CSE 451: Operating Systems

Spring 2022

Module 9.5

Synchronization Review

John Zahorjan

Topics

- High level review of synchronization mechanisms
- RCU (read-copy-update) locks
- Non-blocking synchronization

Spinlocks

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while(test_and_set(&lock->held));
}
void release(lock) {
  lock->held = 0;
}
```

1. Why is hardware support required just to implement simple spin locks?

2. Is it required if there is only a single core?

3. What's wrong with this implementation?

4. What is the policy that this mechanism implements (for who to hand the lock to next)?

Blocking Lock (Mutex)

```
Mutex::acquire() {
    disableInterrupts();
    spinLock.acquire();
    if (value == BUSY) {
        waiting.add(myTCB);
        suspend(&spinlock);
    } else {
        value = BUSY;
    }
    spinLock.release();
   enableInterrupts();
}
```

```
Mutex::release() {
    disableInterrupts();
    spinLock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
         value = FREE;
    }
    spinLock.release();
    enableInterrupts();
}
```

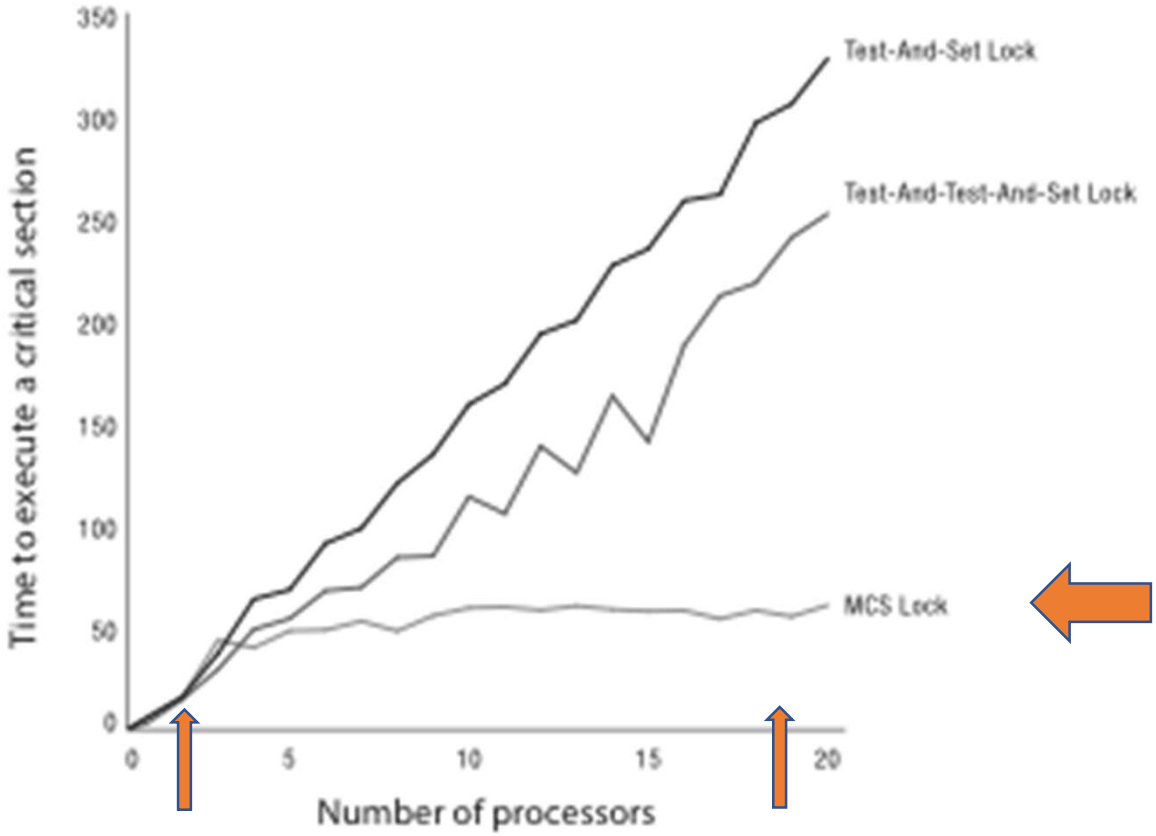
Blocking Lock (Mutex)

