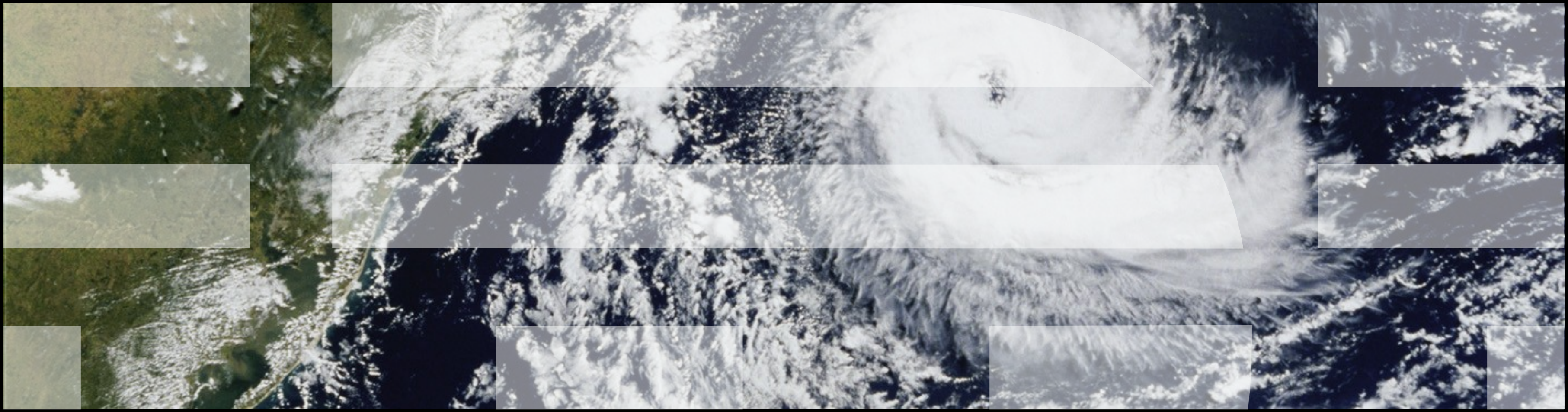
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center

28 October 2011

Validating Core Parallel Software



Overview

- Who is Paul and How Did He Get This Way?
- Avoiding Debugging By Design
- Avoiding Debugging By Process
- Avoiding Debugging By Mechanical Proofs
- Avoiding Debugging By Statistical Analysis
- Coping With Schedule Pressure
- But I Did All This And There Are Still Bugs!!!
- Summary and Conclusions

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
- First use of computer in high school (72-76)
  - IBM mainframe: punched cards and FORTRAN
  - Later ASR33 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
- Sequent Computer Systems (90-00)
  - Communications performance
  - Memory allocators, TLB, RCU, timers, ...
- IBM LTC (00-present)
  - NUMA-aware and brlock-like locking primitive in AIX
    - They didn't want RCU
  - RCU maintainer for Linux kernel

Who is Paul and How Did He Get This Way?

- I have *never*:
  - Used kprobes or SystemTap to find a bug
  - Taken a core dump from a Linux system
  - Used ftrace to find a bug
  - Used “perf” at all
- I *sometimes*:
  - Use debugging printk()s
  - Use event tracing
  - Use WARN_ON_ONCE()
    - Probably more often than printk()
- I *often*:
  - Use special-purpose counters
- Why avoid these debug techniques? What to do instead?

Avoiding Debugging By Design

Avoiding Debugging By Design

- Understand the Hardware
- Understand the Software Environment

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

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

Need to be here! (Partitioning/RCU)

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???

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!!!

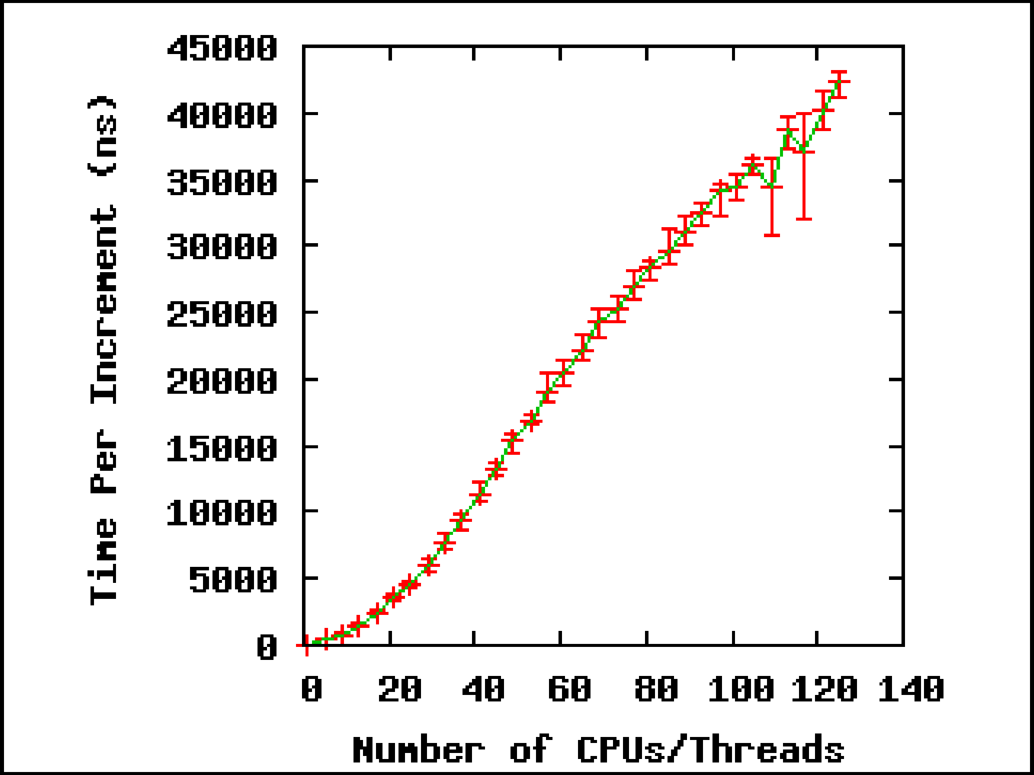
Performance of Synchronization Mechanisms

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

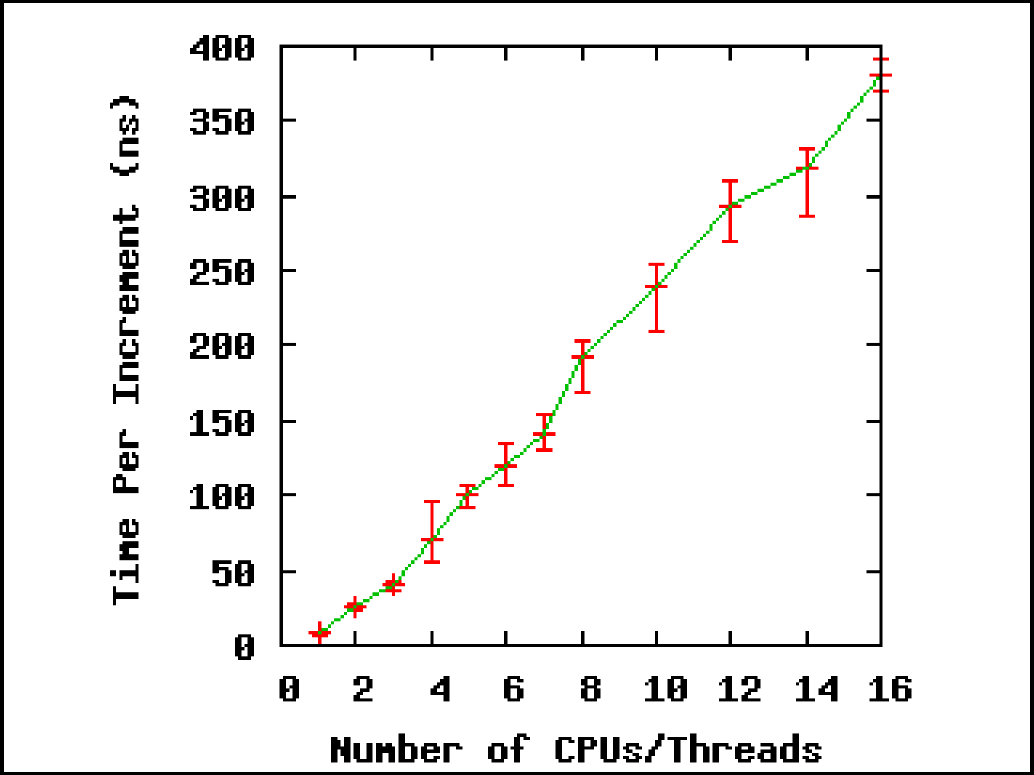
Maybe not such a big difference after all...
And these are best-case values!!! (Why?)

