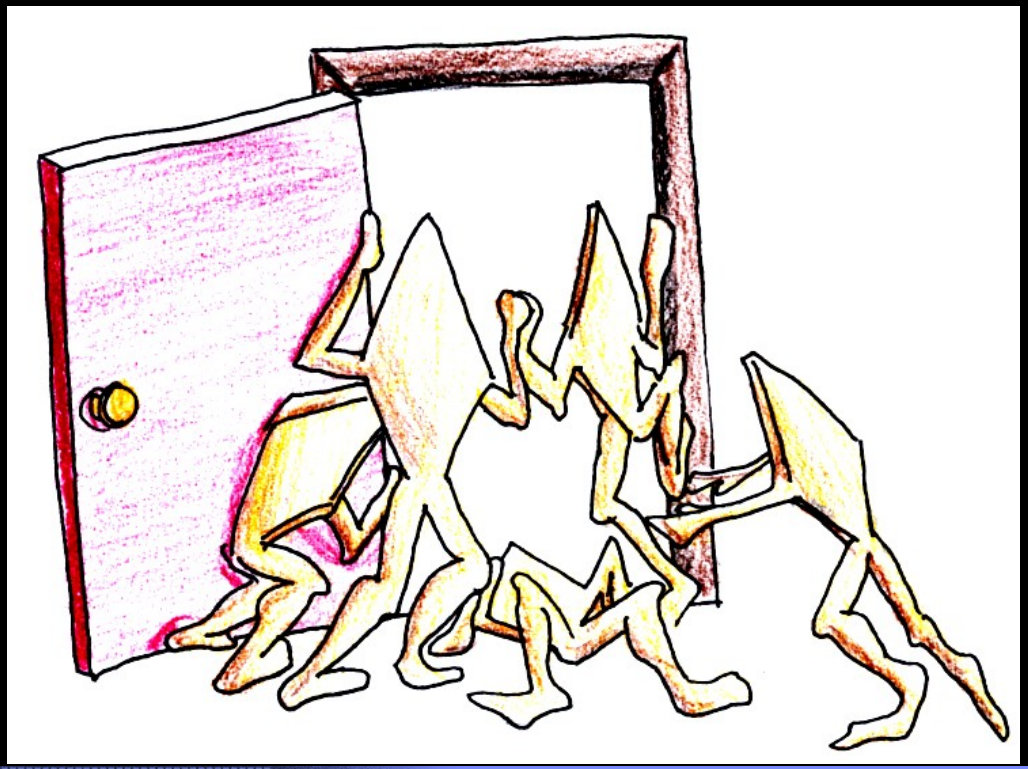
If you thought a *single* atomic operation was slow, try lots of them!!!
(Atomic increment of single variable on 1.9GHz Power 5 system)

Performance of Synchronization Mechanisms



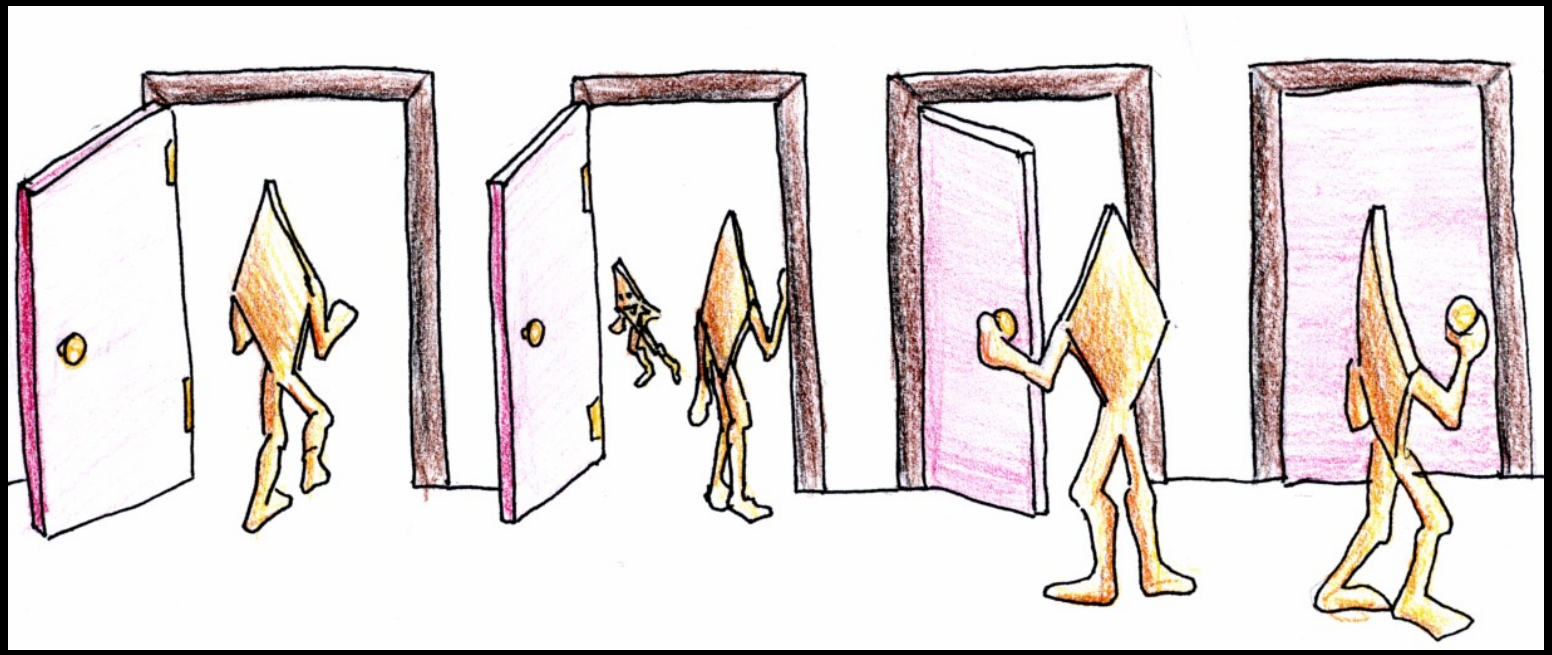
Same effect on a 16-CPU 2.8GHz Intel X5550 (Nehalem) system

Design Principle: Avoid Bottlenecks



Only one of something: bad for performance and scalability

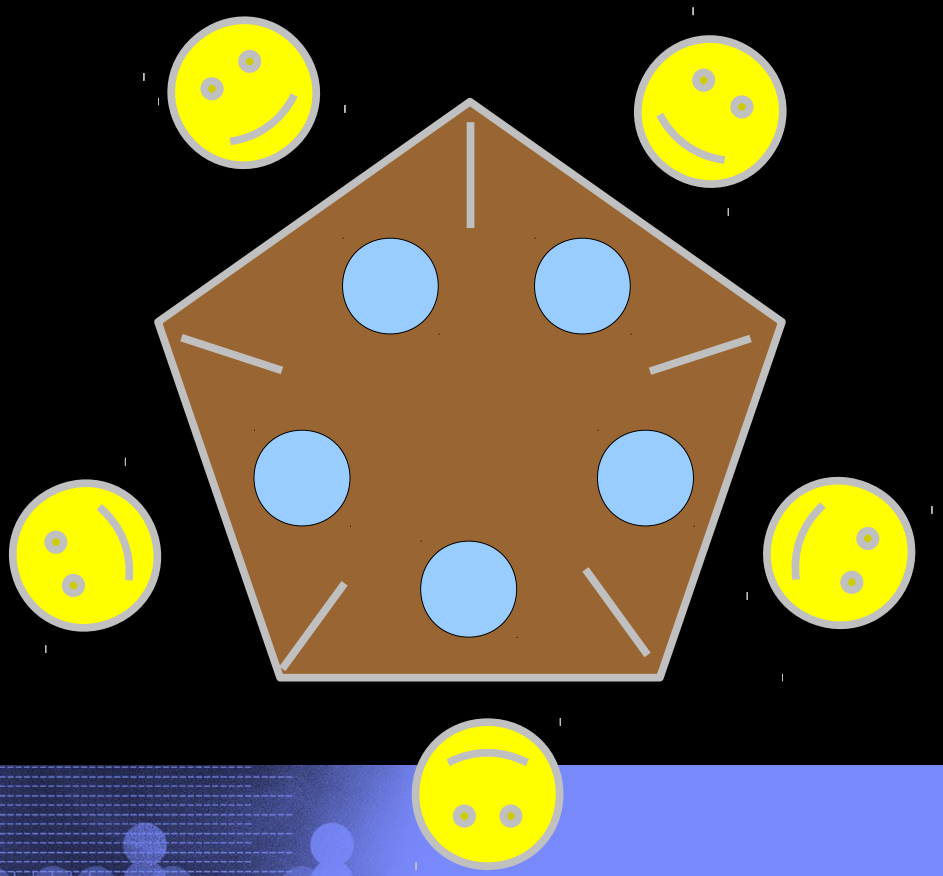
Design Principle: Avoid Bottlenecks



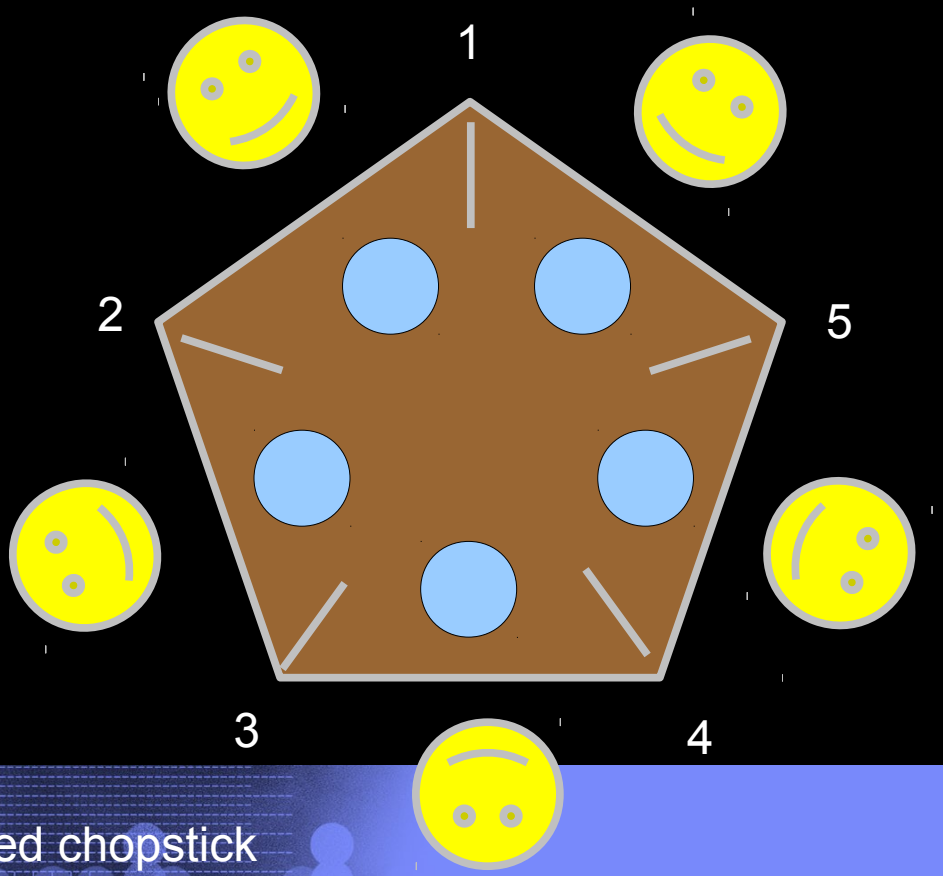
Many instances of something: great for performance and scalability!
Any exceptions to this rule?

Exercise: Dining Philosophers Problem

Each philosopher requires two chopsticks to eat.
Need to avoid starvation.



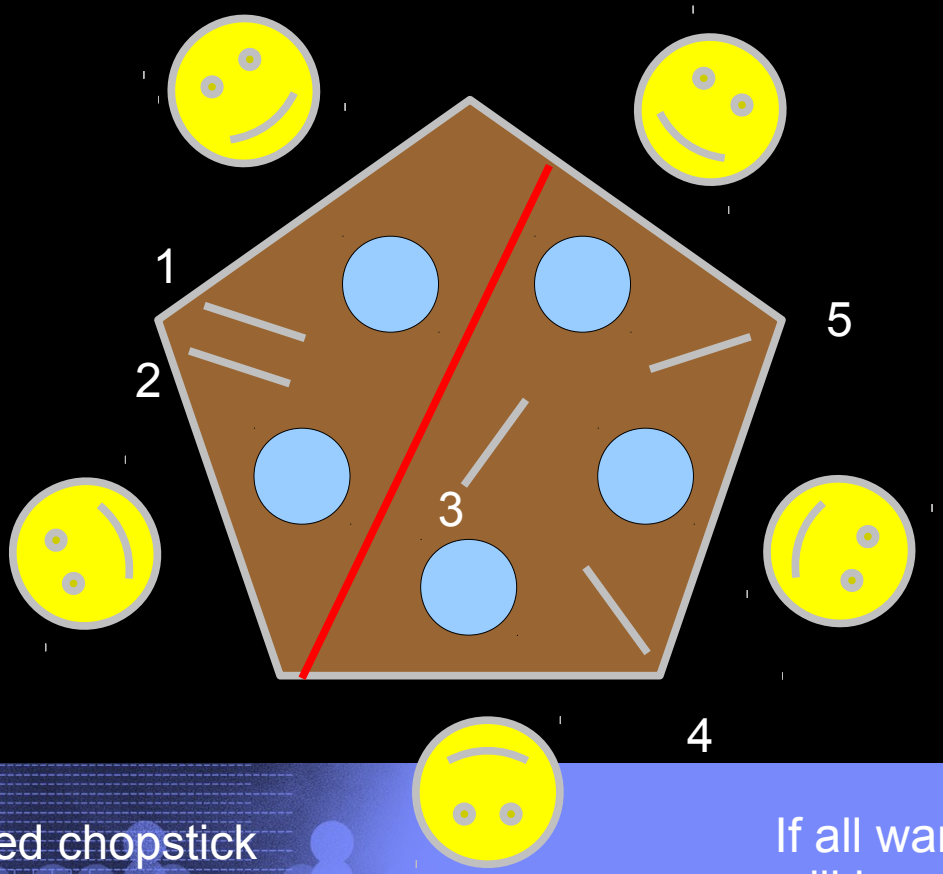
Exercise: Dining Philosophers Solution #1



Locking hierarchy.
Pick up low-numbered chopstick first, preventing deadlock.

Is this a good solution???

