

Lecture Topics

- semaphores and their uses [was a demo in 438...]
- interesting examples
 - non-linearizable algorithm
 - non-blocking and wait-free implementations
 - read-copy update

Administrivia

- Note #3 online
- let me know if you'll be late with Lab #3

Semaphores and Their Uses

- a semaphore is a counter that supports a fixed number of participants
 - up/post/increment and down/wait/decrement operations
 - (for the original Dutch symbols, see the 391 notes)
- consider two groups of threads
 - workers: perform tasks (servers)
 - supervisors: assign tasks to workers (clients?)
- three experiments to demonstrate
 - allowing supervisors to sleep (i.e., not waste CPU cycles)
 - fairness issues between supervisors
- Experiment #1: several workers, one supervisor, no semaphore
 - workers repeat: wait for task in your queue, execute task
 - unique supervisor repeats
 - wait for empty worker position (repeat check in round robin order)
 - place a task sheet in the empty position
 - How much time does the supervisor spend idling?
(This time corresponds to wasted CPU cycles.)
- Experiment #2: several workers, fast and slow supervisors, no semaphore
 - workers as before
 - supervisors repeat
 - look for an empty position (check once in round robin order)
 - if none is available, delay for time commensurate with your speed, then start looking again
 - place a task sheet in the empty position
(atomically w.r.t. other supervisor!)
 - Do slow and fast supervisors assign a roughly equal number of tasks?

- Experiment #3: N workers, fast and slow supervisors, with a semaphore!
 - semaphore: starts at N
 - workers repeat
 - wait for task in your queue, execute task
 - NEW: up the semaphore
 - supervisors repeat
 - NEW: down the semaphore
 - look for an empty position (**always** available)
 - place a task sheet in the empty position (atomically w.r.t. other supervisor!)

- same questions from first two experiments...
 - How much time do supervisors spend idling?
 - None.
 - all workers busy means
 - semaphore is at 0
 - supervisors go to sleep immediately

 - Do slow and fast supervisors assign a roughly equal number of tasks?
 - Yes.
 - when a worker finishes, one supervisor woken up
 - semaphore maintains a queue, so they take turns being woken
 - If workers finish tasks faster than slow supervisor can assign,
 - fast supervisor will still have advantage
 - But who cares? Slow supervisor assigning as fast as it can!

- Note: semaphore with initial value 1 is a mutex lock.

Some Interesting Examples

- 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:

```
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

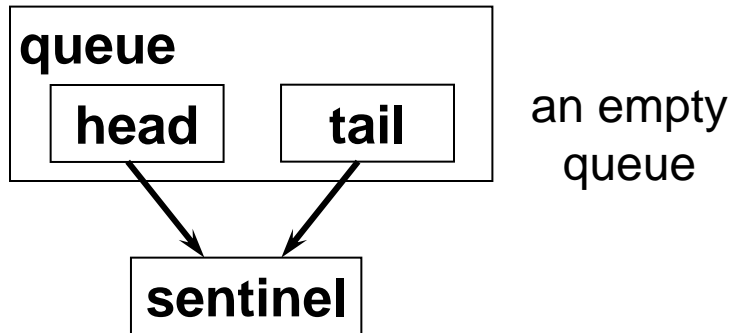
    // (memory and compiler barriers implied by SWAP)

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

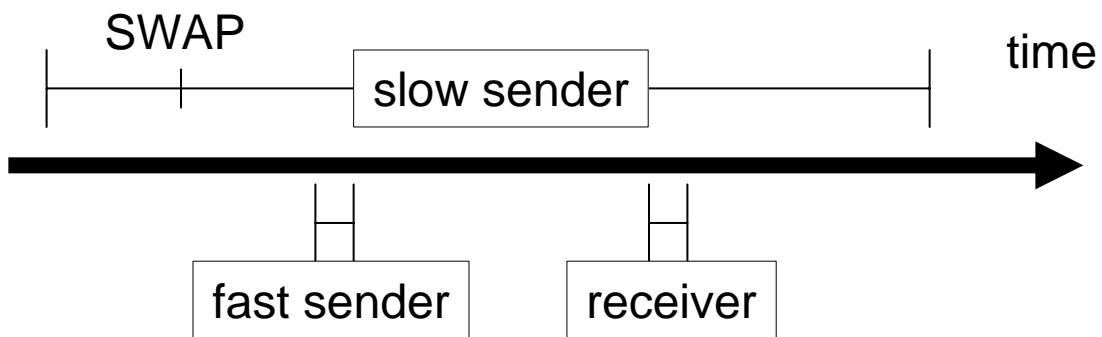
    // (dependence on SWAP return value suffices for
    // barriers)
    old_tail->next = msg;
}

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



- consider the following scenario



- 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 fetch-and-increment implementation using CAS

```

fetch_and_increment_nb (int32_t* addr)
{
    int32_t value, next;
    do {
        value = *addr;
        next = value + 1;
        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.
 - `NUM_THREADS + 1` copy blocks exist at all times
 - `ME` is a unique thread id for each thread

```

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
 - see McKenney, Slingwine, “Read-Copy Update: Using Execution History to Solve Concurrency Problems,” PDCS, 1998.
 - or Linux Journal (e.g., www.rdrop.com/users/paulmck/rclock)