```
Mutex::acquire() {
   disableInterrupts();
   spinLock.acquire();
   if (value == BUSY) {
      waiting.add(myTCB);
      suspend(&spinlock);
   } else {
      value = BUSY;
   }
   spinLock.release();
  enableInterrupts();
}
```

```
Mutex::release() {
   disableInterrupts();
   spinLock.acquire();
   if (!waiting.Empty()) {
      next = waiting.remove();
      scheduler->makeReady(next);
   } else {
         value = FREE;
   }
   spinLock.release();
   enableInterrupts();
}
```

1. Why do we need a spinlock?
2. Why does the code disable interrupts?
3. Where is the implementation of the policy for who to give the lock to next?
    1. Where should it be?

More Robust Spinlock Performance



MCS Lock Implementation

```
MCSLock::acquire() {
   Queue *oldTail = tail;

   myTCB->next = NULL;
   myTCB->needToWait = TRUE;
   while (!compareAndSwap(&tail,
                          oldTail, &myTCB)) {
      oldTail = tail;
   }
   if (oldTail != NULL) {
      oldTail->next = myTCB;
      memory_barrier();
      while (myTCB->needToWait)   ;
   }
}
```

```
MCSLock::release() {
   if (!compareAndSwap(&tail,
                       myTCB, NULL)) {
      while (myTCB->next == NULL)  ;
      myTCB->next->needToWait=FALSE;
   }
}
```

race

Why is this fast?

- *Under low lock contention*
- *At high lock contention*

Spin on thread-specific location

What is the Lesson of MCS Locks?

- ?

R/W Locks Possible Implementation

```
void startRead() {
    lock.lock();
    while ( numWriters > 0 )
            wait(readWaitCV, lock);
    numReaders++;
    lock.unlock();
}
void endRead() {
    lock.lock();
    if ( --numReaders == 0 )
            signal(writeWaitCV);
    lock.unlock();
}
```