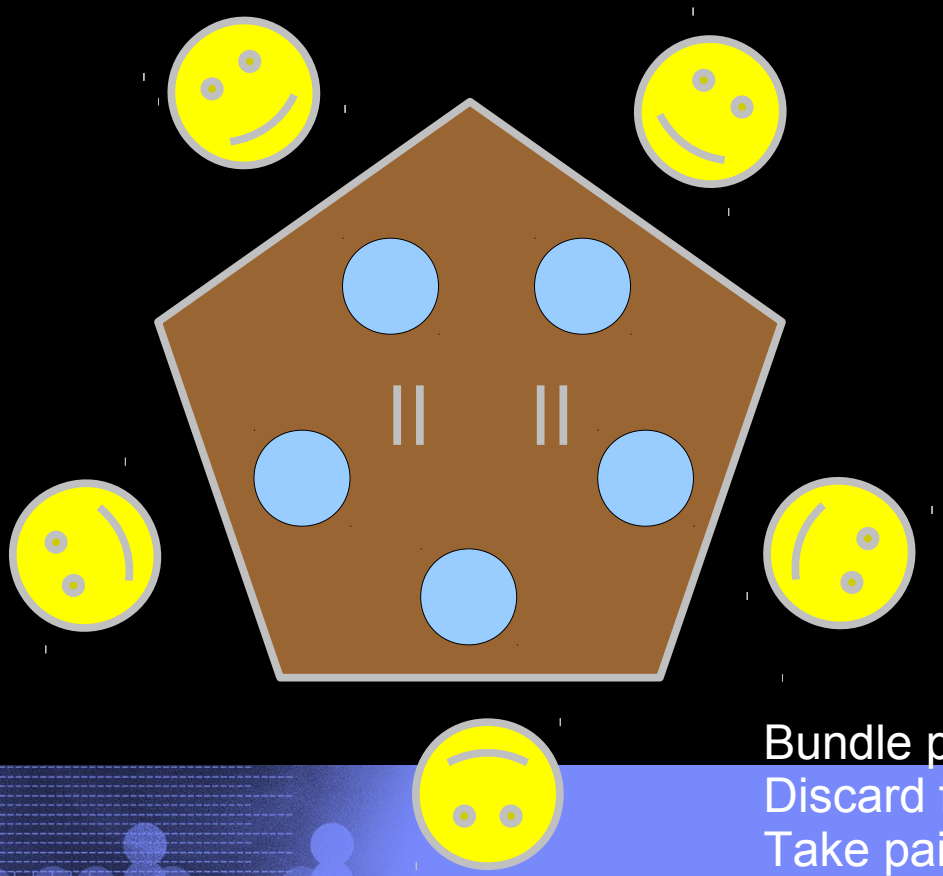
Exercise: Dining Philosophers Solution #2



Locking hierarchy.
Pick up low-numbered chopstick first, preventing deadlock.

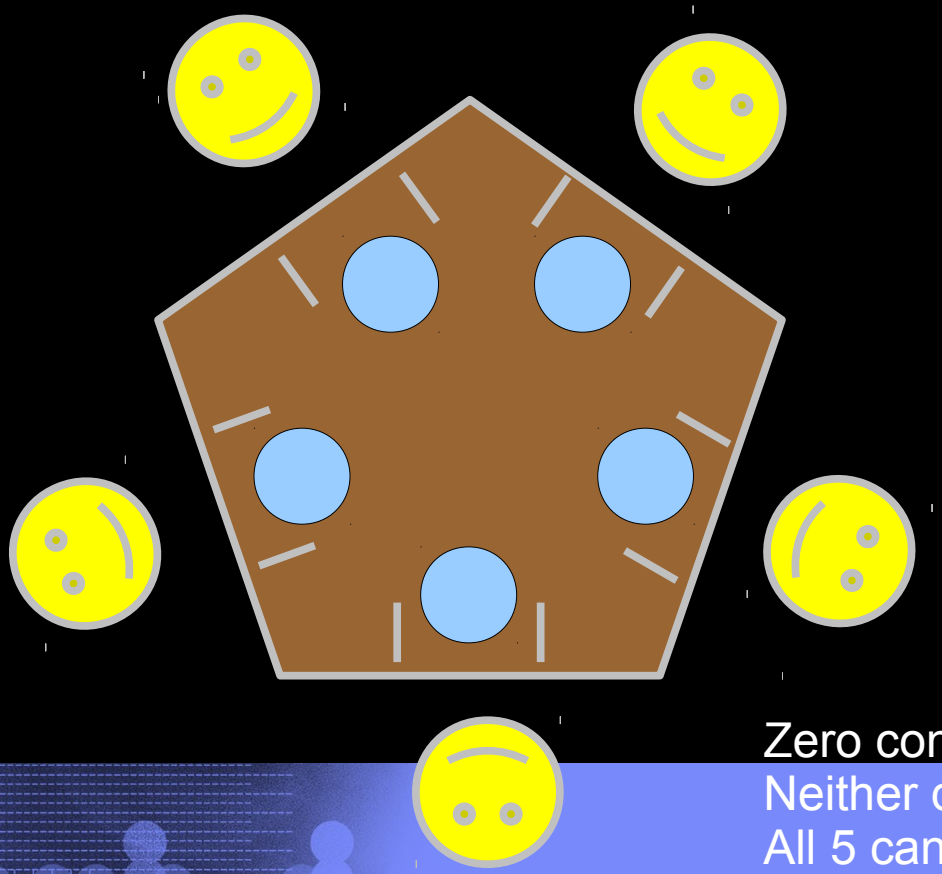
If all want to eat, at least two will be able to do so.

Exercise: Dining Philosophers Solution #3



Bundle pairs of chopsticks.
Discard fifth chopstick.
Take pair: no deadlock.
Starvation still possible.

Exercise: Dining Philosophers Solution #4



Zero contention.
Neither deadlock nor livelock.
All 5 can eat concurrently.
Excellent disease control.

Exercise: Dining Philosophers Solutions

- **Objections to solution #2 and #3:**
  - **"You can't just change the rules like that!!!"**
    - No rule against moving or adding chopsticks!!!
  - **"Dining Philosophers Problem valuable lock-hierarchy teaching tool – #3 just destroyed it!!!"**
    - Lock hierarchy is indeed very valuable and widely used, so the restriction "there can only be five chopsticks positioned as shown" does indeed have its place, even if it didn't appear in this instance of the Dining Philosophers Problem.
    - But the lesson of transforming the problem into perfectly partitionable form is also very valuable, and given the wide availability of cheap multiprocessors, most desperately needed.
  - **"But what if each chopstick costs a million dollars?"**
    - Then we make the philosophers eat with their fingers... ☺

But What About Real-World Software?

Sequential Programs Often Highly Optimized

- Highly optimized means difficult to modify
  - And no wand can wish this away

Sequential Programs Often Highly Optimized

- Highly optimized means difficult to modify
  - And no wand can wish this away
- But there are some tricks that can often help:
  - Partition the problem at high level
    - Reduce communication overhead between partitions
    - Good approach for multitasking smartphones
  - Use existing parallel software
    - Single-threaded user-interface programs talking to central database software in servers
    - Changes in protocols may be needed to open up similar opportunities in embedded
      - ✓ Hardware acceleration can also be attractive
  - Parallelize only the performance-critical code

Some Issues With Real-World Software

- Object-oriented spaghetti code
- Highly optimized sequential code
- Parallel-unfriendly APIs
- Parallelization can be a major change

Object-Oriented Spaghetti Code

Let's see *you* create a valid locking hierarchy!!!
What can you do about this?

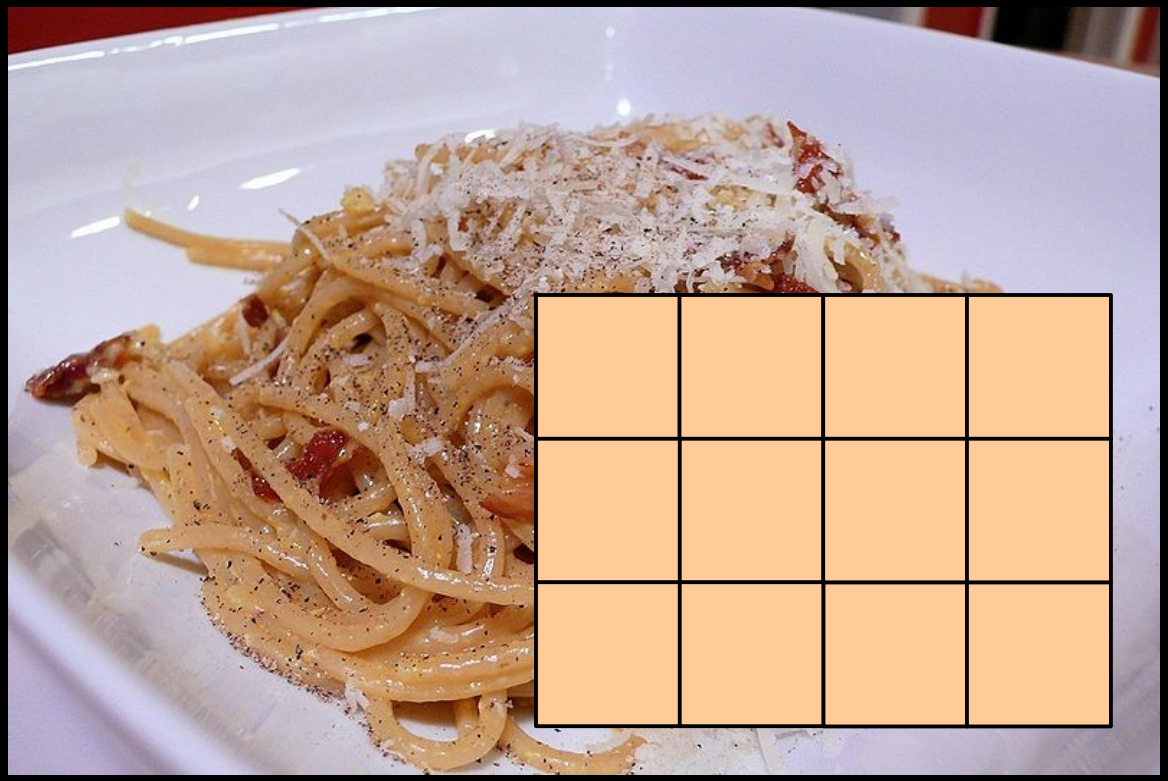
Photo: © 2007 Ed Hawco all cc-gy-sa

Object-Oriented Spaghetti Code: Solution 1

Run multiple instances in parallel.
Maybe with the sh “&” operator.

Photo: © 2007 Ed Hawco all cc-gy-sa

Object-Oriented Spaghetti Code: Solution 2



Find the high-cost portion of the code, and rewrite that portion to be parallel-friendly.
Or implement in hardware.
Either way, a careful redesign will be required.

Photo: © 2007 Ed Hawco all cc-gy-sa

Object-Oriented Spaghetti Code: Solution 3

- Use explicit locking
  - No “synchronized” clauses
  - Provide a global hashed array of locks
    - Hash from object address to corresponding lock
  - Also provide a single global lock
- Conditionally acquire needed locks before making any changes
  - If any lock acquisition fails, drop all locks and then acquire the global lock
  - Then unconditionally acquire the needed locks
- Make changes, then release all the acquired locks
- From “Concurrent Programming in Java: Design Principles and Patterns” by Doug Lea

Highly Optimized Sequential Code

- Highly optimized sequential code is difficult to change, especially if the result is to remain optimized
- Parallelization is a change, so parallelizing highly optimized sequential code is likely to be difficult
  - But it is easy to run multiple instances in parallel
  - It may well be that the sequential optimizations work better than parallelization
    - Parallelism is but one optimization of many: Use the best optimization for the task at hand

Parallel-Unfriendly APIs

- Consider a hash-table addition function that returns the number of entries in the table
- What is the best way to implement this in parallel?

Parallel-Unfriendly APIs

- Consider a hash-table addition function that returns the number of entries in the table
- What is the best way to implement this in parallel?
- There is no good way to do so!!!
- The problem is that concurrent additions must communicate, and this communication is inherently expensive
  - http://lwn.net/Articles/423994/
- What should be done instead?

Parallel-Unfriendly APIs

- Don't return the count!
  - Without the count, a hash-table implementation can handle two concurrent “add” operations with no conflicts: fully in parallel
  - With the count, the two concurrent “add” operations must update the count
- Alternatively, return an approximate count
- Either way, sequential APIs may need to be reworked to allow efficient implementation

Parallelization Can Be A Major Change

- Software resembles building construction
  - Start with architects
  - Then construction
  - Finally maintenance
- If you have a old piece of software, your staff might consist only of software janitors
  - Nothing wrong with janitors: I used to be one
  - But odds are against janitors remodeling skyscrapers
- Use the right people for the job
- Can current staff do a major change?
  - And yes, you might need time and budget to suit...

Conclusions

Summary and Problem Statement

- Writing new parallel code is quite doable
  - Many open-source projects prove this point
- Converting existing sequential code to run in parallel can be easy...
  - … or arbitrarily difficult
- So don't do things the hard way!

Is Parallel Programming Hard, And If So, Why?

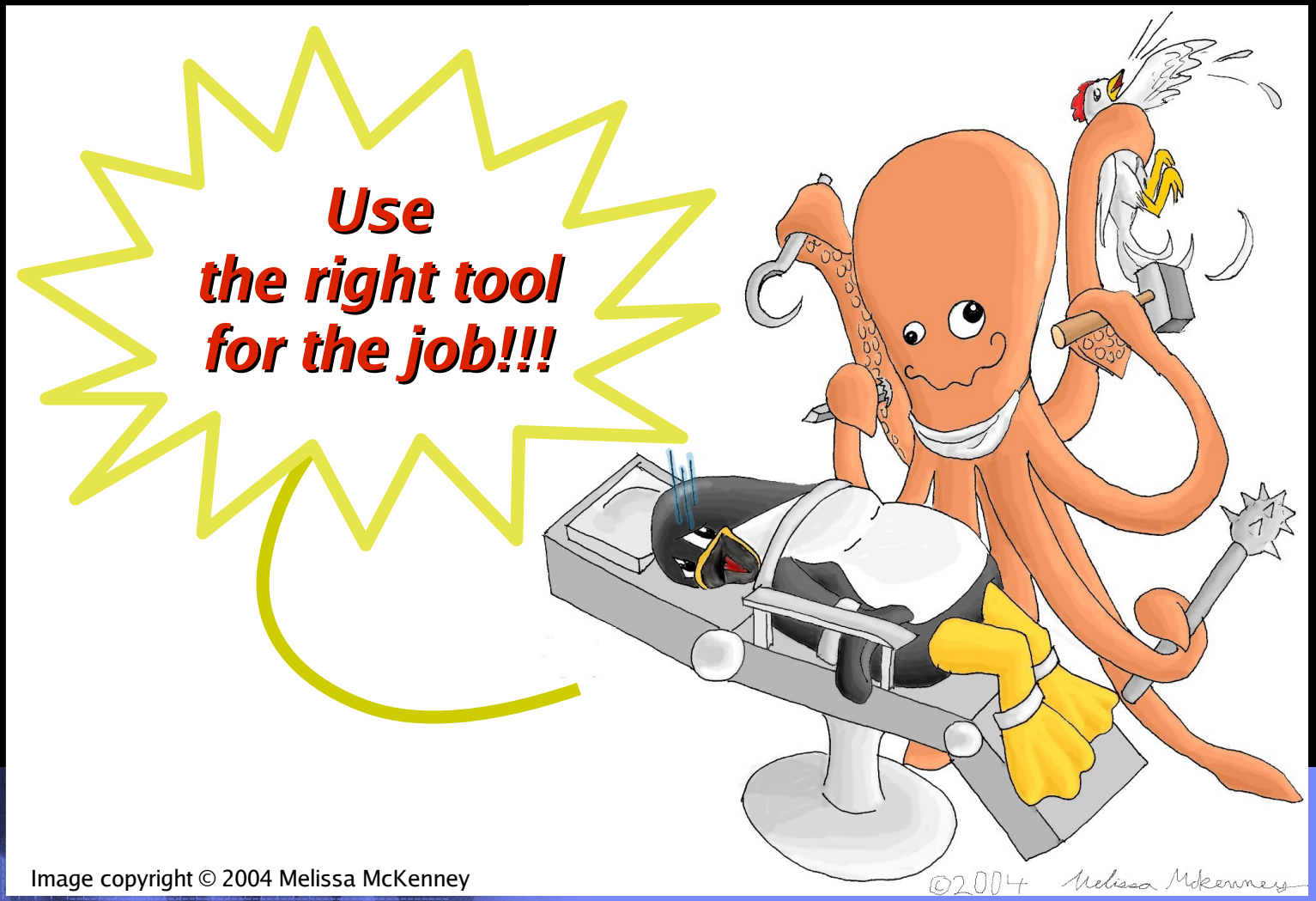
Parallel Programming is as Hard or as Easy as We Make It.

