Creating Performant and Scalable Linux Applications

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

ACACES July 15, 2009

Course Objectives and Goals

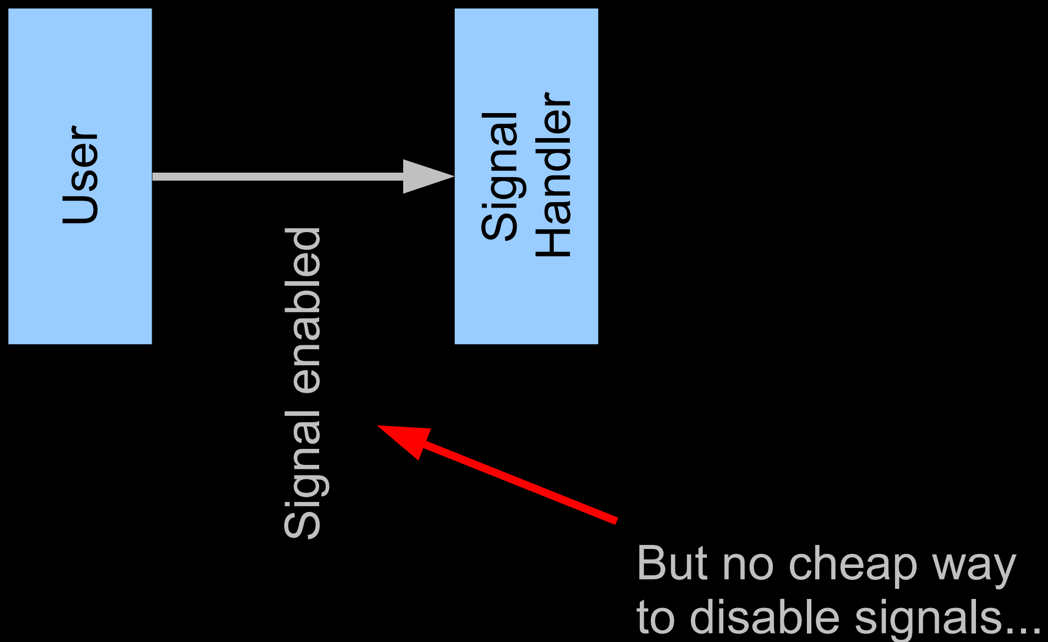
- Introduction to Performance, Scalability, and Real-Time Issues on Modern Multicore Hardware: Is Parallel Programming Hard, And If So, Why?
- Performance and Scalability Technologies in the Linux Kernel
- Creating Performant and Scalable Linux Applications
- Real-Time Technologies in the Linux Kernel
- Creating Real-Time Linux Applications

Overview

- Programming Environments in Linux Apps
- Synchronization Primitives
- Per-Thread Variables
- Solutions to the Existence Problem

Programming Environments in Linux Apps

Programming Environments in Linux Apps



Signal-Handler Synchronization Strategies

- Don't use signal handlers (my favorite)
- Use locking, accept expensive sigvec() calls
- Use non-blocking synchronization, accept restricted set of algorithms or great complexity
- Use RCU, accept read-only access from signal handler
- Some combination of the above

Synchronization Primitives

Synchronization Primitives (Partial POSIX)

- pthread_mutex_t
  - pthread_mutex_init()
  - pthread_mutex_lock()
  - pthread_mutex_trylock()
  - pthread_mutex_unlock()
- pthread_rwlock_t
  - pthread_rwlock_init()
  - pthread_rwlock_rdlock()
  - pthread_rwlock_tryrdlock()
  - pthread_rwlock_wrlock()
  - pthread_rwlock_trywrlock()
  - pthread_rwlock_unlock()

http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

Synchronization Primitives (Partial POSIX)

- pthread_cond_t
  - pthread_cond_init()
  - pthread_cond_wait()
  - pthread_cond_timedwait()
  - pthread_cond_signal()
  - pthread_cond_broadcast()

