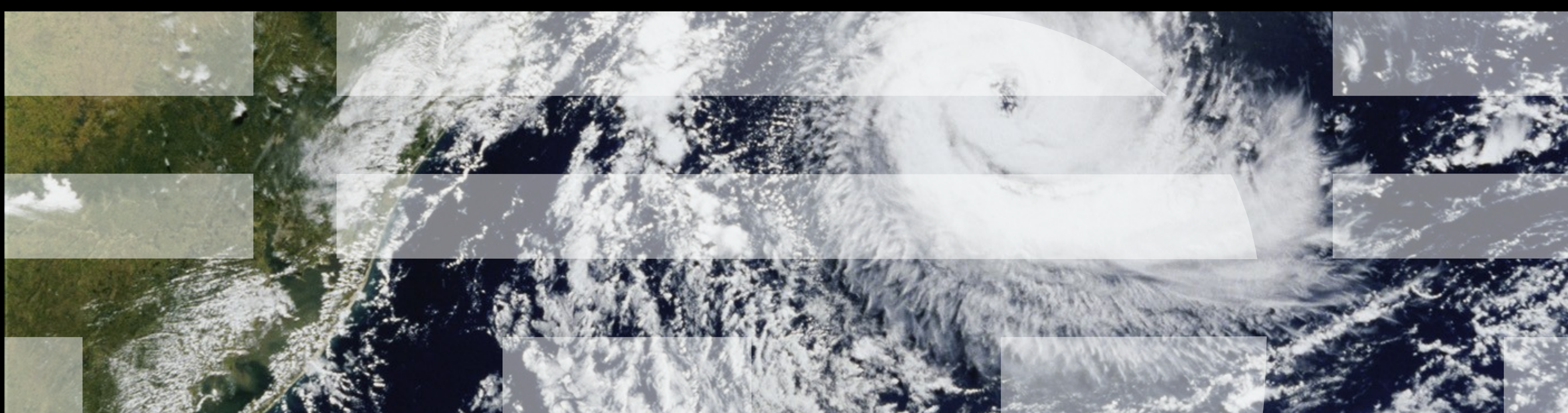
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center
Member, IBM Academy of Technology
Multicore World, February 21, 2017

Does RCU Really Work?

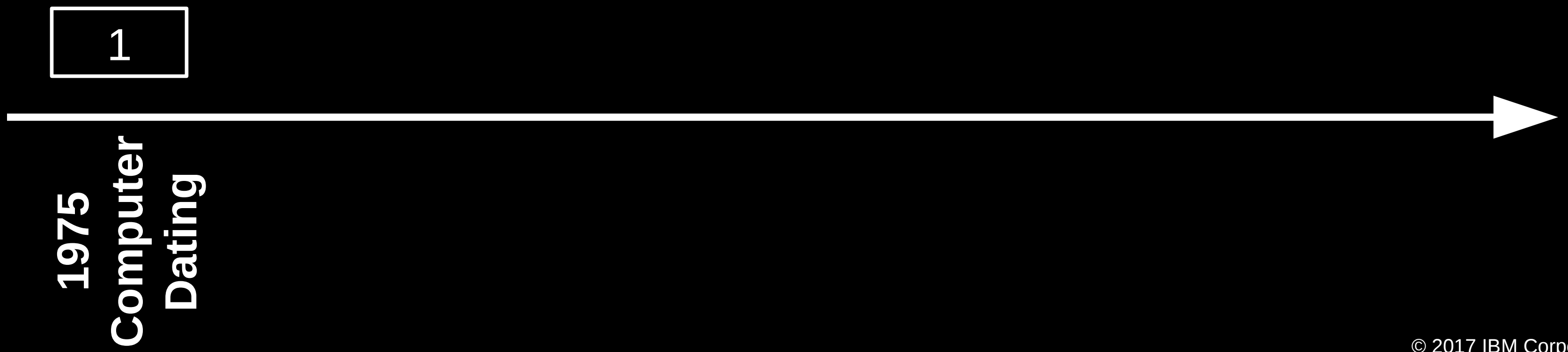
And if so, how would we know?



Isn't Making Software Work A Solved Problem?

Paul's Installed Base Over The Past Four Decades

Million-Year Bug: Once per million years

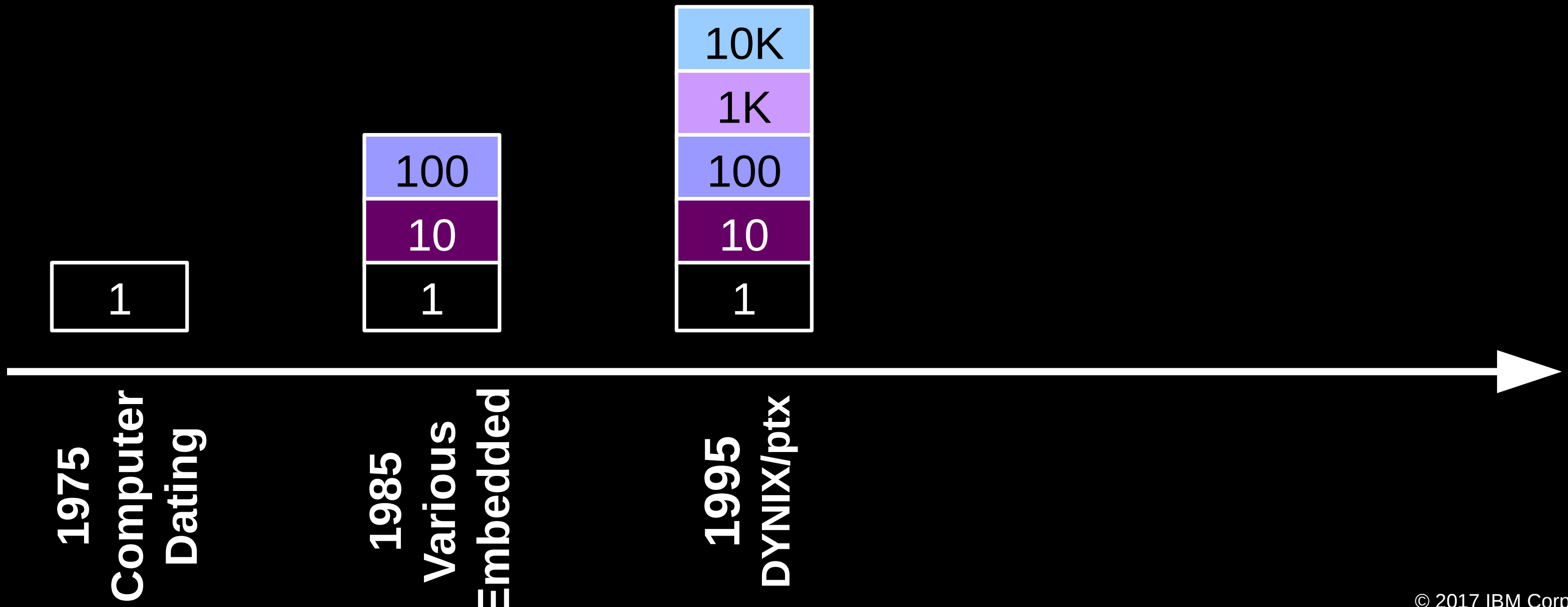


Paul's Installed Base Over The Past Four Decades

Million-Year Bug: Once per ten millenia

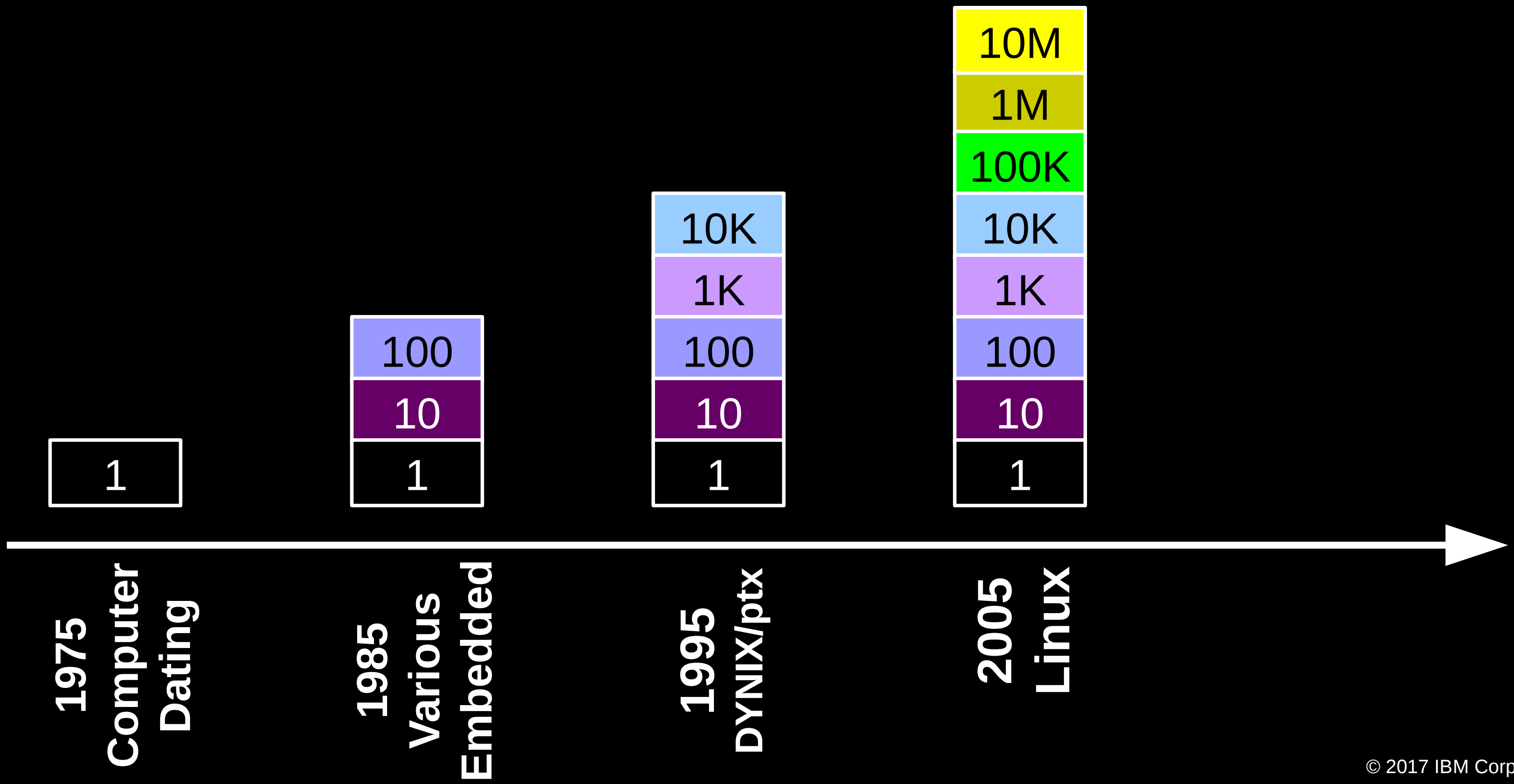
Paul's Installed Base Over The Past Four Decades

Million-Year Bug: Once per century



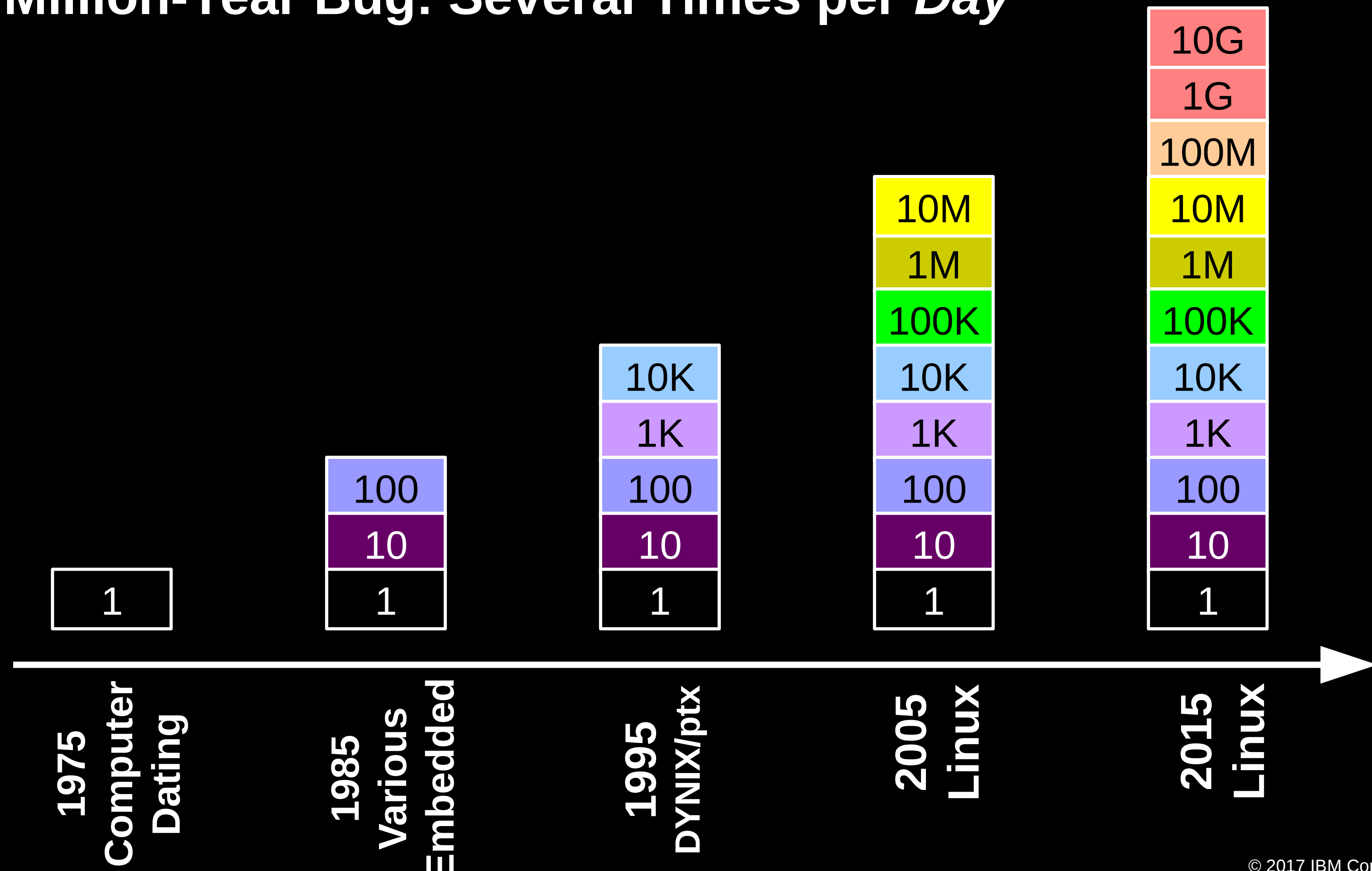
Paul's Installed Base Over The Past Four Decades