It is that hard (or that easy) because we make it that way!!!

Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.
- This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.

Summarized Summary



To Probe Further...

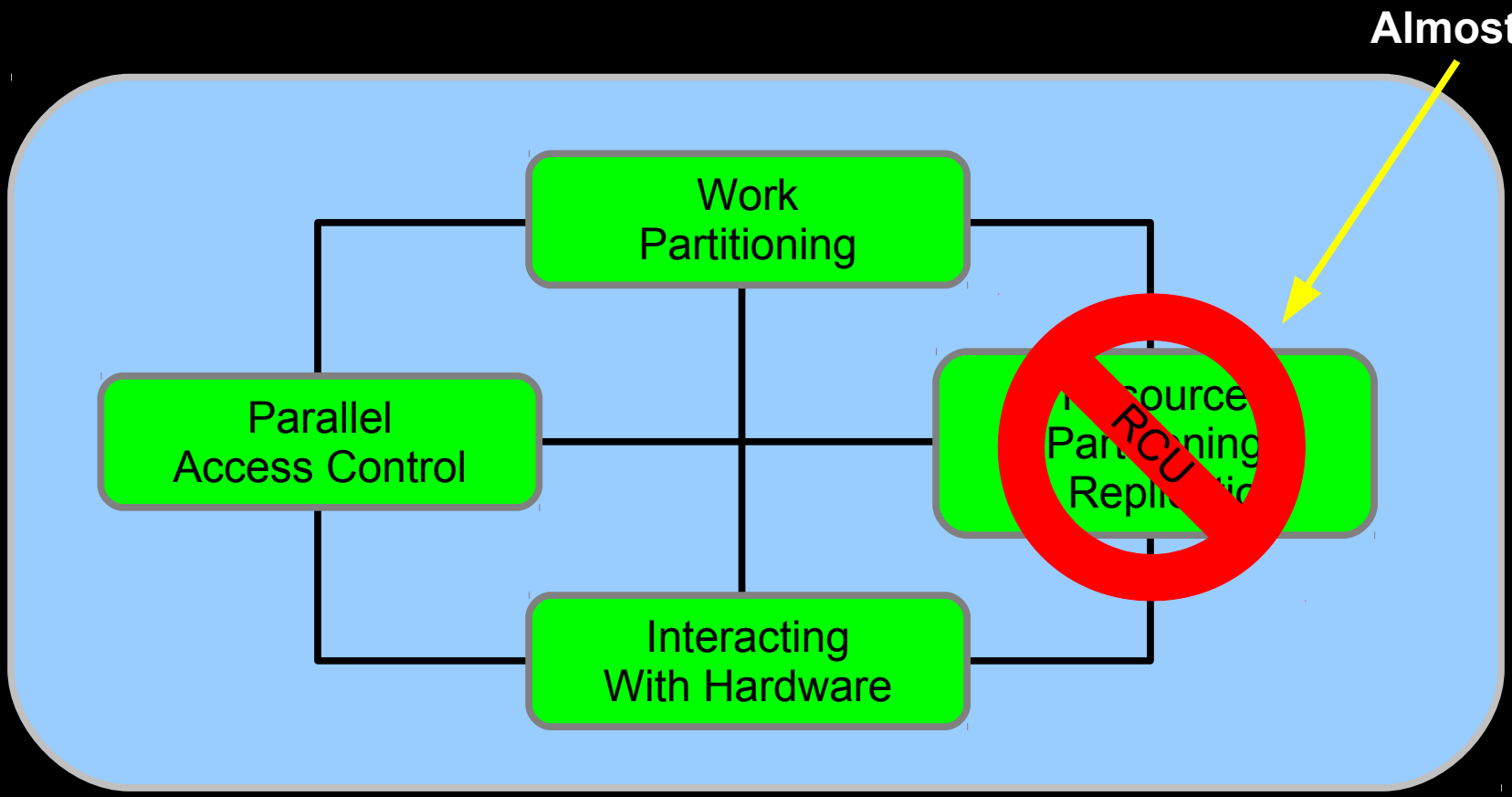
- Pattern-Oriented Software Architecture, vol 2&4, Schmidt et al.
- Programming with POSIX Threads, Butenhof
- Intel Threading Building Blocks, Reinders
- Patterns for Parallel Programming, Mattson et al.
- Concurrent Programming in Java, Lea
- Effective Concurrency, Sutter
- The Art of Multiprocessor Programming, Herlihy and Shavit
- Design and Validation of Computer Protocols, Holzmann
- http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html
  - Online pthreads reference
- "Is Parallel Programming Hard, And If So, What Can You Do About It?"
  - git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git

BACKUP

Parallel Programming Advanced Topic: RCU

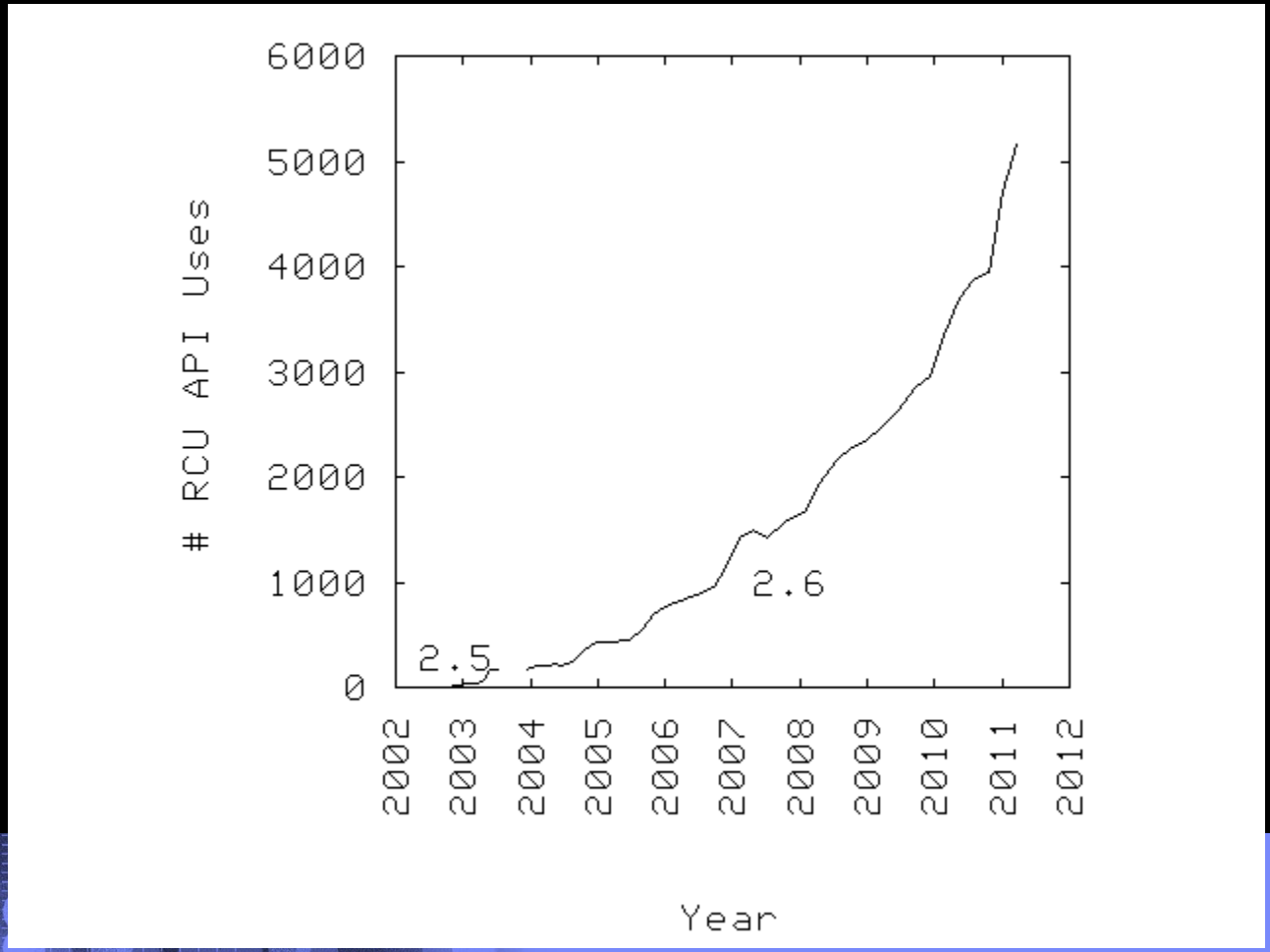
- For read-mostly data structures, RCU provides the benefits of the data-parallel model
  - But without the need to actually partition or replicate the RCU-protected data structures
  - Readers access data without needing to exclude each others or updates
    - Extremely lightweight read-side primitives
- And RCU provides additional read-side performance and scalability benefits
  - With a few limitations and restrictions....

RCU for Read-Mostly Data Structures



RCU data-parallel approach: ~~first partition resources~~, then partition work, and only then worry about parallel access control, and only for updates.

RCU Usage in the Linux Kernel

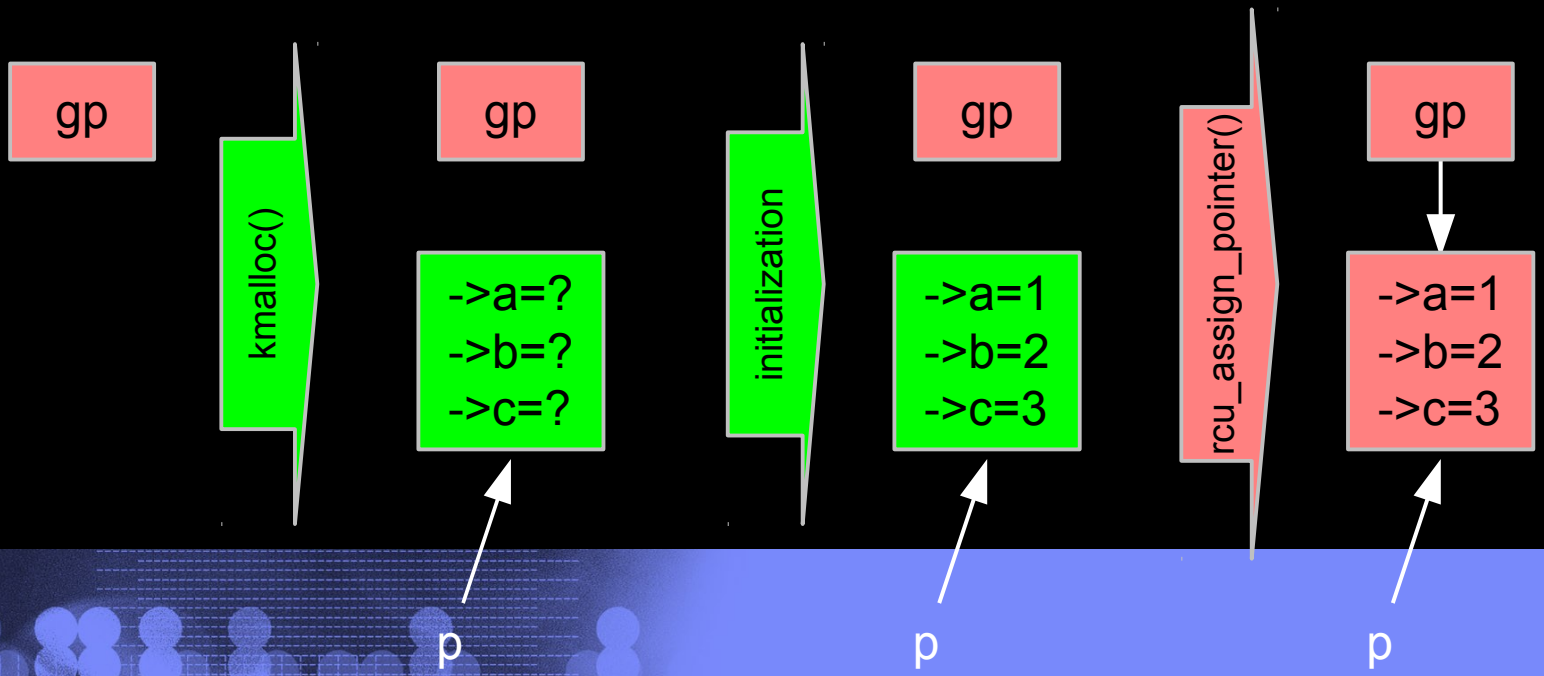


What Is RCU?

