Kernel Scalability

Adam Belay <abelay@mit.edu>

Motivation

- Modern CPUs are predominantly multicore
- Applications rely heavily on kernel for networking, filesystem, etc.
- If kernel can’t scale across many cores, applications that rely on it won’t scale either
- Have to be able to execute system calls in parallel

Problem is sharing

- OS maintains many data structures
  - Process table, file descriptor table, buffer cache, scheduler queues, etc.
- They depend on locks to maintain invariants
- Applications may contend on locks, limiting scalability

OS evolution

- Early kernels depended on a single “big lock” to protect kernel data
- Later, kernels transitioned to fine-grained locking
- Now, many lock-free approaches are used too
- Extreme case: Some research kernels attempted to share nothing
  - E.g. FOS and Barrelfish
  - Could potentially run without cache-coherence
  - Downside: Poor load balancing

Agenda for today

1. Read-copy-update (today's reading assignment)
2. Per-CPU reference counters
3. Scalable commutativity rule

Read-heavy data structures

- Kernels often have data that is read much more often than it is modified
  - Network tables: routing, ARP
  - File descriptor arrays, most types of system call state
  - **RCU optimizes for these use cases**
  - Over 10,000 RCU API uses in the Linux Kernel!

1. Goal: Concurrent reads even during updates
2. Goal: Low space overhead
3. Goal: Low execution overhead

Plan #1: spin locks

- Problem: Serializes all critical sections
- Read-only critical sections would have to wait for other read-only sections to finish
- Idea: Could we allow parallel readers but still serialize writers with respect to both readers and others writers

Plan #2: Read-write locks

- A modification to spin locks that allows parallel reads
- How to change spin lock implementation to support this feature?

Read-write lock implementation

```
typedef struct { volatile int cnt; } rwlock_t;

void read_lock(rwlock_t *l) {
  int x;
  while (true) {
    x = l->cnt;
    if (x < 0) // is write lock held?
      continue;
    if (CMPXCHG(&l->cnt, x, x + 1))
      break;
  }
}

void read_unlock(rwlock_t *l) {
  ATOMIC_DEC(&l->cnt);
}
```

Read-write lock implementation

```
typedef struct { volatile int cnt; } rwlock_t;

void write_lock(rwlock_t *l) {
  int x;
  while (true) {
    x = l->cnt;
    if (x != 0) // is the lock held?
      continue;
    if (CMPXCHG(&l->cnt, 0, -1))
      break;
  }
}

void write_unlock(rwlock_t *l) {
  ATOMIC_INC(&l->cnt);
}
```

Q: Why check before CMPXCHG?

```
typedef struct { volatile int cnt; } rwlock_t;

void write_lock(rwlock_t *l) {
  int x;
  while (true) {
    x = l->cnt;
    if (x != 0) // is the lock held?
      continue;
    if (CMPXCHG(&l->cnt, 0, -1))
      break;
  }
}

void write_unlock(rwlock_t *l) {
  ATOMIC_INC(&l->cnt);
}
```

Why?