Million-Year Bug: Once a month



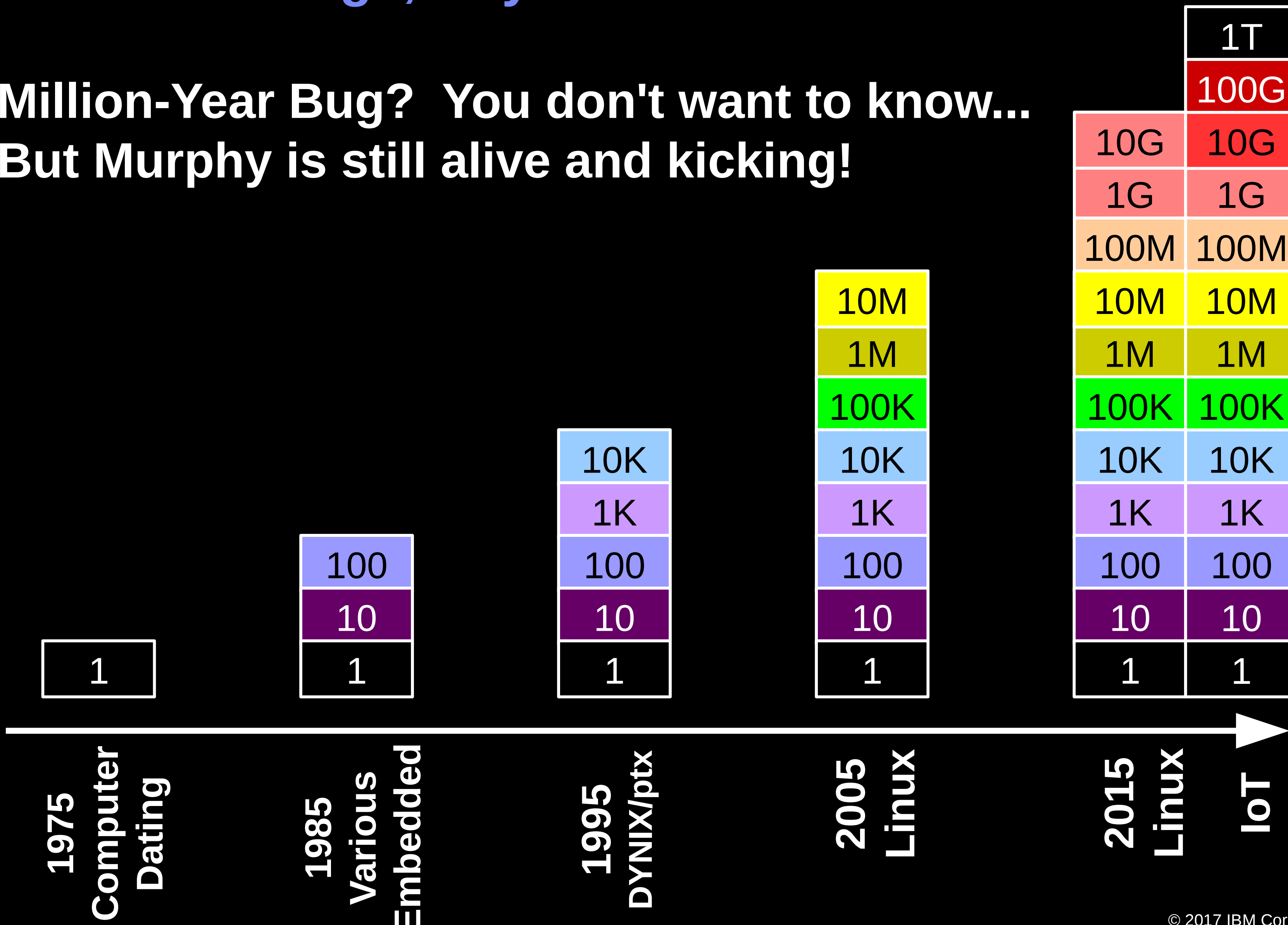
Paul's Installed Base Over The Past Four Decades

Million-Year Bug: Several Times per *Day*



Internet of Things, Anyone???

Million-Year Bug? You don't want to know...
But Murphy is still alive and kicking!



Why Stress About Potential Low-Probability Bugs?

- Almost any bug might become a security exploit
  - Internet access means physical presence no longer required
- RCU's low level does not necessarily mean low risk
  - If Row Hammer can hit DRAM, RCU is not invulnerable
- Internet of Things could mean a trillion computers on Earth
  - Even low failure probability translates to huge numbers of failures
  - Some of which might put the general public at risk
    - Linux is already used in some safety-critical applications
    - Murphy transitions from nice guy to real jerk to homocidal maniac
- It is therefore not too early to think about reducing risk
  - And RCU is a good well-contained test case for proofs of concept

Does RCU Really Work?
If So, How Would We Know?

Does RCU Really Work? If So, How Would We Know?

- What is RCU (read-copy update) supposed to do?
- What are the odds of RCU “just working”?
- RCU validation

What is RCU Supposed To Do?

What is RCU Supposed To Do? (Brief Overview!)

- Structured deferral: synchronization via procrastination
  - The waiters: *RCU grace periods*
    - synchronize_rcu(), call_rcu(), …
  - The waited upon: *RCU read-side critical sections*
    - *rcu_read_lock() and rcu_read_unlock, …*
    - RCU's read-side primitives have exceedingly low overhead, great scalability
- RCU grace periods must wait for pre-existing RCU read-side critical sections
  - How could this possibly be useful? See next slides...
- Other examples of synchronization via procrastination:
  - Reference counting, sequence locking, hazard pointers, garbage collectors
  - Arguably also locking (new acquisition must wait for old acquisition)

What RCU is Supposed To Do

```
void thread0(void)
{
  rcu_read_lock();
  /* p = gp, sort of. */
  p = rcu_dereference(gp);



  do_something_with(p->a);
  rcu_read_unlock();
}
```

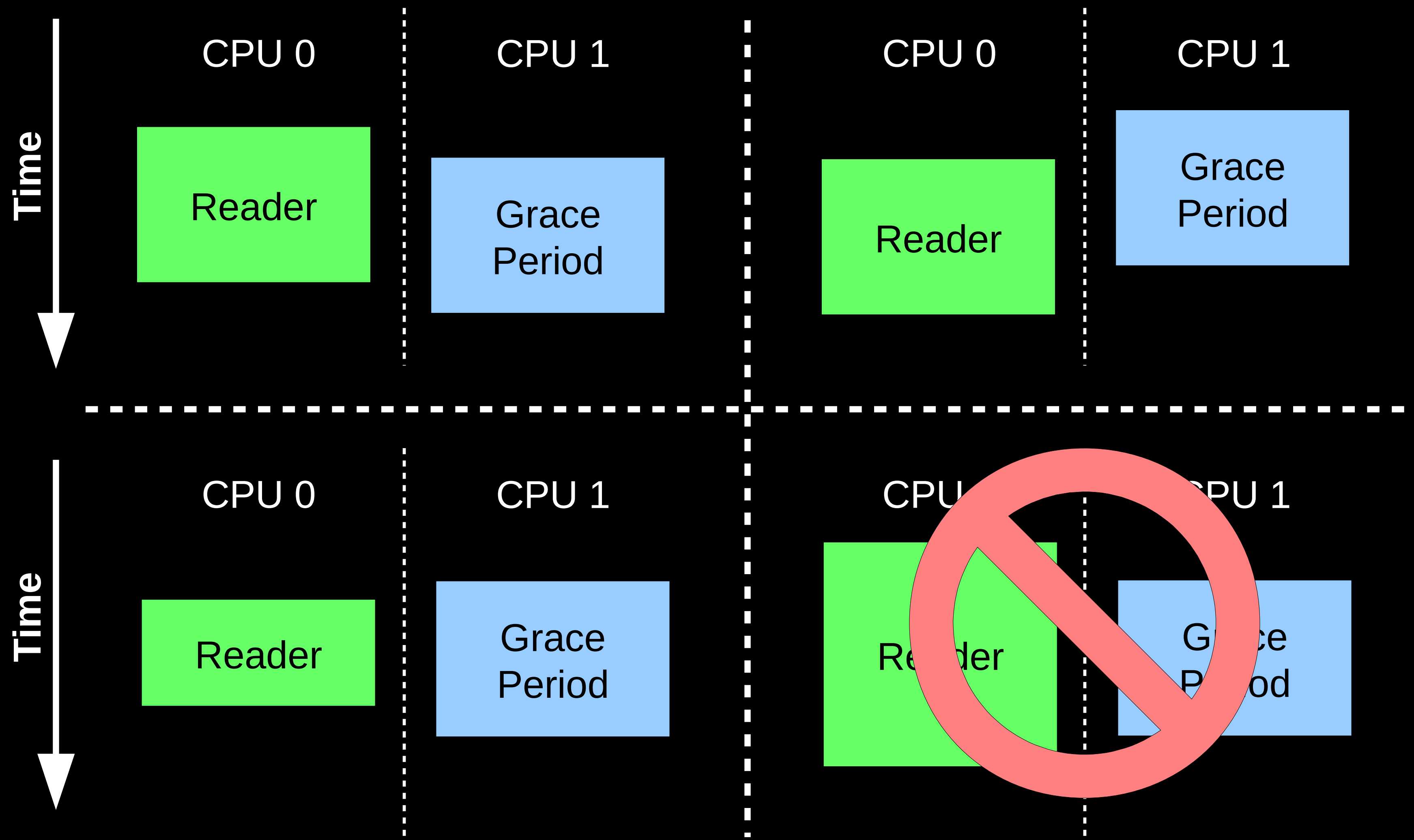
```
void thread1(void)
{
  q = alloc_something();
  p = gp;
  /* gp = p, sort of. */
  rcu_assign_pointer(gp, q);
  synchronize_rcu();
  /* wait */
  /* wait */
  /* wait */
  /* wait */
  free(p);
}
```

What RCU is Supposed To Do

```
void thread1(void)
{
  q = alloc_something();
  p = gp;
  rcu_assign_pointer(gp, q);
  synchronize_rcu();


  free(p);
}
```

```
void P2(void)
{



    rcu_read_lock();
    p = rcu_dereference(gp);



    do_something_with(p->a);
    rcu_read_unlock();
}
```

What RCU Is Supposed To Do and Not...



