Nonblocking Synchronization

- Idea: avoid problems with blocking synchronization mechanisms (e.g., locks) by…..
  - Not blocking
- Also want to allow multiple threads to make progress concurrently

Problems with locks

- Error prone
  - Deadlock possible, for example, if locks acquired in wrong order
    - Locks aren't composable; for example all modules must adhere to the same convention for multiple locks
- Overhead in case where contention is unlikely
- Vulnerable to failures and faults
  - E.g., thread holding lock infinite loops
- Pre-emption while holding lock

Problems with locks

- Deadlock that is not programmer induced
  - Consider following case: medium priority thread running, and low priority thread owns lock that high priority thread is waiting for
    - This is priority inversion, even if the lock has a blocking-based implementation
  - Lock convoys
    - If there is contention for a lock, threads may do little work before blocking (too much context switching time)

Load-linked with Store conditional

- Used together (but note they are *two separate instructions*)
  - LL returns value at a shared memory location
  - SC stores new value there if previous value has not changed
    - Returns true if stored, false if not stored

Concurrent Counter with LL/SC

```
int ConcurrentAdd(ref int x, int value)  {
  repeat
      old = LL(x)
  until
      SC(x, old+value)
  return old   // return previous value
}
```

Critical Section Solution with LL/SC

```
Init: s = 0

Entry: while (1) {
           while (LL(s) == 1) ;
           if (SC(s, 1))
               return    // success; go into critical section
       }

Exit: s = 0
```

Concurrent Stack

```
structure pointer_t     {ptr: pointer to node_t, count: unsigned integer}
structure node_t        {value: data type, next: pointer_t}
structure stack_t       {Top: pointer_t}

INITIALIZE(S: pointer to stack_t)
    S→Top.ptr = NULL                                    # Empty stack. Top points to NULL

PUSH(S: pointer to stack_t, value: data type)
    node = new_node()                                   # Allocate a new node from the free list
    node→value = value                                  # Copy stacked value into node
    node→next.ptr = NULL                                # Set next pointer of node to NULL
    repeat                                              # Keep trying until Push is done
        top = S→Top                                     # Read Top.ptr and Top.count together
        node→next.ptr = top.ptr                         # Link new node to head of list
    until CAS(&S→Top, top, [node, top.count+1])         # Try to swing Top to new node

POP(S: pointer to stack_t, pvalue: pointer to data type): boolean
    repeat                                              # Keep trying until Pop is done
        top = S→Top                                     # Read Top
        if top.ptr == NULL                              # Is the stack empty?
            return FALSE                                # The stack was empty, couldn't pop
        endif
    until CAS(&S→Top, top, [top.ptr→next.ptr, top.count+1])   # Try to swing Top to the next node
    *pvalue = top.ptr→value                             # Pop is done. Read value
    free(top.ptr)                                       # It is safe now to free the old node
    return TRUE                                         # The stack was not empty, pop succeeded
```

FIG. 1. Structure and operation of Treiber's nonblocking concurrent stack algorithm [38].

Alternate Concurrent Stack (pushes/pops pointers)
Source: Wikipedia
(https://en.wikipedia.org/wiki/ABA_problem)

Stack just points to stack top

Obj is structure of interest;
push and pop pointers to Obj

```
Push(Stack *S, Obj *p)
    while (1) {
        Obj *top = S
        p→next = top
        if CAS(S, top, p)
            return
    }
```

```
Obj *Pop( )
    while (1) {
        Obj *ret = S;
        if (!ret) return NULL
        next = ret→next
        if CAS(S, ret, next)
            return ret
    }
```

Why is there a counter as part of the stack pointer?

