6828 2011 L17: Lock-free coordination

Required reading: Linux scalability

Plan
  programming without locks
    example: lists
  rcu: plan widely-used in Linux
  paper discussion

Problem:
  Locks limit scalability (serialize lock holders)
  Transferring lock from one holder to another is expensive
    Even when locks are scalable
   Can we do better?
      lock-free concurrent data structures

Example: a stack
  Sequential program:

```
struct element {
    int key;
    int value;
    struct element *next;
};

struct element *top;

void push(struct element *e) {
    e->next = top;
    top = e;
}

struct element *pop(void) {
    struct element *e = top;
    top = e->next;
    return e;
}

int search(int key) {
    struct element *e = top;
    while (e) {
        if (e->key == key)
            return e->value;
        e = e->next;
    }
    return -1;
}
```

this is clearly not going to work on a concurrent system
   what is a race?

global spinlock: correct, but what about performance?

how many concurrent ops? just one CPU at a time
bus interactions? bounce cache line for both reads, writes

global read-write lock: correct, but what about performance
how many concurrent ops? theoretically, could do one per CPU
bus interactions? bounce cache line for both reads, writes
draw timing diagram of CPU interactions
we might be slower on two CPUs than on a single CPU!

is it going to get better if we allocate a read-write lock for each item?
concurrency still possible but penalty even worse: N_{elem}
cache line bounces for each search traversal

other possible solutions
partition data into n lists (e.g., n free lists, one per core)
if one runs out of memory, steal memory from other core's free lists

why do we want to avoid locks?
performance
complexity
deadlock
priority inversion

what's the plan?
reduce the operation we want to perform to some atomic x86 instruction
x86 LOCK prefix makes many read-modify-write instructions atomic
simple example: implement atomic counters by adding LOCK prefix
most general thing is cmpxchg (which we seen many times now)

```
int cmpxchg(int *addr, int old, int new) {
    int was = *addr;
    if (was == old)
        *addr = new;
    return was;
}
```

cmpxchg can be used to implement locks, but we can also use it
directly for concurrent, correct access to the linked list.

example: concurrent stack with out locks

```
void push(struct element *e) {
again:
    e->next = top;
    if (cmpxchg(&top, e->next, e) != e->next)
        goto again;
}

struct element *pop(void) {
again:
    struct element *e = top;
    if (cmpxchg(&top, e, e->next) != e)
        goto again;
    return e;
```

}

search can be the same as in the non-concurrent case (almost..)

why is this better than not having locks?
  readers no longer generate spurious updates, which incurred performance hit
  may be not having to think about deadlock is great

problem 1: lock-free data structures require careful design, hw support
  suppose we want to remove arbitrary elements, not just the first one
  what could go wrong if we try to use the same cmpxchg?
    race condition when two processors
    one processor deletes a node next to the other node being deleted
  need DCAS (double-compare-and-swap) to implement remove properly
    must make sure neither previous nor next element changed
    x86 hardware doesn't have DCAS
  one approach:
    on delete, mark node's next pointer to signal node is deleted
    now the CAS that removes a subsequent node if this previous node is delete concurrently too

problem 2: memory reuse
  when can we free a memory block, if other CPUs could be accessing it?
    other CPU's search might be traversing any of the elements

  reusing a memory block can corrupt list
    stack contains three elements
      top -> A -> B -> C
    CPU 1 about to pop off the top of the stack,
      preempted just before cmpxchg(&top, A, B)
    CPU 2 pops off A, B, frees both of them
      top -> C
    CPU 2 allocates another item (malloc reuses A) and pushes onto stack
      top -> A -> C
    CPU 1: cmpxchg succeeds, stack now looks like
      top -> B -> C
    this is called the "ABA problem"
     (memory switches from A-state to B-state and back to A-state without being able to tell)

  strawman solution for specific problem (actually used by some systems):
    type-stable memory (stack-elements never become non-stack-elements)
    each stack element has a reuse counter
    include generation# along with each pointer (so that cmpxchg notices)

  but for more complex data structures this becomes hard to solve
    for example, readers must check that data structure hasn't been freed

  need a general-purpose garbage-collection plan
    see below

RCU: ready-copy update
  concurrent data structure approach used in the Linux kernel for read-intensive concurrent data structures
    many usages in kernel, for hash tables, lists, etc.
  3 components:
    - lock-free readers, but serialize writers using locks

- writers don't update in place
  on write, make copy
  update copy
  switch one pointer atomically
  => readers seen atomic switch from old to new
- garbage collection to allow readers to read old versions and avoid ABA problem

Can be applied to many data structures

Example: RCU stack

```
void
rcu_push(int k, int v)
{
  acquire(l);
  push(k, v);
  release(l);
}

elem_t*
rcu_pop()
{
  elem_t *e;
  acquire(l);
  e = pop();
  release(l);
  return e;
}

int
rcu_search(k)
{
  int v;
  v = search(k);
  return v;
}
```

problem: garbage collection
  after pop, can we just free e?
  no! such may be still looking at it!
  in a garbage-collected language, the garbage collection will make this safe

many different gc schemes possible for C
  on free, put element on a list for delayed freeing
  remove an element when sure that no reader is looking at the element

one scheme: epochs
   global epoch counter
   writers increment epoch number
     they hold lock anyway
   on free of element, record epoch number in element
  at beginning of read, threads stores global epoch in thread-local state
  gc: free all elements with epoch # < minimum of all readers

example stack

```
int
rcu_search(k)
{
  int v;
  rcu_start_read()
  v = search(k);
  rcu_end_read()
  return v;
}

rcu_begin_read(int tid)
{
  epochs[tid].epoch = global_epoch;
  __sync_synchronize();
}

void
rcu_end_read(int tid)
{
  epochs[tid].epoch = INF;
}

void free(e)
{
    // record global epoch into e
}

void gc()
{
  unsigned long min = global_epoch;

  for (i = 0; i < nthread; i++) {
    if (min > epochs[i].epoch)
      min = epochs[i].epoch;
  }
  // free all e whose epoch < min
}
```

RCU
  concurrent code very similar to sequential code
  general approach for read-intensive data structures
  but serializes all writers
    ok for stack case
    but what if inserting at the beginning and end of list concurrently?
  may make unnecessary copies

Future
  lock-free data structures for readers and writers?
  exist for lists, hash tables, skip lists, etc.

Paper
  What is the current state of scalability in current kernels?
  Many short sections

Partitioning and replication of data structures for scalability
RCU for lock-free sections

Example: exim
Many processes do path name lookup concurrent
Linux has a directory entry cache for mappings from directory entry to inode #
[i#,name] -> file struct
RCU has been used for the directory entry cache
lookups for different pathnames run in parallel without locks
lookups for the same dir entry of a lock

Problem:
Look ups for "/", however, hit the same dentry
Contention on spinlock, and collapse.
Not simple fix:
reads increase refcnt
writes (e.g., rename), too + [i#,name]->file struct

Strawman solution: scalable locks
Avoid collapse, but not more scalability

Solution: lock-free dentry?
add generation #
write: lock dentry, g = gen, gen=0, change, gen=g+1
read:
g <- gen#
if g== 0, use locks (some core is modifying)
copy relevant fields
g1 <- gen#
if g1 != g locking protocol
if match, refcnt++ (if refcnt != 0)
Gen # is well-known trick
Result: all lookups run lock free

Future: ?