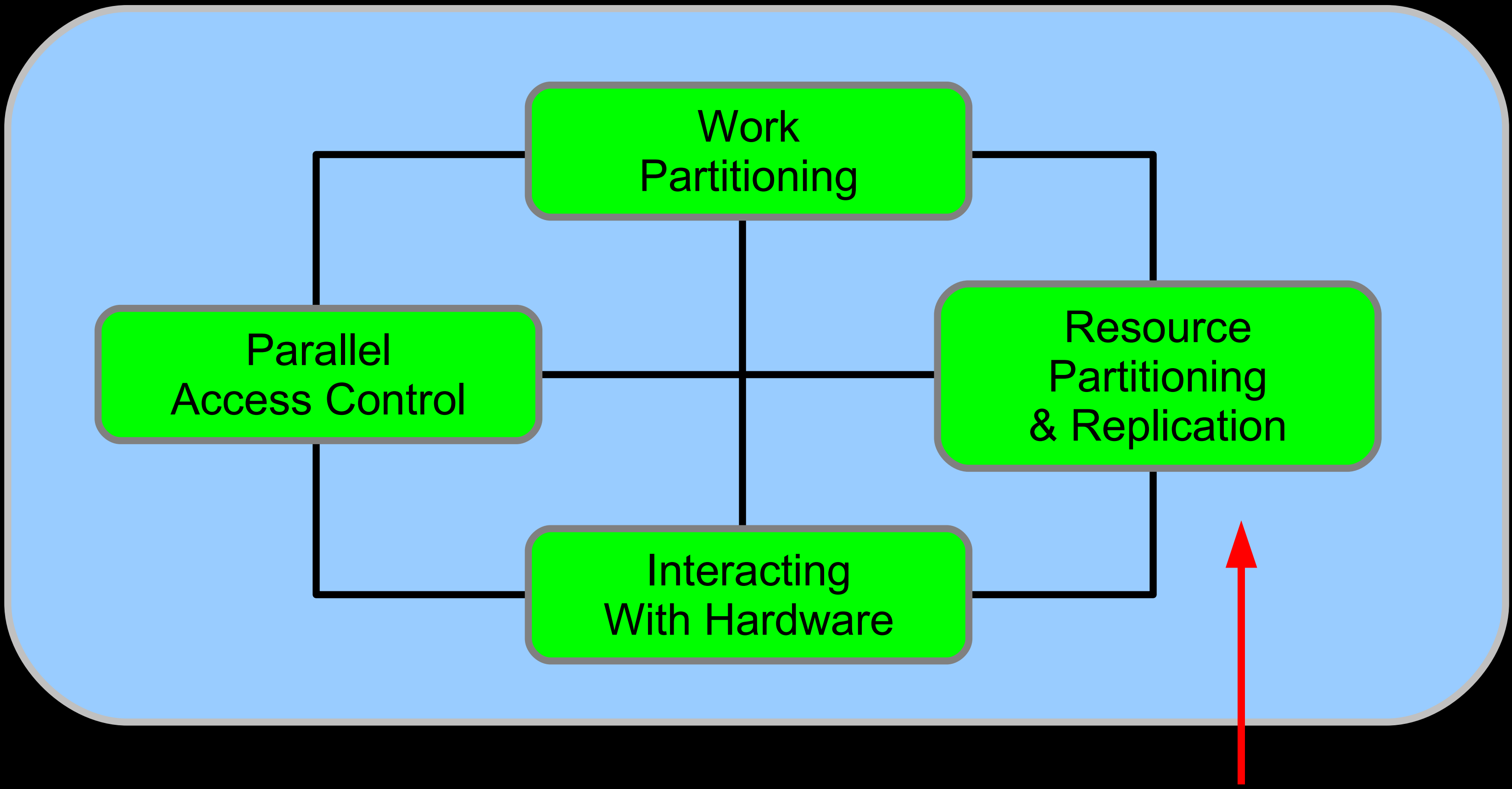
http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

Atomic __sync Intrinsics

- Fetch-and-op (returns old value):
  - type __sync_fetch_and_add (type *ptr, type value, ...)
  - type __sync_fetch_and_sub (type *ptr, type value, ...)
  - type __sync_fetch_and_or (type *ptr, type value, ...)
  - type __sync_fetch_and_and (type *ptr, type value, ...)
  - type __sync_fetch_and_xor (type *ptr, type value, ...)
  - type __sync_fetch_and_nand (type *ptr, type value, ...)
- Op-and-fetch (returns new value):
  - type __sync_add_and_fetch (type *ptr, type value, ...)
  - type __sync_sub_and_fetch (type *ptr, type value, ...)
  - type __sync_or_and_fetch (type *ptr, type value, ...)
  - type __sync_and_and_fetch (type *ptr, type value, ...)
  - type __sync_xor_and_fetch (type *ptr, type value, ...)
  - type __sync_nand_and_fetch (type *ptr, type value, ...)
- Compare-and-swap:
  - bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)
  - type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)
- Memory fences:
  - __sync_synchronize (...)
  - type __sync_lock_test_and_set (type *ptr, type value, ...)
  - void __sync_lock_release (type *ptr, ...)

http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html

Synchronization Primitives: Just as with Kernel



Do this first!!!!
Job #1 is *not* selecting primitives!