```
void startWrite() {
    lock.lock();
    while ( numWriters > 0 || numReaders > 0 )
            wait(writeWaitCV, lock);
    numWriters = 1;
    lock.unlock();
}
void endWrite() {
    lock.lock();
    numWriters = 0;
    broadcast(readWaitCV);
    signal(writeWaitCV);
    lock.unlock();
}
```

R/W Locks

1. What is their purpose?

2. What is the policy issue?

3. What is/are the technical term(s) associated with the policy issue?

Something New:
Read-Copy-Update Locks

Read-Copy-Update

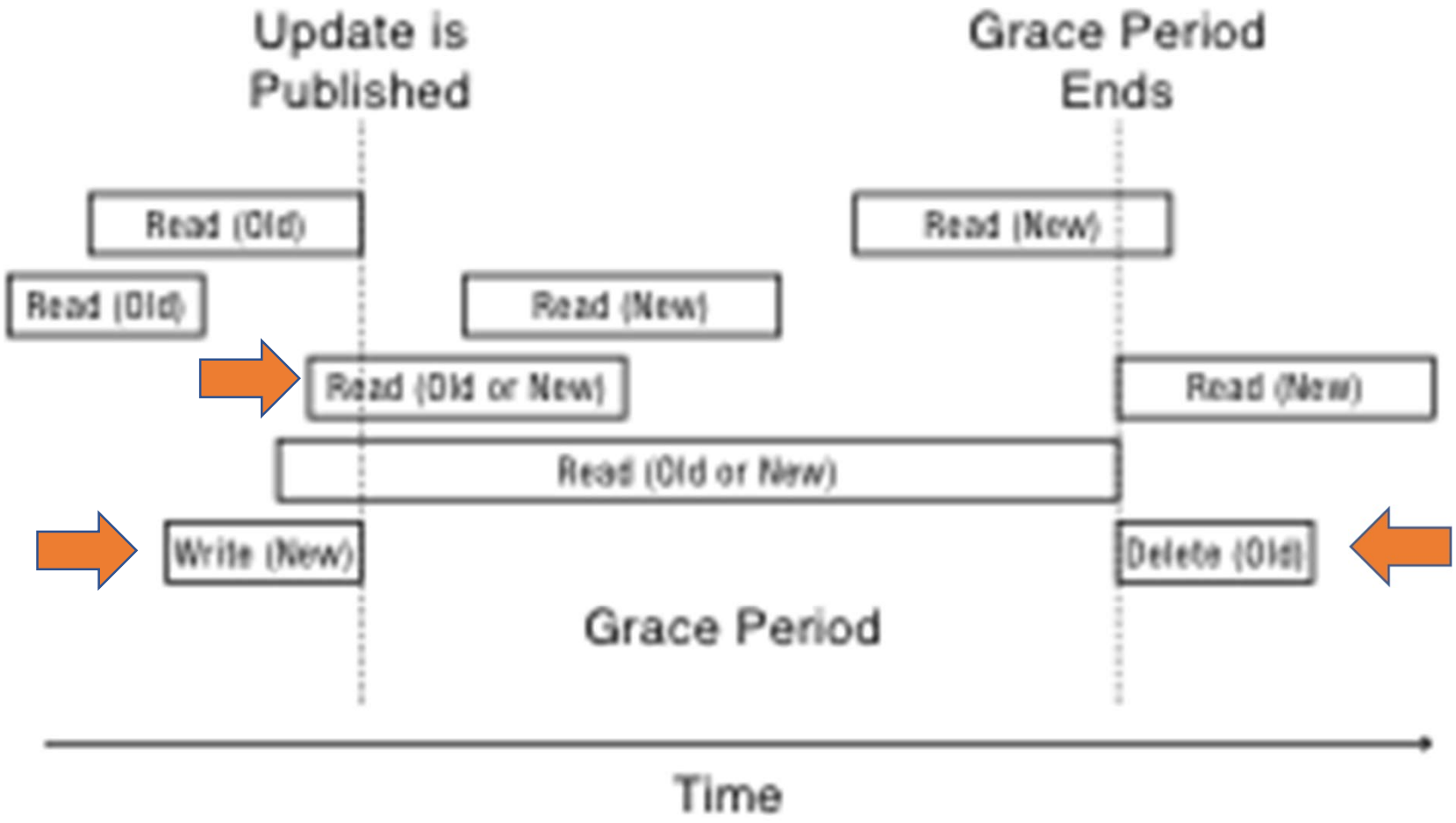
- Goal: low latency reads to shared data
  - Reads proceed without first acquiring a lock
  - It's OK if we get this by making writes (very) slow
    - Best use scenario: writes are infrequent

- Writers: Restricted update
  - Writer creates a new version (copy) of data structure
  - Publishes new version with a single atomic instruction

- Readers: Unimpeded by writes because writers never write a data structure that is being read
  - This results in multiple concurrent versions
  - Which means that readers may see an "old version" for a limited time

- When is it safe to clean up old version?
  - Relies on integration with thread scheduler
  - Guarantee all readers complete within grace period, and then garbage collect old version

Read-Copy-Update



RCU Lock Basic Idea

- Use an atomic update to install next version of data structure
  - Reader sees either the last version or the new version, but never a mixture of the two
- Don't know if any readers are still using an old version
  - Problem: can't "clean up" old versions as part of publishing new versions
- Solution: version generation numbers
  - Increment a generation number (counter) associated with data structure each time a new version is published
  - Each thread advertises the highest version number it has seen
  - So... just wait until all threads are saying they've seen at least version N to clean up versions before N
- RCU Locks: do that, but on a processor basis rather than a thread basis
  - Why not on a per-thread basis?

Read-Copy-Update Implementation

- Readers disable interrupts on entry
  - Guarantees they complete critical section in a timely fashion
  - Prevents scheduler from running on that core
  - No need for a read or write lock

- Writers
  - Acquire write lock
    - One writer at a time
  - Copy-Update
    - Create new data structure
  - Publish new version with atomic instruction
  - Release write lock

  - Wait for scheduler time slice on each CPU
  - Only then, garbage collect old version of data structure

Writer Operation