- Publication of new data
- Subscription to existing data
- Wait for pre-existing RCU readers
  - Once all pre-existing RCU readers are done, old versions of the data may be discarded

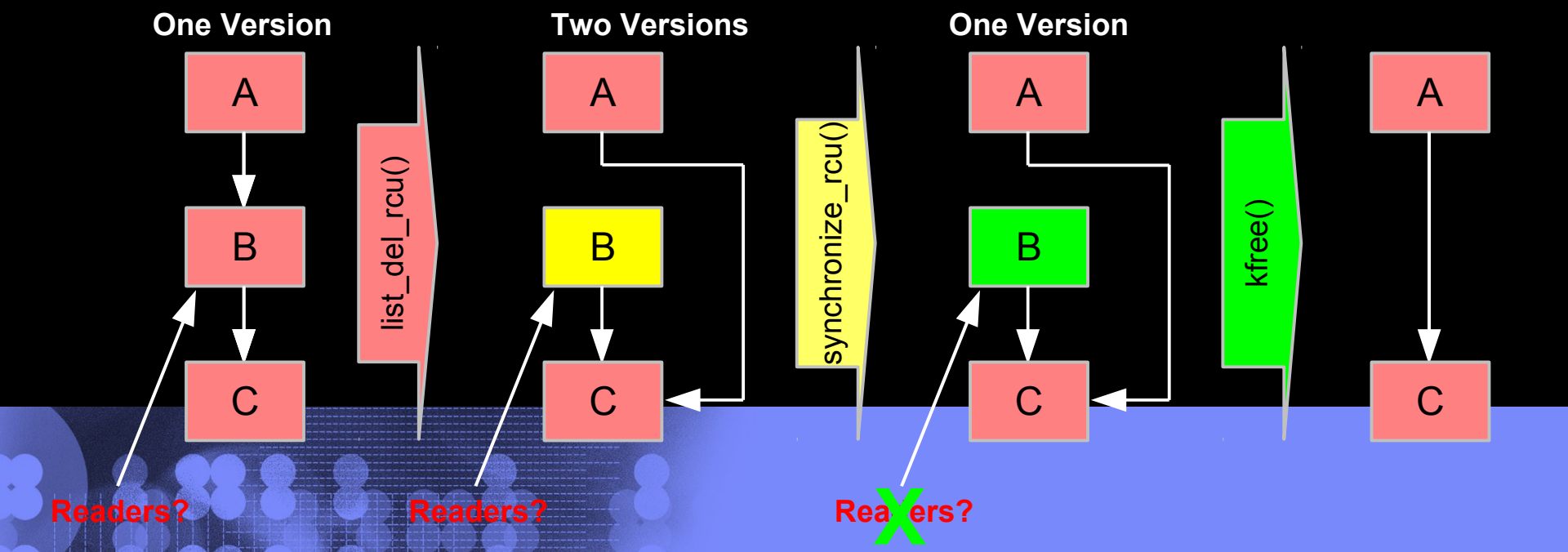
Publication of And Subscription To New Data

Key:
- Dangerous for updates: all readers can access
- Still dangerous for updates: pre-existing readers can access
- Safe for updates: inaccessible to all readers



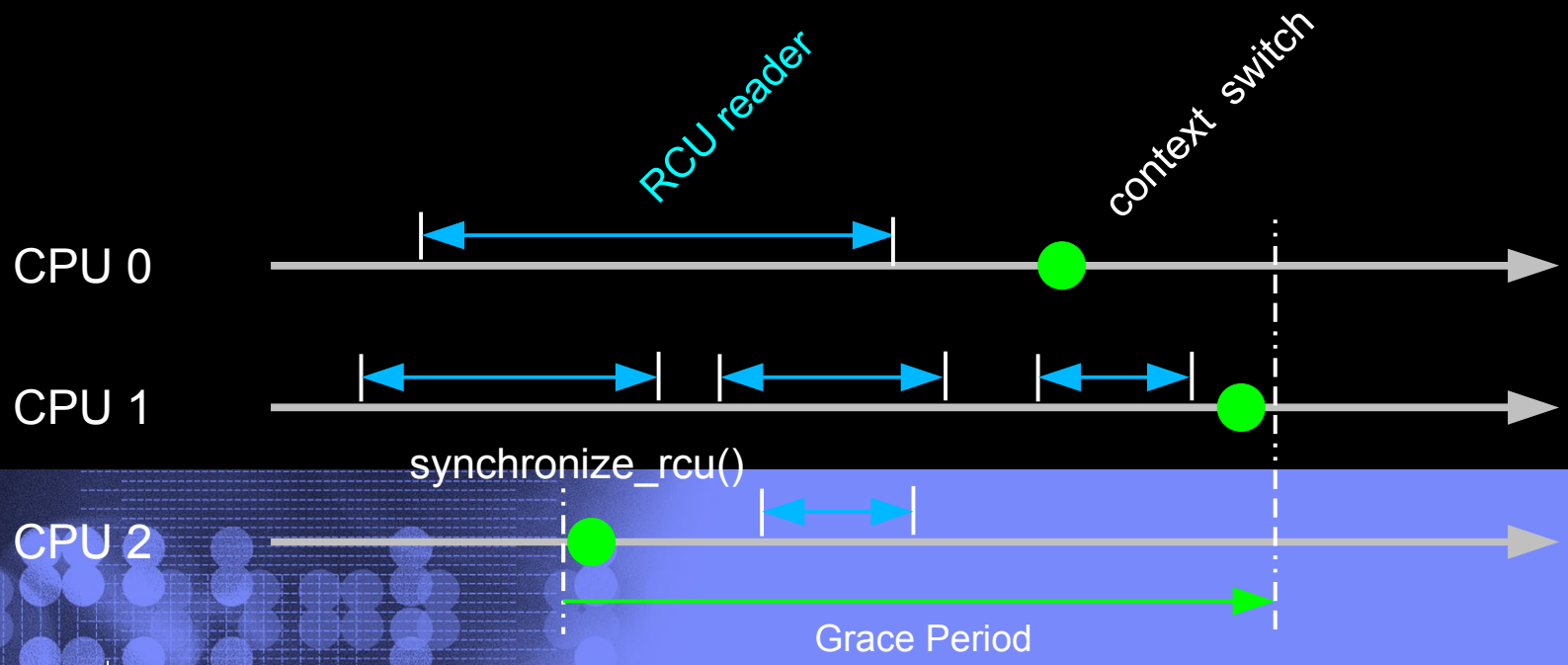
RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes element B from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free B (kfree())

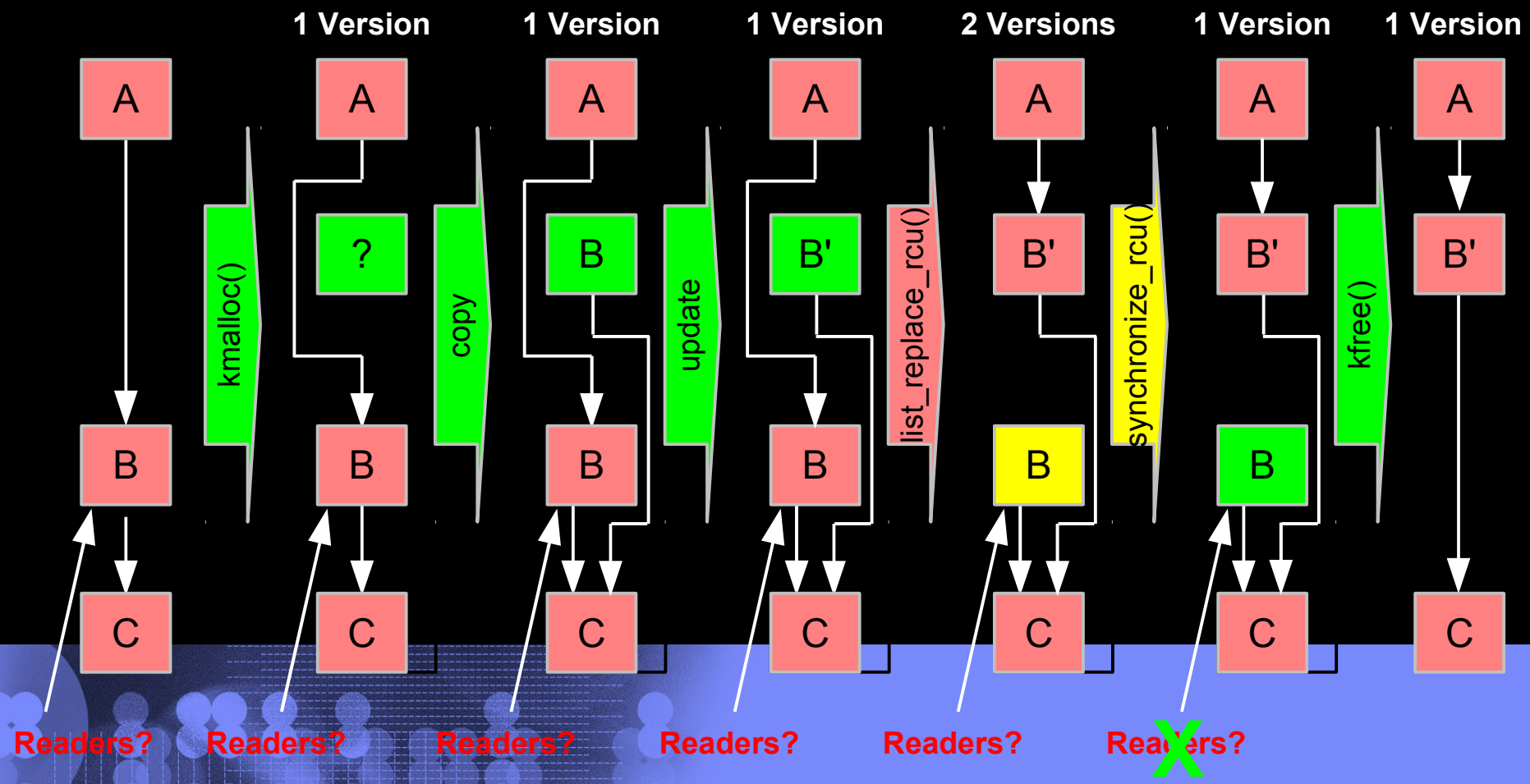


Waiting for Pre-Existing Readers: QSBR

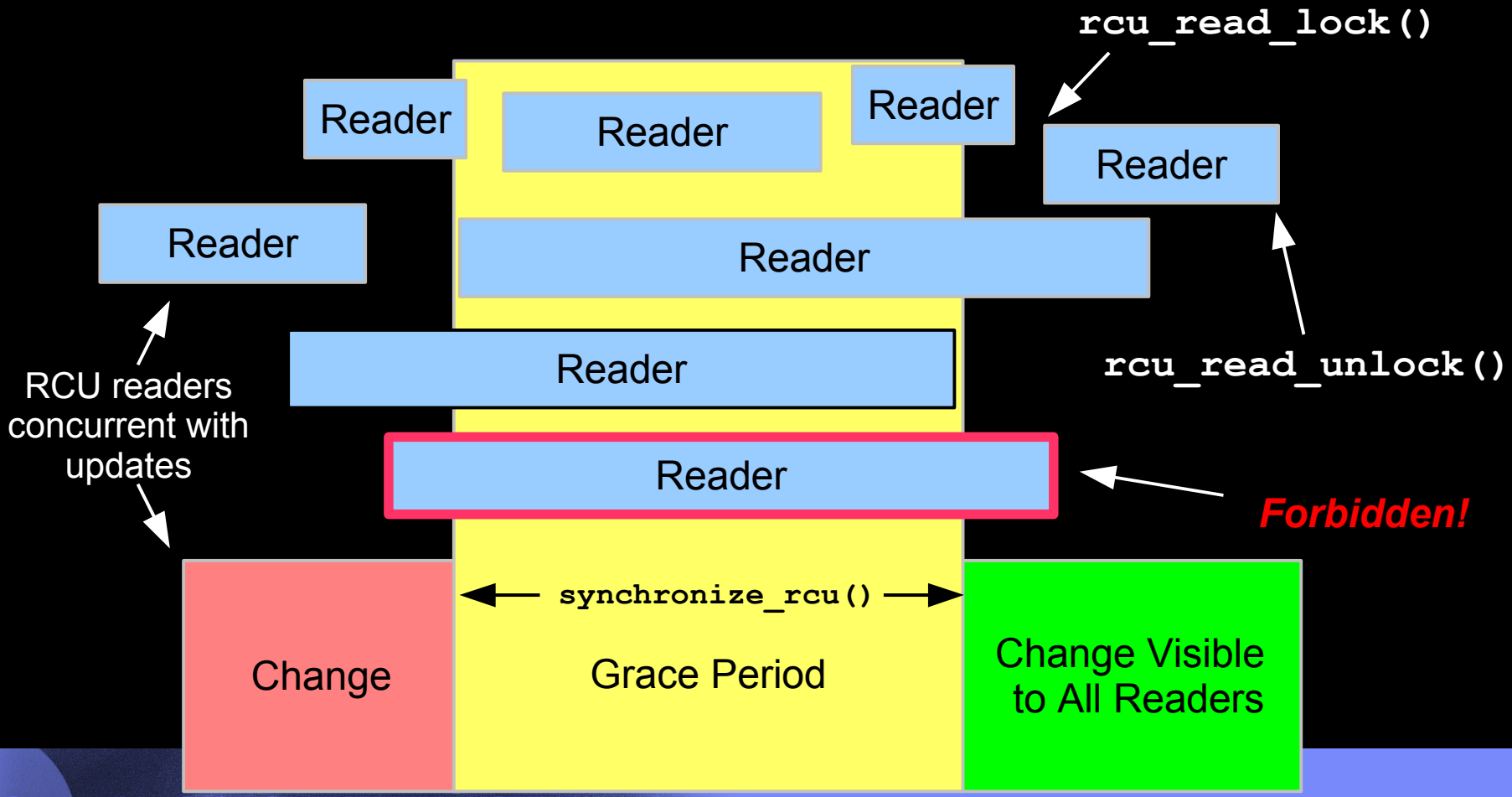
- Non-preemptive environment (CONFIG_PREEMPT=n)
  - RCU readers are not permitted to block
  - Same rule as for tasks holding spinlocks
- CPU context switch means all that CPU's readers are done
- *Grace period* ends after all CPUs execute a context switch