What RCU is Supposed To Do

- Read-side primitives are exceedingly low overhead
  - rcu_read_lock(), rcu_read_unlock(), rcu_dereference(), …
  - Free is a ***very*** good price!!!
- RCU therefore provides high scalability and performance for access to read-mostly linked data structures
  - And is therefore heavily used in the Linux kernel and elsewhere
- But the devil is in the details!
  - CPU hotplug, idle CPUs, energy efficiency, 4096-CPU systems, real-time response, boot vs. runtime...
  - RCU's specification is empirical in nature!
    - https://lwn.net/Articles/652156/, https://lwn.net/Articles/652677/, and https://lwn.net/Articles/653326/
    - Linux kernel source: Documentation/RCU/Design/Requirements/

RCU Area of Applicability

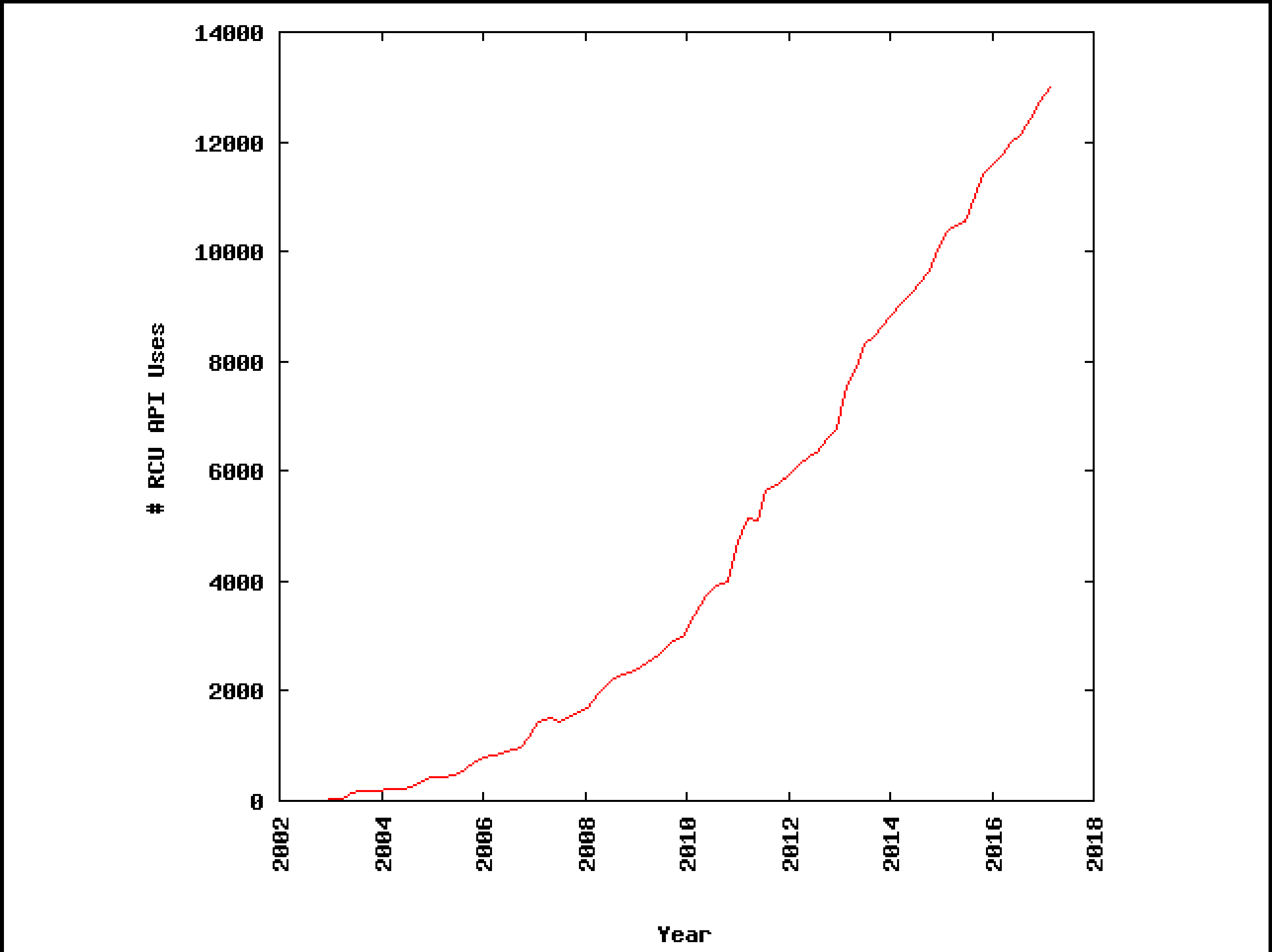
Read-Mostly, Stale &
Inconsistent Data OK
(RCU Works Great!!!)

Read-Mostly, Need Consistent Data
(RCU Works OK)

Read-Write, Need Consistent Data
(RCU *Might* Be OK...)

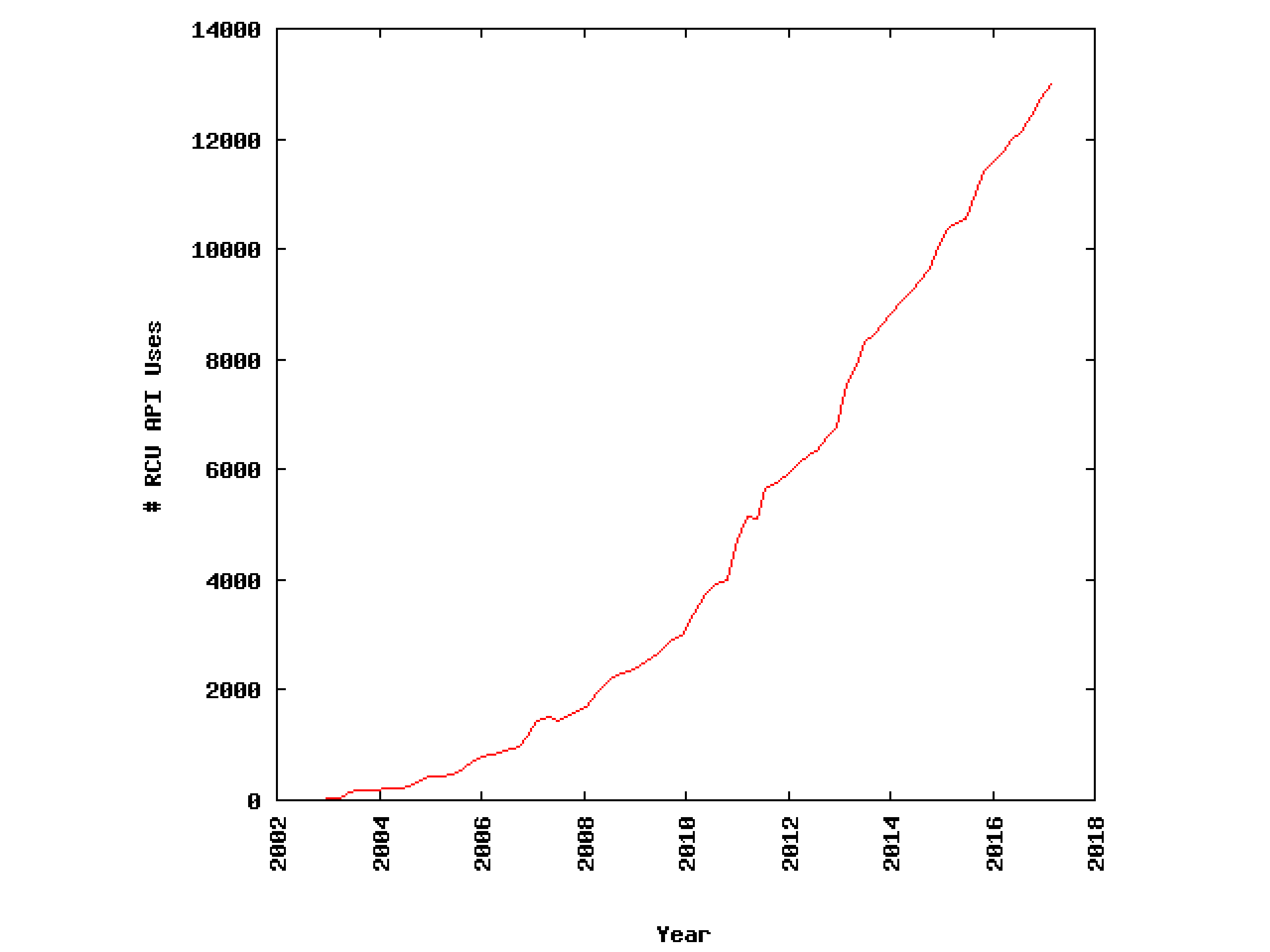
Update-Mostly, Need Consistent Data
(RCU is ***Really*** Unlikely to be the Right Tool For The Job, But It Can:
(1) Provide Existence Guarantees For Update-Friendly Mechanisms
(2) Provide Wait-Free Read-Side Primitives for Real-Time Use)

RCU Applicability to the Linux Kernel



In 1996, I thought I knew everything there was to know about RCU

RCU Applicability to the Linux Kernel



In 1996, I thought I knew everything there was to know about RCU
The Linux kernel community proved me wrong many times!!!

What Are The Odds of RCU “Just Working”?

Two Definitions and a Consequence

Two Definitions and a Consequence

- A ***bug-free software system*** is a trivial software system
- A ***reliable software system*** contains no known bugs

Two Definitions and a Consequence

- A bug-free software system is a trivial software system
- A reliable software system contains no known bugs

- Therefore, any non-trivial reliable software system contains at least one bug that you don't know about

Two Definitions and a Consequence

- A bug-free software system is a trivial software system
- A reliable software system contains no known bugs

- Therefore, any non-trivial reliable software system contains at least one bug that you don't (yet) know about
- I assert that Linux-kernel RCU is both non-trivial and reliable, thus containing at least one bug that I don't (yet) know about

Two Definitions and a Consequence

- A bug-free software system is a trivial software system
- A reliable software system contains no known bugs

- Therefore, any non-trivial reliable software system contains at least one bug that you don't know about
- I assert that Linux-kernel RCU is both non-trivial and reliable, thus containing at least one bug that I don't (yet) know about
- But how many bugs?
  - Analyze from a software-engineering viewpoint...

Software-Engineering Analysis

Software-Engineering Analysis

- RCU contains 11,534 lines of code (including comments, etc.)
- 1-3 bugs/KLoC for production-quality code: ***11-36 bugs***
  - Best case I have seen: 0.04 bugs/KLoC for safety-critical code
    - Extreme code-style restrictions, single-threaded, formal methods, …
    - And still way more than zero bugs!!! :-)
- Median age of a line of RCU code is less than four years
  - And young code tends to be buggier than old code!

Software-Engineering Analysis

- RCU contains 11,534 lines of code (including comments, etc.)
- 1-3 bugs/KLoC for production-quality code: ***11-36 bugs***
  - Best case I have seen: 0.04 bugs/KLoC for safety-critical code
    - Extreme code-style restrictions, single-threaded, formal methods, …
    - And still way more than zero bugs!!! :-)
- Median age of a line of RCU code is less than four years
  - And young code tends to be buggier than old code!
- We should therefore expect a few tens more bugs in RCU!

RCU Validation

Current RCU Regression Testing

Current RCU Regression Testing

- Stress-test suite: “rcutorture”
  - http://lwn.net/Articles/154107/, http://lwn.net/Articles/622404/
- “Intelligent fuzz testing”: “trinity”
  - http://codemonkey.org.uk/projects/trinity/
- Test suite including static analysis: “0-day test robot”
  - https://lwn.net/Articles/514278/
- Integration testing: “linux-next tree”
  - https://lwn.net/Articles/571980/
- Above is old technology – but quite effective
  - 2010: wait for -rc3 or -rc4. 2013: Usually no problems with -rc1

Current RCU Regression Testing

- Stress-test suite: “rcutorture”
  - http://lwn.net/Articles/154107/, http://lwn.net/Articles/622404/
- “Intelligent fuzz testing”: “trinity”
  - http://codemonkey.org.uk/projects/trinity/
- Test suite including static analysis: “0-day test robot”
  - https://lwn.net/Articles/514278/
- Integration testing: “linux-next tree”
  - https://lwn.net/Articles/571980/
- Above is old technology – but quite effective
  - 2010: wait for -rc3 or -rc4. 2013: Usually no problems with -rc1
- Formal verification in design, but not in regression testing
  - http://lwn.net/Articles/243851/, https://lwn.net/Articles/470681/, https://lwn.net/Articles/608550/

January 30, 2017 rcutorture Output

```
tools/testing/selftests/rcutorture/bin/kvm.sh --cpus 50 --duration 1800
SRCU-N ------- 610414 grace periods (5.65198 per second)
SRCU-P ------- 13349 grace periods (0.123602 per second)
TASKS01 ------- 70971 grace periods (0.657139 per second)
TASKS02 ------- 70238 grace periods (0.650352 per second)
TASKS03 ------- 69972 grace periods (0.647889 per second)
TINY01 ------- 8152793 grace periods (75.4888 per second)
TINY02 ------- 17916244 grace periods (165.891 per second)
TREE01 ------- 4376468 grace periods (40.5229 per second)
TREE02 ------- 3034531 grace periods (28.0975 per second)
TREE03 ------- 1048736 grace periods (9.71052 per second)
TREE04 ------- 637788 grace periods (5.90544 per second)
TREE05 ------- 2415024 grace periods (22.3613 per second)
TREE06 ------- 1791390 grace periods (16.5869 per second)
TREE07 ------- 551532 grace periods (5.10678 per second)
TREE08 ------- 1072103 grace periods (9.92688 per second)
TREE09 ------- 7543572 grace periods (69.8479 per second)
```

January 30, 2017 rcutorture Output

```
tools/testing/selftests/rcutorture/bin/kvm.sh --cpus 50 --duration 1800
SRCU-N ------- 610414 grace periods (5.65198 per second)
SRCU-P ------- 13349 grace periods (0.123602 per second)
TASKS01 ------- 70971 grace periods (0.657139 per second)
TASKS02 ------- 70238 grace periods (0.650352 per second)
TASKS03 ------- 69972 grace periods (0.647889 per second)
TINY01 ------- 8152793 grace periods (75.4888 per second)
TINY02 ------- 17916244 grace periods (165.891 per second)
TREE01 ------- 4376468 grace periods (40.5229 per second)
TREE02 ------- 3034531 grace periods (28.0975 per second)
TREE03 ------- 1048736 grace periods (9.71052 per second)
TREE04 ------- 637788 grace periods (5.90544 per second)
TREE05 ------- 2415024 grace periods (22.3613 per second)
TREE06 ------- 1791390 grace periods (16.5869 per second)
TREE07 ------- 551532 grace periods (5.10678 per second)
TREE08 ------- 1072103 grace periods (9.92688 per second)
TREE09 ------- 7543572 grace periods (69.8479 per second)
```