Per-Thread Variables

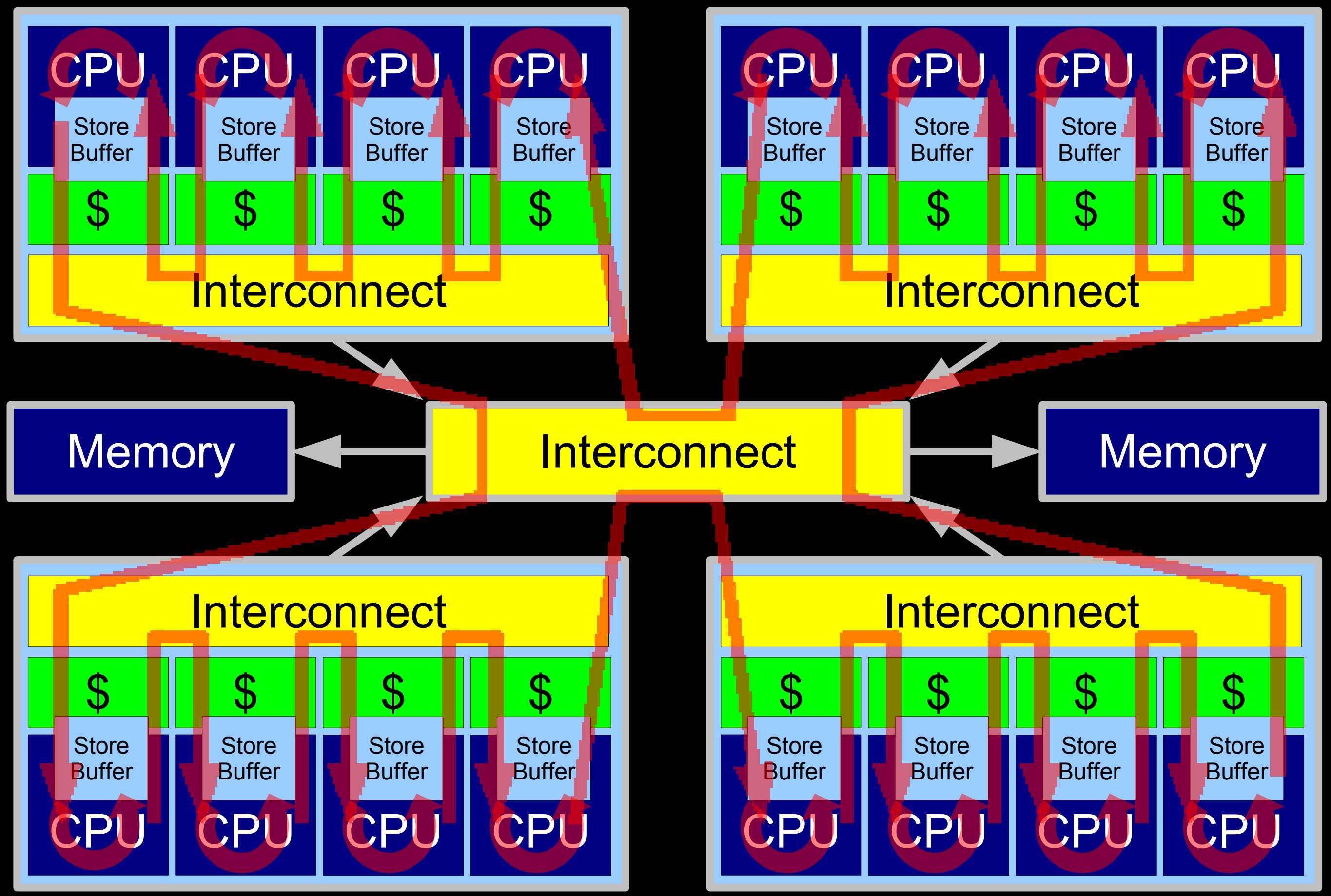
Per-Thread Variables

- Use the “__thread” storage class
- However, no standard way to access other thread's per-thread variables in C/C++
  - Is this important?
  - If so, what can you do about it?
- And why are per-thread variables so important?

