Lecture Topics

- the concept of consistency models
- serializability and linearizability
- non-blocking and wait-free implementations
- read-copy update
- a few lock algorithms

Administrivia

- …
The Concept of Consistency Models [this slide is review]

- a simple thread interaction model
  - starring Byron’s Parasina and Azo, threads P and A
  - Parasina and Azo work concurrently, so their tasks can be logically interleaved...
- sequential consistency
  - instructions from different threads appear to execute in some interleaving
    - all instructions from any given thread are in execution order
    - all processors perceive the same total order of instructions
  - nice model, but not really how many modern machines work
• some common variations and examples
  – operations from a single processor may be reordered
    • execute load before earlier store if addresses known not to conflict
    • What if load is inside a critical section, and store is the lock
      acquisition for the critical section?
  – perception of execution may be different on different processors
    • two stores go to different memory banks
    • Why wait for the first to finish before sending the second?
    • first is data, second is synchronization flag
      • Now can we change the perceived order of stores?
      • What if another processor loads data after flag changes
        but before data is stored?
Serializability and Linearizability (definitions)

• properties of parallel data structures

• serializability
  – a data structure with operations is serializable if
    • the state of any possible parallel execution, including
      – arguments to operations
      – return values from operations
      – internal state of data structure
    • is equivalent to some serialization of the operations applied
  – in other words
    • there must exist an ordering of operations
    • such that the operations executed in parallel appear to have executed serially in that order

• linearizability
  – paper difficult to understand in my opinion
  – addresses lack of causality with serializability
    • parallel data structure P
    • thread 1 applies op A to P then signals thread 2
    • thread 2 waits for thread 1’s signal and then applies op B to P
    • can the serial order place op B before op A? YES.
  – With linearizability, the answer to the last question is NO.
  – in practice, hard to provide this property
  – if provided, gives some other nice properties “for free,” such as composition of linearizable operations/data structures
Serializability and Linearizability

- Vijay Karamcheti & Andrew Chien developed an interesting non-blocking queue for the Cray T3D
  - rationale: avoid output contention/cache flushing
  - let receiver “pull” data to itself
  - sender simply chains object pointer to end of queue

- Why look at this algorithm?
  - simple but interesting non-blocking, many-to-one queue
  - easy to understand example of something that is
    - serializable, but NOT linearizable
    - serializable means operations appear to have occurred in some serial order
    - linearizable further constrains this order as follows
      - for any operation A
      - that finishes far enough in advance of some operation B
      - for communication to occur between the processor that executed A and the processor that executed B
      - B cannot occur before A in the serial order

- Cray T3D (Alpha-based) provided a swap synchronization primitive:

```c
int64_t SWAP (int64_t* addr, int64_t new)  
{  
    int64_t old = *addr;  
    *addr = new;  
    return old;  
}  
```

- (SWAP executes atomically with respect to all other instructions)
• enqueue and dequeue routines (translated to C++; not as methods for clarity)

    void enqueue (Queue* q, void* data)
    {
        // message containing pointer to data and
        // next field equal to NULL
        Message* msg = new Message (data, NULL);

        // we might prefer copy semantics for the
        // data, but need to choose message size for
        // that implementation

        // (memory and compiler barriers implied by SWAP)

        Message* old_tail =
            (Message*)SWAP (&q->tail, (int64_t)msg);

        // (dependence on SWAP return value suffices for
        // barriers)
        old_tail->next = msg;
    }

    bool dequeue (Queue* q, void** dptr)
    {
        Message* head = q->head;
        Message* next = head->next;
        if (NULL == next) {
            return false; // underflow
        }
        q->head = next;
        delete head;
        *dptr = next->data;
        return true;
    }
- queue maintains a sentinel element at all times
- when message received
  - previous sentinel discarded
  - message being delivered becomes new sentinel

![Diagram of queue with head, tail, and sentinel](image)

- consider the following scenario

![Timeline showing SWAP, slow sender, fast sender, and receiver](image)

  - receiver sees empty queue (slow sender hasn’t filled sentinel’s next field)
  - serial order MUST start with receiver
  - but fast sender finished before receiver started, so not allowed
    (fast sender may have used another channel to communicate/synch. completion to receiver)
  - therefore, NOT linearizable
Non-Blocking and Wait-Free Implementations