There are bugs in RCU, and 30 hours of rcutorture failed to find them

January 30, 2017 rcutorture Output

```
tools/testing/selftests/rcutorture/bin/kvm.sh --cpus 50 --duration 1800
SRCU-N ------- 610414 grace periods (5.65198 per second)
SRCU-P ------- 13349 grace periods (0.123602 per second)
TASKS01 ------- 70971 grace periods (0.657139 per second)
TASKS02 ------- 70238 grace periods (0.650352 per second)
TASKS03 ------- 69972 grace periods (0.647889 per second)
TINY01 ------- 8152793 grace periods (75.4888 per second)
TINY02 ------- 17916244 grace periods (165.891 per second)
TREE01 ------- 4376468 grace periods (40.5229 per second)
TREE02 ------- 3034531 grace periods (28.0975 per second)
TREE03 ------- 1048736 grace periods (9.71052 per second)
TREE04 ------- 637788 grace periods (5.90544 per second)
TREE05 ------- 2415024 grace periods (22.3613 per second)
TREE06 ------- 1791390 grace periods (16.5869 per second)
TREE07 ------- 551532 grace periods (5.10678 per second)
TREE08 ------- 1072103 grace periods (9.92688 per second)
TREE09 ------- 7543572 grace periods (69.8479 per second)
```

There are bugs in RCU, and 30 hours of rcutorture failed to find them

This constitutes a critical bug in rcutorture

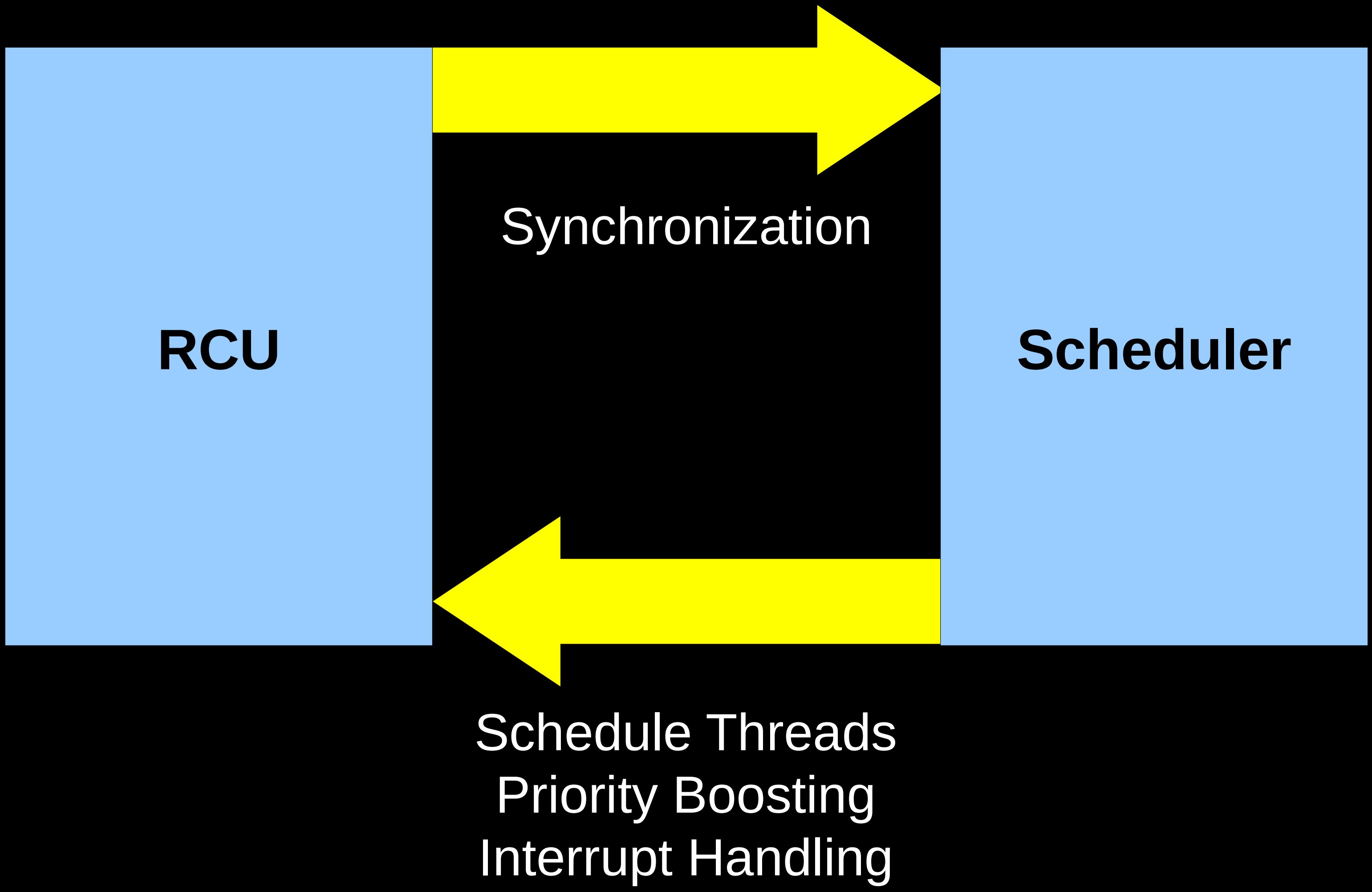
January 30, 2017 rcutorture Output

```
tools/testing/selftests/rcutorture/bin/kvm.sh --cpus 50 --duration 1800
SRCU-N ------- 610414 grace periods (5.65198 per second)
SRCU-P ------- 13349 grace periods (0.123602 per second)
TASKS01 ------- 70971 grace periods (0.657139 per second)
TASKS02 ------- 70238 grace periods (0.650352 per second)
TASKS03 ------- 69972 grace periods (0.647889 per second)
TINY01 ------- 8152793 grace periods (75.4888 per second)
TINY02 ------- 17916244 grace periods (165.891 per second)
TREE01 ------- 4376468 grace periods (40.5229 per second)
TREE02 ------- 3034531 grace periods (28.0975 per second)
TREE03 ------- 1048736 grace periods (9.71052 per second)
TREE04 ------- 637788 grace periods (5.90544 per second)
TREE05 ------- 2415024 grace periods (22.3613 per second)
TREE06 ------- 1791390 grace periods (16.5869 per second)
TREE07 ------- 551532 grace periods (5.10678 per second)
TREE08 ------- 1072103 grace periods (9.92688 per second)
TREE09 ------- 7543572 grace periods (69.8479 per second)
```

There are bugs in RCU, and 30 hours of rcutorture failed to find them
This constitutes a critical bug in rcutorture
On the other hand, first time in over a year that I have see this!

How Well Does Linux-Kernel Testing Really Work?

Example 1: RCU-Scheduler Mutual Dependency



So, What Was The Problem?

- Found during testing of Linux kernel v3.0-rc7:
  - RCU read-side critical section is preempted for an extended period
  - RCU priority boosting is brought to bear
  - RCU read-side critical section ends, notes need for special processing
  - Interrupt invokes handler, then starts softirq processing
  - Scheduler invoked to wake ksoftirqd kernel thread:
    - Acquires runqueue lock and enters RCU read-side critical section
    - Leaves RCU read-side critical section, notes need for special processing
    - Because in_irq() returns false, special processing attempts deboosting
    - Which causes the scheduler to acquire the runqueue lock
    - Which results in self-deadlock
  - (See http://lwn.net/Articles/453002/ for more details.)
- Fix: Add separate “exiting read-side critical section” state
  - Also validated my creation of correct patches – without testing!

Note: Remains a bug even under SC

Example 1: Bug Was Located By Normal Testing

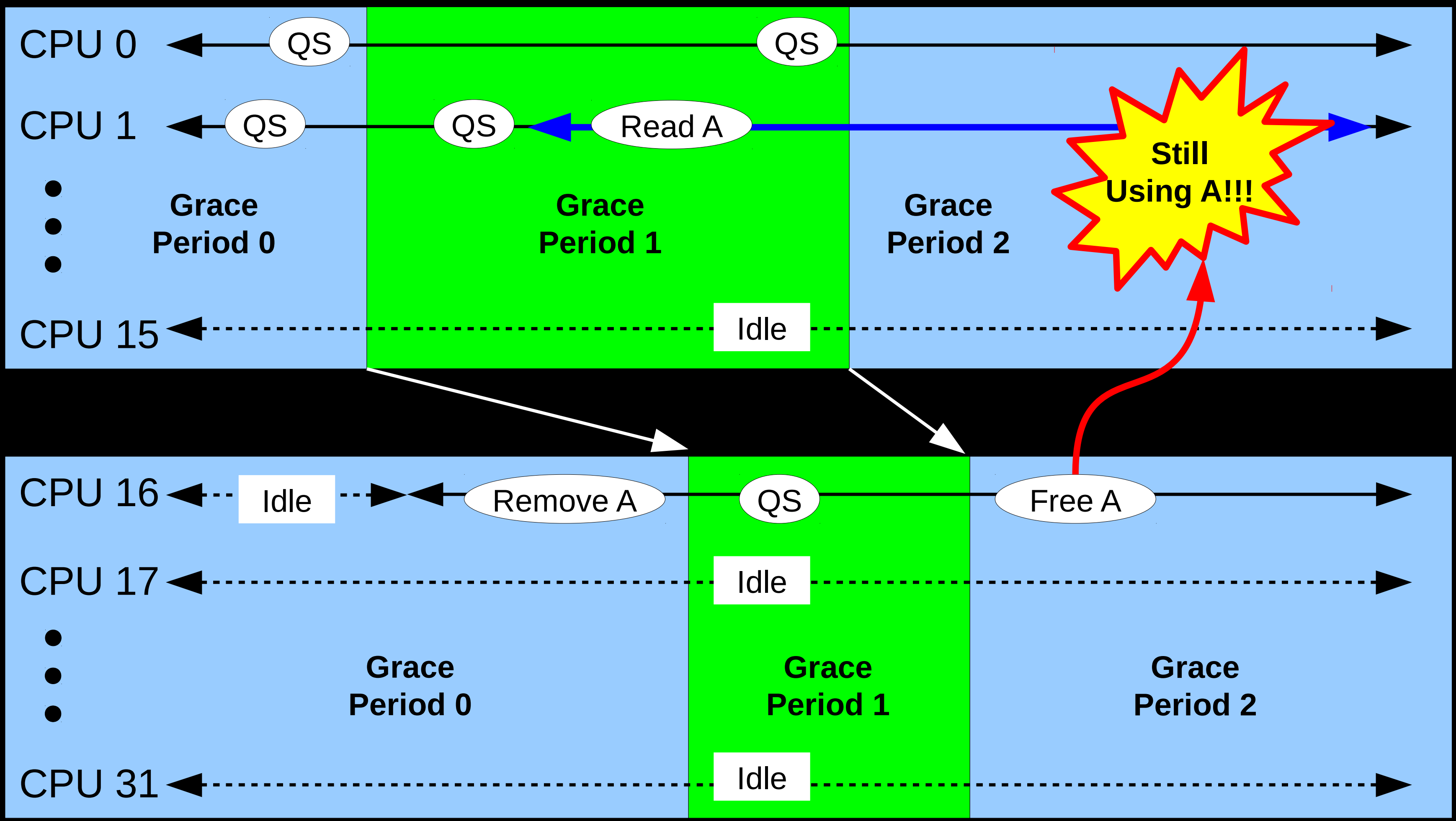
IBM

Example 2: Grace Period Cleanup/Initialization Bug

1. CPU 0 completes grace period, starts new one, cleaning up and initializing up through first leaf rcu_node structure
2. CPU 1 passes through quiescent state (new grace period!)
3. CPU 1 does rcu_read_lock() and acquires reference to A
4. CPU 16 exits dyntick-idle mode (back on ***old*** grace period)
5. CPU 16 removes A, passes it to call_rcu()
6. CPU 16 associates callback with next grace period
7. CPU 0 completes cleanup/initialization of rcu_node structures
8. CPU 16 callback associated with now-current grace period
9. All remaining CPUs pass through quiescent states
10. Last CPU performs cleanup on all rcu_node structures
11. CPU 16 notices end of grace period, advances callback to “done” state
12. CPU 16 invokes callback, freeing A (***too bad CPU 1 is still using it***)