Per-Thread Variables

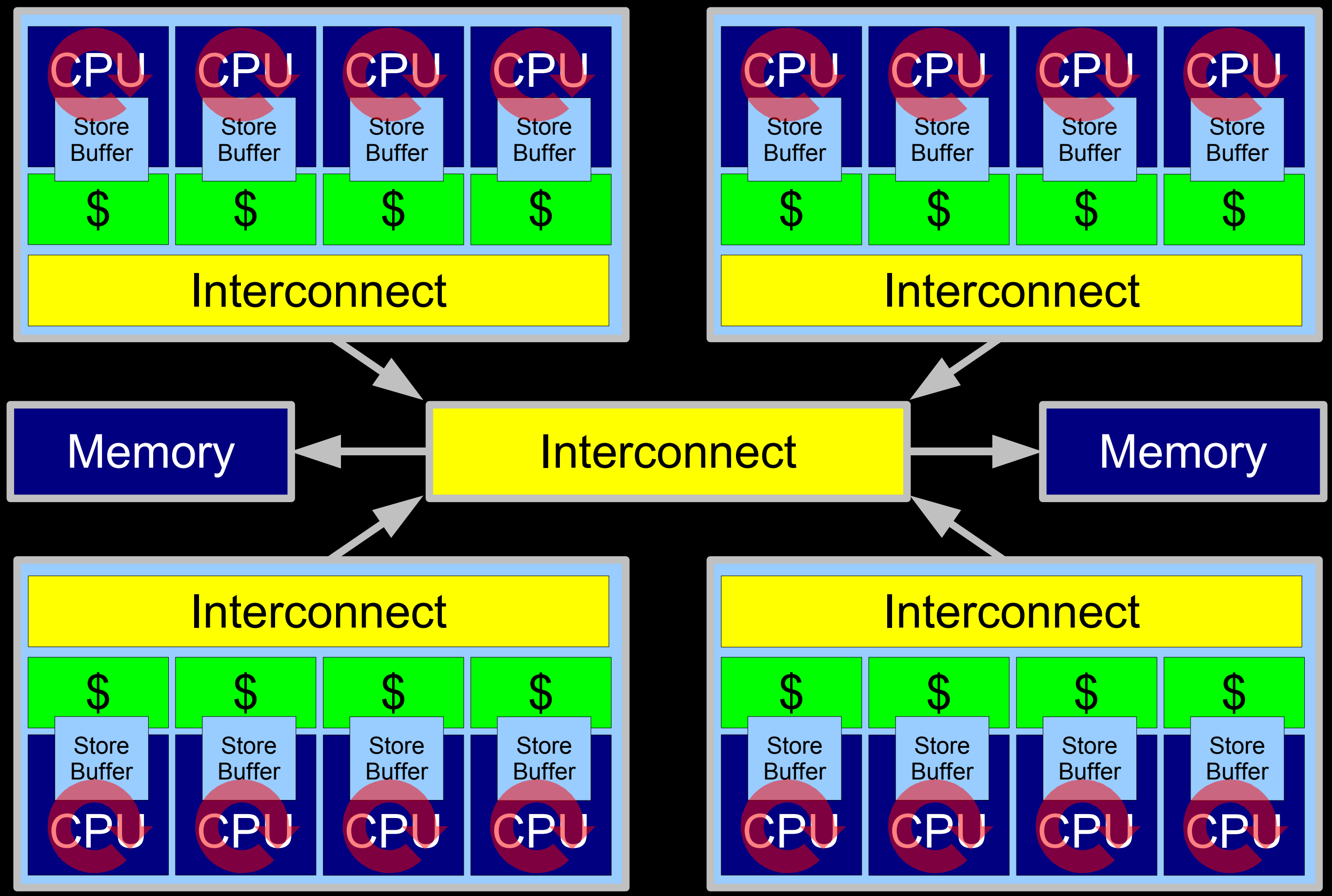
- Access to other thread's __thread variables
  - “Just say 'no'”: combine values after thread exits
    - Requires thread move __thread data before exit, of course
  - “Just say 'no'”: use communication primitives:
    - SysV message queues
    - UNIX-domain sockets
    - TCP/IP
    - POSIX signals
  - Create per-variable arrays containing pointers to each thread's corresponding variable
    - Each thread then records address of relevant variables
  - Use offsets (but good luck with shared libraries!!!)
  - Lobby for an enhancement to the standard

Atomic Increment of Global Variable



Lots and Lots of Latency!!!