RCU Replacement Of Item In Linked List

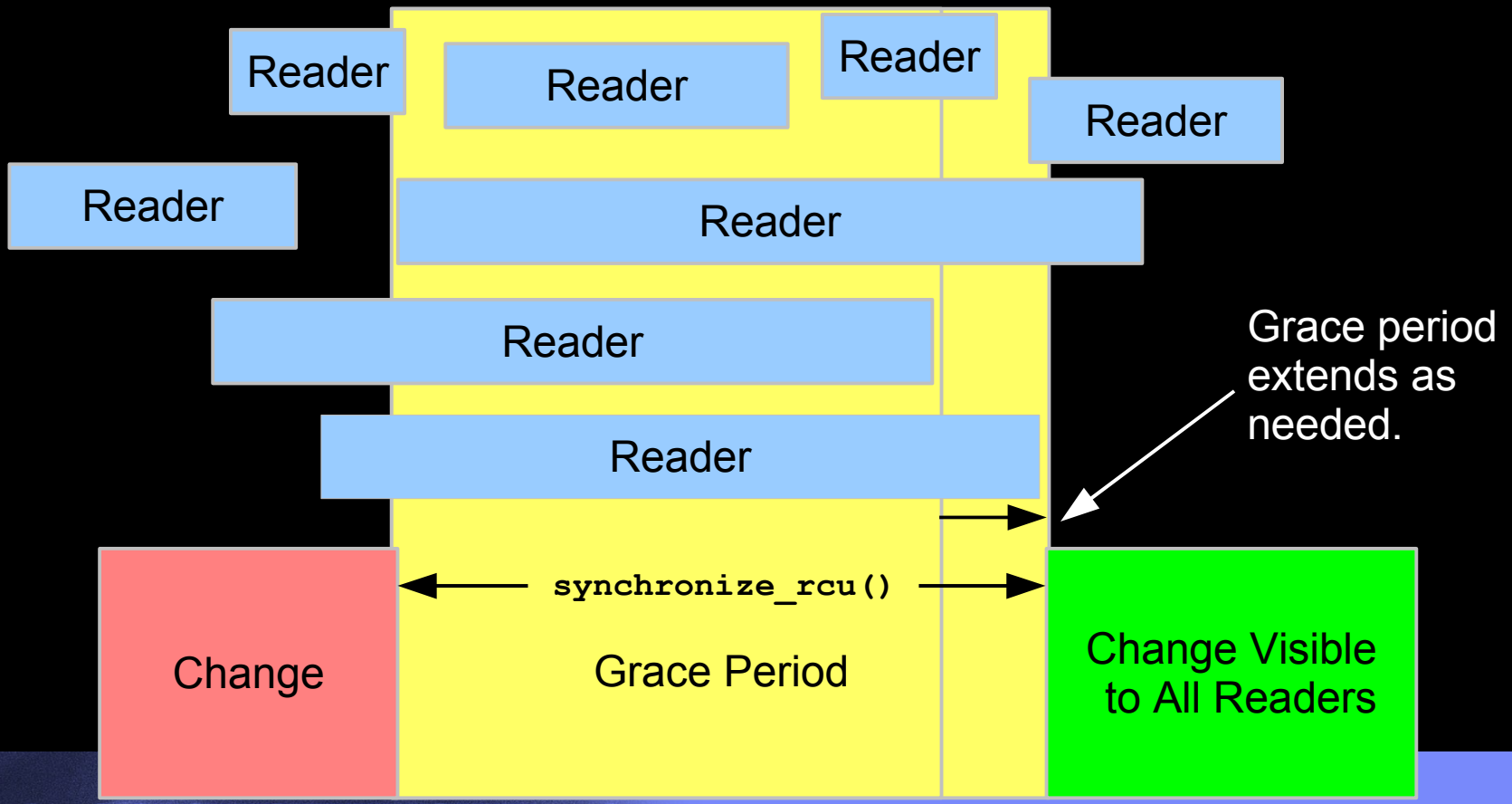


Wait For Pre-Existing RCU Readers



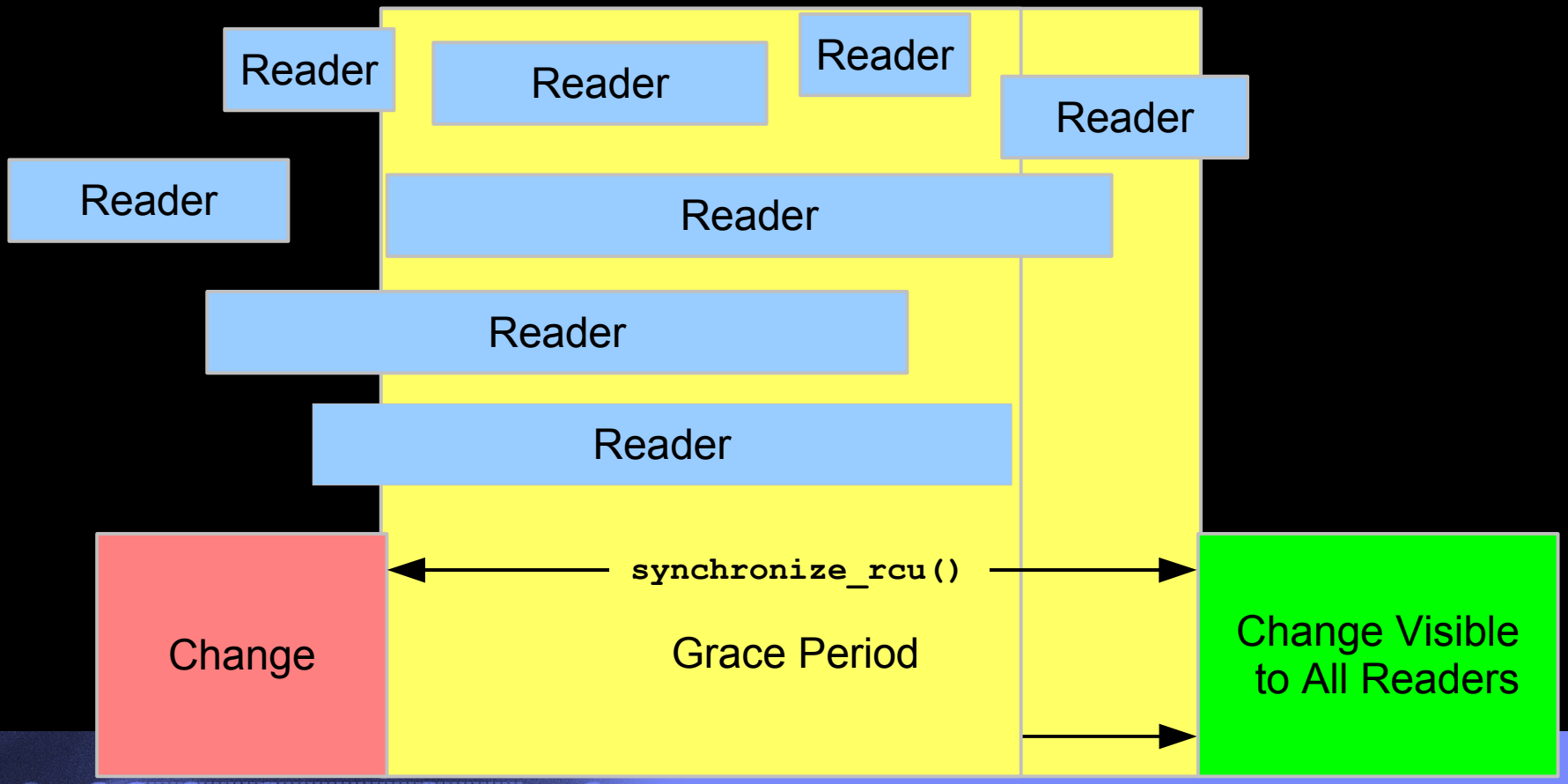
So what happens if you try to extend an RCU read-side critical section across a grace period?

Wait For Pre-Existing RCU Readers



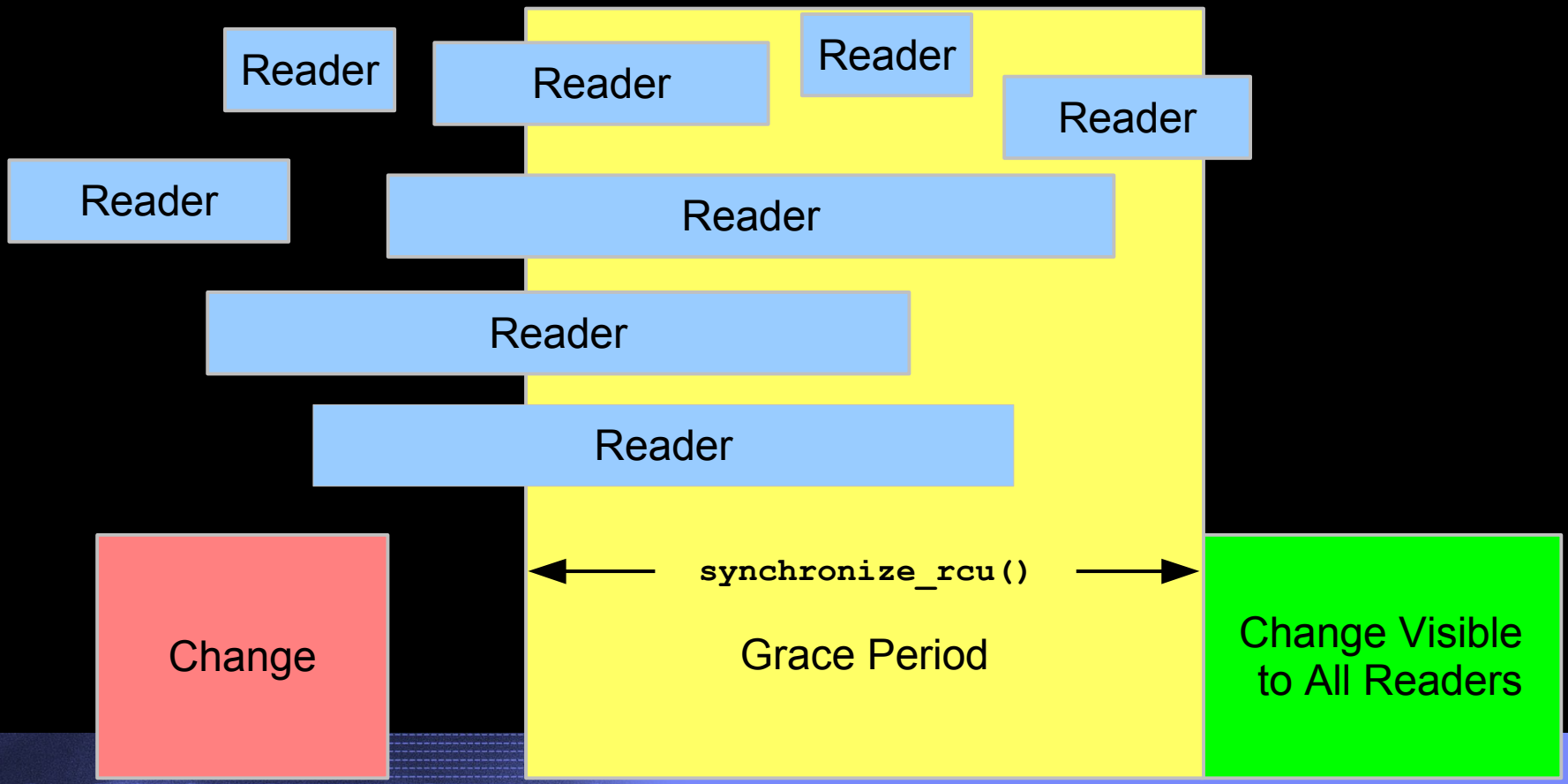
A grace period is not permitted to end until all pre-existing readers have completed.

Wait For Pre-Existing RCU Readers



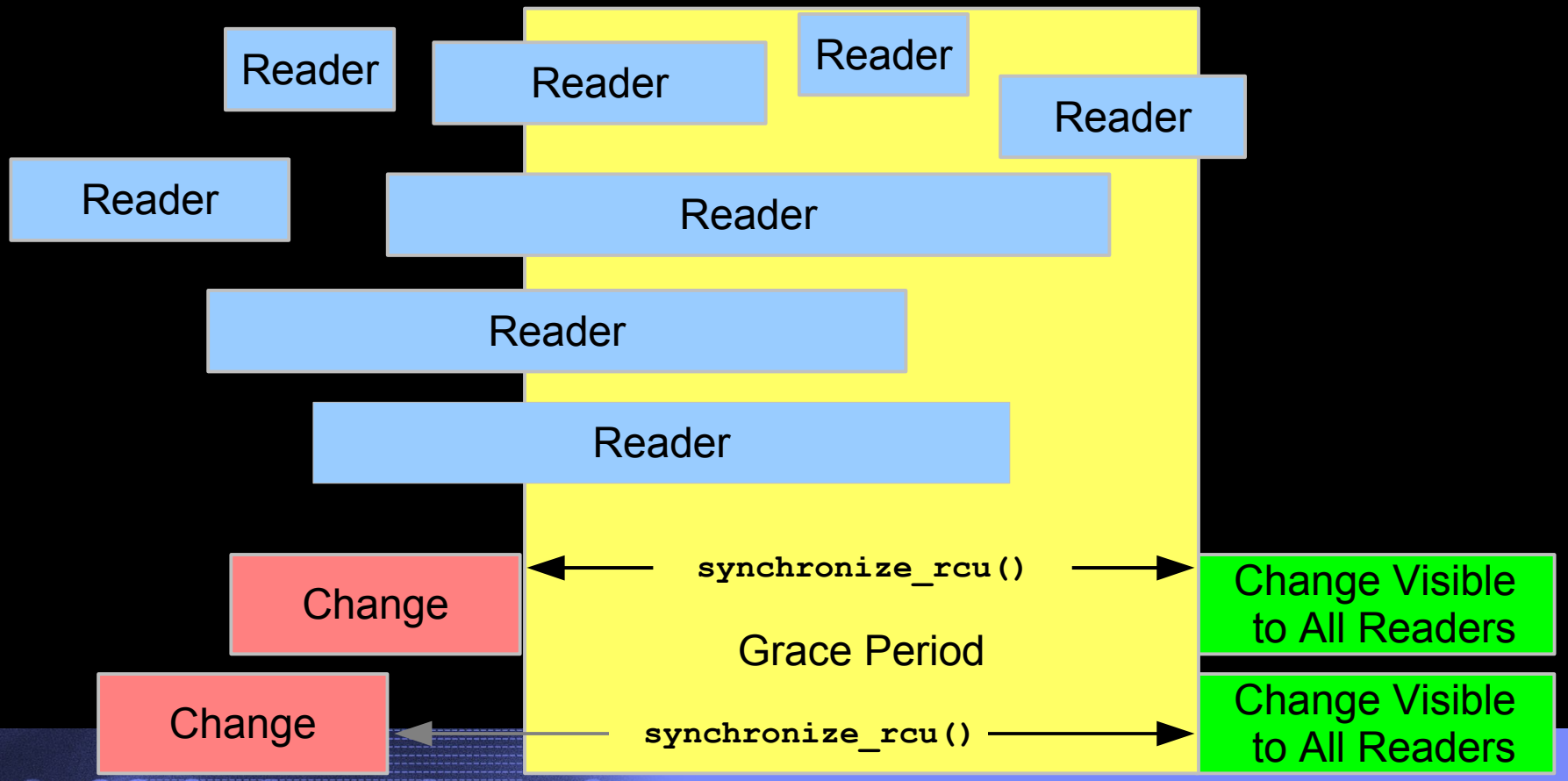
But it is OK for a grace period to extend longer than necessary

Wait For Pre-Existing RCU Readers



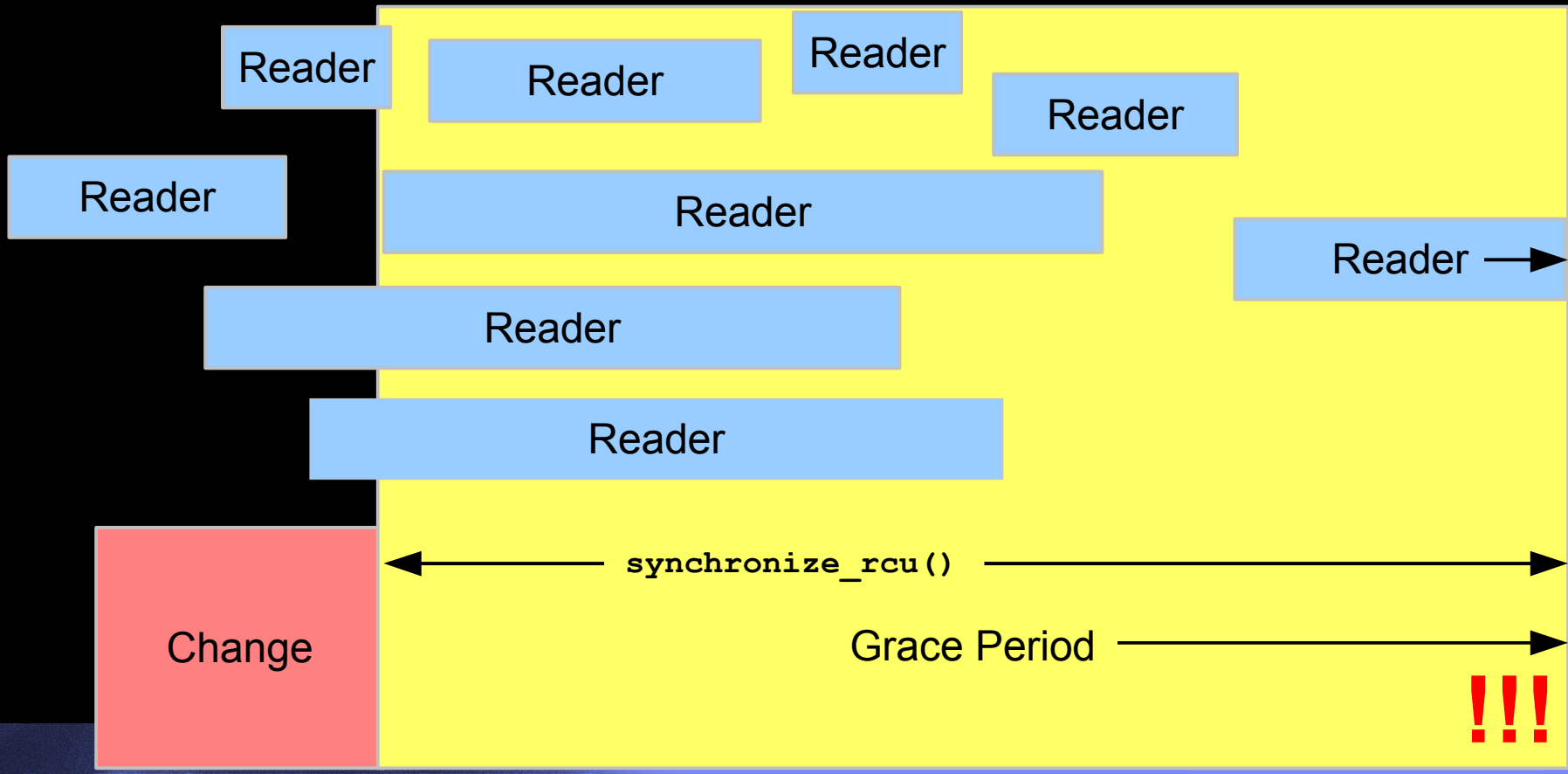
And it is also OK for a grace period to begin later than necessary