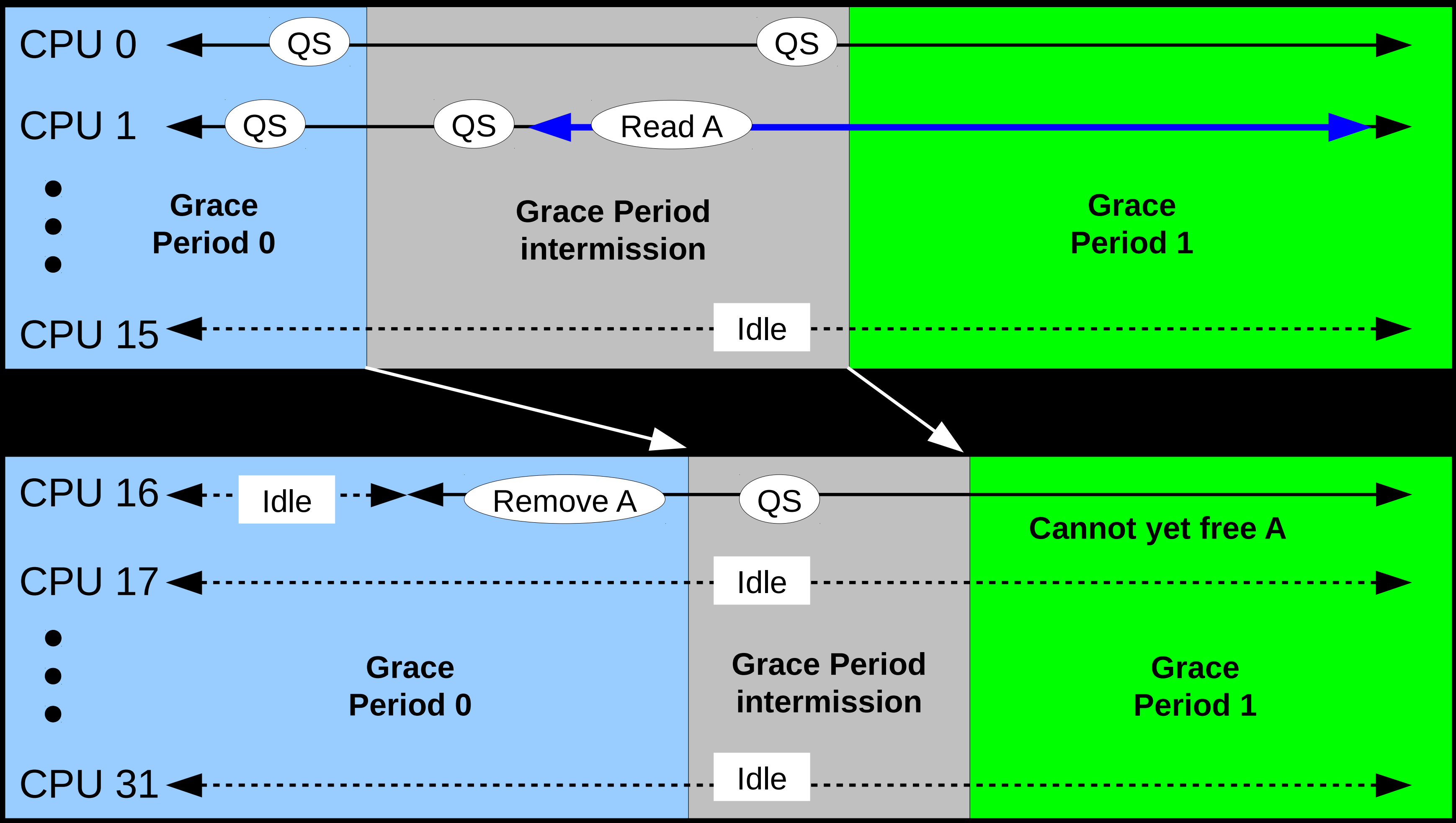
Not found via Linux-kernel validation: In production for 5 years!

Example 2: Grace Period Cleanup/Initialization Bug



Note: Remains a bug even under SC

Example 2: Grace Period Cleanup/Initialization Fix



All agree that grace period 1 starts after grace period 0 ends

Example 1 & Example 2 Results

- Example 1: Bug was located by normal Linux test procedures
- Example 2: Bug was missed by normal Linux test procedures
  - Not found via Linux-kernel validation: In production for 5 years!
  - On systems with up to 4096 CPUs...
- Both are bugs even under sequential consistency
- Normal testing is not bad, but improvement is needed
- Can Linux-kernel RCU validation do better?

Example 1 & Example 2 Results

- Example 1: Bug was located by normal Linux test procedures
- Example 2: Bug was missed by normal Linux test procedures
  - Not found via Linux-kernel validation: In production for 5 years!
  - On systems with up to 4096 CPUs...
- Both are bugs even under sequential consistency
- Normal testing is not bad, but improvement is needed
- Can Linux-kernel RCU validation do better?
- But first, what is the validation problem that must be solved?

More Than 1.5 Billion Linux Instances Running!!!

More Than 1.5 Billion Linux Instances Running!!!
Woo-Hoo!!! Linux Has Won!!!

More Than 1.5 Billion Linux Instances Running!!!
Woo-Hoo!!! Linux Has Won!!!

But How The #@$&! Do I Validate RCU For This???

How The #@$&! Do I Validate RCU For This???

- A race condition that occurs once in a million years happens ***several times per day*** across the installed base
  - I am very proud of rcutorture, but it simply cannot detect million-year races when running on a reasonable test setup

How The #@$&! Do I Validate RCU For This???

- A race condition that occurs once in a million years happens ***several times per day*** across the installed base
  - I am very proud of rcutorture, but it simply cannot detect million-year races when running on a reasonable test setup
  - Even given expected improvements in rcutorture

How The #@$&! Do I Validate RCU For This???

- A race condition that occurs once in a million years happens ***several times per day*** across the installed base
  - I am very proud of rcutorture, but it simply cannot detect million-year races when running on a reasonable test setup
  - Even given expected improvements in rcutorture
  - Even with help from mutation testing
    - Groce et al., “How Verified is My Code? Falsification-Driven Verification” https://www.cs.cmu.edu/~agroce/ase15.pdf

RCU Validation Options?

- Other failures mask RCU's, including hardware failures
  - I know of no human artifact with a million-year MTBF
  - But I do know of Linux uses that put the public's safety at risk...
- Increasing CPUs on test system increases race probability
- Rare critical operations forced to happen more frequently
- Knowledge of possible race conditions allows targeted tests
  - Plus other dirty tricks from 25 years of testing concurrent software
  - Provide harsh environment to force software to evolve quickly
- Formal verification used for some aspects of RCU design

RCU Validation Options?

- Other failures mask RCU's, including hardware failures
  - I know of no human artifact with a million-year MTBF
  - But I do know of Linux uses that put the public's safety at risk...
- Increasing CPUs on test system increases race probability
- Rare critical operations forced to happen more frequently
- Knowledge of possible race conditions allows targeted tests
  - Plus other dirty tricks from 25 years of testing concurrent software
  - Provide harsh environment to force software to evolve quickly
- Formal verification used for some aspects of RCU design

- Should I use formal verification in RCU's regression testing?

Formal Verification and Regression Testing: Requirements

Formal Verification and Regression Testing: Requirements

(1)Either automatic translation or no translation required

– Automatic discarding of irrelevant portions of the code
– Manual translation provides opportunity for human error

Formal Verification and Regression Testing: Requirements

(1) Either automatic translation or no translation required

(2) Correctly handle environment, including memory model
- The QRCU validation benchmark is an excellent cautionary tale

Formal Verification and Regression Testing: Requirements

(1) Either automatic translation or no translation required

(2) Correctly handle environment, including memory model

(3) Reasonable memory and CPU overhead

- Bugs must be located in practice as well as in theory
- Linux-kernel RCU is 15KLoC and release cycles are short

Formal Verification and Regression Testing: Requirements

(1) Either automatic translation or no translation required

(2) Correctly handle environment, including memory model

(3) Reasonable memory and CPU overhead

(4) Map to source code line(s) containing the bug
  - "Something is wrong somewhere" is not a helpful diagnostic: I **know** bugs exist

Formal Verification and Regression Testing: Requirements

(1) Either automatic translation or no translation required

(2) Correctly handle environment, including memory model

(3) Reasonable memory and CPU overhead

(4) Map to source code line(s) containing the bug

(5) Modest input outside of source code under test
- Preferably glean much of the specification from the source code itself (empirical spec!)
- Specifications are software and can have their own bugs

Formal Verification and Regression Testing: Requirements

(1) Either automatic translation or no translation required

(2) Correctly handle environment, including memory model

(3) Reasonable memory and CPU overhead

(4) Map to source code line(s) containing the bug

(5) Modest input outside of source code under test

(6) Find relevant bugs
- Low false-positive rate, weight towards likelihood of occurrence (fixes create bugs!)

Formal Verification and Regression Testing: Requirements

(1)Either automatic translation or no translation required
- Automatic discarding of irrelevant portions of the code
- Manual translation provides opportunity for human error

(2)Correctly handle environment, including memory model
- The QRCU validation benchmark is an excellent cautionary tale

(3)Reasonable memory and CPU overhead
- Bugs must be located in practice as well as in theory
- Linux-kernel RCU is 15KLoC and release cycles are short

(4)Map to source code line(s) containing the bug
- “Something is wrong somewhere” is not a helpful diagnostic: I **know** bugs exist

(5)Modest input outside of source code under test
- Preferably glean much of the specification from the source code itself (empirical spec!)
- Specifications are software and can have their own bugs

(6)Find relevant bugs
- Low false-positive rate, weight towards likelihood of occurrence (fixes create bugs!)

Formal Validation Tools Used and Regression Testing

- Promela and Spin
  - Holzmann: “The Spin Model Checker”
  - I have used Promela/Spin in design for more than 20 years, but:
    - Limited problem size, long run times, large memory consumption
    - Does not implement memory models (assumes sequential consistency)
    - Special language, difficult to translate from C
- ARMMEM and PPCMEM (2)
  - Alglave, Maranget, Pawan, Sarkar, Sewell, Williams, Nardelli: “PPCMEM/ARMMEM: A Tool for Exploring the POWER and ARM Memory Models”
    - Very limited problem size, long run times, large memory consumption
    - Restricted pseudo-assembly language, manual translation required
- Herd (2, 3)
  - Alglave, Maranget, and Tautschnig: “Herding Cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory”
    - Very limited problem size (but much improved run times and memory consumption)
    - Restricted pseudo-assembly language, manual translation required

Useful, but not for regression testing

C Bounded Model Checker (CBMC)

- Nascent concurrency and weak-memory functionality
- Valuable property: “Just enough specification”
  - Assertions in code act as specifications!
  - Can provide additional specifications in “verification driver” code
- Verified rcu_dereference() and rcu_assign_pointer()
  - Daniel Kroening, Oxford
- Verified Tiny RCU
  - http://paulmck.livejournal.com/39343.html
- Verified substantial portion of Tree RCU
  - Lihao Liang, Oxford: https://arxiv.org/abs/1610.03052
- Added Lance Roy's CBMC SRCU verification to rcutorture

Kroening, Clarke, and Lerda, “A tool for checking ANSI-C programs”, *Tools and Algorithms for the Construction and Analysis of Systems, 2004,* pp. 168-176.

C Bounded Model Checker (CBMC): Usage

- C Bounded Model Checker (CBMC) applies long-standing hardware verification techniques to software
- Easy to use: Given recent Debian-derived distributions:

```
sudo apt-get install cbmc
```

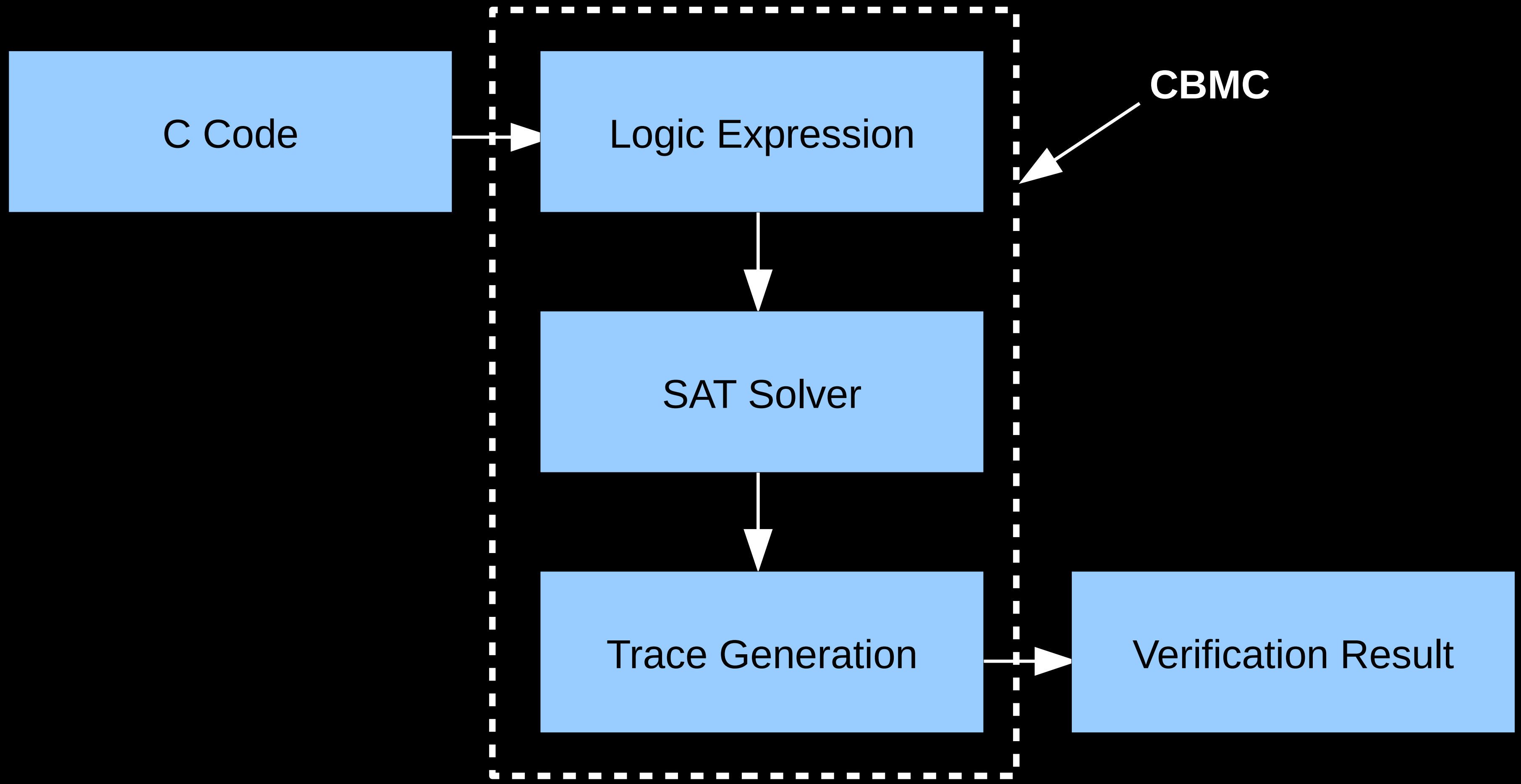
```
cbmc filename.c
```

- If no combination of inputs can trigger an assertion or cause an array-out-of-bounds error, it prints:

```
VERIFICATION SUCCESSFUL
```

- And since 2015, CBMC handles concurrency!!!