Performance of Synchronization Mechanisms



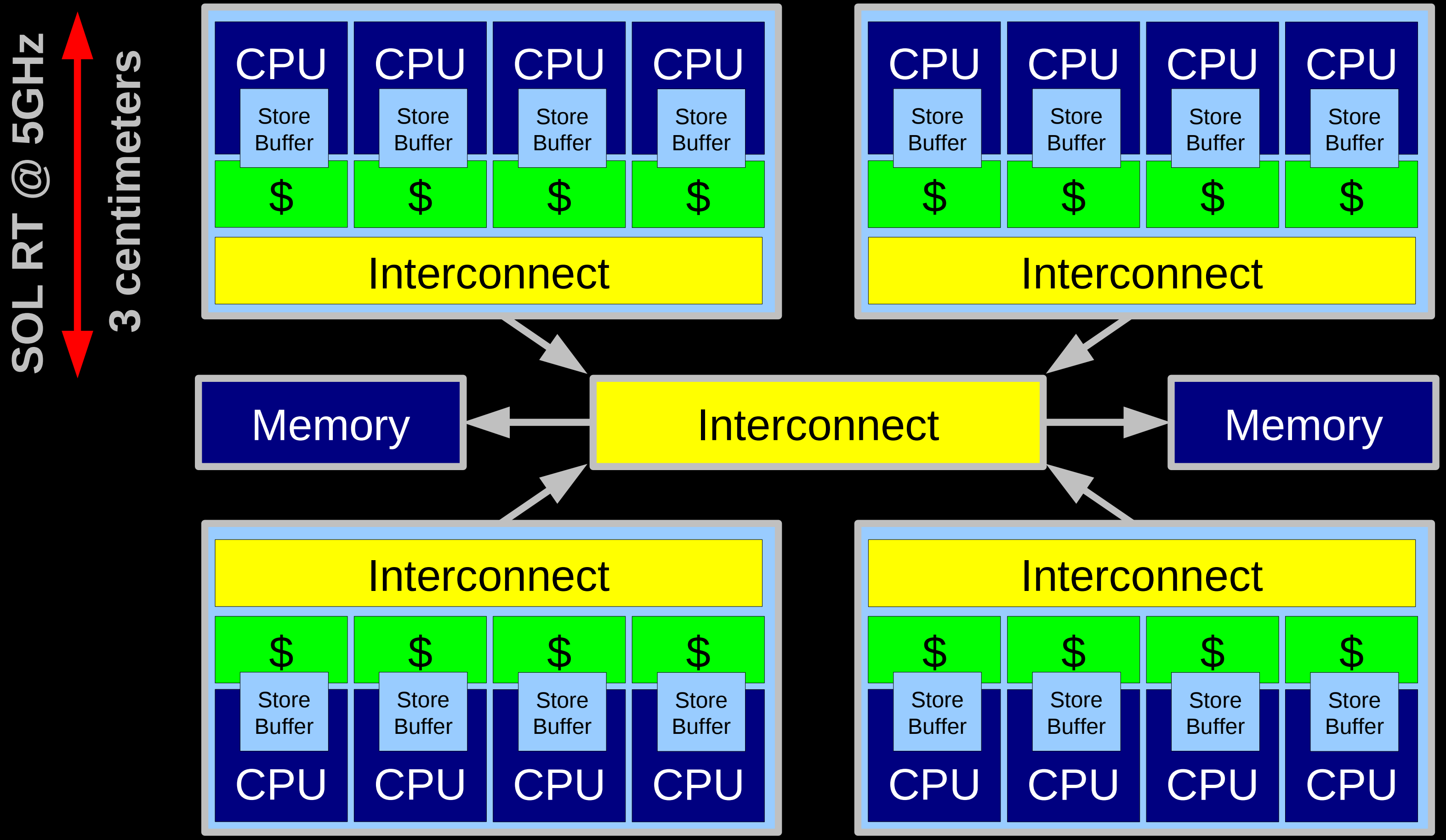
If you thought a *single* atomic operation was slow, try lots of them!!!
(Parallel atomic increment of single variable on 1.9GHz Power 5 system)

Performance of Synchronization Mechanisms



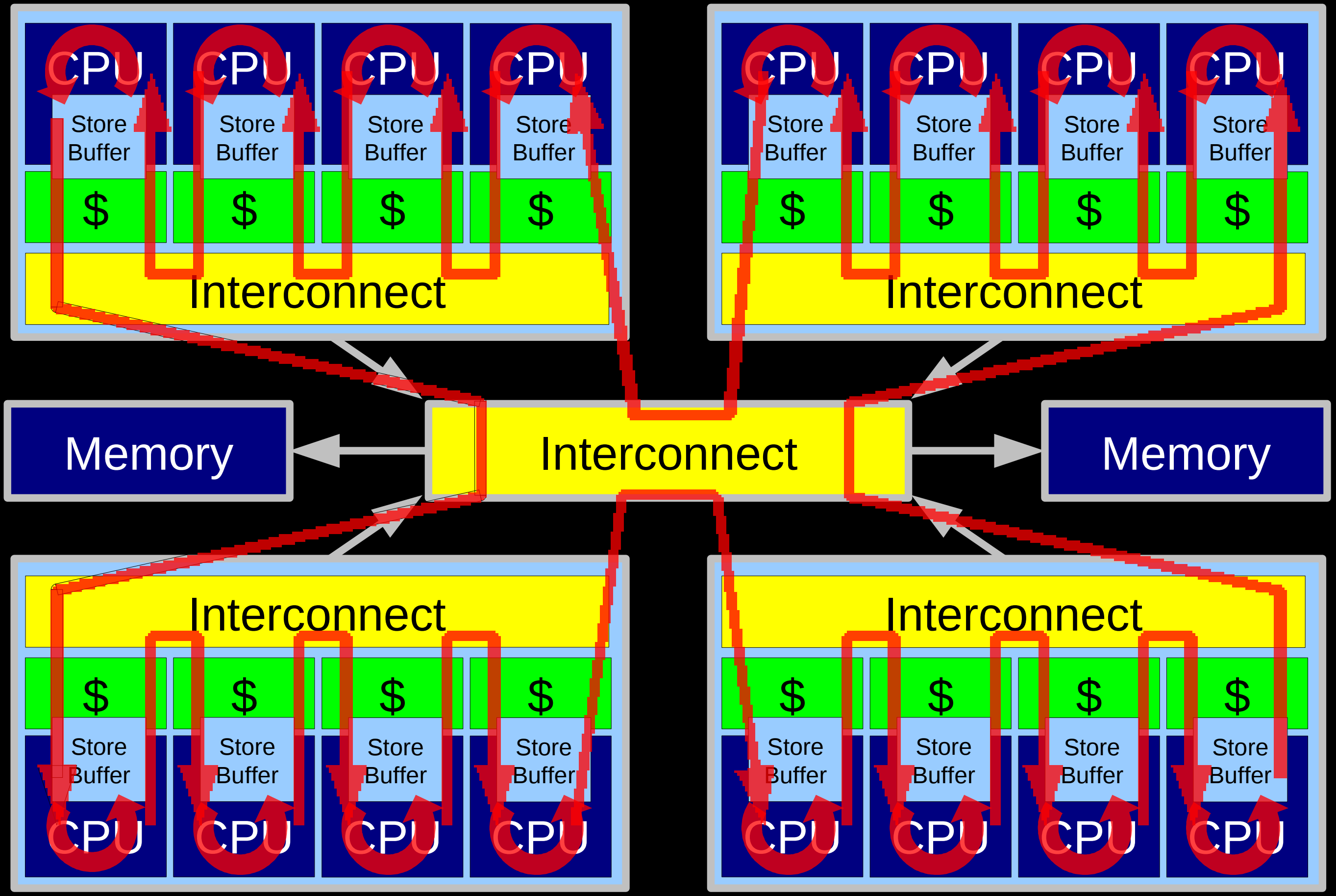
Same effect on a 16-CPU 2.8GHz Intel X5550 (Nehalem) system