- Example sequence of events (Wikipedia version):
  1. State at a given point: Top → A → B → C
  2. Threads 1 and 2 both invoke Pop
  3. Thread 1 starts running Pop, and sets *ret* to A and *next* to B
  4. Before Thread 1 can execute CAS, context switch to Thread 2
  5. Thread 2 starts running Pop and sets *ret'* to A and *next'* to B
  6. Thread 2 successfully executes CAS, so now stack is Top → B → C
  7. Thread 2 starts running Pop again and sets *ret'* to B and *next'* to C
  8. Thread 2 successfully executes CAS, so now stack is Top → C
  9. Thread 2 deallocates B
  10. Thread 2 invokes Push(A) and succeeds, so now stack is Top → A → C
  11. Thread 1 resumes, and Top and *ret* are both A, so CAS succeeds and Top now points to B, which has been deallocated. To say the least, this is bad.

Why is there a counter as part of the stack pointer?

- ABA problem
  - Means that value is changed and changed back, but from A to B to A---so Compare and Swap does not detect the change!
  - Count up, so that this can't happen
    - Well, it could if there is overflow, but…
  - Requires doubleword Compare and Swap
    - So that we can store a pointer and a counter
  - If have LL/SC, can avoid the ABA problem
    - But not all architectures have LL/SC

Concurrent Queue

```
structure pointer_t     {ptr: pointer to node_t, count: unsigned integer}
structure node_t        {value: data type, next: pointer_t}
structure queue_t       {Head: pointer_t, Tail: pointer_t}


INITIALIZE(Q: pointer to queue_t)
    node = new_node()                         # Allocate a free node
    node→next.ptr = NULL                      # Make it the only node in the linked list
    Q→Head.ptr = Q→Tail.ptr = node            # Both Head and Tail point to it


ENQUEUE(Q: pointer to queue_t, value: data type)
    node = new_node()                         # Allocate a new node from the free list
    node→value = value                        # Copy enqueued value into node
    node→next.ptr = NULL                      # Set next pointer of node to NULL
    loop                                      # Keep trying until Enqueue is done
        tail = Q→Tail                         # Read Tail.ptr and Tail.count together
        next = tail.ptr→next                  # Read next ptr and count fields together
        if tail == Q→Tail                     # Are tail and next consistent?
            if next.ptr == NULL               # Was Tail pointing to the last node?
                if CAS(&tail.ptr→next, next, [node, next.count+1])  # Try to link node at the end of the linked list
                    break                     # Enqueue is done. Exit loop
                endif
            else                              # Tail was not pointing to the last node
                CAS(&Q→Tail, tail, [next.ptr, tail.count+1])  # Try to swing Tail to the next node
            endif
        endif
    endloop
    CAS(&Q→Tail, tail, [node, tail.count+1])  # Try to swing Tail to the inserted node
```

Concurrent Queue

```
DEQUEUE(Q: pointer to queue_t, pvalue: pointer to data type): boolean
    loop                                                            # Keep trying until Dequeue is done
        head = Q→Head                                               # Read Head
        tail = Q→Tail                                               # Read Tail
        next = head.ptr→next                                        # Read Head.ptr→next
        if head == Q→Head                                           # Are head, tail, and next consistent?
            if head.ptr == tail.ptr                                 # Is queue empty or Tail falling behind?
                if next.ptr == NULL                                 # Is queue empty?
                    return FALSE                                    # Queue is empty, couldn't dequeue
                endif
                CAS(&Q→Tail, tail, [next.ptr, tail.count+1])        # Tail is falling behind. Try to advance it
            else                                                    # No need to deal with Tail
                # Read value before CAS, otherwise another dequeue might free the next node
                *pvalue = next.ptr→value
                if CAS(&Q→Head, head, [next.ptr, head.count+1])     # Try to swing Head to the next node
                    break                                           # Dequeue is done. Exit loop
                endif
            endif
        endif
    endloop
    free(head.ptr)                                                  # It is safe now to free the old dummy node
    return TRUE                                                     # Queue was not empty, dequeue succeeded
```

FIG. 3. Structure and operation of a nonblocking concurrent queue.