How Does CBMC Work?



Scorecard For Linux-Kernel C Code (Incomplete)

	Promela	PPCMEM	Herd	CBMC
(1) Automated				
(2) Handle environment	(MM)		(MM)	(MM)
(3) Low overhead				SAT?
(4) Map to source code				
(5) Modest input				
(6) Relevant bugs	???	???	???	???
Paul McKenney's first use	1993	2011	2014	2015

Promela MM: Only SC: Weak memory must be implemented in model
Herd MM: Some PowerPC and ARM corner-case issues
CBMC MM: Only SC and TSO
Note: All four handle concurrency! (Promela has done so for 25 years!!!)

Scorecard For Linux-Kernel C Code

	Promela	PPCMEM	Herd	CBMC	Test
(1) Automated					
(2) Handle environment	(MM)		(MM)	(MM)	
(3) Low overhead				SAT?	
(4) Map to source code					
(5) Modest input					
(6) Relevant bugs	???	???	???	???	
Paul McKenney's first use	1993	2011	2014	2015	1973

So why do anything other than testing?

Scorecard For Linux-Kernel C Code

	Promela	PPCMEM	Herd	CBMC	Test
(1) Automated					
(2) Handle environment	(MM)		(MM)	(MM)	
(3) Low overhead				SAT?	
(4) Map to source code					
(5) Modest input					
(6) Relevant bugs	???	???	???	???	
Paul McKenney's first use	1993	2011	2014	2015	1973

So why do anything other than testing?

- Low-probability bugs can require expensive testing regimen
- Large installed base will encounter low-probability bugs
- Safety-criitcal applications are sensitive to low-probability bugs

Other Possible Approaches

- By-hand formalizations and proofs
  - Stern: Semi-formal proof of URCU (2012 IEEE TPDS)
  - Gotsman: Separation-logic RCU semantics (2013 ESOP)
  - Tasserotti et al.: Formal proof of URCU linked list: (2015 PLDI)
  - Excellent work, but not useful for regression testing
- seL4 tooling: Lacks support for concurrency and RCU idioms
  - Might be applicable to Tiny RCU callback handling
  - Impressive work nevertheless!!!
- Apply Peter O'Hearn's Infer to the Linux kernel
- Nidhugg: Work by Michalis Kokologiannakis and Kostis Sagonas
  - https://github.com/michalis-/rcu/blob/master/rcupaper.pdf
  - Appears to be more scalable than CBMC, but some restrictions
  - Nevertheless, Nidhugg finds all my injected bugs

Summary and Challenges

Summary

- RCU's specification is empirical
- RCU's implementation is unlikely to be bug-free, reliable though it might be
- Currently relying on stress testing augmented by mutation analysis, adding formal verification

Summary

- RCU's specification is empirical
- RCU's implementation is unlikely to be bug-free, reliable though it might be
- Currently relying on stress testing augmented by mutation analysis, adding formal verification
  - Formal verification currently weak on forward-progress guarantees
  - And has not yet found any RCU bugs that I didn't already know about
  - But RCU validation is difficult, so I am throwing everything I can at it!!!

Challenges

- Find bug in rcu_preempt_offline_tasks()
  - Note: No practical impact because this function has been removed
  - http://paulmck.livejournal.com/37782.html
- Find bug in RCU_NO_HZ_FULL_SYSIDLE
  - http://paulmck.livejournal.com/38016.html
- Find bug in RCU linked-list use cases
  - http://paulmck.livejournal.com/39793.html
- Find lost wakeup bug in the Linux kernel (or maybe qemu)
  - Heavy rcutorture testing with CPU hotplug on two-socket system
  - Detailed repeat-by: https://lkml.org/lkml/2016/3/28/214
  - Can you find this before we do? (Sorry, too late!!!)
- Find any other bug in popular open-source software
  - A verification researcher has provoked a SEGV in Linux-kernel RCU

More Challenges (AKA Current Limitations)

- Incorporate Linux-kernel memory model into analysis
  - And/or the ARM and PowerPC memory models
- Detect race conditions leading to deadlocks and hangs
  - CBMC and Nidhugg can detect unconditional deadlocks and hangs
- Analyze bugs involving networking and mass storage
- Use induction techniques to fully analyze indefinite recursion and unbounded looping
  - Spinloops should be easy: Yes, there are halting-problem limitations
- Analyze larger programs: RCU is not exactly huge!!!
  - Automatically decompose large programs and combine results?

To Probe Deeper (RCU)

- https://queue.acm.org/detail.cfm?id=2488549
  - "Structured Deferral: Synchronization via Procrastination" (also in July 2013 CACM)
- http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159 and http://www.computer.org/cms/Computer.org/dl/trans/td/2012/02/extras/ttd2012020375s.pdf
  - "User-Level Implementations of Read-Copy Update"
- git://lttng.org/userspace-rcu.git (User-space RCU git tree)
- http://people.csail.mit.edu/nickolai/papers/clements-bonsai.pdf
  - Applying RCU and weighted-balance tree to Linux mmap_sem.
- http://www.usenix.org/event/atc11/tech/final_files/Triplett.pdf
  - RCU-protected resizable hash tables, both in kernel and user space
- http://www.usenix.org/event/hotpar11/tech/final_files/Howard.pdf
  - Combining RCU and software transactional memory
- http://wiki.cs.pdx.edu/rp/: Relativistic programming, a generalization of RCU
- http://lwn.net/Articles/262464/, http://lwn.net/Articles/263130/, http://lwn.net/Articles/264090/
  - "What is RCU?" Series
- 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://read.seas.harvard.edu/cs261/2011/rcu.html
  - Harvard University class notes on RCU (Courtesy of Eddie Koher)
- http://www.rdrop.com/users/paulmck/RCU/ (More RCU information)

To Probe Deeper (1/5)

- Hash tables:
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e1.html Chapter 10
- Split counters:
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Chapter 5
  - http://events.linuxfoundation.org/sites/events/files/slides/BareMetal.2014.03.09a.pdf
- Perfect partitioning
  - Candide et al: “Dynamo: Amazon's highly available key-value store”
    - http://doi.acm.org/10.1145/1323293.1294281
  - McKenney: “Is Parallel Programming Hard, And, If So, What Can You Do About It?”
    - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Section 6.5
  - McKenney: “Retrofitted Parallelism Considered Grossly Suboptimal”
    - Embarrassing parallelism vs. humiliating parallelism
    - https://www.usenix.org/conference/hotpar12/retro%EF%AC%81tted-parallelism-considered-grossly-sub-optimal
  - McKenney et al: “Experience With an Efficient Parallel Kernel Memory Allocator”
    - http://www.rdrop.com/users/paulmck/scalability/paper/mpalloc.pdf
  - Bonwick et al: “Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources”
    - http://static.usenix.org/event/usenix01/full_papers/bonwick/bonwick_html/
  - Turner et al: “PerCPU Atomics”
    - http://www.linuxplumbersconf.org/2013/ocw//system/presentations/1695/original/LPC%20-%20PerCpu%20Atomics.pdf

To Probe Deeper (2/5)

- Stream-based applications:
  - Sutton: “Concurrent Programming With The Disruptor”
    - http://www.youtube.com/watch?v=UvE389P6Er4
    - http://lca2013.linux.org.au/schedule/30168/view_talk
  - Thompson: “Mechanical Sympathy”
    - http://mechanical-sympathy.blogspot.com/
- Read-only traversal to update location
  - Arcangeli et al: “Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel”
    - https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli_html/index.html
  - Corbet: “Dcache scalability and RCU-walk”
    - https://lwn.net/Articles/419811/
  - Xu: “bridge: Add core IGMP snooping support”
    - http://kerneltrap.com/mailarchive/linux-netdev/2010/2/26/6270589
  - Triplett et al., “Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming”
    - http://www.usenix.org/event/atc11/tech/final_files/Triplett.pdf
  - Howard: “A Relativistic Enhancement to Software Transactional Memory”
    - http://www.usenix.org/event/hotpar11/tech/final_files/Howard.pdf
  - McKenney et al: “URCU-Protected Hash Tables”
    - http://lwn.net/Articles/573431/

To Probe Deeper (3/5)

- Hardware lock elision: Overviews
  - Kleen: “Scaling Existing Lock-based Applications with Lock Elision”
    - http://queue.acm.org/detail.cfm?id=2579227
- Hardware lock elision: Hardware description
  - POWER ISA Version 2.07
    - http://www.power.org/documentation/power-isa-version-2-07/
  - Intel® 64 and IA-32 Architectures Software Developer Manuals
    - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
  - Jacobi et al: “Transactional Memory Architecture and Implementation for IBM System z”
    - http://www.microsymposia.org/micro45/talks-posters/3-jacobi-presentation.pdf
- Hardware lock elision: Evaluations
  - http://pcl.intel-research.net/publications/SC13-TSX.pdf
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Section 16.3
- Hardware lock elision: Need for weak atomicity
  - Herlihy et al: “Software Transactional Memory for Dynamic-Sized Data Structures”
    - http://research.sun.com/scalable/pubs/PODC03.pdf
  - Shavit et al: “Data structures in the multicore age”
    - http://doi.acm.org/10.1145/1897852.1897873
  - Haas et al: “How FIFO is your FIFO queue?”
    - http://dl.acm.org/citation.cfm?id=2414731
  - Gramoli et al: “Democratizing transactional programming”
    - http://doi.acm.org/10.1145/2541883.2541900

To Probe Deeper (4/5)

- RCU
  - Desnoyers et al.: “User-Level Implementations of Read-Copy Update”
    - http://www.rdrop.com/users/paulmck/RCU/urcu-main-accepted.2011.08.30a.pdf
    - http://www.computer.org/cms/Computer.org/dl/trans/td/2012/02/extras/ttd2012020375s.pdf
  - McKenney et al.: “RCU Usage In the Linux Kernel: One Decade Later”
    - http://rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf
    - http://rdrop.com/users/paulmck/techreports/RCUUsage.2013.02.24a.pdf
  - McKenney: “Structured deferral: synchronization via procrastination”
    - http://doi.acm.org/10.1145/2483852.2483867
  - McKenney et al.: “User-space RCU” https://lwn.net/Articles/573424/
- Possible future additions
  - Boyd-Wickizer: “Optimizing Communications Bottlenecks in Multiprocessor Operating Systems Kernels”
    - http://pdos.csail.mit.edu/papers/sbw-phd-thesis.pdf
  - Clements et al: “The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors”
    - http://www.read.seas.harvard.edu/~kohler/pubs/clements13scalable.pdf
  - McKenney: “N4037: Non-Transactional Implementation of Atomic Tree Move”
    - http://www.rdrop.com/users/paulmck/scalability/paper/AtomicTreeMove.2014.05.26a.pdf
  - McKenney: “C++ Memory Model Meets High-Update-Rate Data Structures”
    - http://www2.rdrop.com/users/paulmck/RCU/C++Updates.2014.09.11a.pdf

To Probe Deeper (5/5)

- RCU theory and semantics, academic contributions (partial list)
  - Gamsa et al., “Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System”
    - http://www.usenix.org/events/osdi99/full_papers/gamsa/gamsa.pdf
  - McKenney, “Exploiting Deferred Destruction: An Analysis of RCU Techniques”
    - http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf
  - Hart, “Applying Lock-free Techniques to the Linux Kernel”
    - http://www.cs.toronto.edu/~tomhart/masters_thesis.html
  - Olsson et al., “TRASH: A dynamic LC-trie and hash data structure”
    - http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4281239
  - Desnoyers, “Low-Impact Operating System Tracing”
    - http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf
  - Dalton, “The Design and Implementation of Dynamic Information Flow Tracking ...”
    - http://csl.stanford.edu/~christos/publications/2009.michael_dalton.phd_thesis.pdf
  - Gotsman et al., “Verifying Highly Concurrent Algorithms with Grace (extended version)”
    - http://software.imdea.org/~gotsman/papers/recycling-esop13-ext.pdf
  - Liu et al., “Mindicators: A Scalable Approach to Quiescence”
    - http://dx.doi.org/10.1109/ICDCS.2013.39
  - Tu et al., “Speedy Transactions in Multicore In-memory Databases”
    - http://doi.acm.org/10.1145/2517349.2522713
  - Arbel et al., “Concurrent Updates with RCU: Search Tree as an Example”
    - http://www.cs.technion.ac.il/~mayaarl/podc047f.pdf

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.