System Hardware Structure



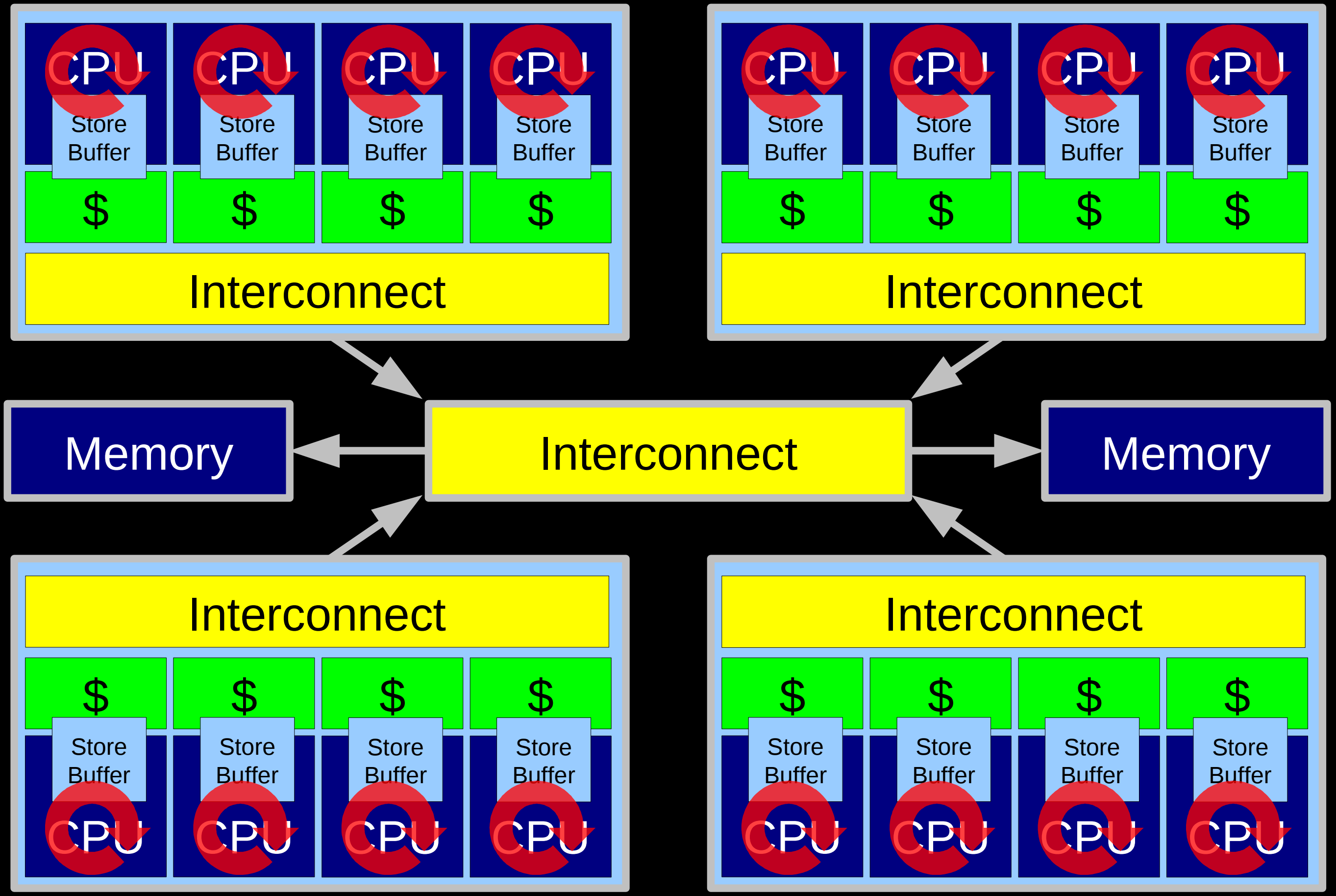
Electrons move at 0.03C to 0.3C in transistors and, so lots of waiting. 3D???

Atomic Increment of Global Variable



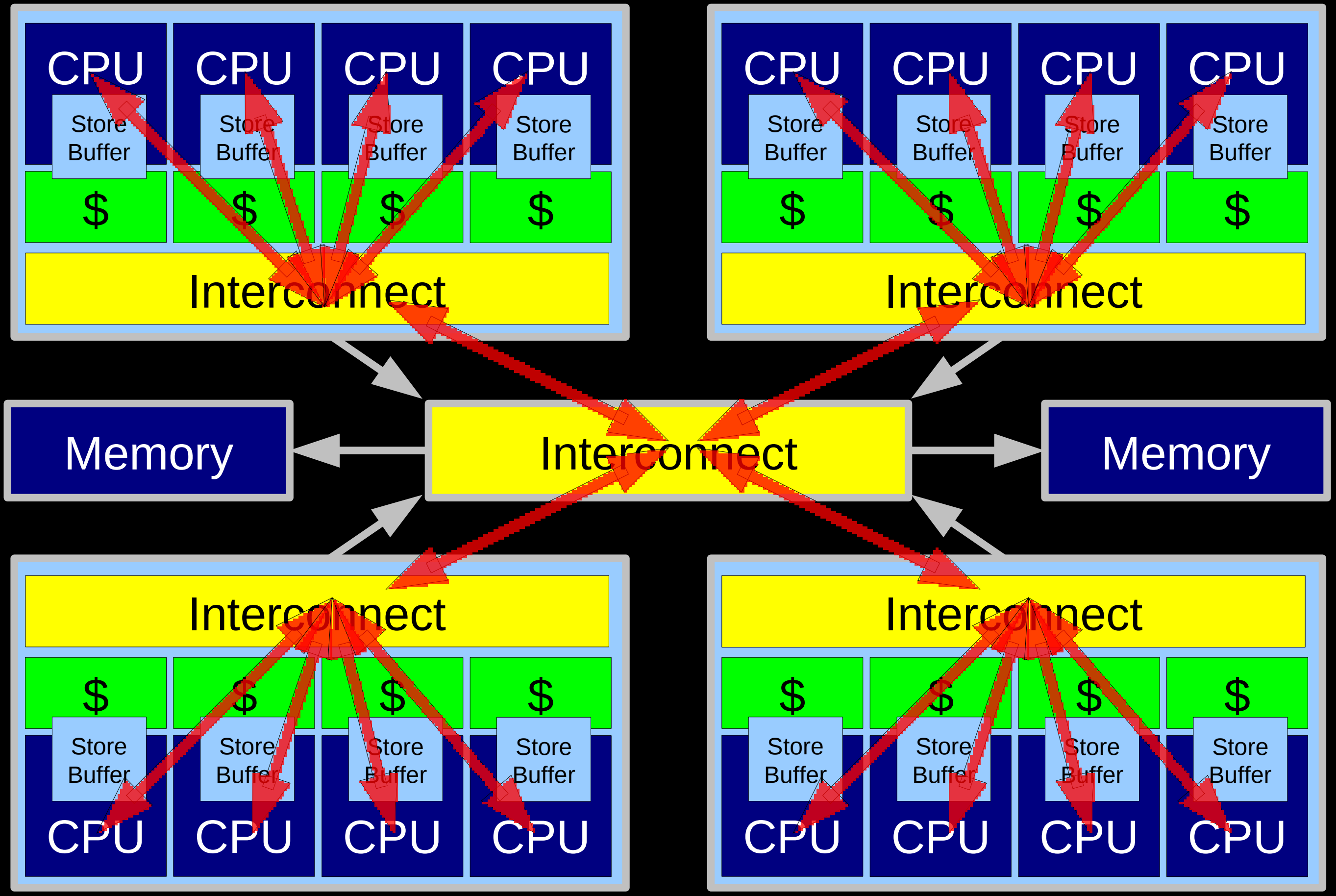
Lots and Lots of Latency!!!