Atomic Increment of Per-CPU Variable



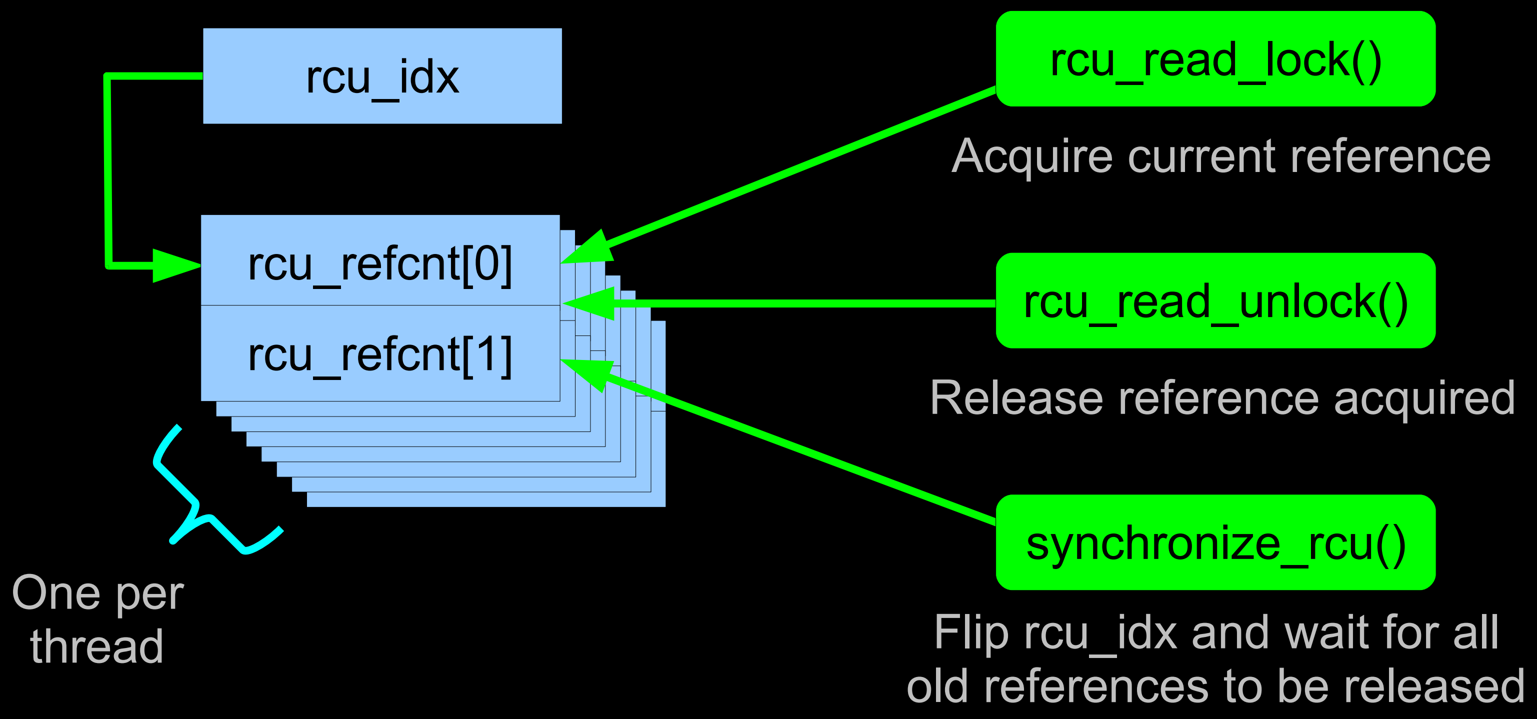
Little Latency, Lots of Increments at Core Clock Rate