- non-blocking fetch-and-increment implementation using CAS
  ```c
  int32_t fetch_and_increment_nb (int32_t* addr)
  {
    int32_t value, next;
    do {
      value = *addr;
      next = value + 1;
      while (!CAS (addr, value, next);
    } while (!CAS (addr, value, next);
    return value;
  }
  ```

- wait-free fetch-and-increment implementation using CAS
  - cooperative approach based on Herlihy’s general method
  - copy block holds data + interprocess synch. info.
  - \( \text{NUM\_THREADS} + 1 \) copy blocks exist at all times
  - \( \text{ME} \) is a unique thread id for each thread
  ```c
  struct cblock_t {
    int32_t value; // value being fetched & incremented
    int32_t responses[NUM_THREADS]; // per-thread return values
    bool toggle[NUM_THREADS]; // per-thread completion handshake
  };
  struct epoch_t { // small number of cblocks! ABA looms!
    cblock_t* ptr;
    int32_t count;
  };
  struct object_t {
    bool announce[NUM_THREADS]; // per-thread announcement of op
    epoch_t cblock;
  };
  static cblock_t* my_block[NUM_THREADS];
  ```
[write code, then talk about it]

fetch_and_increment_wf (object_t* obj)
{
    bool toggle = !obj->announce[ME];
    obj->announce[ME] = toggle;
    // mem + comp. barriers (announcement must be visible)
    for (try = 0; 2 > try; try++) {
        if (obj->cblock.ptr->toggle[ME] == toggle) {
            if (obj->cblock.ptr->toggle[ME] == toggle) {
                break;
            }
        }
    }
    epoch_t old_cblock = obj->cblock;
    memcpy (my_cblock[ME], old_cblock.ptr,
            sizeof (*my_cblock[ME]));
    for (int32_t i = 0; NUM_THREADS > i; i++) {
        if (obj->announce[i] != my_cblock[ME]->toggle[i]) {
            my_cblock[ME]->responses[i] =
                my_cblock[ME]->value++;
            my_cblock[ME]->toggle[i] = obj->announce[i];
        }
    }
    epoch_t new_cblock (my_cblock[ME], old_cblock.count + 1);
    if (CAS (&obj->cblock, old_cblock, new_cblock)) {
        my_cblock[ME] = old_cblock.ptr;
        break;
    }
}
// two tries; some process will have done this thread’s op
return obj->cblock.ptr->response[ME];
}
• general approach
  – thread announces intent to perform operation
    • toggle/announce are 1-bit serialization markers
    • when they differ, thread is trying to perform op
    • announce is changed in the current cblock
    • toggle is changed in the replacement cblock
    • when they match, op is complete
  – in a more complex operation
    • need to post arguments first
    • then memory + compiler barriers
    • finally flip announce
  – after announcing
    • try to perform ops for all waiting threads
    • atomically swap in new copy block
    • can’t fail more than twice (first winning thread might not see announcement, but second winning thread must have)

• some complexities
  – must be sure that no thread reads bad data
  – never free any copy block; someone may be reading it
  – if thread has pointer to copy block, that copy must be “correct”; either:
    • block indicates that thread’s op is not complete
    • block contains correct answer for thread’s op
    • even if block is repeatedly reused before thread reads values!

• before trying to perform op, thread checks twice
  – does toggle match announce?
  – cblock pointer may change after being read
    • may point to some thread that is trying to perform thread’s op
    • but will ultimately fail, so can’t assume success
  – so check again… (two changes to cblock again implies success)
• two attempt reasoning…

  – theorem: if two changes are made to the copy block pointer after a thread announces its intention to perform an op, the op is complete

  – proof
    • let TA be the announcing process
    • let T1 and T2 be the threads that change the copy block pointer (T1 may or may not equal T2, but neither equals TA)
    • four events
      – A: TA reads pointer
      – B: T1 changes pointer
      – C: T2 reads pointer
      – D: T2 changes pointer
    • A precedes B by given condition
    • C precedes D by definition of the operation
    • since T2 succeeded in changing the pointer
      – it must have read the pointer after T1 changed it
      – so B precedes C
    • by transitivity, A precedes C
      – but A occurs after TA has announced intent
      – so T2 sees announcement
      – so T2 performs TA’s operation
Read-Copy Update (RCU)