Atomic Increment of Per-CPU Variable

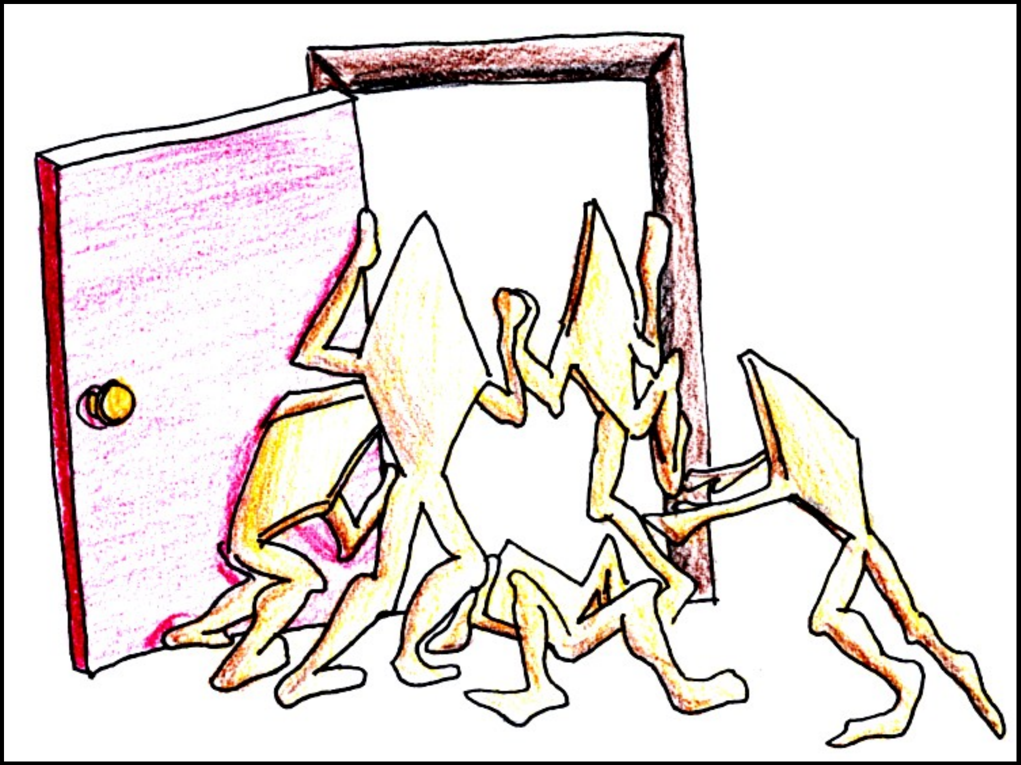


Little Latency, Lots of Increments at Core Clock Rate

HW-Assist Atomic Increment of Global Variable

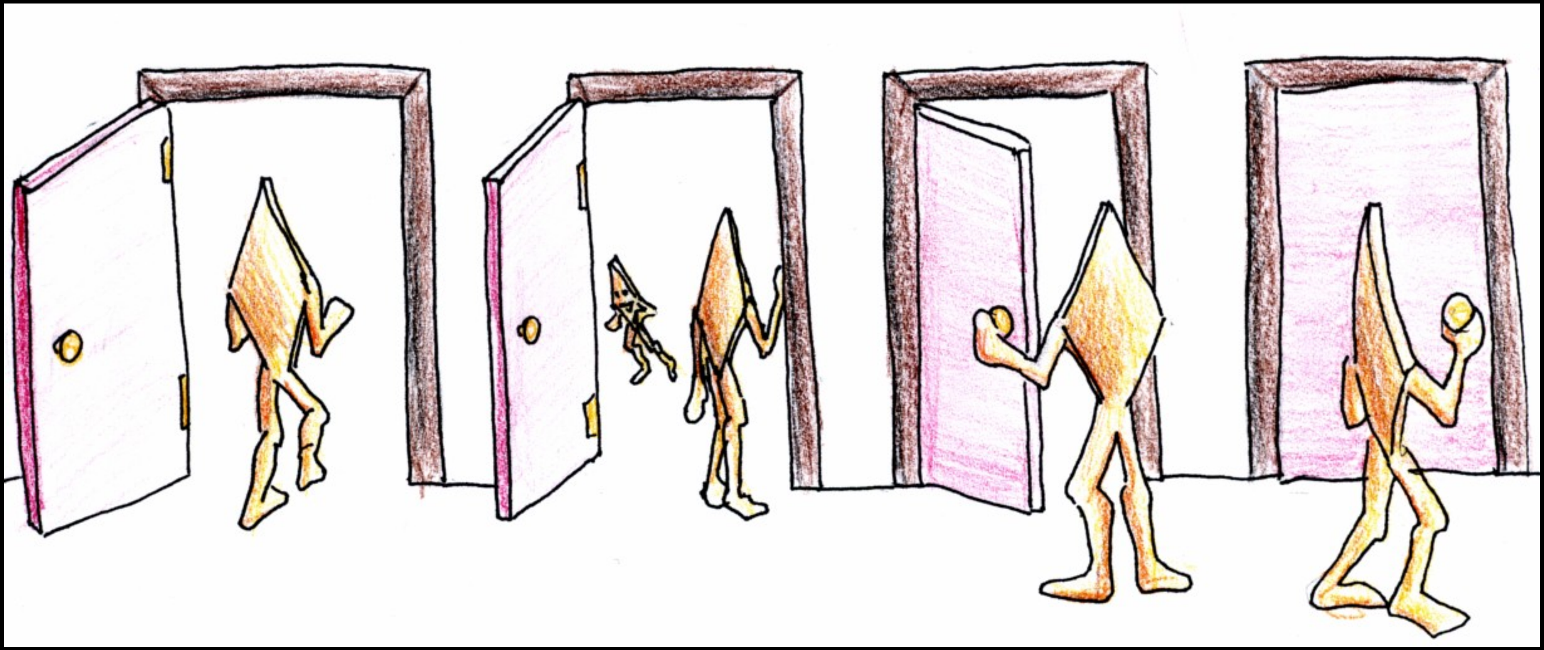


Design Principle: Avoid Bottlenecks



Only one of something: bad for performance and scalability

Design Principle: Avoid Bottlenecks



Many instances of something good!
Any exceptions to this rule?

Understand the Hardware: Summary

- A strong understanding of the hardware helps rule out infeasible designs early in process
- Understanding hardware trends helps reduce the amount of future rework required
- Ditto for low-level software that your code depends on

Understand the Software Environment

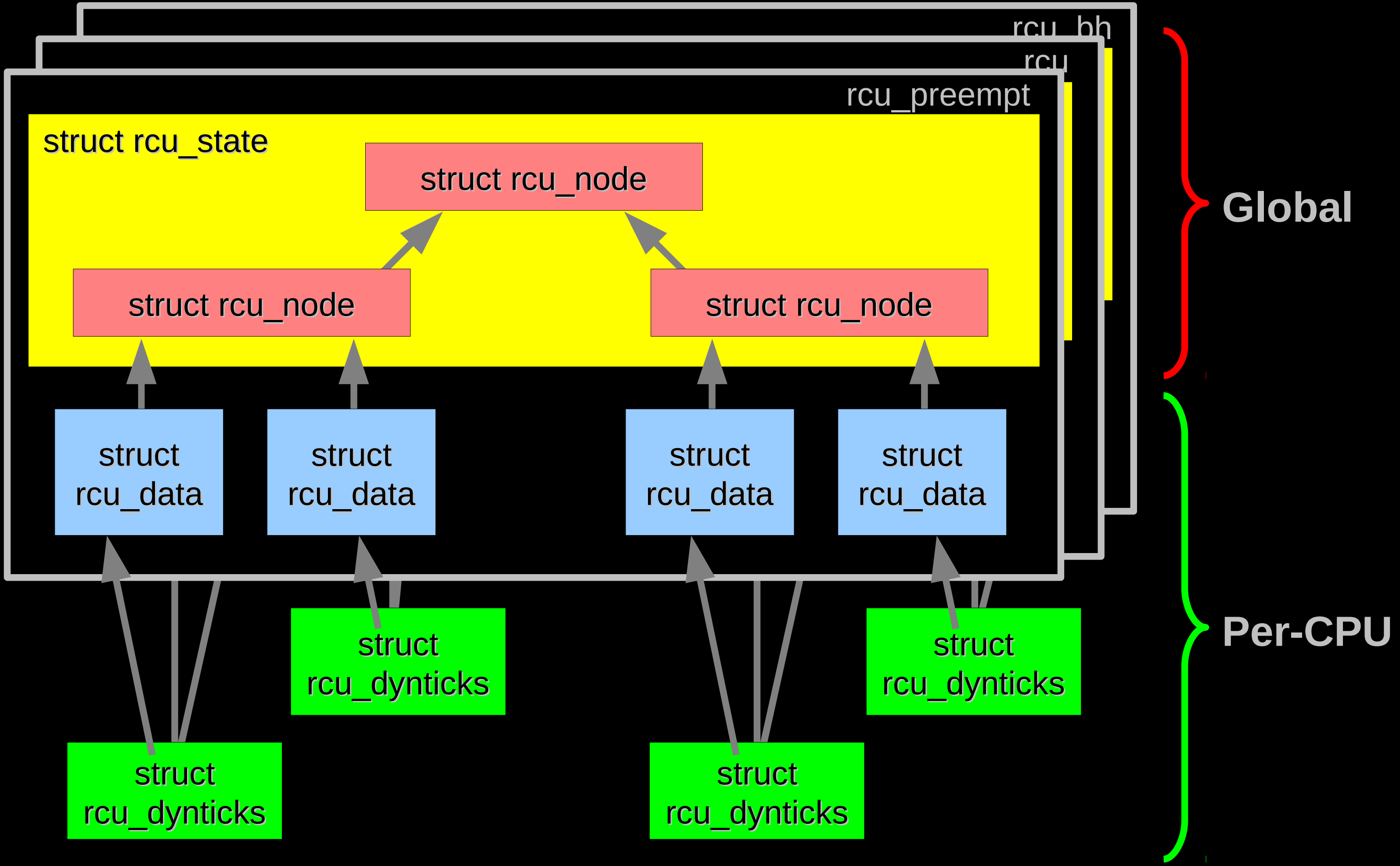
- Understand the Workloads
  - Which for Linux means a great many of them
  - Your code must take whatever shows up
- Google-Search LWN
  - But you knew this already
- Test Unfamiliar Primitives
  - And complain on LKML if they break
  - Preferably accompanying the complaint with a fix
- Review Others' Code
  - See recent ltc-interlock discussion for how-to info
- Make a Map
  - See next slides...