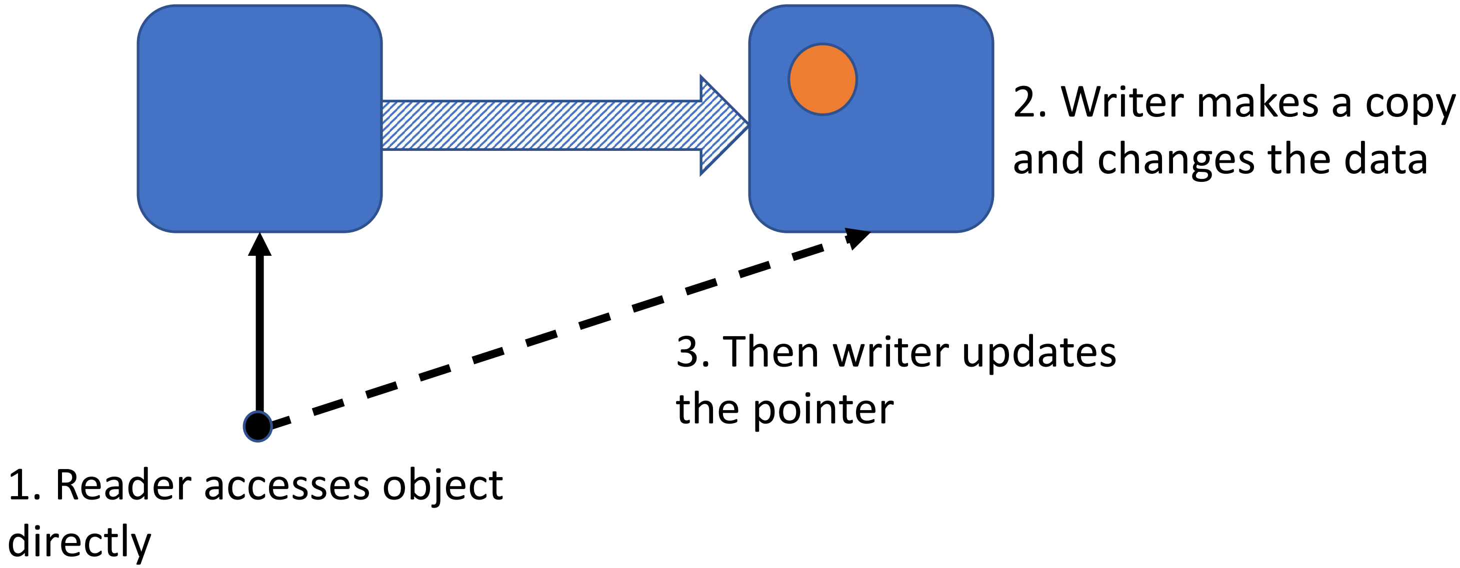
Q: What's the execution overhead?

Q: What’s the execution overhead?

- Every reader uses CMPXCHG instruction
    - S -> M cache coherence state transition
    - Find + invalidate messages for contended read_lock()
    - And for read_unlock() too!
- If writer holds lock, readers must spin and wait
    - Violates goal of concurrent read, even during updates

Plan #3: Read-copy-update (RCU)

- Readers just access objects directly (no locks)
- Writers make a copy of object, change it, then update the pointer to the new copy



When to free old objects?

- At any given moment, readers could be accessing the latest copy or older copies of an object
- Idea: Can safely free objects when they are no longer "reachable"
- Usually only one pointer to an RCU object
  - Can't be copied, stored on the stack, or in registers (except inside critical sections)
- Need to define a "quiescent period", after which it's safe to free
  - Wait until all cores have passed through a context switch
  - Pointer can only be dereferenced inside a critical section
  - Read critical sections disable preemption

Q: Why disable preemption during RCU read critical sections?

Q: Why disable preemption during RCU read critical sections?

- If we didn’t, waiting for all cores to context switch wouldn’t be an effective quiescent period
- A task could still hold a pointer to an RCU object while it is preempted
  - Hard to determine when its safe to free
  - Unless we wait until all current tasks are killed
- Need to define a read critical section such that references to RCU objects can’t persist outside the section

RCU API (simplified)

```
void rcu_read_lock() {
  preempt_disable[cpu_id()]++;
}

void rcu_read_unlock() {
  preempt_disable[cpu_id()]--;
}

void synchronize_rcu(void) {
  for_each_cpu(int cpu)
    run_on(cpu);
}
```

Real RCU API

- **rcu_read_lock():** Begin an RCU critical section
- **rcu_read_unlock():** End an RCU critical section
- **synchronize_rcu():** Wait for existing RCU critical sections to complete
- **call_rcu(callback, argument):** Call the callback when existing RCU critical sections complete
- **rcu_dereference(pointer):** Signal the intent to dereference a pointer in an RCU critical section
- **rcu_dereference_protected(pointer, check):** signals the intent to dereference a pointer outside of an RCU critical section
- **rcu_assign_pointer(pointer_addr, pointer):** Assign a value to a pointer that is read in RCU critical sections

How to synchronize writes?

Against other writers:

- Allow only one writer
- Or just use normal synchronization like locks!