```
WriteLock();            // only one writer at a time

<prepare updated data structure>

publish(updated data structure);  // make new version visible by CAS
                                  // pointer

WriteUnlock();      // allow another writer to start

synchronize();      // wait until all readers are at at least the version
                    //  you published

<free anything that needs freeing from the version you replaced>
```

RCU Lock Implementation

```
void ReadLock() { disableInterrupts(); }
void ReadUnlock() { enableInterrupts(); }

void WriteLock() { writerSpin.lock(); }
void WriteUnlock() { writerSpin.unlock(); }

void publish( void **ppHead, void *pNew) {
    memory_barrier();
    *ppHead = pNew;   // atomic assignment needed...
     memory_barrier();
}
```

RCU Lock Implementation

```
// called after each modification (after releasing write lock)
void synchronize() {
     c = atomicIncrement(globalCounter);
     for (p=0; p<NUM_CORES; p++ )
          while (PER_PROC_VAR(quiescentCount,p) < c)
               sleep(10);      // about a scheduling quantum
}
 // called by scheduler  (if scheduler is running, there is no reader running or
// suspended on that processor)
void QuiescentState() {
    memory_barrier();
     PER_PROC_VAR(quiescentCount) = globalCounter;
     memory_barrier();
}
```

RCU Lock Question

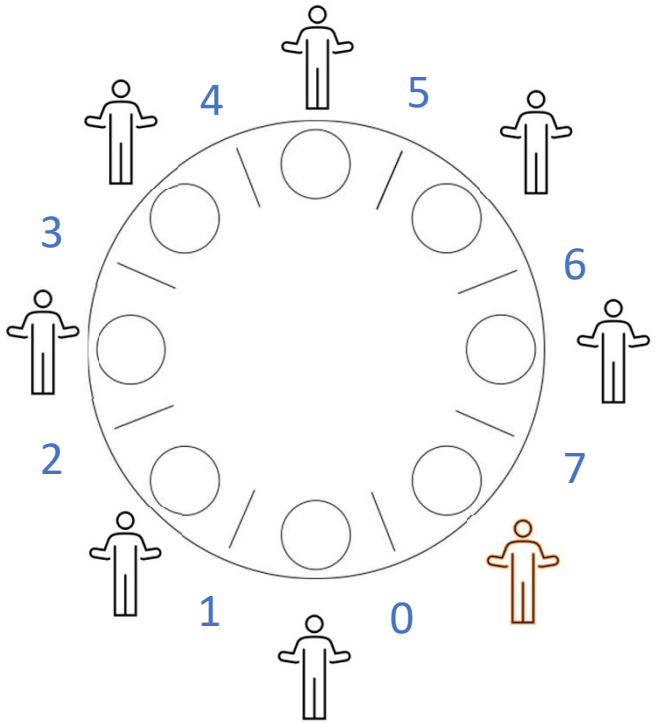
- We require that the new version of the update be published with a single, atomic instruction, so…

- Why do we need a write lock?
  - Why not just produce the updated data structure without a lock and then install it using the atomic instruction?

Deadlock

Traditional Dining Lawyers Solution

- Static rules are:
  - All lawyers but one pick up right fork and then left fork
  - One lawyer picks up left fork then right fork
- This is an example of resource ordering



[Bonus Slide] Acquire All Locks At Once – C++

std::lock

template <class Mutex1, class Mutex2, class... Mutexes> void lock (Mutex1& a, Mutex2& b, Mutexes&... cde);

Lock multiple mutexes

Locks all the objects passed as arguments, blocking the calling thread if necessary.

The function locks the objects using an unspecified sequence of calls to their members lock, try_lock and unlock that ensures that all arguments are locked on return (without producing any deadlocks).

If the function cannot lock all objects (such as because one of its internal calls threw an exception), the function first *unlocks* all objects it successfully locked (if any) before failing.

From http://www.cplusplus.com/reference/mutex/lock/

See code sample there for clearer connection to deadlock issues.

New: Non-Blocking Data Structures

Yet Another Approach: Non-Blocking Algorithms

- An algorithm is non-blocking if a slow thread cannot prevent another faster thread from making progress
  - Using locks is not non-blocking because a thread may acquire the lock and then run really really slowly
    - (Why?)

- Non-blocking algorithms are often built on an atomic hardware instruction, Compare And Swap (CAS), whose semantics are:

```
bool CAS(ptr, old, new) {
    if ( *ptr == old ) { *ptr = new;  return true; }
    return false;
}
```

Example: Non-blocking atomic integer

```
int atomic_int_add(atomic_int *p, int val) {
    int oldval;
    do {
        oldval = *p;
    } while ( !CAS(p, oldval, oldval+val);
);
```

- What happens if multiple threads execute this concurrently?
  - Does every thread make progress?
  - Does at least one thread make progress in bounded number of steps?
- Suppose a thread currently executing this routine is pre-empted?

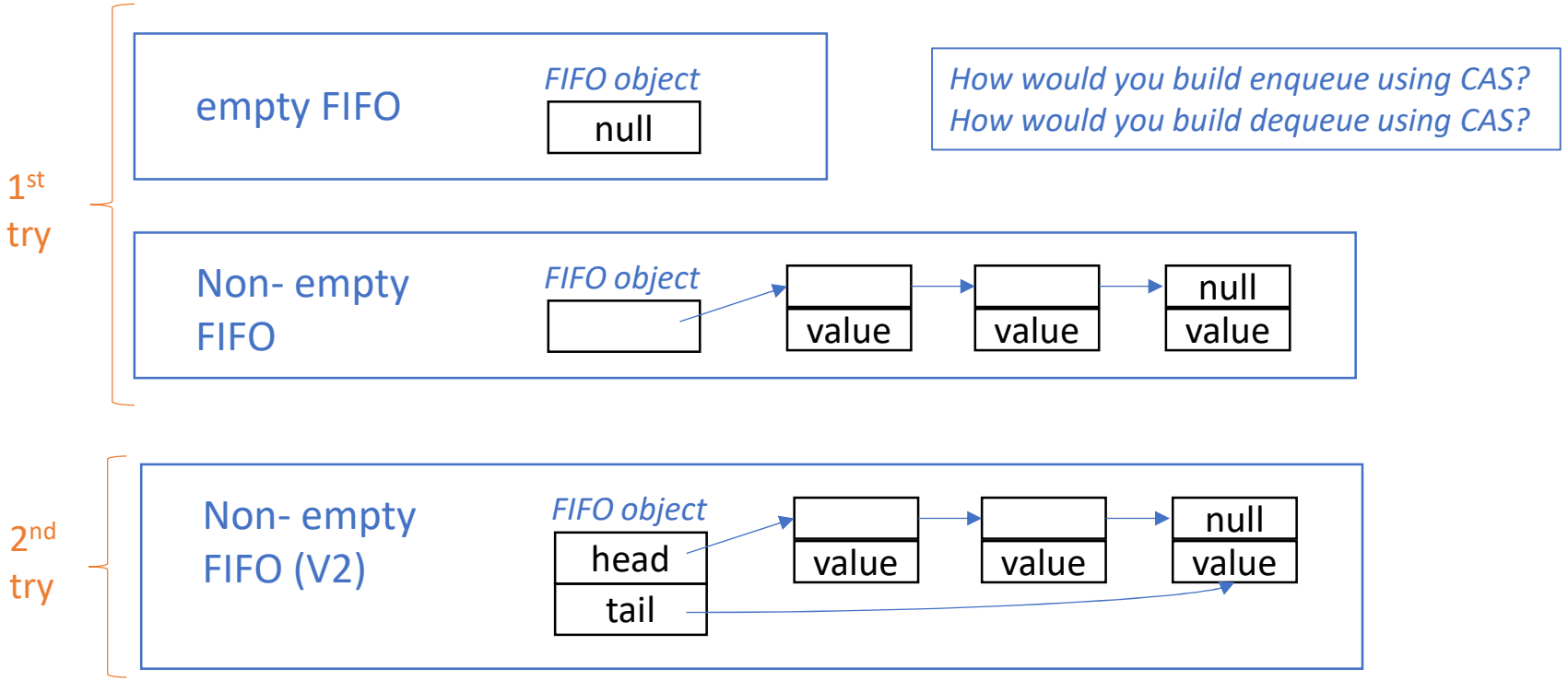
Why Non-blocking?

Two words: No locks!

- With locks, what happens if a thread is pre-empted while holding a lock?
- With locks, deadlock might be possible.
  - Is it possible when there are no locks?
- Priority inversion and locks
  - Assume threads have been assigned priorities, and we'd like to preferentially allocate cores to the highest priority runnable threads