Making a Map of Software

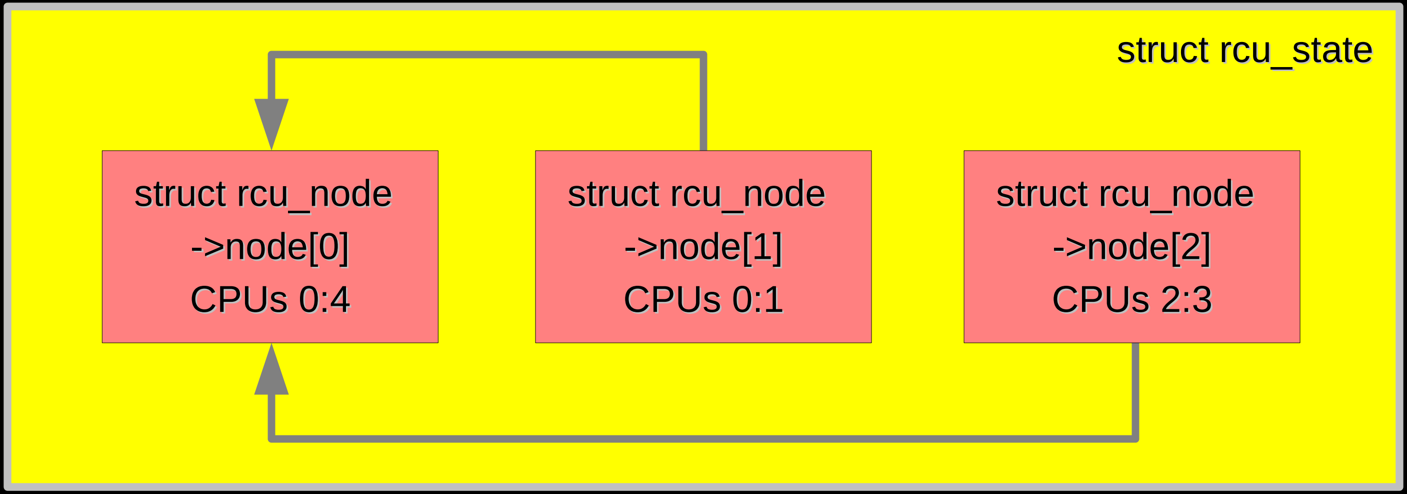
Hierarchical RCU Data Structures

```
struct rcu_dynticks {
  int dynticks_nesting;
  int dynticks;
  int dynticks_nmi;
};

struct rcu_node {
  spinlock_t lock;
  long  gpnum;
  long  completed;
  unsigned long qsmask;
  unsigned long qsmaskinit;
  unsigned long grpmask;
  int  grplo;
  int  grphi;
  u8  grpnum;
  u8  level;
  struct rcu_node *parent;
  struct list_head blocked_tasks[2];
}

struct rcu_data {
  long    completed;
  long    gpnum;
  long    passed_quiesc_completed;
  bool    passed_quiesc;
  bool    qs_pending;
  bool    beenonline;
  bool    preemptable;
  struct rcu_node *mynode;
  unsigned long grpmask;
  struct rcu_head *nxtlist;
  struct rcu_head **nxttail[RCU_NEXT_SIZE];
  long    qlen;
  long    qlen_last_fqs_check;
  unsigned long  n_force_qs_snap;
  long    blimit;
#ifdef CONFIG_NO_HZ
  struct rcu_dynticks *dynticks;
  int dynticks_snap;
  int dynticks_nmi_snap;
#ifdef CONFIG_NO_HZ
  unsigned long dynticks_fqs;
#endif /* #ifdef CONFIG_NO_HZ */
  unsigned long offline_fqs;
  unsigned long resched_ipi;
  long n_rcu_pending;
  long n_rp_qs_pending;
  long n_rp_cb_ready;
  long n_rp_cpu_needs_gp;
  long n_rp_gp_completed;
  long n_rp_gp_started;
  long n_rp_need_fqs;
  long n_rp_need_nothing;
  int cpu;
};

struct rcu_state {
  struct rcu_node node[NUM_RCU_NODES];
  struct rcu_node *level[NUM_RCU_LVLS];
  u32 levelcnt[MAX_RCU_LVLS + 1];
  u8 levelspread[NUM_RCU_LVLS];
  struct rcu_data *rda[NR_CPUS];
  u8  signaled;
  long  gpnum;
  long  completed;
  spinlock_t onofflock;
  struct rcu_head *orphan_cbs_list;
  struct rcu_head **orphan_cbs_tail;
  long orphan_qlen;
  spinlock_t fqslock;
  unsigned long jiffies_force_qs;
  unsigned long n_force_qs;
  unsigned long n_force_qs_lh;
  unsigned long n_force_qs_ngp;
#ifdef CONFIG_RCU_CPU_STALL_DETECTOR
  unsigned long gp_start;
  unsigned long jiffies_stall;
#endif /* #ifdef CONFIG_RCU_CPU_STALL_DETECTOR */
  long dynticks_completed;
};
```

Mapping Data Structures



Placement of rcu_node Within rcu_state


struct rcu_state
struct rcu_node ->node[0] CPUs 0:4
struct rcu_node ->node[1] CPUs 0:1
struct rcu_node ->node[2] CPUs 2:3

