Read Copy Update

MIT 6.S081 Lec 23
Michael

Agenda

- Problem Motivation
- Read Write Lock
- Read Copy Update

Workload in Question

When:

- **write does happen but infrequently**
    - mostly read, rarely updated linked list accessible by kernel code
    - read the data, do a lot of computation and finally publish the result(scientific computing)
    - dynamically growing container: lots of read, but infrequently resizing

Since:

- we have many cores and we want to keep these cores busy

We need:

- concurrency control

Approaches

Performance Comparison

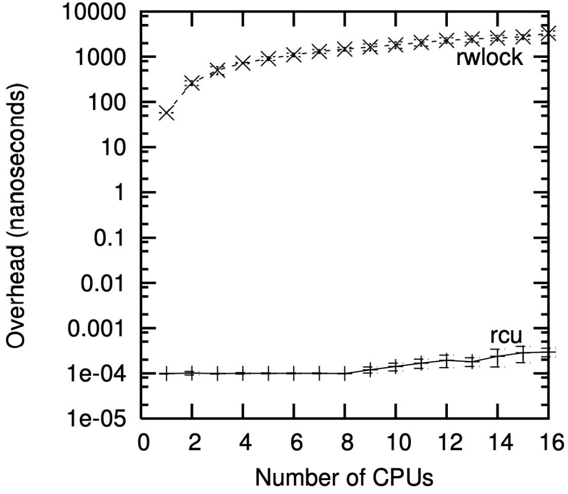
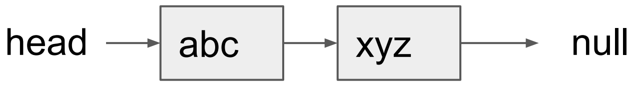


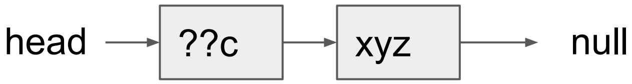
Figure 8: The overhead of entering and completing an RCU critical section, and acquiring and releasing a read-write lock.

Linked List: What can go wrong without synchronization

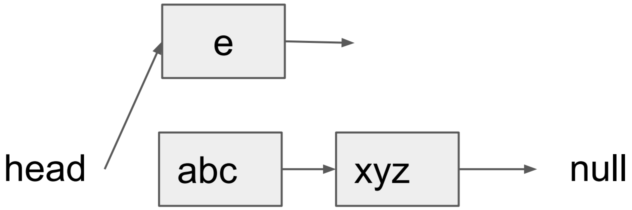


in kernel, many readers one writer, without sync:

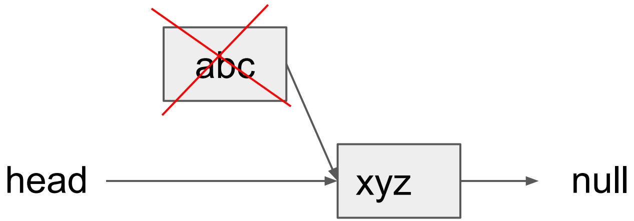
1. modify:



2. add node:



3. delete node:



Spinlock

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
void
acquire(struct spinlock *lk)
{
  push_off(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // On RISC-V, sync_lock_test_and_set turns into an atomic swap:
  //   a5 = 1
  //   s1 = &lk->locked
  //   amoswap.w.aq a5, a5, (s1)
  while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen strictly after the lock is acquired.
  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Record info about lock acquisition for holding() and debugging.
  lk->cpu = mycpu();
}
```

1

2

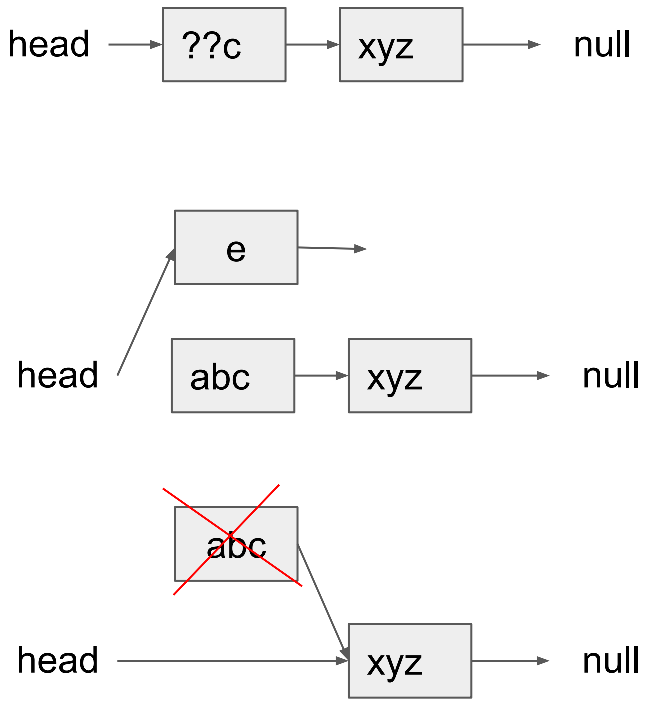
```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->cpu = 0;

