

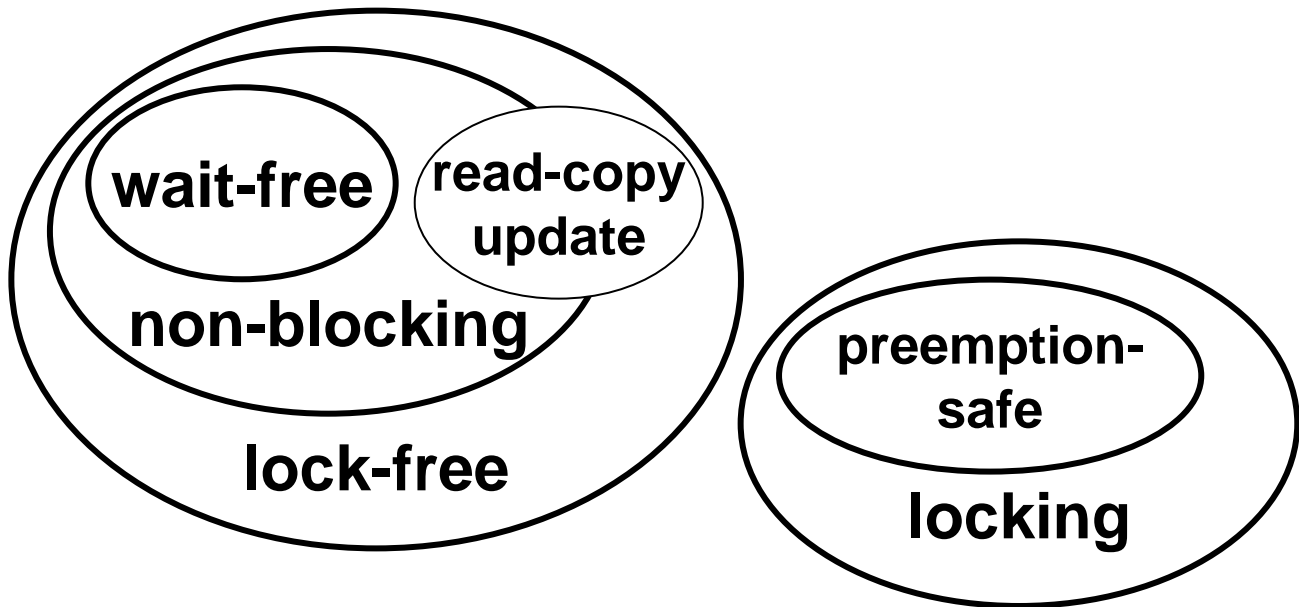
Lecture Topics

- synchronization classes (theory)
- primitives and universality
- a few lock algorithms

Administrivia

- lab 3
 - sequential version now available to you
 - along with sample images

Synchronization Classes



- most of you should be familiar with lock-based synchronization
 - mutual exclusion primitive (a lock)
 - for atomicity (with respect to all other operations executed with lock)
 - obtain lock
 - execute operation
 - release lock
 - for dependence
 - set state
 - consumer waits for state to change (perhaps go to sleep)
 - producer changes state under protection of lock
 - locks can sometimes be omitted
 - for certain simple state changes
 - usually only if consumer is unique / change is a broadcast

- pre-emption safe locking implies OS support for locking
 - the problem
 - thread idles waiting for lock
 - wastes processor cycles
 - potentially causes deadlock
 - also leads to priority inversion
 - low-priority job holds lock, but is sleeping
 - high-priority job also sleeps until lock released
 - meanwhile, medium-priority job runs!
 - some solutions
 - priority inheritance
 - lock-holder scheduled at highest priority of waiters and self
 - used with Posix mutex
 - process holding a lock cannot be preempted
 - used for spin locks in Linux 2.4-
 - still used in certain cases in Linux 2.6+
 - also useful at user level in some cases
- lock-free algorithms/data structures do not make use of locks
 - strongest class: wait-free
 - what you expect for instruction execution, for example
 - each (executing) thread makes progress
 - in a bounded number of its own time steps
 - weaker class: non-blocking
 - some thread makes progress
 - in a bounded number of “system” time steps
 - lock-free with no guarantees
 - can starve or livelock
 - may be unlikely/impossible or an unstable equilibrium
 - progress in a bounded expected number of time steps

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...

```
object_t* head; // global for simplicity of example
```

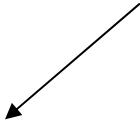
```
void insert (object_t* elt)
```

```
{
    object_t* cur_head;
    do {
        cur_head = head;
        elt->next = cur_head;
    } while (!CAS (&head, cur_head, elt));
}
```

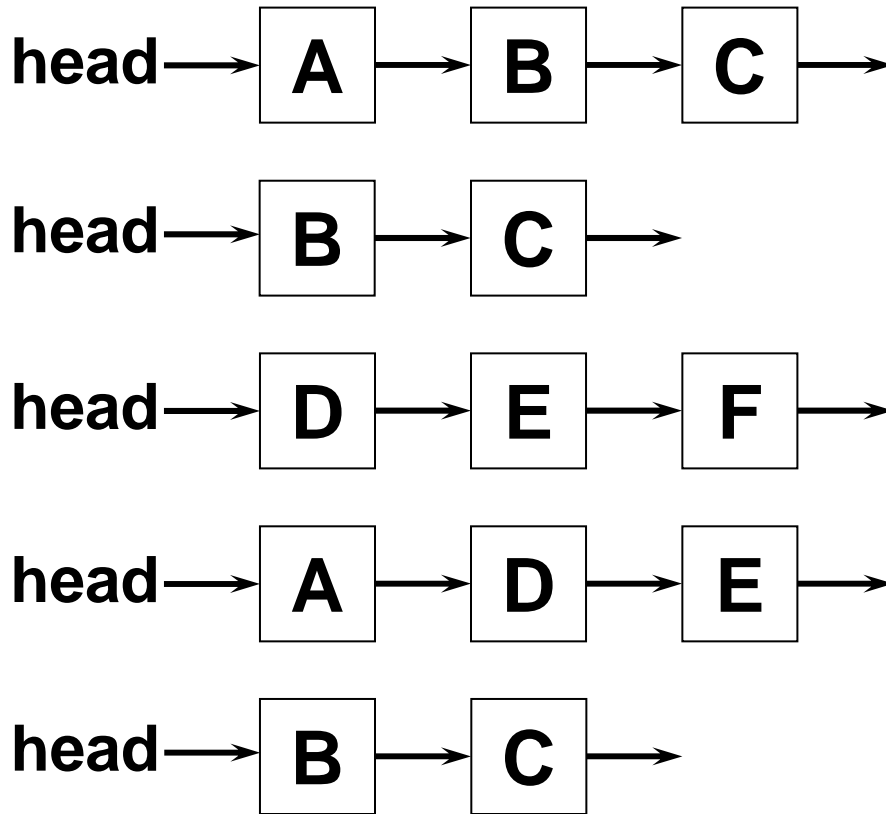
```
object_t* remove ()
```

```
{
    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

```

obj_and_epoch_t head; // global for simplicity of example
void insert (object_t* elt)
{
    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));
}
  
```

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)
- 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

```
void lock_T_and_S (lock_t* lock)
{
    while (T_and_S (lock));
}
```

```
void lock_T_and_T_and_S (lock_t* lock)
{
    while (1) {
        while (1 == *lock);
        if (!T_and_S (lock)) {return;}
    }
}
```