Avoiding Debugging By Process

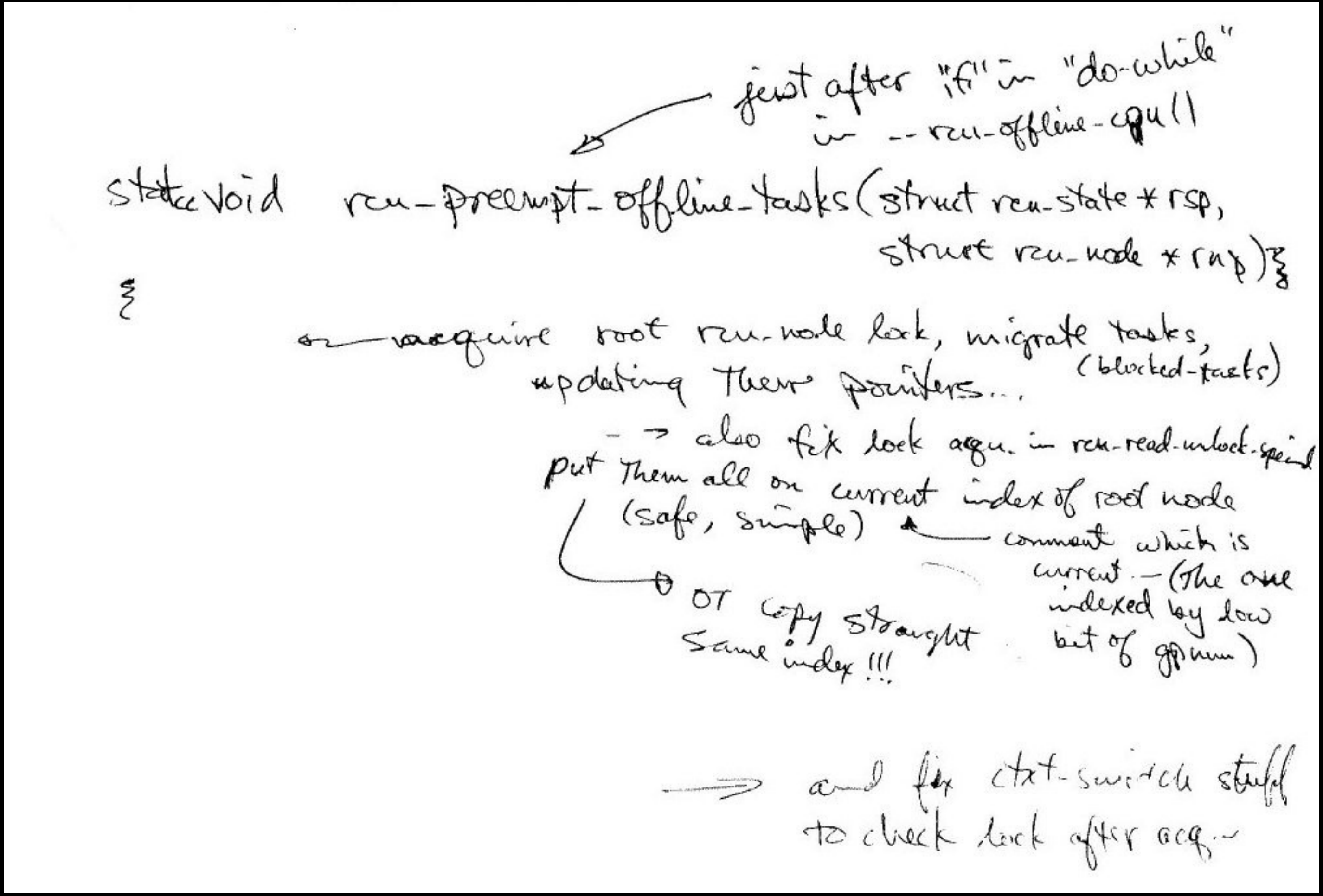
Avoiding Debugging By Process

- Review your own work carefully
  - See following slides
- Test early, test often, test in small pieces
  - Debugging is 2-3 times harder than writing code
  - Debugging effort rises as the square of the amount of new code added to the testing effort
- Where possible, use existing well-tested code
  - Even if it is a lot more fun to re-invent the wheel
- I would have scorned this advice as late as the early 1990s, but have since learned it the hard way
- And still sometimes has difficulty following it:
  - http://paulmck.livejournal.com/14639.html

Review Your Own Code Carefully

- Paul E. McKenney's self-review rules for complex code:
  - Write the code long hand in pen on paper
  - Correct bugs as you go
  - Copy onto a clean sheet of paper
  - Repeat until the last two versions are identical

- What constitutes “not complex”?
  - Sequential code, *and*
  - You test it line-by-line
    - For example, bash script or single-threaded C-code with gdb)



just after "if" in "do-while" in __rcu_offline_cpu()

static void rcu_preempt_offline_tasks(struct rcu_state *rsp,
struct rcu_node *rnp)}
{

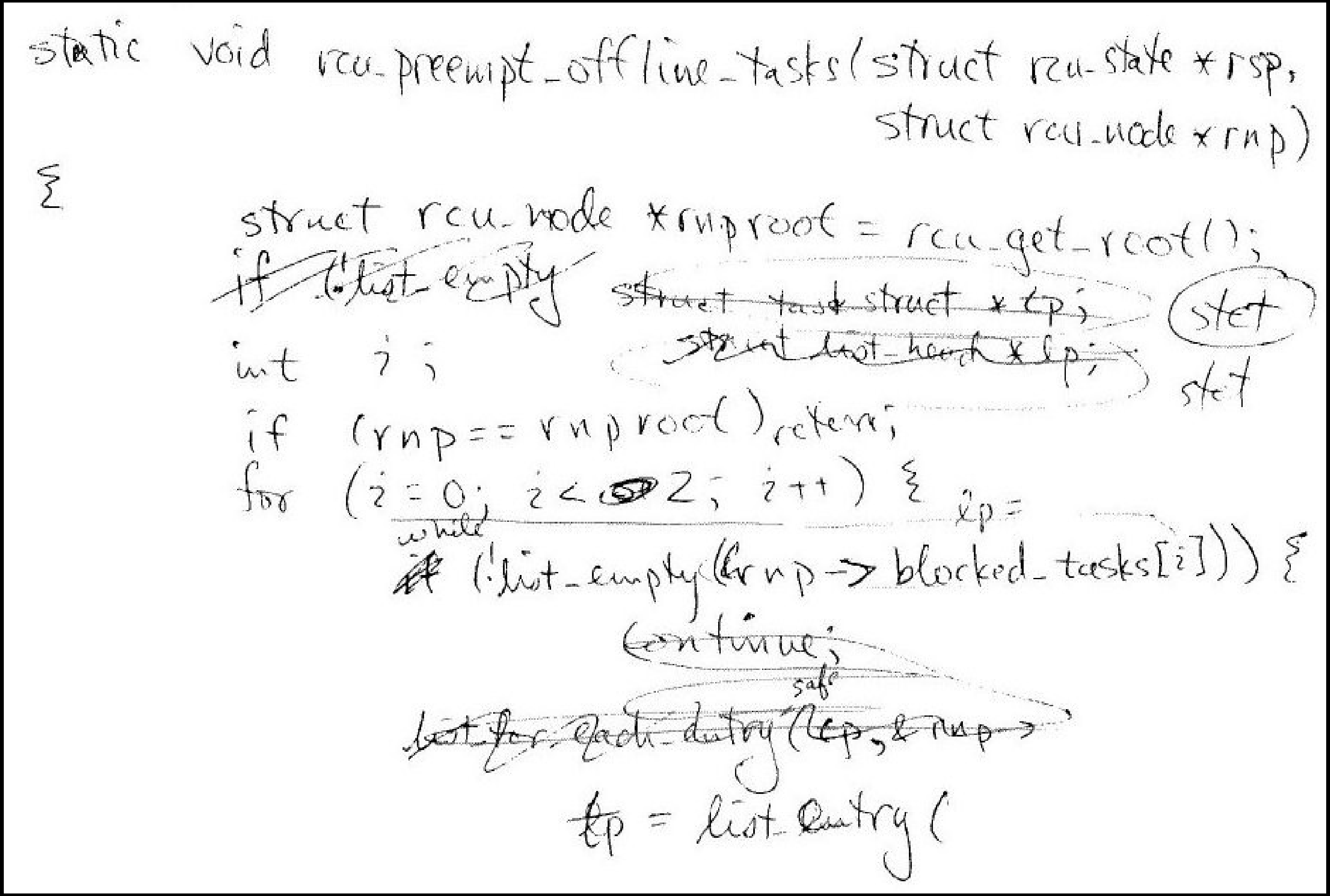
~~or~~ — ~~re~~acquire root rcu_node lock, migrate tasks, (blocked_tasks) updating their pointers...

