Lecture Topics

• primitives and universality
• ownership transfer & queue examples (algorithms)
• the concept of consistency models
• serializability and linearizability

Administrivia

• corezilla accounts yet?
Primitives and Universality

- Can you build synchronization with loads and stores?
  - not easily and not well
  - load/modify/store sequence is not atomic with respect to itself
  - subject to severe delays, often livelock/starvation
- a synchronization “primitive” is an instruction
  - executed atomically with respect to other instructions
  - usually some form of read/modify/write memory
  - note that x86 ADDL to memory address is NOT atomic without a LOCK prefix; it is NOT a synchronization primitive
- most basic primitive
  - test and set
  - set a memory location to 1, and return its previous value
  - treat 1 as locked, 0 as unlocked
  - if several threads compete, only one of them receives a 0 in return
- Can you build wait-free and non-blocking synchronization with T&S?
  - No.
  - You need something more powerful.
  - primitive capable of implementing any kind of synch. called universal
  - two common universal primitives
    - compare and swap: if value matches current, swap with new
    - load-linked/store conditional
      - load a value and have h/w track address
      - store succeeds iff value has not been changed by another thread
      - store returns success/failure
- most ISAs support one/both universal primitives
- other common primitives: fetch & add, swap
• load-linked/store conditional (LL/SC) “better” than compare-and-swap (CAS)
  – LL/SC requires hardware support
  – but avoids need for more complex software
• why? consider linked list with atomic insert/removal using CAS...

```c
object_t* head; // global for simplicity of example
void insert (object_t* elt)
{
    object_t* cur_head;
    do {
        cur_head = head;
        elt->next = cur_head;
    } while (!CAS (&head, cur_head, elt));
}
object_t* remove ()
{
    object_t* ret_val;
    object_t* replace;
    do {
        ret_val = head;
        if (NULL == ret_val) { return NULL; }
        replace = ret_val->next;
    } while (!CAS (&head, ret_val, replace));
    return ret_val;
}
```

What happens if I replace one/both ret_val’s with head?

• CAS subject to the ABA problem; example scenario follows
  – list initially starts with ABC…
  – thread 1 observes A, loads next pointer (to B) into register, and goes to sleep before removing A from list
  – thread 2 comes in, removes A, and works on it for a while
  – other threads insert some nodes; now list starts DEF…
  – finally, thread 2 finishes with A and returns it to front of list
  – thread 1 wakes up, executes CAS on head: if A, change to B…(Oops.)
- solution
  - extend pointers with epoch #'s
  - 32-bit count of # of operations (for example)
  - extremely unlikely (want practically impossible) to match by accident
- example with linked list insertion

```c
obj_and_epoch_t head; // global for simplicity of example
do
    { 
        obj_and_epoch_t replace;
        do
            { 
                obj_and_epoch_t cur_head = head; // one load instruction!
                elt->next = cur_head.ptr;
                replace.ptr = elt.ptr;
                replace.epoch = cur_head.epoch + 1);
            } while (!CAS (&head, cur_head, replace));
    } 
```

Ownership Transfer

- critical sections must be small
  - large critical sections increase contention
  - more likely to be a bottleneck
  - more likely to require extra work to add locks
- instead, many use ownership transfer models
  - only one thread can touch a particular object
  - possibly temporary
    - under lock, change object state to “private”
    - other threads steer clear for a while
  - possibly always (with thread varying)
    - when thread is done, pass to next thread
    - use queue abstraction
    - message-passing on shared memory
- queue models
  - one to one (single producer, single consumer)
    - this structure you can build with loads and stores!
    - still need memory + compiler barriers on many ISAs
    - often used, but can use lots of memory
      and be challenging with dynamic threads
  - many to one (multiple producer, single consumer)
    - need synchronization primitives
    - efficient in memory
    - any thread can find queue for known target thread
    - slower in theory
      - but polling many queues is not fast
      - so competitive in practice
      - also, most communication patterns have limited degree
single-producer, single-consumer FIFO queue (review?)

```c
#define Q_LEN 256 // a power of 2—why?
void* queue[Q_LEN];
volatile int32_t q_head = 0;
volatile int32_t q_tail = 0;

bool enqueue (void* data)
{
    if (q_tail == q_head + Q_LEN) { return false; }
    queue[(q_tail % Q_LEN)] = data;
    q_tail++; // volatile disallows load to move upward
    // which in turn prevents store from moving...
    return true;
}

void block_enqueue (void* data)
{
    // now volatility of head is even more important!
    while (!enqueue (data));
}

void* dequeue ()
{
    void* ret_val;
    if (q_head == q_tail) { return NULL; }
    ret_val = queue[(q_head % Q_LEN)]; // read after increment?
    q_head++; // again, volatility is important here
    return ret_val;
}

void* block_dequeue ()
{
    void* ret_val;
    while (NULL == (ret_val = dequeue ()));
    return ret_val;
}
```
- multi-producer, single consumer queue (see Ch. 3 of my thesis on web page)
  - data structure
    - queue head & tail (separate cache lines)
    - array of one-cache-line “packets” with state + contents
  - code uses Fetch & Increment and Compare & Swap
  - enqueue sequence
    - packet assignment
    - packet claim (atomic state change: FREE to CLAIMED)
    - fill packet
    - packet ready for delivery (state change: CLAIMED TO READY)
    - packet read
    - packet ready for reuse (READY TO FREE; can be claimed again)

// returns index of claimed packet in array
def claim_packet (queue_t* q)
{
    int32_t index = (fetch_and_increment (q->tail) % Q_LEN);
    while (1) {
        if (CAS (&q->packet[index].state, FREE, CLAIMED)) {
            return index;
        }
        // back off exponentially (& poll incoming queue...)
    }
}

- receiver simply checks state of packet at head of queue for READY
- assigned packets can exceed Q_LEN
  - threads compete for some packet slots
  - back off exponentially to avoid extra memory traffic
- this queue is neither FIFO nor linearizable!
The Concept of Consistency Models

• 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