Wait For Pre-Existing RCU Readers



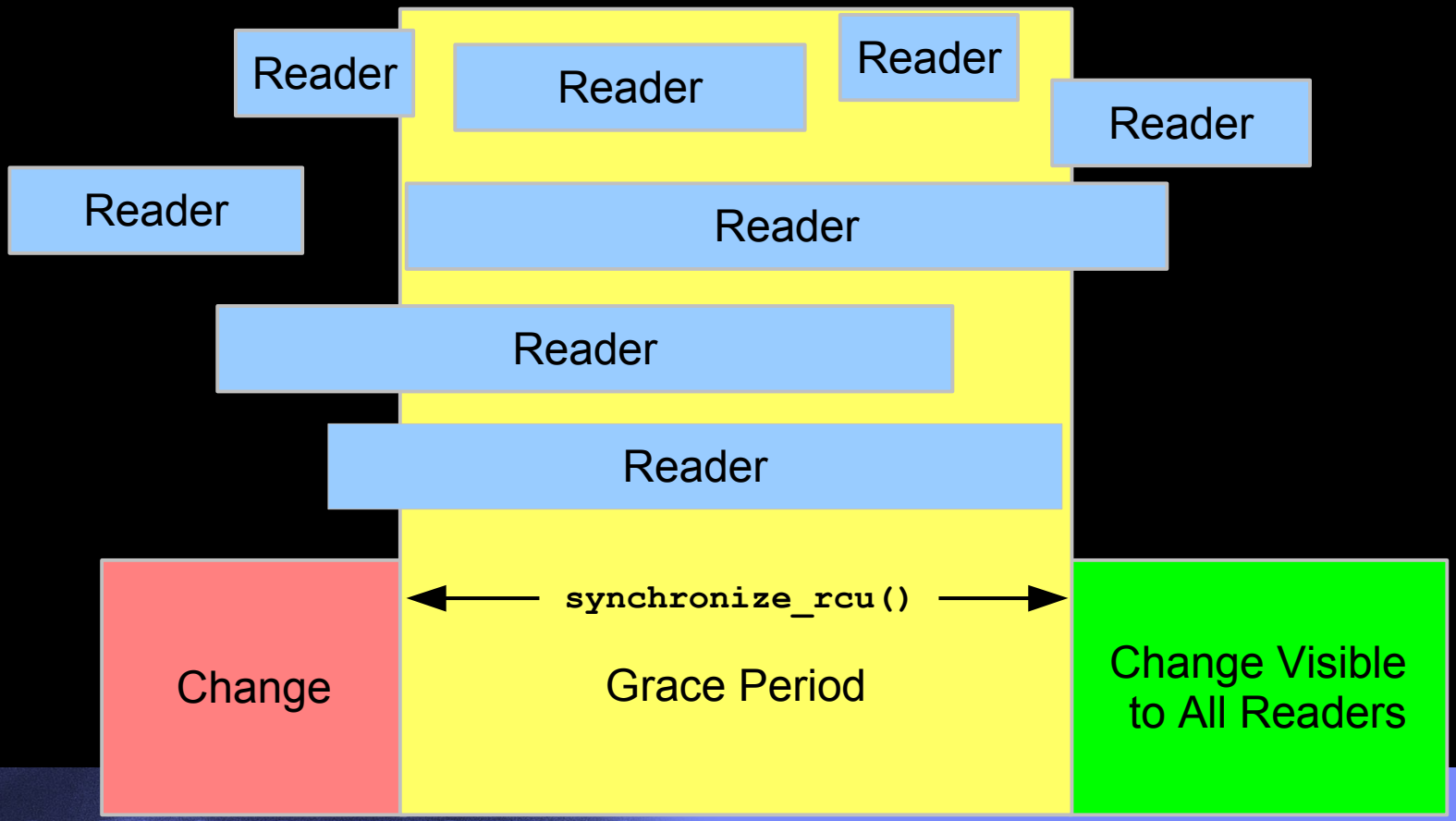
Starting a grace period late can allow it to serve multiple updates.

Wait For Pre-Existing RCU Readers



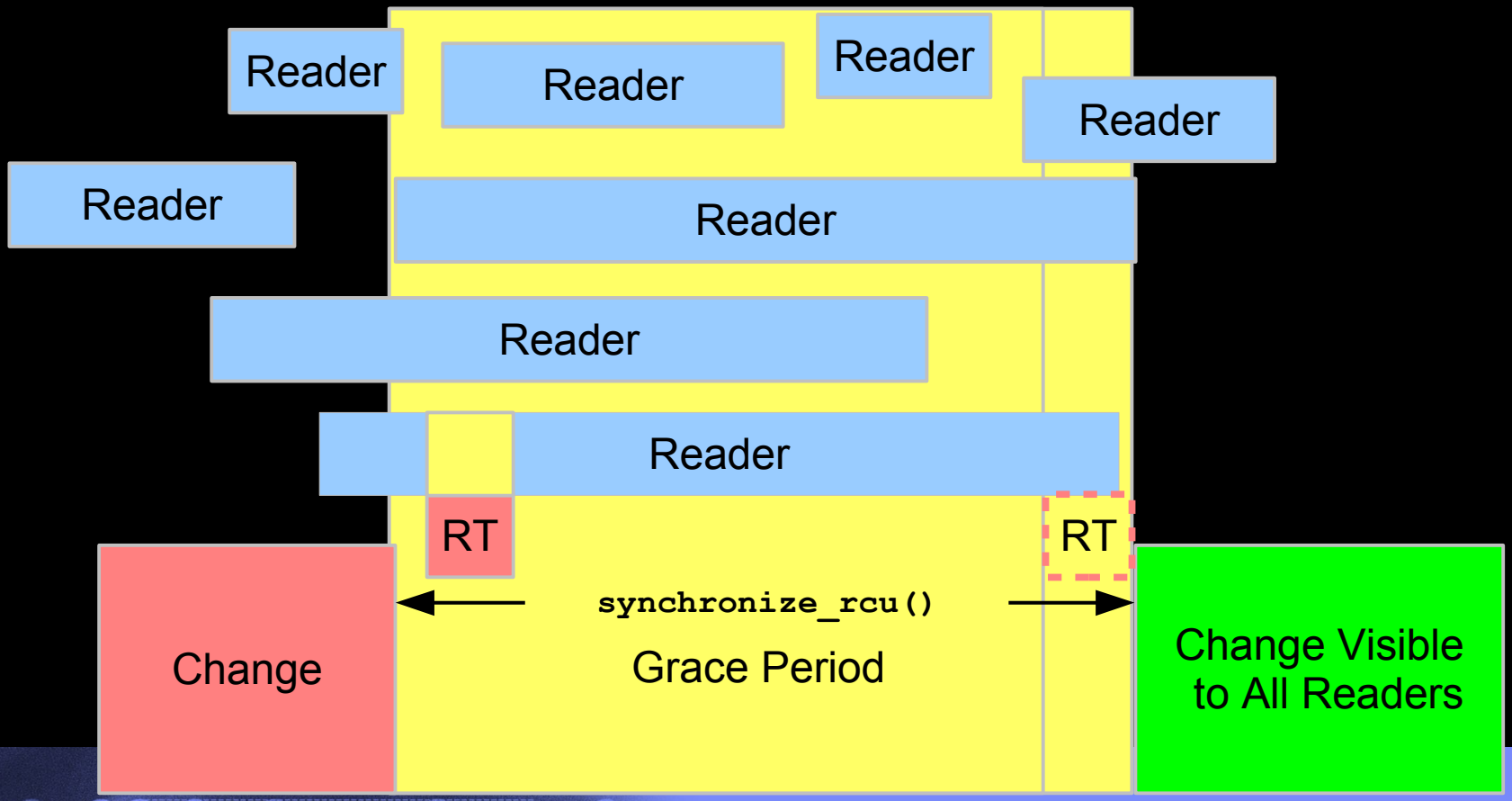
And it is OK for the system to complain (or even abort) if a grace period extends too long. Too-long of grace periods are likely to result in death by memory exhaustion anyway.

Wait For Pre-Existing RCU Readers



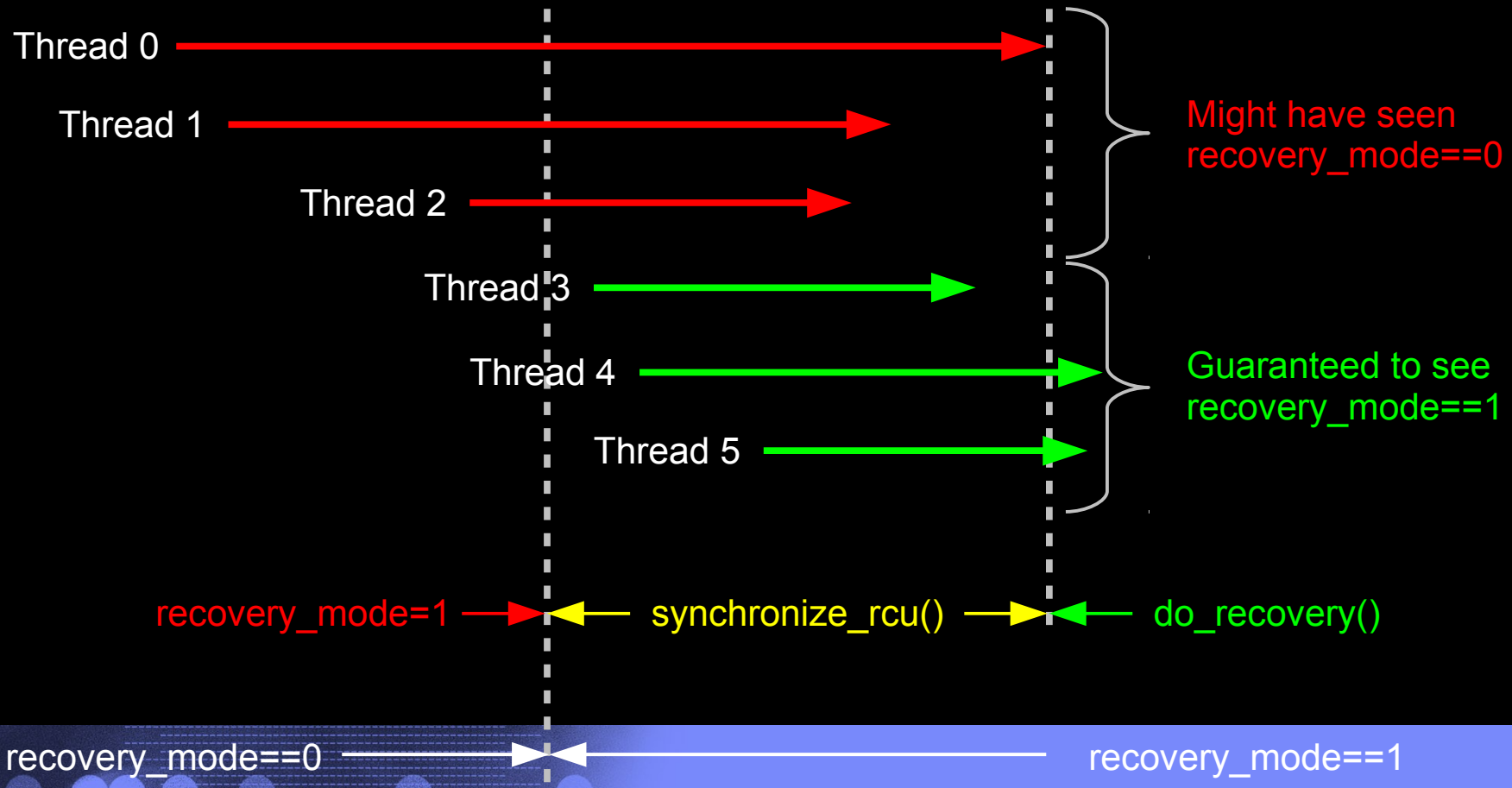
Returning to the minimum-duration grace period.

Wait For Pre-Existing RCU Readers

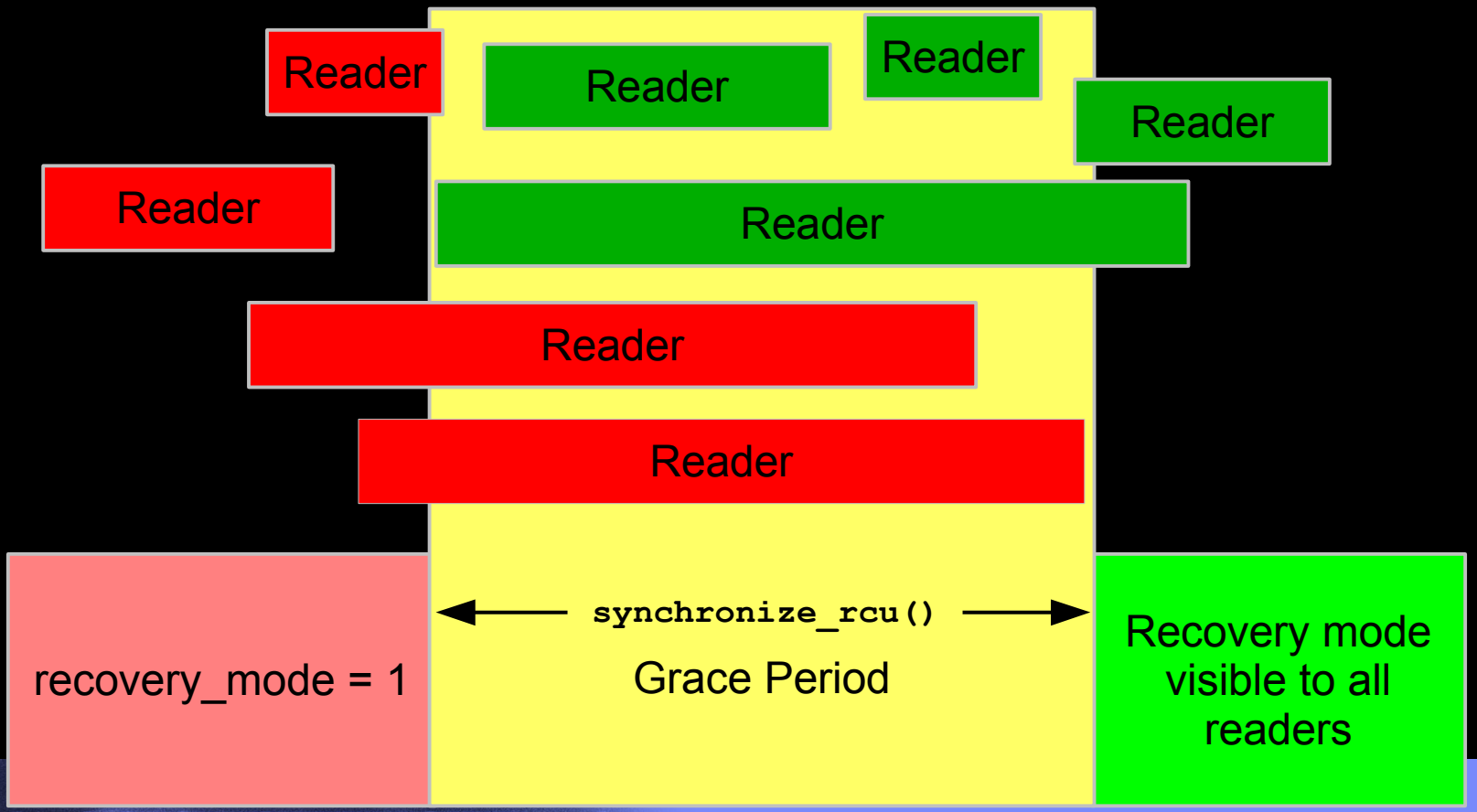


Real-time scheduling constraints can extend grace periods by preempting RCU readers.
It is sometimes necessary to priority-boost RCU readers.