→ also fix lock acqu. in rcu_read_unlock_special

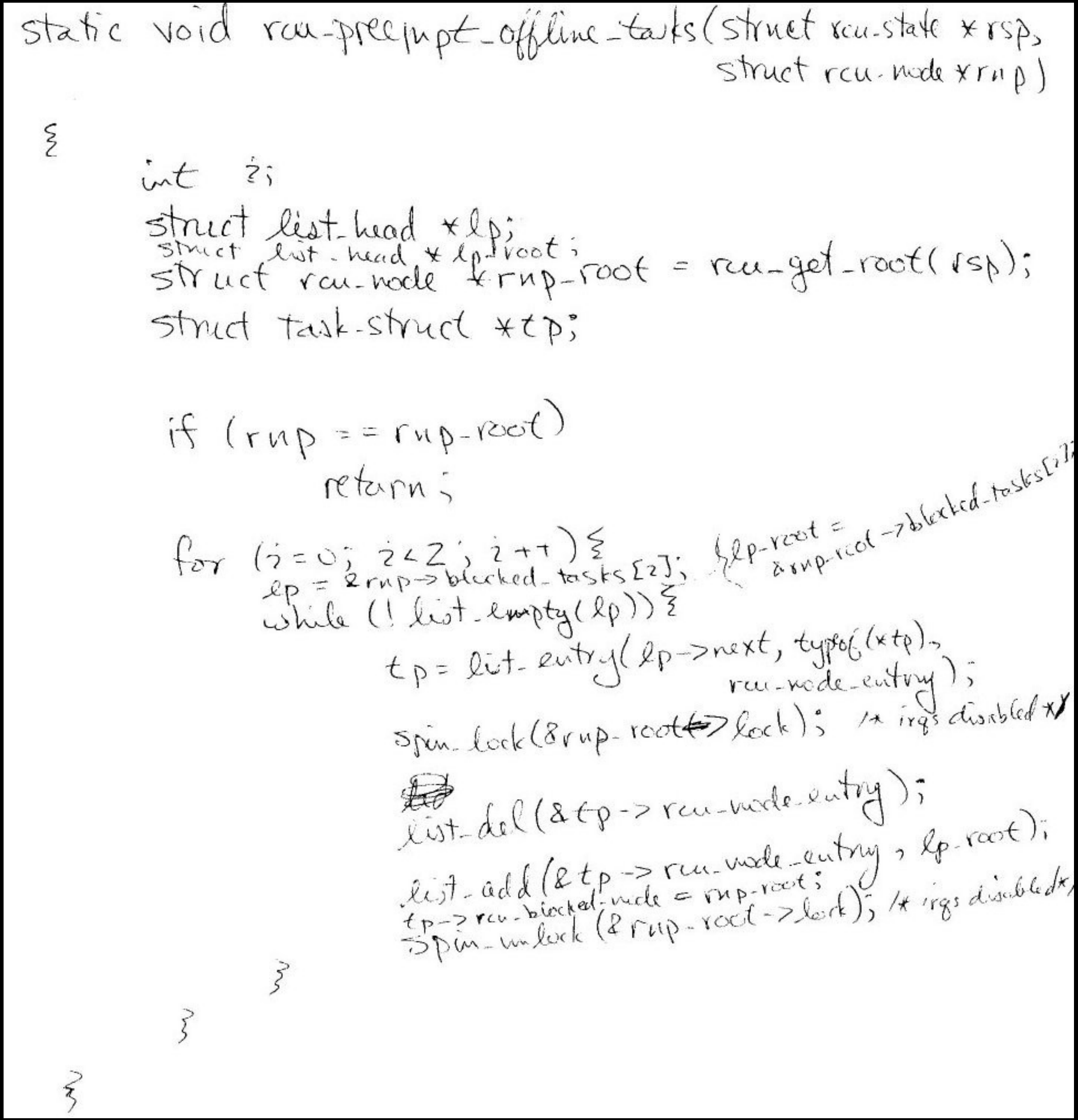
put them all on current index of root node (safe, simple) ← comment which is current — (the one indexed by low bit of gpnum)

or copy straight same index !!!

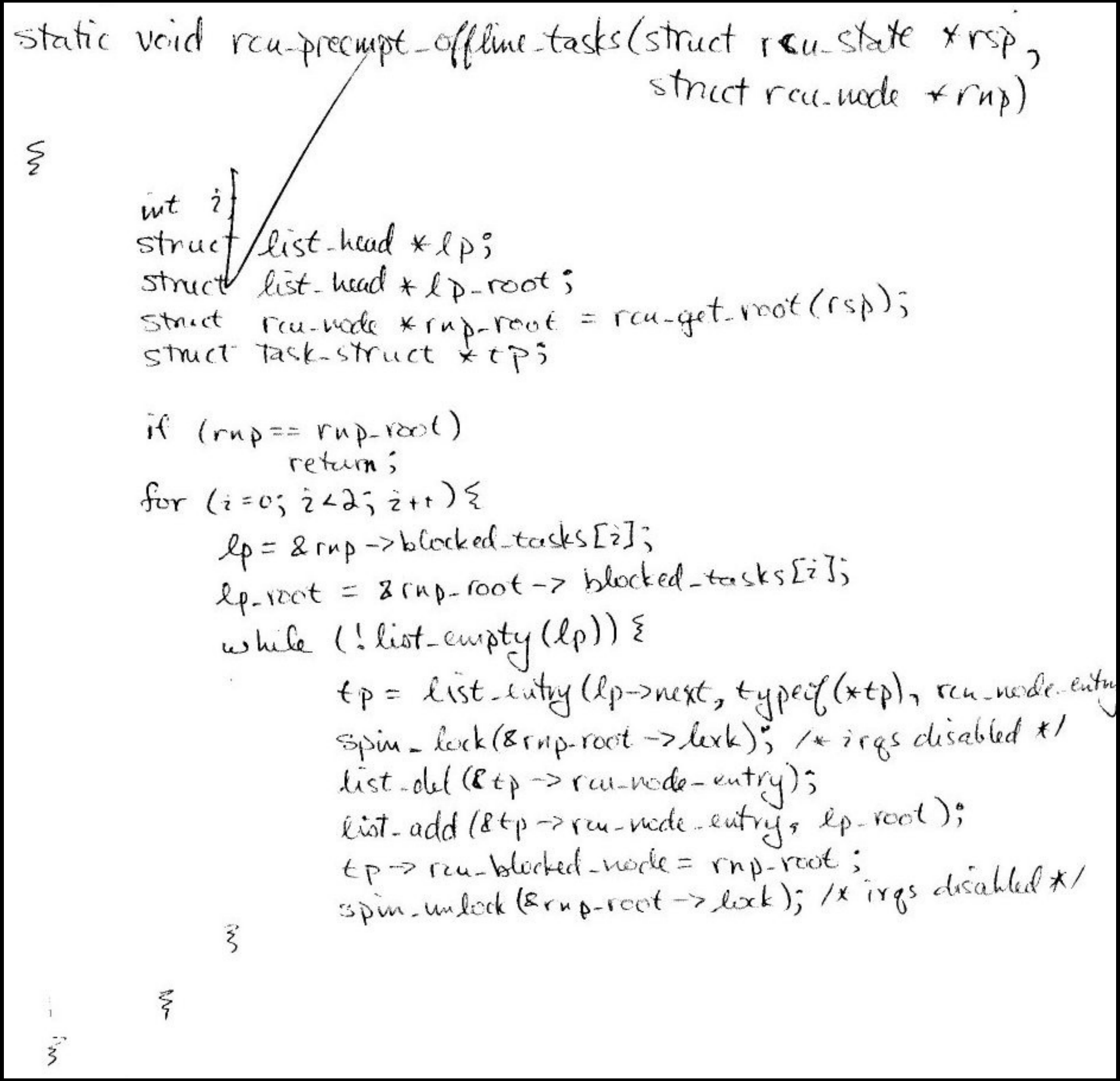
⟹ and fix ctxt-switch stuff to check lock after acq.



```
static void rcu_preempt_offline_tasks(struct rcu_state *rsp,
                                      struct rcu_node *rnp)
{
        struct rcu_node *rnp_root = rcu_get_root();
        ~~if (!list_empty~~    ~~struct task_struct *tp;~~    stet
        int i;                     ~~struct list_head *lp;~~    stet
        if (rnp == rnp_root()) return;
        for (i = 0; i < 2; i++) {              lp =
                while
                ~~if~~ (!list_empty(&rnp->blocked_tasks[i])) {
                        ~~continue;~~
                                          safe
                        ~~list_for_each_entry(tp, &rnp->~~
                        tp = list_entry(
```



```
static void rcu-preempt_offline_tasks(struct rcu-state *rsp,
                                       struct rcu-node *rnp)

{
        int  i;
        struct list_head *lp;
        struct list-head *lp-root;
        struct rcu-node *rnp_root = rcu_get_root(rsp);
        struct task-struct *tp;


        if (rnp == rnp-root)
                return;

        for (i = 0; i < 2; i ++) {
          lp = &rnp->blocked_tasks[i];   {lp-root =
                                          &rnp-root->blocked-tasks[i];
          while (! list_empty(lp)) {
                tp = list-entry(lp->next, typeof(*tp),
                                          rcu-node-entry);
                spin_lock(&rnp-root->lock);  /* irqs disabled */

                list_del(&tp->rcu-node-entry);
                list_add(&tp->rcu-node-entry, lp-root);
                tp->rcu-blocked-node = rnp-root;
                spin-unlock(&rnp-root->lock); /* irqs disabled*/

          }
        }
}
```



```
static void rcu_preempt_offline_tasks(struct rcu_state *rsp,
                                      struct rcu_node *rnp)
{
        int i;
        struct list_head *lp;
        struct list_head *lp_root;
        struct rcu_node *rnp_root = rcu_get_root(rsp);
        struct task_struct *tp;

        if (rnp == rnp_root)
                return;
        for (i = 0; i < 2; i++) {
                lp = &rnp->blocked_tasks[i];
                lp_root = &rnp_root->blocked_tasks[i];
                while (!list_empty(lp)) {
                        tp = list_entry(lp->next, typeof(*tp), rcu_node_entry);
                        spin_lock(&rnp_root->lock); /* irqs disabled */
                        list_del(&tp->rcu_node_entry);
                        list_add(&tp->rcu_node_entry, lp_root);
                        tp->rcu_blocked_node = rnp_root;
                        spin_unlock(&rnp_root->lock); /* irqs disabled */
                }
        }
}
```