  - Now suppose a low priority thread holds a lock needed by a high priority thread
  - Medium priority threads might steal the core from the low priority thread, indefinitely delaying the high priority thread!
- *Alternative solution (to non-blocking): priority inheritance*
  - Raise the priority of a thread holding a lock to the maximum priority of any thread waiting for the lock

Why Not Non-Blocking?

- 1 word: complicated! [fragile, error prone, special cases…]
- Let's build a non-blocking FIFO queue
- What problems do we anticipate with these?



Why not non-blocking? (Non-blocking FIFO implementation)

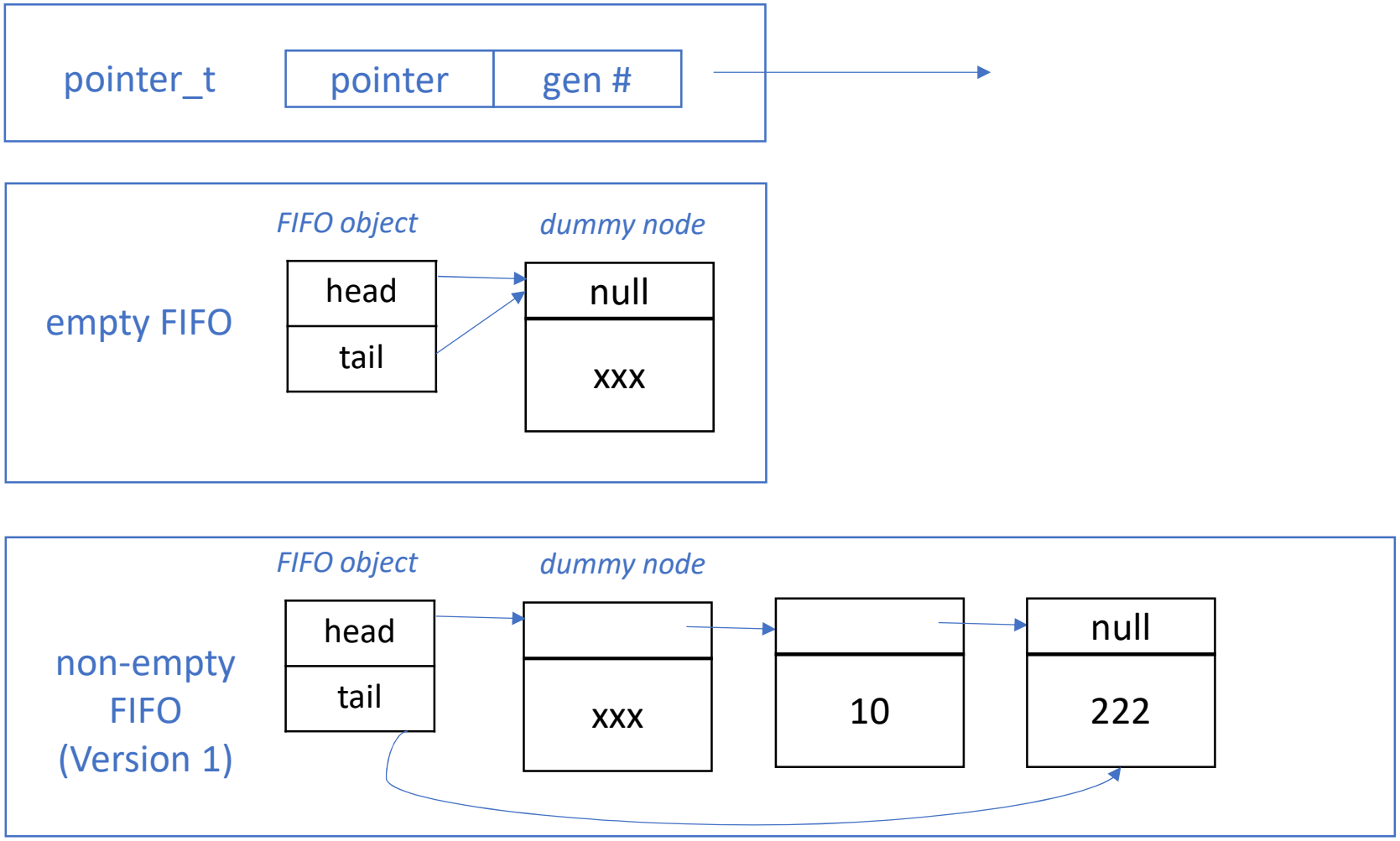
```
structure pointer_t      {ptr: pointer to node_t, count: unsigned integer}
structure node_t         {value: data type, next: pointer_t}
structure queue_t        {Head: pointer_t, Tail: pointer_t}