Questions?

BACKUP

Promela/spin: Design-Time Verification

- 1993: Shared-disk/network election algorithm (pre-Linux)
  - Hadn't figured out bug injection: Way too trusting!!!
  - Single-point-of failure bug in specification: Fixed during coding
    - But fix had bug that propagated to field: Cluster partition
  - **Conclusion**: Formal verification is trickier than expected!!!
- 2007: RCU idle-detection energy-efficiency logic
  - (http://lwn.net/Articles/243851/)
  - Verified, but much simpler approach found two years later
  - **Conclusion**: The need for formal verification is a symptom of a too-complex design
- 2012: Verify userspace RCU, emulating weak memory
  - Two independent models (Desnoyers and myself), **bug injection**
- 2014: NMIs can nest!!! Affects energy-efficiency logic
  - Verified Andy's code, and no simpler approach apparent thus far!!!
  - Note: Excellent example of **empirical specification**

Promela Model of Incorrect Atomic Increment (1/2)

```
#define NUMPROCS 2

byte counter = 0;
byte progress[NUMPROCS];

proctype incrementer(byte me)
{
  int temp;

  temp = counter;
  counter = temp + 1;
  progress[me] = 1;
}
```

Promela Model of Incorrect Atomic Increment (2/2)

```
init {
  int i = 0;
  int sum = 0;

  atomic {
    i = 0;
    do
    :: i < NUMPROCS ->
      progress[i] = 0;
      run incrementer(i);
      i++
    :: i >= NUMPROCS -> break
    od;
  }
  atomic {
    i = 0;
    sum = 0;
    do
    :: i < NUMPROCS ->
      sum = sum + progress[i];
      i++
    :: i >= NUMPROCS -> break
    od;
    assert(sum < NUMPROCS || counter == NUMPROCS)
  }
}
```

PPCMEM and Herd

- Verified suspected bug in Power Linux atomic primitives
- Found bug in Power Linux spin_unlock_wait()
- Verified ordering properties of locking primitives
- Excellent memory-ordering teaching tools
  - Starting to be used more widely within IBM as a design-time tool
- PPCMEM: (http://lwn.net/Articles/470681/)
  - Accurate but slow
- Herd: (http://lwn.net/Articles/608550/)
  - Faster, but some correctness issues with RMW atomics and lwsync
  - Work in progress: Formalize Linux-kernel memory model
    - With Alglave, Maranget, Parri, and Stern, plus lots of architects
    - Hopefully will feed into improved tooling

Alglave, Maranget, Pawan, Sarkar, Sewell, Williams, Nardelli:
“PPCMEM/ARMMEM: A Tool for Exploring the POWER and ARM Memory Models”
Alglave, Maranget, and Tautschnig: “Herding Cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory”

PPCMEM Example Litmus Test for IRIW

```
PPC IRIW.litmus
""
(* Traditional IRIW. *)
{
0:r1=1; 0:r2=x;
1:r1=1;         1:r4=y;
2:      2:r2=x; 2:r4=y;
3:      3:r2=x; 3:r4=y;
}
 P0            | P1            | P2                   | P3                   ;
 stw r1,0(r2)  | stw r1,0(r4)  | lwz r3,0(r2)         | lwz r3,0(r4)         ;
               |               | sync                 | sync                 ;
               |               | lwz r5,0(r4)         | lwz r5,0(r2)         ;

exists
(2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)
```

Fourteen CPU hours and 10 GB of memory

Herd Example Litmus Test for Incorrect IRIW

```
PPC IRIW-lwsync-f.litmus
""
(* Traditional IRIW. *)
{
0:r1=1; 0:r2=x;
1:r1=1;         1:r4=y;
2:      2:r2=x; 2:r4=y;
3:      3:r2=x; 3:r4=y;
}
 P0            | P1            | P2                    | P3                    ;
 stw r1,0(r2)  | stw r1,0(r4)  | lwz r3,0(r2)          | lwz r3,0(r4)          ;
               |               | lwsync                | lwsync                ;
               |               | lwz r5,0(r4)          | lwz r5,0(r2)          ;

exists
(2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)

. . .

Positive: 1 Negative: 15
Condition exists (2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)
Observation IRIW Sometimes 1 15
```

What Exactly is a Relevant Bug???

- Suppose RCU has 19 million-year bugs and one 10-year bug
  - Suppose tool finds all 19 million-year bugs, but misses the 10-year bug
  - Further suppose I fix all 19 bugs located by the tool
  - What is the effect on RCU robustness?

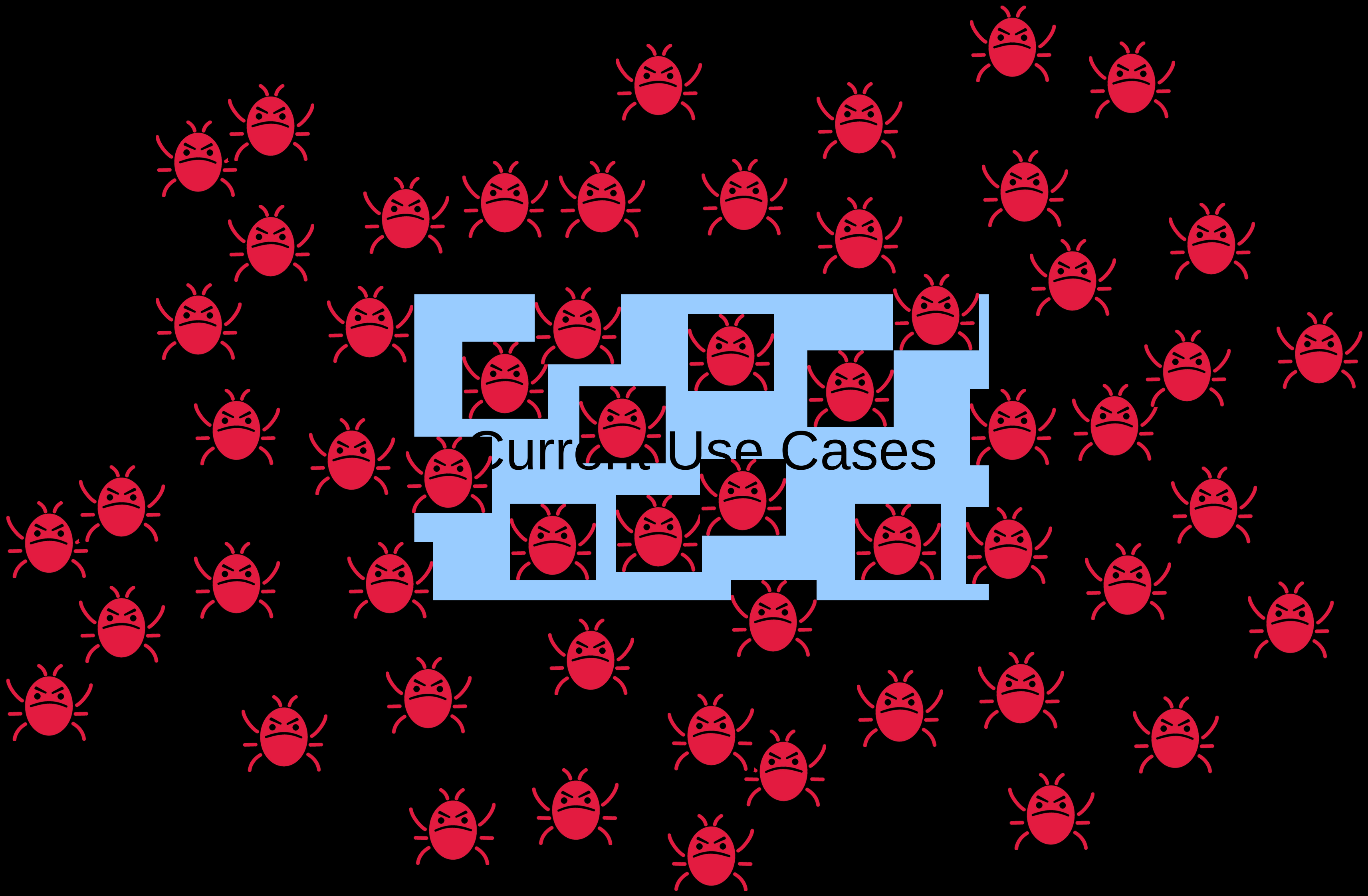
What Exactly is a Relevant Bug???

- Suppose RCU has 19 million-year bugs and one 10-year bug
  - Suppose tool finds all 19 million-year bugs, but misses the 10-year bug
  - Further suppose I fix all 19 bugs located by the tool
  - What is the effect on RCU robustness?
- Negligible net improvement from the 19 fixes
  - And possible large degradation from these fixes
  - Statistically, one in every six fixes injects a new bug!
- Of course both severity and frequency are important
  - Loss of time, loss of money, loss of accuracy, loss of life, ...
  - But be careful – refusing to fix “minor” bugs can build a wall of bugs preventing your code from being adopted for new uses