RCU-Mediated Mode Change Diagram



RCU-Mediated Mode Change Diagram Redux

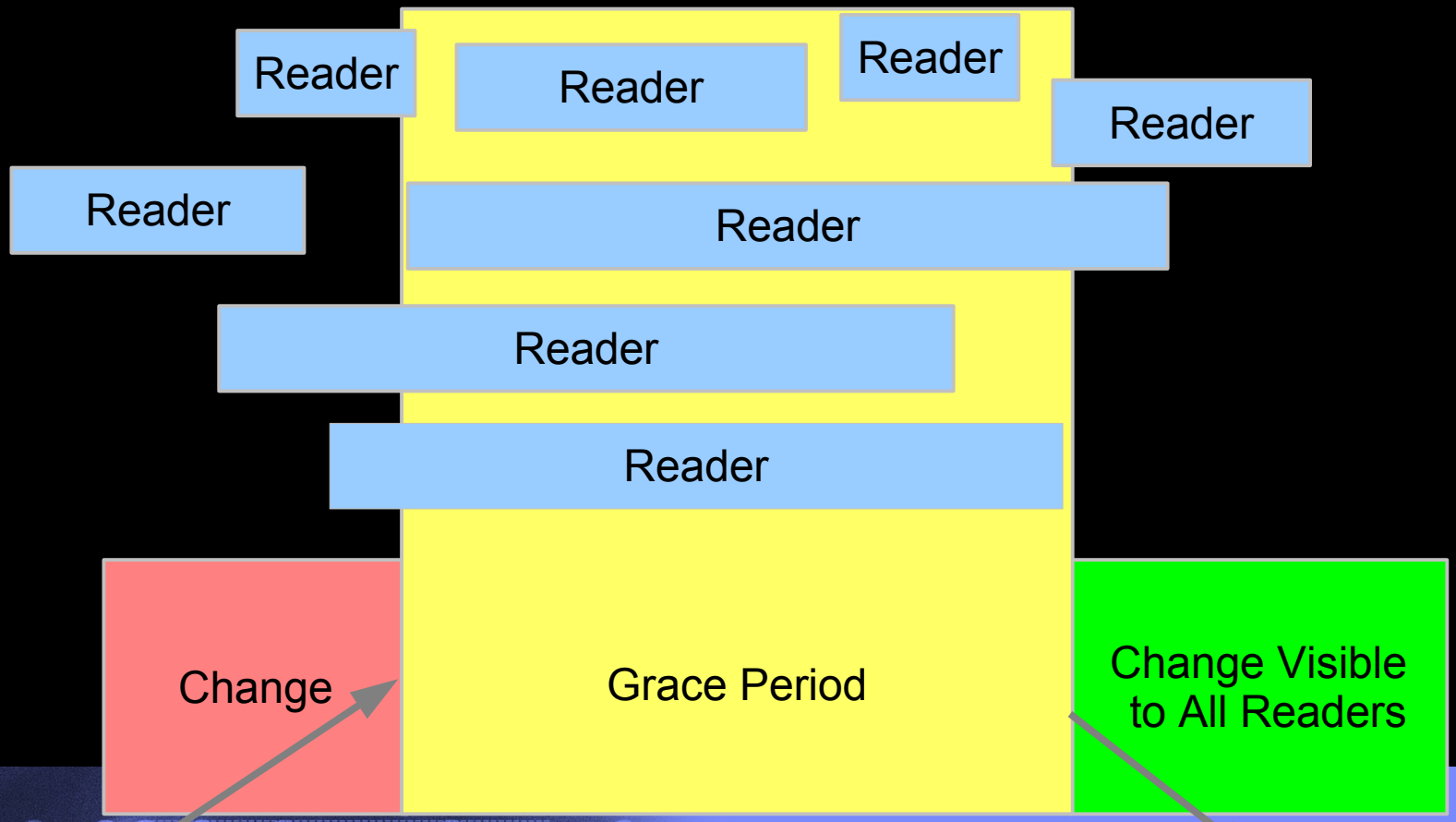


Red readers might be unaware of recovery_mode==1, green readers guaranteed to be aware.

RCU Asynchronous Grace-Period Detection

- The call_rcu() function registers an RCU callback, which is invoked after a subsequent grace period elapses
- API:
  - `call_rcu(struct rcu_head head,`
    `void (*func)(struct rcu_head *rcu));`
- The `rcu_head` structure:
  - `struct rcu_head {`
  - `struct rcu_head *next;`
  - `void (*func)(struct rcu_head *rcu);`
  - `};`
- The `rcu_head` structure is normally embedded within the RCU-protected data structure

RCU Asynchronous Grace-Period Detection




```
call_rcu(&p->rcu, func);
```

```
func(&p->rcu);
```

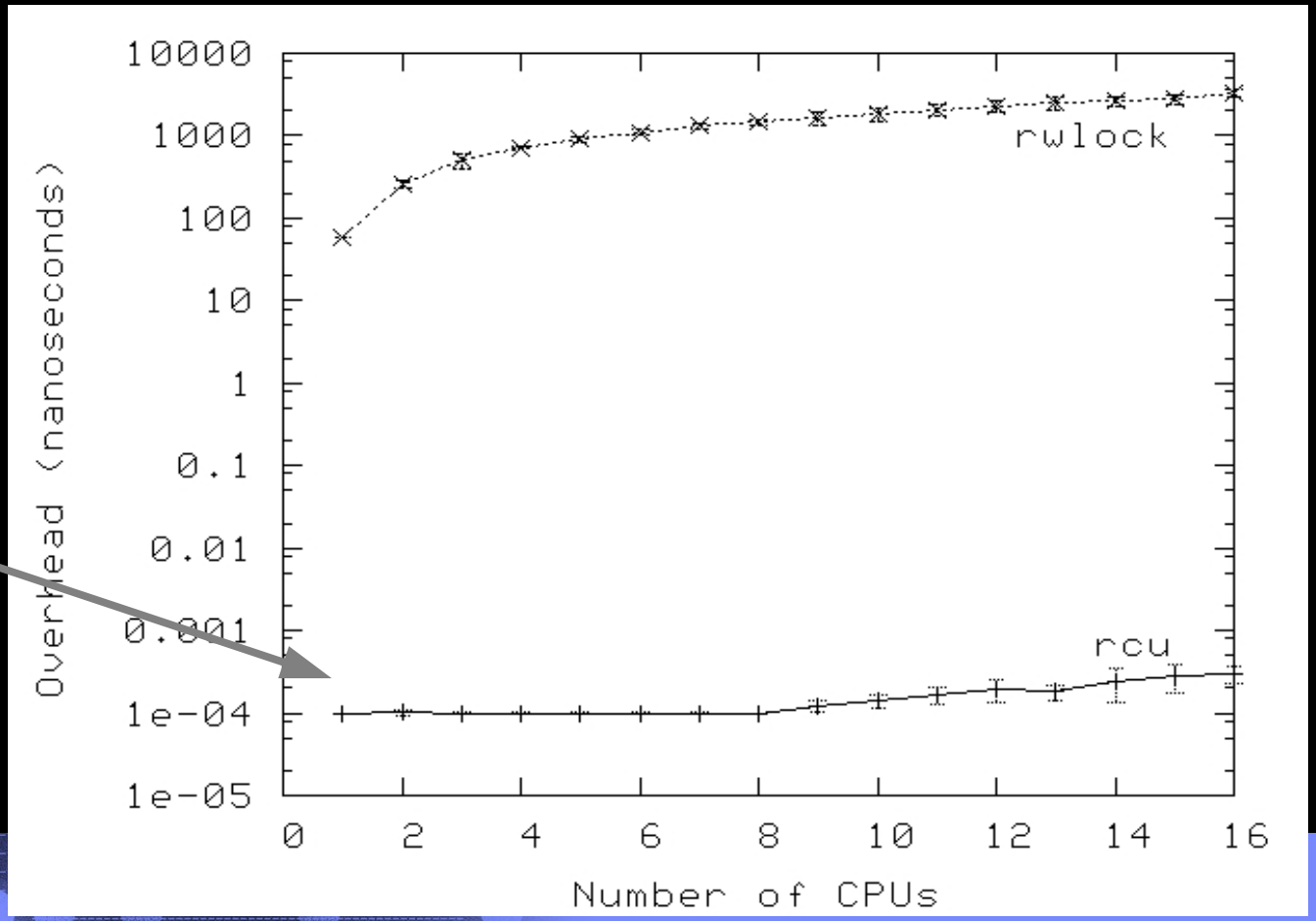
RCU Asynchronous Grace-Period Detection

- But suppose `call_rcu()` is invoked from a kernel module, which is later unloaded?
  - When is it safe to actually unload the module?

RCU Asynchronous Grace-Period Detection

- But suppose `call_rcu()` is invoked from a kernel module, which is later unloaded?
  - When is it safe to actually unload the module?
  - Only after all outstanding RCU callbacks registered by that module have been invoked!
  - Otherwise, later RCU callbacks will try to reference that module's code and data, which have now been unloaded!!!
- Use `rcu_barrier()`: waits until all currently registered RCU callbacks have been invoked
  - After `rcu_barrier()` returns, safe to unload module

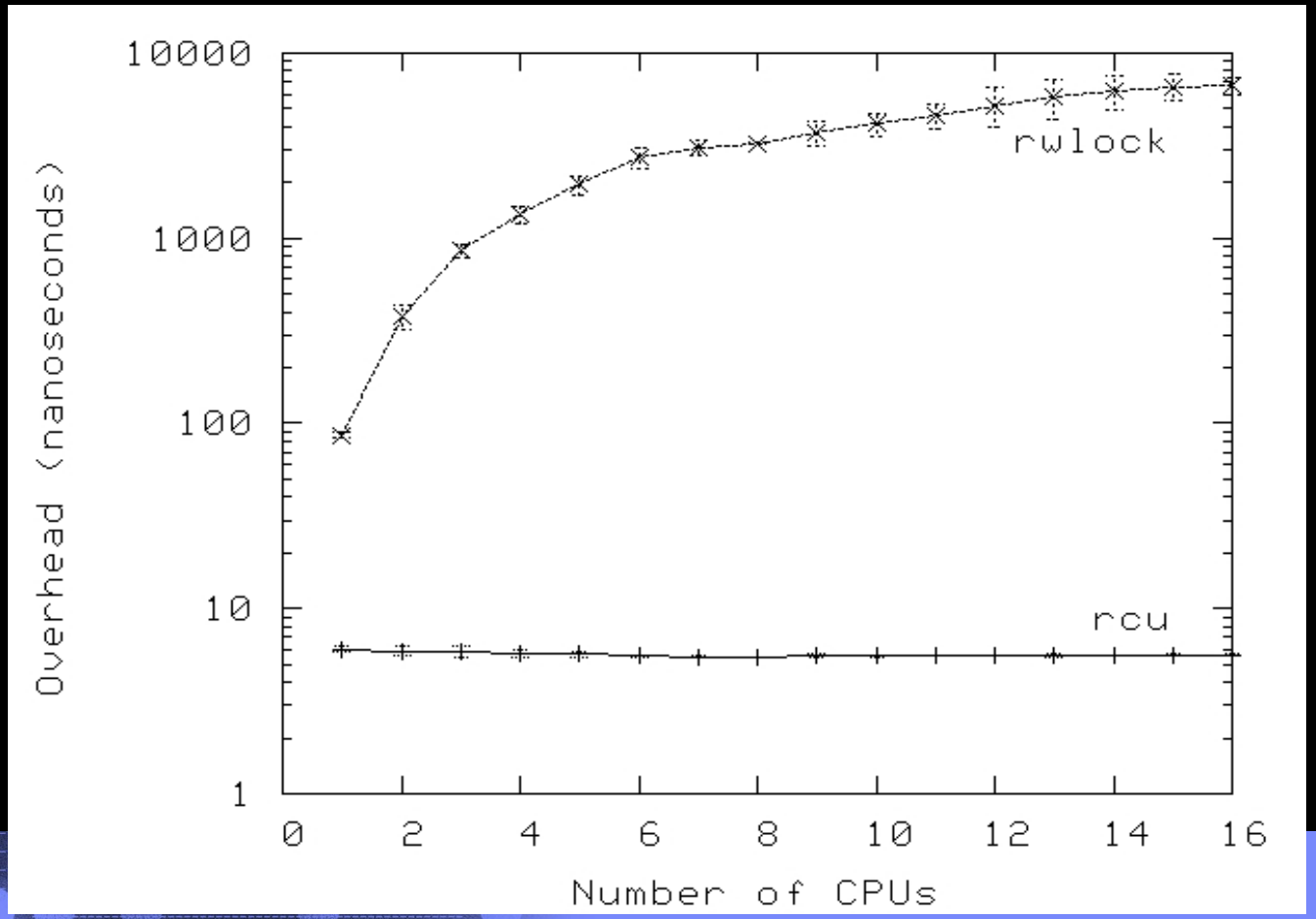
RCU vs. Reader-Writer Locking Performance



How is this possible?