Solutions to the Existence Problem

Solutions to the Existence Problem

- Use of “Running on CPU” as a reference does not translate well from kernel to user apps
  - No reliable way to suppress preemption
    - Existing user-level facilities are usually only hints
  - If there was a reliable way to suppress preemption, it would be subject to abuse
- Kernels have strictly enforced architectures
  - Can trust each kernel thread to reach a quiescent state in a timely fashion
    - Not so for user applications
    - Even less so for libraries – the application has not yet been thought of, much less architected!!!
- Thus must revisit reference-counting schemes

Per-Thread Reference Count Pair Data



Per-Thread Reference Count Pair Data

```
DEFINE_PER_THREAD(int, rcu_refcnt[2]);
atomic_t rcu_idx;
DEFINE_SPINLOCK(rcu_gp_lock);
DEFINE_PER_THREAD(int, rcu_nesting);
DEFINE_PER_THREAD(int, rcu_read_idx);
```

Per-Thread Ref-Count Pair Reader Primitives

```
static void rcu_read_lock(void)
{
  int i;
  int n;

  n = __get_thread_var(rcu_nesting);
  if (n == 0) {
    i = atomic_read(&rcu_idx);
    __get_thread_var(rcu_read_idx) = i;
    __get_thread_var(rcu_refcnt)[i]++;
  }
  __get_thread_var(rcu_nesting) = n + 1;
  smp_mb();
}

static void rcu_read_unlock(void)
{
  int i;
  int n;

  smp_mb();
  n = __get_thread_var(rcu_nesting);
  if (n == 1) {
    i = __get_thread_var(rcu_read_idx);
    __get_thread_var(rcu_refcnt)[i]--;
  }
  __get_thread_var(rcu_nesting) = n - 1;
}
```

Acquire reference

Release reference

Per-Thread Ref-Count Pair Updater Primitives

```
static void flip_counter_and_wait(int i)
{
  int t;

  atomic_set(&rcu_idx, !i);
  smp_mb();
  for_each_thread(t) {
    while (per_thread(rcu_refcnt, t)[i] != 0) {
      barrier();
    }
  }
  smp_mb();
}

void synchronize_rcu(void)
{
  int i;

  smp_mb();
  spin_lock(&rcu_gp_lock);
  i = atomic_read(&rcu_idx);
  flip_counter_and_wait(i);
  flip_counter_and_wait(!i);
  spin_unlock(&rcu_gp_lock);
  smp_mb();
}
```

Flip counter once

Wait for references to be released

Flip counter twice

Per-Thread Ref-Count Pair Issues

- No read-side memory contention
- No read-side atomic operations
- Complex read-side primitives
  - Array indexing and lots of operations, need something simpler
- Double counter flip and update-side lock slow
- No updater starvation