  // Tell the C compiler and the CPU to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other CPUs before the lock is released,
  // and that loads in the critical section occur strictly before
  // the lock is released.
  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code doesn't use a C assignment, since the C standard
  // implies that an assignment might be implemented with
  // multiple store instructions.
  // On RISC-V, sync_lock_release turns into an atomic swap:
  //   s1 = &lk->locked
  //   amoswap.w zero, zero, (s1)
  __sync_lock_release(&lk->locked);

  pop_off();
}
```

RWLock: API

- Reader:
    - reader_lock(l)
    - reader_unlock(l)
- Writer:
    - writer_lock(l)
    - writer_unlock(l)

RWLock: Implementation

```
//
// A simplified version of Linux's read/write lock.
//

// n=0  -> not locked
// n=-1 -> locked by one writer
// n>0  -> locked by n readers
struct rwlock {
  int n;
};

r_lock(l):
  while 1:
    x = l->n
    if x < 0
      continue
    if CAS(&l->n, x, x + 1)
      return

// CAS(p, a, b) is atomic compare-and-swap instruction
//   if *p == a, set *p = b, return true
//   else return false

w_lock(l):
  while 1:
    if CAS(&l->n, 0, -1)
      return
```



RWLock: limitations

- High cache invalidation cost
  - Successful reader lock acquisition requires a cache invalidation for all other cores.
  - Does not scale well with the number of cores.
- Can we avoid this cost?
  - If the readers does not write to the shared integer, can there be speed up?

RCU: API

Readers:

- rcu_read_lock() // enter rcu critical section
- rcu_read_unlock() // leave rcu critical section

Writers:

- synchronize_rcu()

RCU Idea 1: Publishing Protocol

- Writer not allowed to directly modify the linked list instead, it will
  - read the current data(‘s next pointer),
  - copy this information to the new node,
  - and update the current data(‘s next pointer)
- Only works for data structures that can be atomically updated via one write
  - So doubly linked list is not a good data structure for RCU.
  - Tree is a good data structure
  - Not always a pointer: https://www.youtube.com/watch?v=rxQ5K9lo034@14:01
    - something if you have it gives you access to the rest of the data
    - reader gets it, gives it access to whatever data it reveals.
  - Some readers sees the old data, others see the new data; very common.

RCU Idea 2: Memory Barrier

- There is no “after this(commiting write) then”
  - Compiler & CPU will reorder instructions
  - Use memory barrier to ensure the “then” semantic
  - **Because we did not use a lock**(or other sync mechanism)
- For writer
  - the barrier goes before the “commiting write”
- For readers
  - the barrier goes after following the pointer
  - so later read from the node reads the correct(non-cached) value

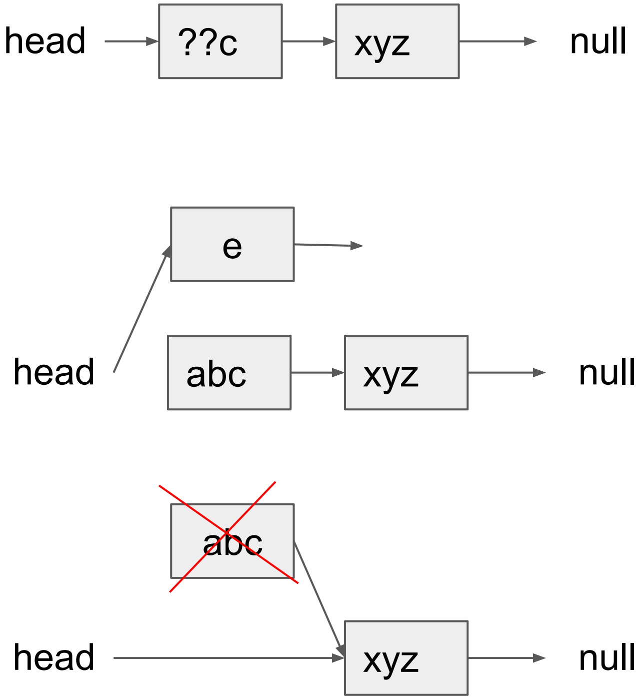
RCU Idea 3: Delayed Memory Reclaim

- This is the protocol(collaboration between reader/writer) that makes RCU
- Readers
    - announce access by calling reader_lock
    - promise not to yield CPU in RCU critical session
        - because context switch is dual purposed to signal a reader is done accessing the shared data structure(reader_unlock)
    - Will always get things via this root token. If used up, traverse via the new root token up to the same location.
- Writer
    - delay free until every call has context switched at least once → by calling rcu_synchronize to start waiting

```
// list reader:

  rcu_read_lock()
  e = head
  while(p){
    e = rcu_dereference(e)
    look at e->x ...
    e = e->next
  }
  rcu_read_unlock()

// replace the first list element:

  acquire(lock)
  old = head
  e = alloc()
  e->x = ...
  e->next = head->next
  rcu_assign_pointer(&head, e)
  release(lock)

  synchronize_rcu()
  free(old)
```

```
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);
}
```

Figure 2: A simplified version of the Linux RCU implementation.



References

1. CppCon 2017: Fedor Pikus “Read, Copy, Update, then what? RCU for non-kernel programmers”
2. 6.S081 Fall 2020 Lecture 23: RCU - YouTube