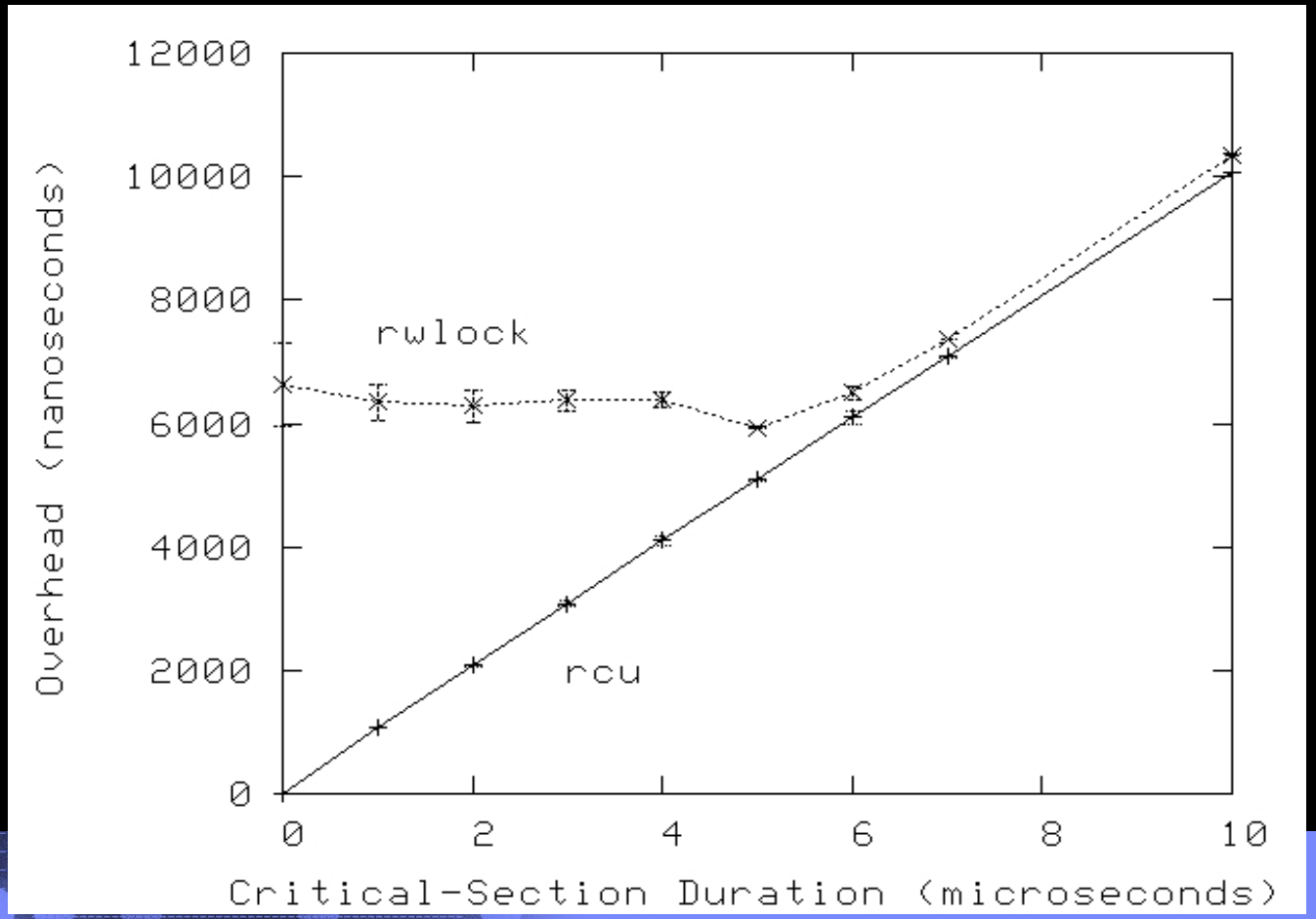
Non-CONFIG_PREEMPT kernel build (QSBR)

RCU vs. Reader-Writer Locking Performance



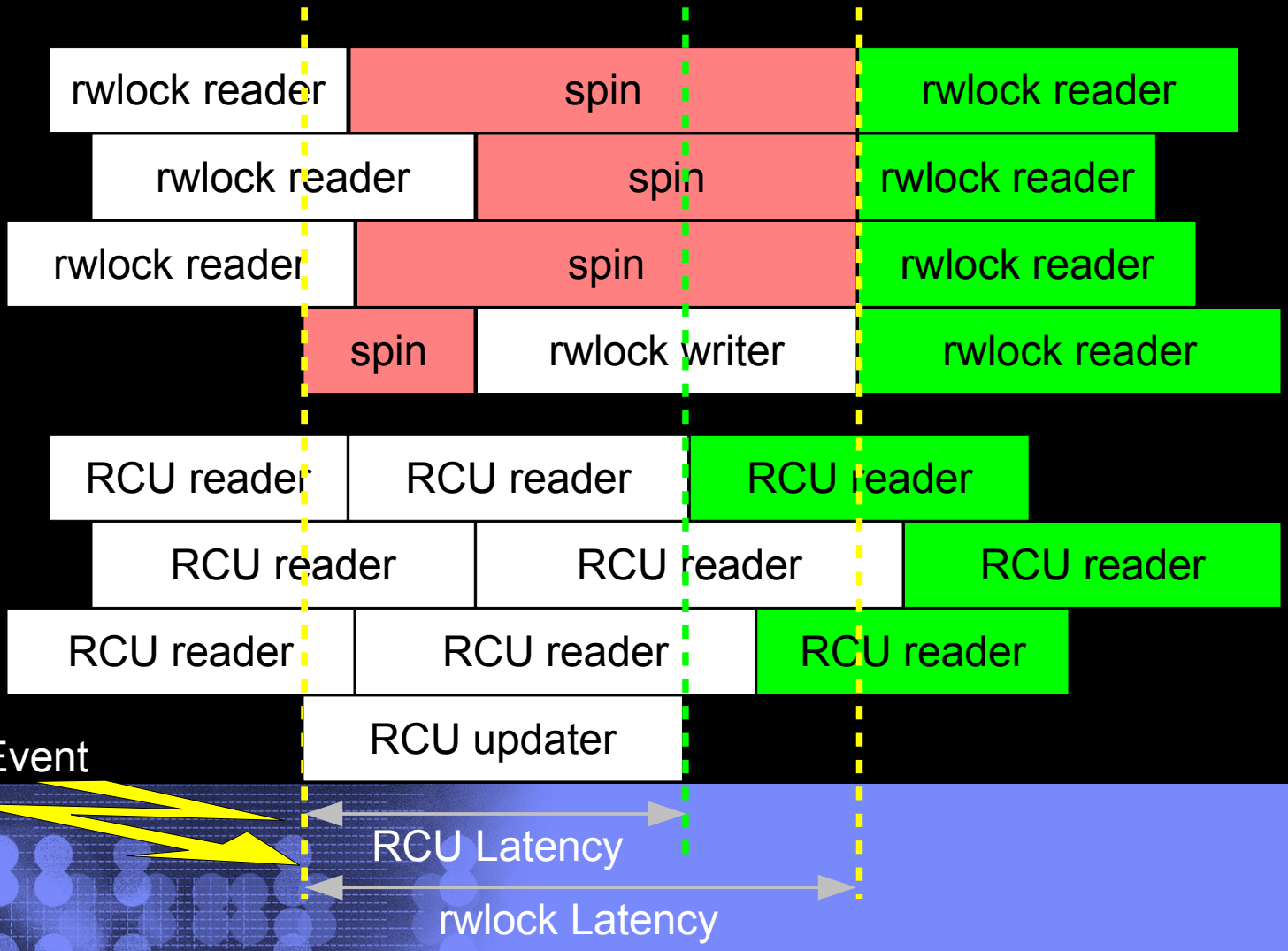
CONFIG_PREEMPT kernel build

RCU vs. Reader-Writer Locking Performance

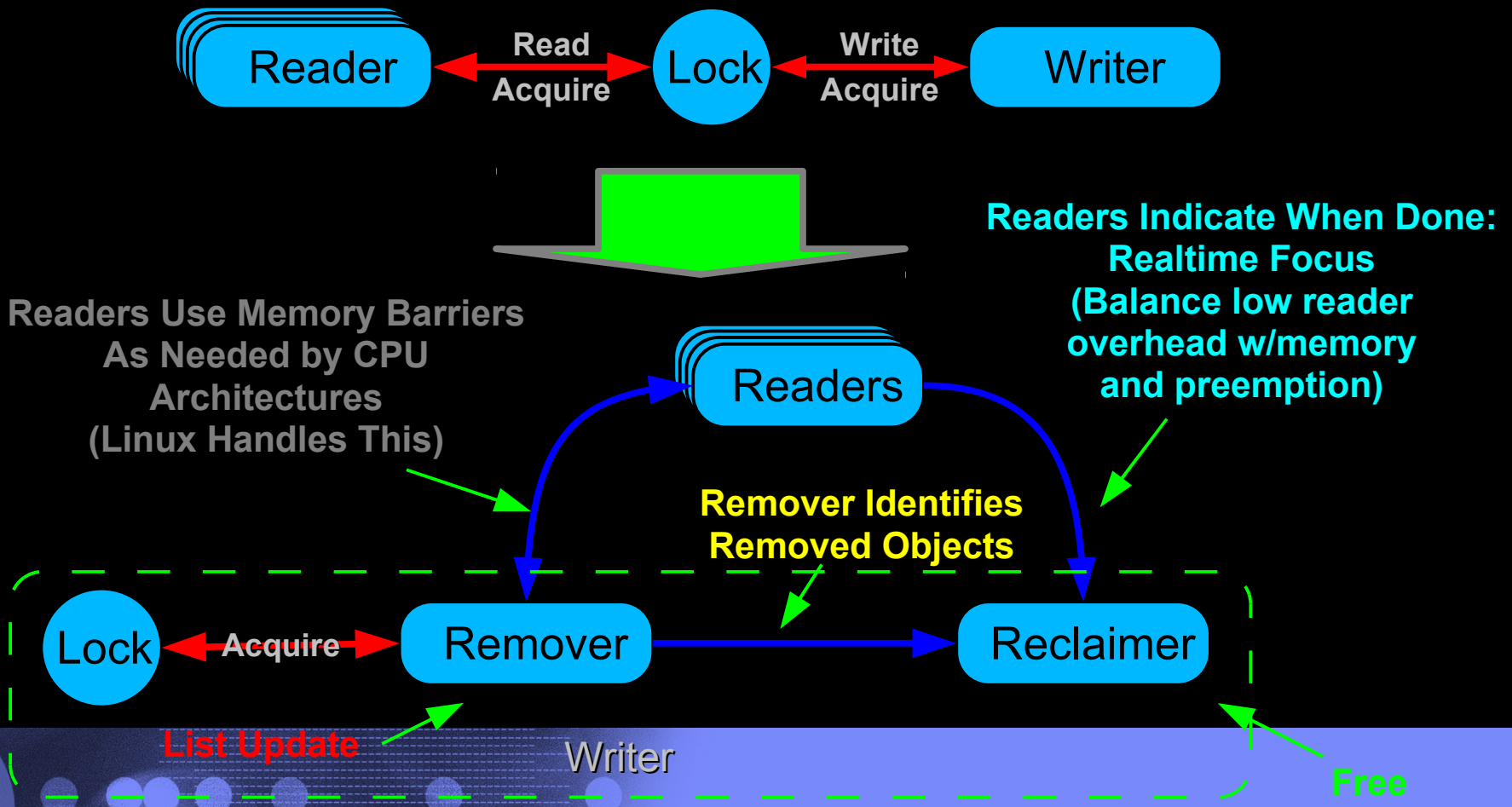


CONFIG_PREEMPT kernel build, 16 CPUs

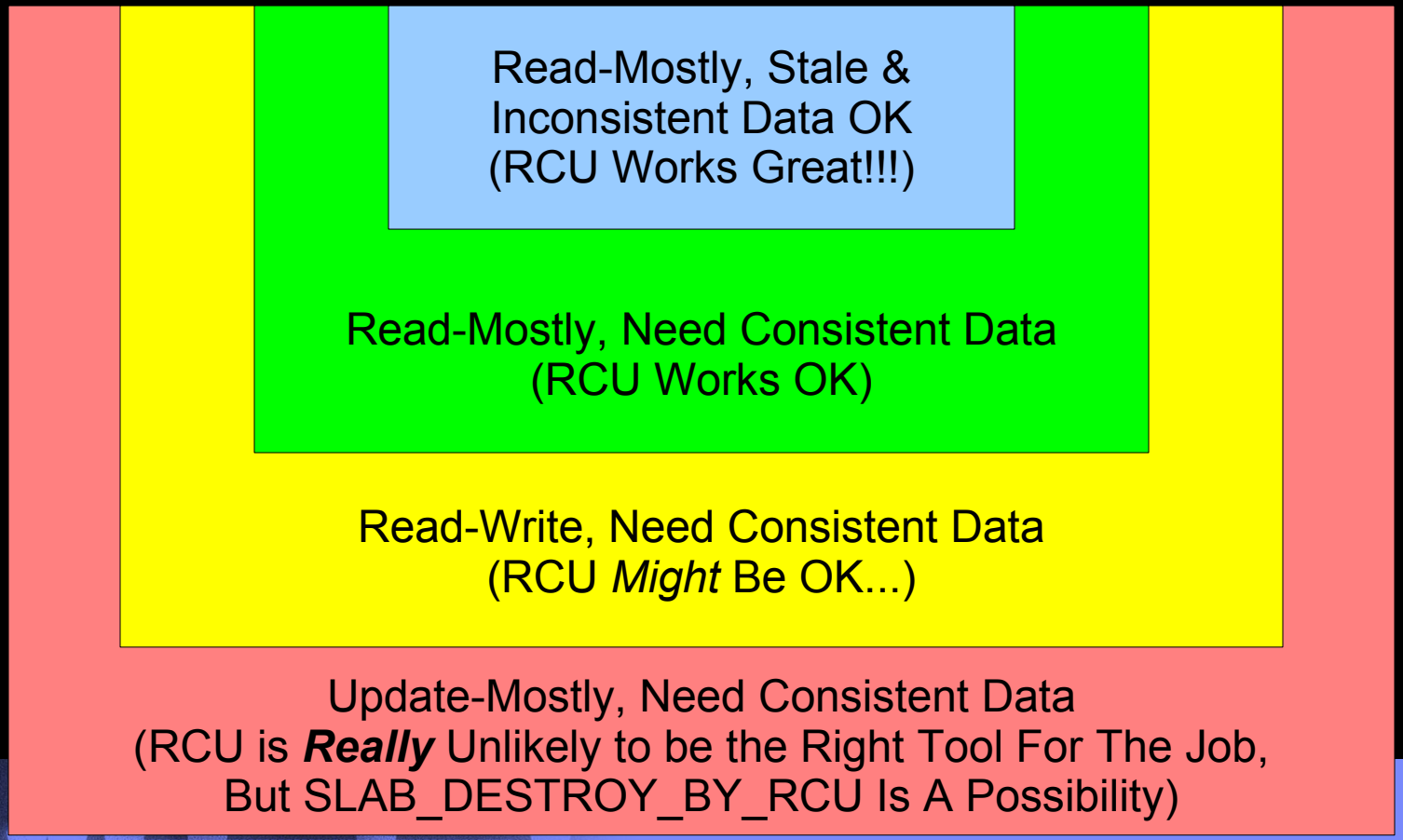
RCU and Reader-Writer Lock Response Latency



Analogy: Reader-Writer Lock vs. RCU



RCU Area of Applicability



To Probe Further into RCU...

- http://lwn.net/Articles/262464/ (What is RCU, Fundamentally?)
- http://lwn.net/Articles/263130/ (What is RCU's Usage?)
- http://lwn.net/Articles/264090/ (What is RCU's API?)
- http://www.rdrop.com/users/paulmck/RCU/lockperf.2004.01.17a.pdf
  - linux.conf.au paper comparing RCU vs. locking performance
- http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf
  - RCU motivation, implementations, usage patterns, performance (micro+sys)
- http://www.livejournal.com/users/james_morris/2153.html
  - System-level performance for SELinux workload: >500x improvement
- http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf
  - Comparison of RCU and NBS (later appeared in JPDC)
- http://doi.acm.org/10.1145/1400097.1400099
  - History of RCU in Linux (Linux changed RCU more than vice versa)
- http://www.rdrop.com/users/paulmck/preprints/
  - User-level implementations of RCU (IEEE TPDS)
- http://www.rdrop.com/users/paulmck/RCU/ (More RCU information)