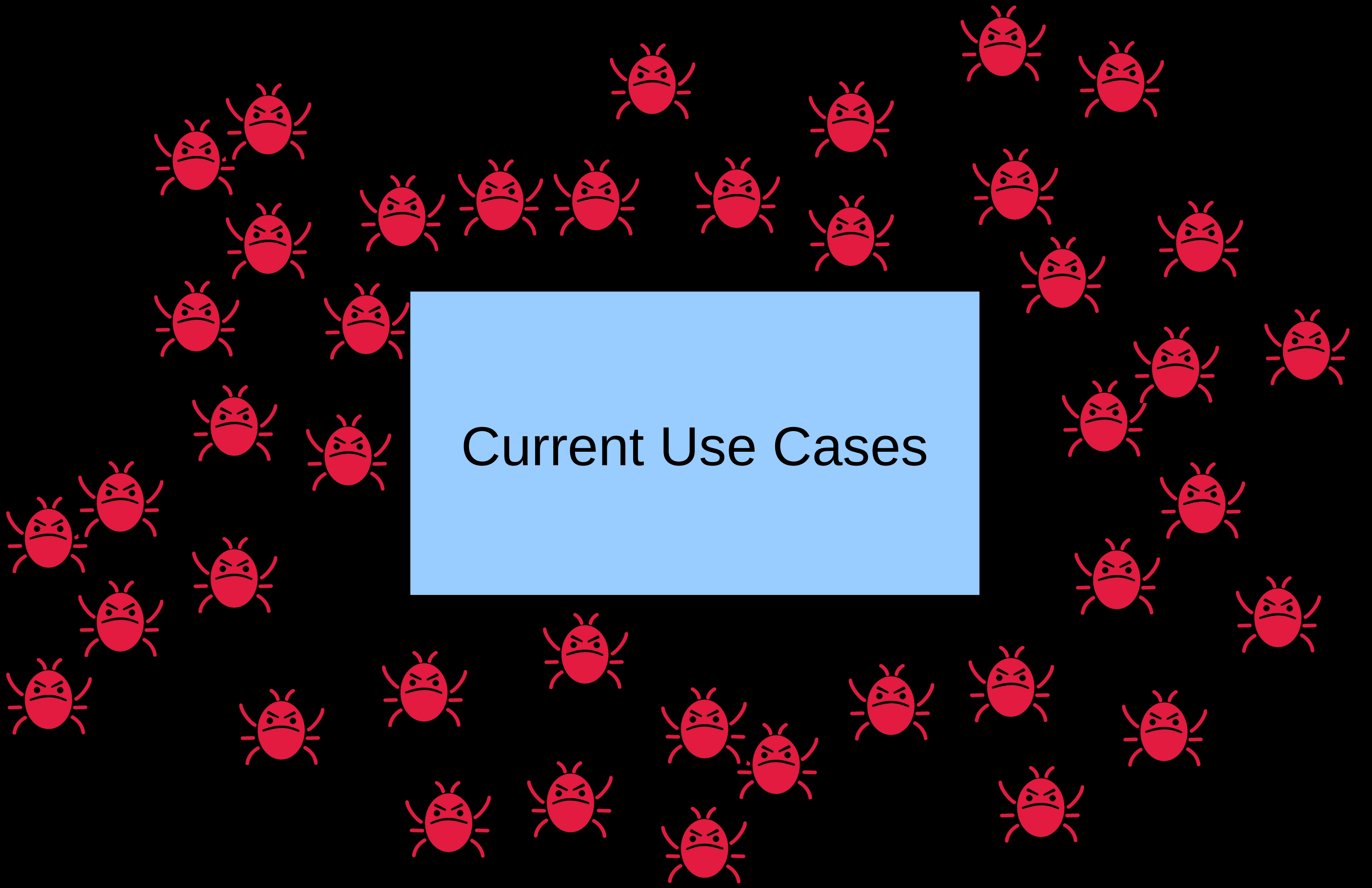
Creating a Wall of Bugs

Current Use Cases

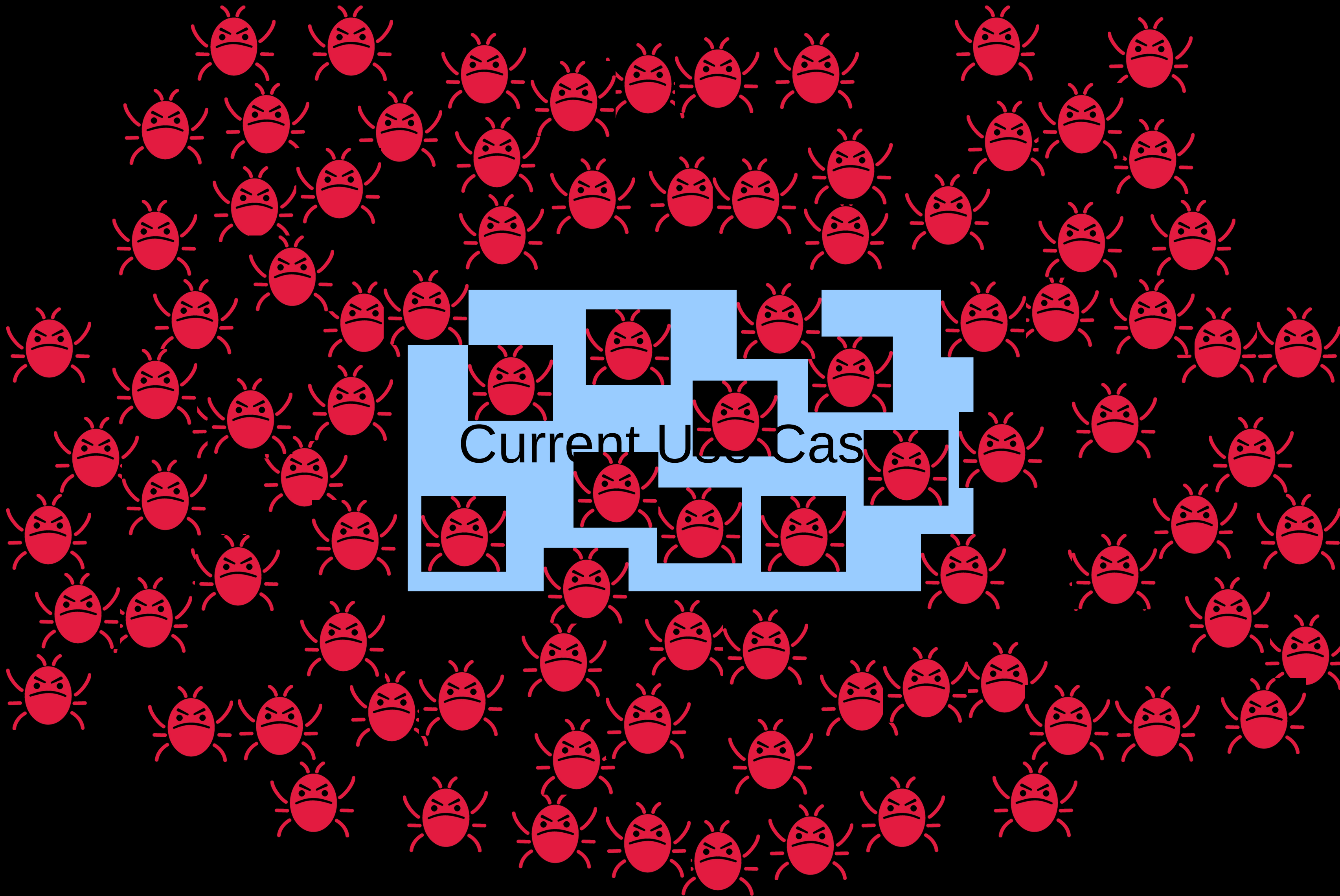
Creating a Wall of Bugs: First Round of Testing



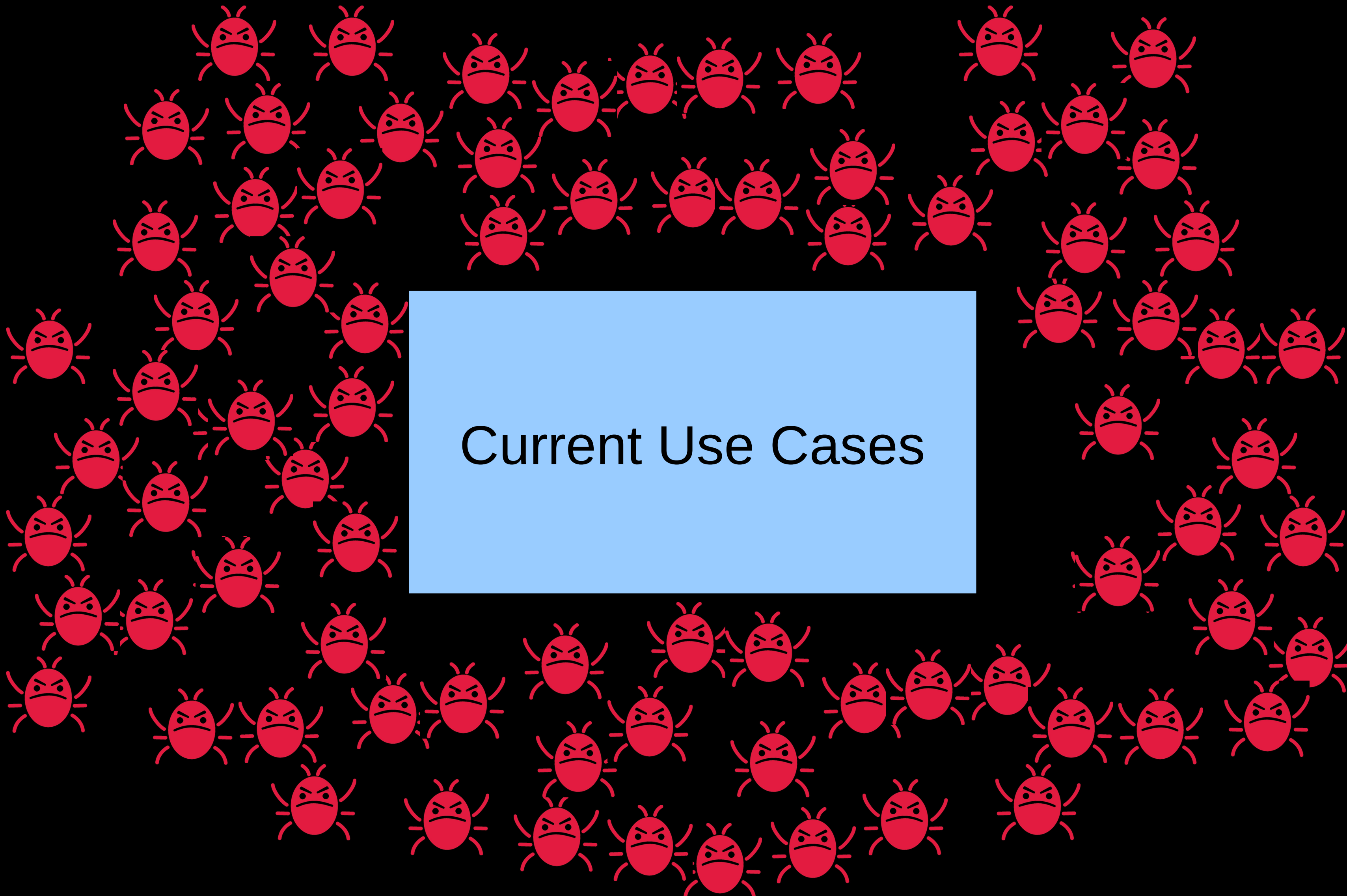
Creating a Wall of Bugs: Fix Relevant Bugs



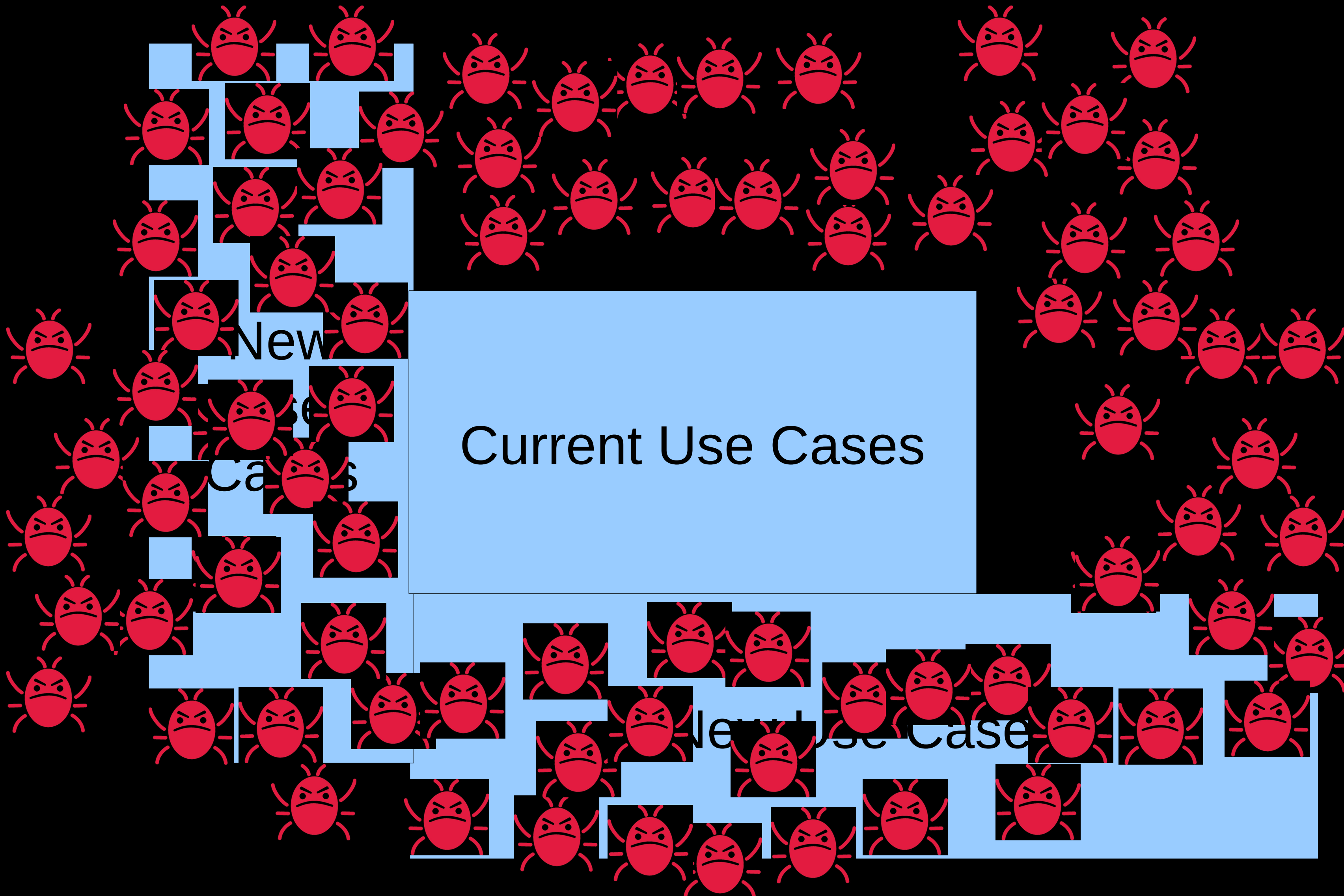
Creating a Wall of Bugs: Second Round of Testing



Creating a Wall of Bugs: Fix Additional Relevant Bugs



Creating a Wall of Bugs: New Use Cases: Game Over!



Cautiously Optimistic For Future CBMC Version

(1)Either automatic translation or no translation required
- No translation required from C, discards irrelevant code quite well

(2)Correctly handle environment, including memory model
- SC, TSO and PSO, hopefully will do other memory models in the future

(3)Reasonable memory and CPU overhead
- OK for Tiny RCU and some tiny uses of concurrent RCU
- Jury is out for concurrent linked-list manipulations
- Progress needed in SAT and in mapping from code to SAT

(4)Map to source code line(s) containing the bug
- Yes, reasonably good backtrace capability

(5)Modest input outside of source code under test
- Yes, modest boilerplate required, can use existing assertions

(6)Find relevant bugs
- Jury still out

Kroening, Clarke, and Lerda, “A tool for checking ANSI-C programs”, *Tools and Algorithms for the Construction and Analysis of Systems, 2004,* pp. 168-176.

A Few Questions/Objections You Might Have...

- But C is Turing-complete and logic expressions are not!!!
  - Yes, hence "bounded". You can specify loop/recursion unrolling limits
- But SAT is NP-complete!!!
  - True, but there are now ***amazing*** heuristics for SAT
  - 1990: World-class solver handles 100 variables (three 32-bit variables)
  - 2015: x86 laptop does 2M variables. In ten seconds.
- How CBMC possibly handle concurrency???
  - Convert C program to SSA, wire reads to writes using memory model
- If this is really useful, why don't you apply it to RCU???
  - I checked CBMC verification of SRCU into -rcu on December 31, 2016
  - Implementation courtesy of Lance Roy
- Has CBMC really found any RCU bugs???
  - Yes, though only injected bugs used to test the verification
  - That is, it has not yet found any bugs that I didn't already know about