Against readers: (memory order matters)

- Writers must fully finish writes to new object before updating pointer
- Readers must not reorder reads such that contents of an object are read before its pointer (NOTE: the DEC Alpha can actually do this!)
- **rcu_dereference()** and **rcu_assign_pointer()** automatically insert the appropriate compiler and memory barriers

Example RCU usage

- Imagine a simple online store
- Need an object to represent the price of each item

```
typedef struct {
  const char *name;
  float price;
  float discount;
} item_t;
__rcu item_t *item;
lock_t item_lock;
```

NOTE: total_cost = price - discount

Example RCU usage (reader)

```
float get_cost(void) {
  item_t *p;
  float cost;
  rcu_read_lock();
  p = rcu_dereference(item); // read
  cost = p->price – p->discount;
  rcu_read_unlock();
  return cost;
}
```

Example RCU usage (writer)

```
void set_cost(float price, float discount) {
  item_t *oldp, *newp;
  spin_lock(&item_lock);
  oldp = rcu_dereference_protected(item, spin_locked(&item_lock));
  newp = kmalloc(sizeof(*newp));
  *newp = *oldp; // copy
  newp->price = price;
  newp->discount = discount;
  rcu_assign_pointer(item, newp); // update
  spin_unlock(&item_lock);
  rcu_synchronize();
  kfree(oldp); // free
}
```

RCU is a very powerful tool

1. Works with more complex data structures like linked lists and hash tables
2. Most common use case is as an alternative to read-write locks
3. Can be used to wait for parallel work to complete
4. Can be used to elide reference counting

See paper for many more examples

Does RCU achieve its goals

1. Goal: Concurrent reads even during updates?
   - Yes! Reads are never stalled by updates.
2. Goal: Low space overhead?
   - Yes! An RCU pointer is the same size as an ordinary pointer. No extra synchronization data is required.
   - However, objects can't be freed until quiescent period has passed. Forcing this to happen immediately incurs overhead.
3. Goal: Low execution overhead?
   - For readers, RCU has practically no execution overhead!
   - For writers, RCU adds a slight amount of overhead due to allocation, freeing, and copying. In practice, this overhead is modest.
   - Fine-grained locking can help to make updates concurrent.

Reference counters

- Counts number of pointer references to an object
- When count reaches zero, safe to free object
- Challenge: involves true sharing
  - Many resources in kernel are reference counted
  - Often a scaling bottleneck (once other bottlenecks are removed)

Standard approach

```
typedef struct { int cnt; } kref_t;

void kref_get(kref_t *r) {
  WARN_ON(r->cnt == 0);
  ATOMIC_INC(&r->cnt);
}

void kref_put(kref_t *r,
              void (*release)(kref_t *r)) {
  if (ATOMIC_DEC_AND_TEST(&r->cnt))
    release(r);
}
```

What's the execution overhead?

What's the execution overhead?

- **kref_get()** and **kref_put()** both require exclusive ownership of cache-line (i.e. place it in M state)
- Tons of cache-line bouncing if object is referenced frequently

Idea: Per-cpu reference counters

- Maintain an array of counters, one per core
- **percpu_ref_get()** and **percpu_ref_put()** operate on the local core’s array entry
  - Data written by only one core, no cache-line bouncing
- **percpu_ref_kill()** reverts to normal, ”shared” reference counting
- Performance improved only if most references added and removed while killer’s reference remains held
  - Often this is true!

How to implement kill?

1. Set shared refcount to a bias value
2. Atomically set a “kill” flag in shared refcount
3. Wait a quiescent period, after which flag is visible to all reference counters
   - How? **rcu_synchronize()** is perfect for this!
4. Sum all refcounts in per-core array and atomically add total to ref count in shared structure
5. Finally, atomically subtract the bias value
6. When shared refcount reaches zero, free object

Can any program be made scalable?

- Today we saw two examples of highly scalable algorithms
- But only applicable in certain situations
- In general, when is it possible to make code scalable?

Scalable commutativity rule

- rule: if two operations, commute then there exists a scalable (conflict-free) implementation
- intuition: if ops commute, order doesn't matter communication between ops must be unnecessary
- Caveat: Scalable implementations may still be possible for operations that fail the rule. However, If they pass the rule, scalability is definitely possible
- See http://pdos.csail.mit.edu/papers/commutativity:sosp13.pdf

Insight: The key to scalability is good interface design

- Example: POSIX requires open() system call to return the smallest available fd number
- Do two open() calls commute?
- How would you change open() to make it more scalable?

Conclusion

- RCU enables zero-cost read-only access at the expense of slightly more expensive updates
  - Very useful for read-mostly data (extremely common in kernels)
- Reference counting can be almost free in cases where objects are long-lived and one task can be designated as the killer
  - Per-CPU refcounting sacrifices space for speed
- Scalable commutativity rule provides guideline for designing scalable interfaces
  - Operations in both RCU and Per-CPU refcount commute!