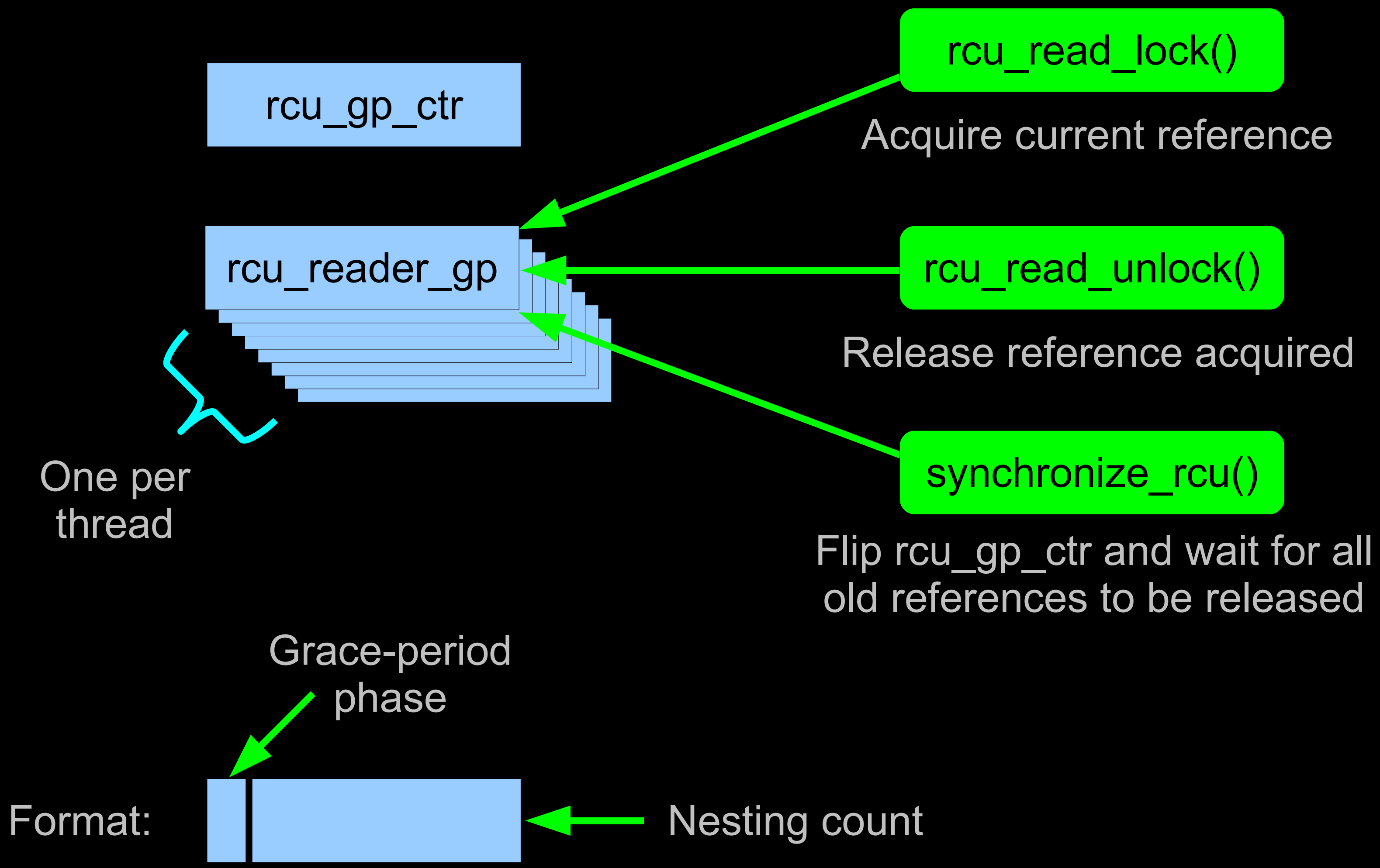
- So combine count and index into single per-thread variable

Per-Thread Ref-Count Pair Issues

- No read-side memory contention
- No read-side atomic operations
- Complex read-side primitives
  - Array indexing and lots of operations, need something simpler
- Double counter flip and update-side lock slow
- Not signal-safe
  - Cannot use both from mainline and signal handler
- No updater starvation

- So combine count and index into single per-thread variable

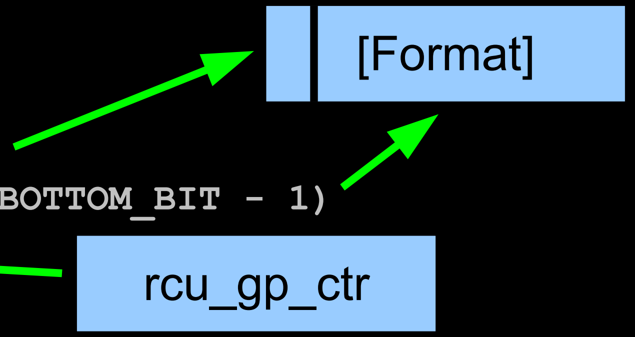
Per-Thread Phase-Counter Data



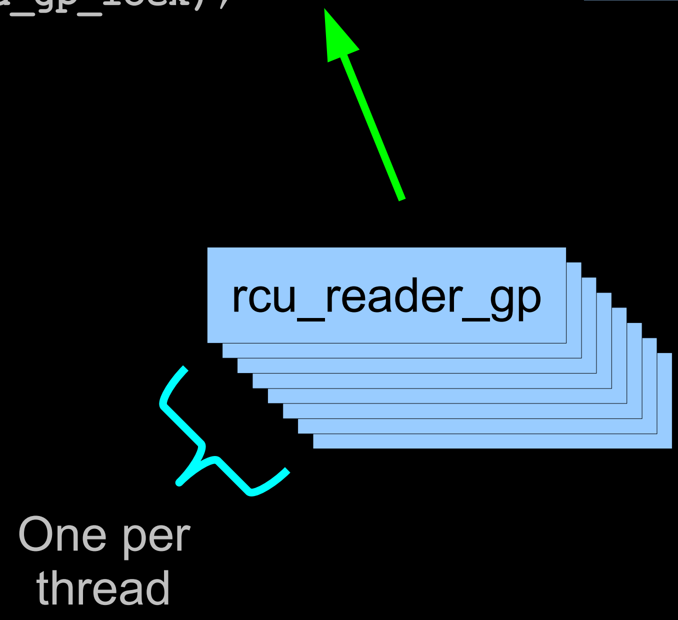
Per-Thread Phase-Counter Data

```
#define RCU_GP_CTR_BOTTOM_BIT 0x80000000
#define RCU_GP_CTR_NEST_MASK (RCU_GP_CTR_BOTTOM_BIT - 1)
long rcu_gp_ctr = 1;
DEFINE_PER_THREAD(long, rcu_reader_gp);
DEFINE_SPINLOCK(rcu_gp_lock);
```