- general idea
  - identify control points (“quiescent states”) at which any thread must have completed an operation, released a resource, etc.
  - wait until all threads have done so
- example #1: linked list removal
  - eagerly move item out of list
  - no thread can find pointer to removed element
  - but some threads may still have pointer in register
  - after passing through quiescent states, thread must have finished with copy in register
  - after all threads have passed through quiescent state, no thread can be using item, so can be destroyed
- example #2: buffer fill
  - use lock/f-and-i/etc. to allocate buffers
  - for sake of example, say that buffer fill cannot be interrupted by scheduler, does not yield, etc.
  - so scheduler is a quiescent point
    (NOT timer tick, but actual change of running thread)
  - at some point X, examine # of buffers allocated
  - after that point, allow all processors to pass through scheduler
  - all buffers allocated by X must be full, so flush to disk
- similar to earlier ideas
  - but in this form first built into Sequent’s Dynix/ptx OS
  - now used in a few places in Linux
  - or Linux Journal (e.g., www.rdrop.com/users/paulmck/rclock)
A Few Lock Algorithms

- T&S: test and set (same as the primitive)
  - a memory location (usually aligned to a cache line)
  - 0 means unlocked, 1 means locked
  - test and set to lock
  - only one thread can own, so simply write 0 to unlock
  - (need memory/compiler barriers for unlock on some machines)
    ```c
    void lock_T_and_S (lock_t* lock)
    {
        while (T_and_S (lock));
    }
    ```

- when threads contend, T&S pounds on the memory system!
  - synch. primitives often executed at/near first shared cache
  - but can’t keep lock in one cache
  - memory traffic can slow thread holding the lock

- T&T&S: test and test and set
  - so long as the lock is held, the line can be read-shared among waiters
  - read the lock
  - if it’s held, wait until it’s not (pound on your own cache!)
  - if it’s not, execute T&S
  - memory traffic effects
    - all waiters still try T&S when lock is released
    - lots of thrashing on each handoff
    ```c
    void lock_T_and_T_and_S (lock_t* lock)
    {
        while (1) {
            while (1 == *lock);
            if (!T_and_S (lock)) {return;}
        }
    }
    ```
• ticket lock: take a number, wait your turn
  – one cache line for the dispenser
  – one cache line for the number being served
  – to lock, pull a number from the dispenser (fetch-and-increment)
  – check the display; when you see your number, you have the lock
  – when you’re done, increment the display to unlock
  – memory traffic effects
    • only one thread changes the display on handoff
    • but all threads immediately want to share the new display value

```c
void lock_ticket (ticket_lock_t* lock)
{
    int32_t mine = Fetch_and_Increment (&lock->dispenser);
    while (mine != lock->display);
}
void unlock_ticket (ticket_lock_t* lock)
{
    // need compiler and memory barriers here!
    lock->display++;
}
```

```c
int32_t lock_anderson (anderson_lock_t* lock)
{
    int32_t mine = (Fetch_and_Increment (&lock->dispenser) % N_CONTENDERS);
    while (0 == lock->slot[mine]);
    return mine;
}
void unlock_anderson (anderson_lock_t* lock, int32_t mine)
{
    lock->slot[mine] = 0;
    // need compiler and memory barriers here!
    lock->slot[(mine + 1) % N_CONTENDERS] = 1;
}
```
• Anderson lock
  – choose maximum number of contenders (power of 2)
  – allocate one cache line per possible contender
  – plus one line to hold a pointer to the current owner
  – to lock, fetch-and-increment the pointer
  – low bits tell you your slot
  – when your slot holds the “token” (say, a 1), you own lock
  – to unlock, write 0 into your slot, and 1 into the next slot
  – memory traffic effects
    • only the next thread’s cache line invalidated on lock handoff
    • best you can do without cache injection support
  – drawbacks
    • bound on threads may not be small (1 cache line each!)
    • pull in & immediately examine two cache lines rather than one, so slow when not contended compared with all previous

• can also build queueing locks and reader-writer locks
• example: Linux reader/writer locks with fetch-and-add
  – determine max threads N
  – start lock value at N
  – reader subtracts 1
  – writer subtracts N
  – negative result means failed acquisition
    • add same value back
    • wait until acceptable value seen (like T&T&S)
    • try again
• note that waiting for an acceptable value precludes livelock scenarios in which multiple threads constantly keep the lock value below the level at which any of them can succeed