So, How Well Did I Do?

```
static void rcu_preempt_offline_tasks(struct rcu_state *rsp,
              struct rcu_node *rnp,
              struct rcu_data *rdp)
{
  int i;
  struct list_head *lp;
  struct list_head *lp_root;
  struct rcu_node *rnp_root = rcu_get_root(rsp);
  struct task_struct *tp;

  if (rnp == rnp_root) {
    WARN_ONCE(1, "Last CPU thought to be offlined?");
    return;
  }
  WARN_ON_ONCE(rnp != rdp->mynode &&
         (!list_empty(&rnp->blocked_tasks[0]) ||
          !list_empty(&rnp->blocked_tasks[1])));
  for (i = 0; i < 2; i++) {
    lp = &rnp->blocked_tasks[i];
    lp_root = &rnp_root->blocked_tasks[i];
    while (!list_empty(lp)) {
      tp = list_entry(lp->next, typeof(*tp), rcu_node_entry);
      spin_lock(&rnp_root->lock); /* irqs already disabled */
      list_del(&tp->rcu_node_entry);
      tp->rcu_blocked_node = rnp_root;
      list_add(&tp->rcu_node_entry, lp_root);
      spin_unlock(&rnp_root->lock); /* irqs remain disabled */
    }
  }
}
```

```
static int rcu_preempt_offline_tasks(struct rcu_state *rsp,
                                     struct rcu_node *rnp,
                                     struct rcu_data *rdp)
{
  int i;
  struct list_head *lp;
  struct list_head *lp_root;
  int retval;
  struct rcu_node *rnp_root = rcu_get_root(rsp);
  struct task_struct *tp;

  if (rnp == rnp_root) {
    WARN_ONCE(1, "Last CPU thought to be offlined?");
    return 0;  /* Shouldn't happen: at least one CPU online. */
  }
  WARN_ON_ONCE(rnp != rdp->mynode &&
         (!list_empty(&rnp->blocked_tasks[0]) ||
          !list_empty(&rnp->blocked_tasks[1])));
  retval = rcu_preempted_readers(rnp);
  for (i = 0; i < 2; i++) {
    lp = &rnp->blocked_tasks[i];
    lp_root = &rnp_root->blocked_tasks[i];
    while (!list_empty(lp)) {
      tp = list_entry(lp->next, typeof(*tp), rcu_node_entry);
      spin_lock(&rnp_root->lock); /* irqs already disabled */
      list_del(&tp->rcu_node_entry);
      tp->rcu_blocked_node = rnp_root;
      list_add(&tp->rcu_node_entry, lp_root);
      spin_unlock(&rnp_root->lock); /* irqs remain disabled */
    }
  }
  return retval;
}
```

Avoiding Debugging By Mechanical Proofs

Avoiding Debugging By Mechanical Proofs

- Works well for small, self-contained algorithms
  - http://lwn.net/Articles/243851/ (QRCU)
  - http://lwn.net/Articles/279077/ (RCU dynticks I/F)
  - git://lttng.org/userspace-rcu formal-model (URCU)

- However, the need for formal proof often indicates an overly complex design!!!
  - Preemptible RCU's dynticks interface being an extreme case in point (http://lwn.net/Articles/279077/)

Avoiding Debugging By Statistical Analysis

Avoiding Debugging By Statistical Analysis

- Different kernel configuration options select different code
- Suppose that more failure occur with CONFIG_FOO=y
  - Focus inspection on code under #ifdef CONFIG_FOO
- But what exactly does “more failures” mean?

Avoiding Debugging By Statistical Analysis

- Different kernel configuration options select different code
- Suppose that more failure occur with CONFIG_FOO=y
  - Focus inspection on code under #ifdef CONFIG_FOO
- But what exactly does “more failures” mean?
  - That is where the statistical analysis comes in
  - The “more failures” must be enough more to be statistically significant
  - One of the most useful classes I took as an undergraduate was a statistics course!

Coping With Schedule Pressure

Coping With Schedule Pressure

- When you are fixing a critical bug, speed counts
- The difference is level of risk
  - The code is *already* broken, so less benefit to using extremely dainty process steps
  - But *only* if you follow up with careful process
  - Which I repeatedly learn the hard way: http://paulmck.livejournal.com/14639.html
  - Failure to invest a few days in early 2009 cost me more than a month in late 2009!!!
- Long-term perspective required
  - And that means *you* – leave the "blame it on management" game to Dilbert cartoons
  - Align with management initiatives, for example, "agile development"

But I Did All This And There Are Still Bugs!!!

But I Did All This And There Are Still Bugs!!!

- “Be Careful!!! It Is A Real World Out There!!!”
- The purpose of careful software-development practices is to reduce risk
  - Strive for perfection, but understand that this goal is rarely reached in this world

But I Did All This And There Are Still Bugs!!!

- "Be Careful!!! It Is A Real World Out There!!!"
- The purpose of careful software-development practices is to reduce risk
  - Strive for perfection, but understand that this goal is rarely reached in this world
- **But you still need to fix your bugs!!!**

Fixing Bugs

- The first challenge is locating the bugs

Fixing Bugs

- The first challenge is locating the bugs
  - The computer knows where the bugs are

Fixing Bugs

- The first challenge is locating the bugs
  - The computer knows where the bugs are
  - So your job is to make it tell you!
- Ways to make the computer tell you where the bugs are:

Fixing Bugs

- The first challenge is locating the bugs
  - The computer knows where the bugs are
  - So your job is to make it tell you!
- Ways to make the computer tell you where the bugs are:
  - Debugging printk()s and assertions
  - Event tracing and ftrace
  - Lock dependency checker (CONFIG_PROVE_LOCKING and CONFIG_PROVE_RCU)
  - Static analysis (and pay attention to compiler warnings!!!)
  - Structured testing: Use an experimental approach
  - **Record all test results, including environment**

Fixing Bugs

- The first challenge is locating the bugs
  - The computer knows where the bugs are
  - So your job is to make it tell you!
  - But getting another person's viewpoint can be helpful
    - To 10,000 *educated and experienced* eyes, all bugs are shallow
- Gaining other people's viewpoints

Fixing Bugs

- The first challenge is locating the bugs
  - The computer knows where the bugs are
  - So your job is to make it tell you!
  - But getting another person's viewpoint can be helpful
    - To 10,000 *educated and experienced* eyes, all bugs are shallow
- Gaining other people's viewpoints
  - Have other people review your code
  - Explain your code to someone else
  - Special case of explaining code: Document it
    - Think of questions you might ask if someone else showed you the code
    - Focus on the parts of the code you are most proud of: Most likely buggy!
    - Try making a copy of the code, removing the comments, and then documenting it: Perhaps the comments are confusing you

But What If The Computer Knows Too Much?

- Event tracing for RCU: 35MB of trace events for failure
- Way too much to read and analyze by hand
- What to do?

But What If The Computer Knows Too Much?

- Event tracing for RCU: 35MB of trace events for each failure
- Way too much to read and analyze by hand all the time
- What to do? Scripting!!!
- How to generate useful scripts:
  - Do it by hand the first few times
  - But keep detailed notes on what you did and what you found
  - Incrementally construct scripts to carry out the most laborious tasks
  - Eventually, you will have a script that analyzes the failures
- But suppose you are working on many different projects?

But What If The Computer Knows Too Much?

- Event tracing for RCU: 35MB of trace events for each failure
- Way too much to read and analyze by hand all the time
- What to do? Scripting!!!
- How to generate useful scripts:
  - Do it by hand the first few times
  - But keep detailed notes on what you did and what you found
  - Incrementally construct scripts to carry out the most laborious tasks
  - Eventually, you will have a script that analyzes the failures
- But suppose you are working on many different projects?
  - Script the common cases that occur in many projects
  - Take advantage of tools others have constructed

Summary and Conclusions

Summary and Conclusions

- Avoid Debugging By Design
- Avoid Debugging By Process
- Avoid Debugging By Mechanical Proofs
- Avoid Debugging By Statistical Analysis
- Avoid Schedule Pressure via Long-Term View
- But Even If You Do All This, You Will Still Do Some Debugging (http://lwn.net/Articles/453002/)
  - Yes, you are living in the real world!!!
  - Might be painful sometimes, but it sure beats all known alternatives...

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.