Per-Thread Phase-Counter Data




```
#define RCU_GP_CTR_BOTTOM_BIT 0x80000000
#define RCU_GP_CTR_NEST_MASK (RCU_GP_CTR_BOTTOM_BIT - 1)
long rcu_gp_ctr = 1;
DEFINE_PER_THREAD(long, rcu_reader_gp);
DEFINE_SPINLOCK(rcu_gp_lock);
```



Per-Thread Phase-Counter Reader Primitives

```
static void rcu_read_lock(void)
{
  long tmp;
  long *rrgp;

  rrgp = &__get_thread_var(rcu_reader_gp);
  tmp = *rrgp;
  if ((tmp & RCU_GP_CTR_NEST_MASK) == 0)
    *rrgp = ACCESS_ONCE(rcu_gp_ctr);
  else
    *rrgp = tmp + 1;
  smp_mb();
}

static void rcu_read_unlock(void)
{
  long tmp;

  smp_mb();
  __get_thread_var(rcu_reader_gp)--;
}
```

Acquire reference

Release reference

Per-Thread Phase-Counter Updater Primitives

```
static inline int rcu_old_gp_ongoing(int t)
{
  int v = ACCESS_ONCE(per_thread(rcu_reader_gp, t));

  return (v & RCU_GP_CTR_NEST_MASK) &&
         ((v ^ rcu_gp_ctr) & ~RCU_GP_CTR_NEST_MASK);
}

static void flip_counter_and_wait(void)
{
  int t;

  rcu_gp_ctr ^= RCU_GP_CTR_BOTTOM_BIT;
  smp_mb();
  for_each_thread(t) {
    while (rcu_old_gp_ongoing(t)) {
      barrier();
    }
  }
}

void synchronize_rcu(void)
{
  smp_mb();
  spin_lock(&rcu_gp_lock);
  flip_counter_and_wait();
  barrier();
  flip_counter_and_wait();
  spin_unlock(&rcu_gp_lock);
  smp_mb();
}
```

Flip counter once

Wait for references to be released

Flip counter twice

Per-Thread Ref-Count Pair Issues

- No read-side memory contention
- No read-side atomic operations
- Simpler read-side primitives
  - Still have memory barriers
  - Can eliminate these by stealing a POSIX signal:
    - Upgrades compiler barrier to full memory barrier
    - git://lttng.org/userspace-rcu.git
    - Paper in preparation
- Double counter flip and update-side lock slow
  - Can batch grace periods similar to Linux kernel
- No updater starvation

Current State-of-the-Art for User-Mode RCU

- We do not yet have a single universal RCU algorithm for user-space applications
- However, there are three promising algorithms:
  - Full control of user application?
    - The use quiescent-state-based reclamation
    - Explicit quiescent states invoked periodically by all threads
    - Zero read-side overhead: free is a very good price!!!
  - Little control of application, but can use signal?
    - Mathieu Desnoyers's signal-based algorithm
    - Read-side overhead in the single-digit cycle range
  - No control of application, not even free signal?
    - Per-thread phase counter

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.
  - Joint work with Mathieu Desnoyers, Michel R. Dagenais, Alan Stern, and Jonathan Walpole

Questions?

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
- git://lttng.org/userspace-rcu.git
  - Mathieu Desnoyers's user-space RCU implementation
  - Also has quiescent-state-based implementation

- And there is *still* no substitute for running tests on real hardware!!!
  - For examples, see "CodeSamples" directory in: git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git
  - And "CodeSamples/defer" directory for user-level RCU implementations
    - rcu_nest32.[hc] has per-thread phase counter algorithm