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 = Q->Tail = node             # Both Head and Tail point to it
```

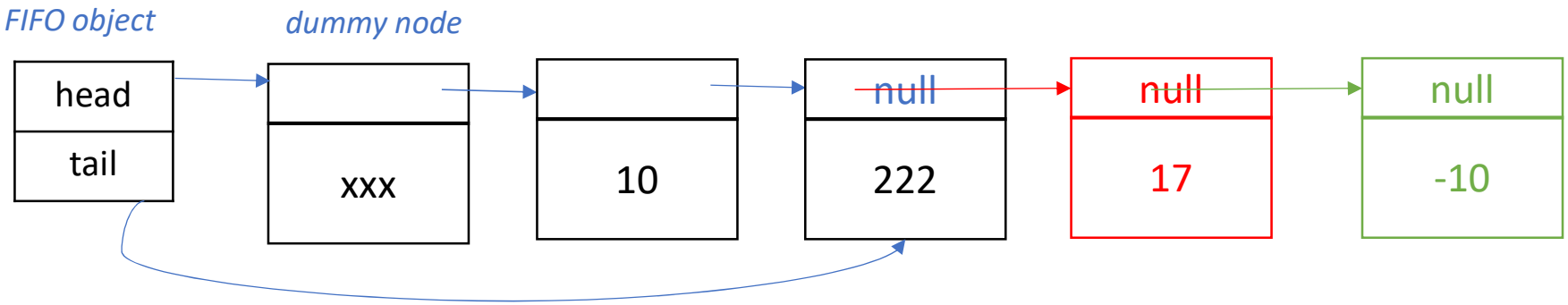
Pointers are stored with a generation number in one 8-byte quantity
(32-bit pointer + 32-bit generation number)

From Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms *by Michael & Scott.*

Non-blocking FIFO



Non-blocking FIFO: enqueue value 17



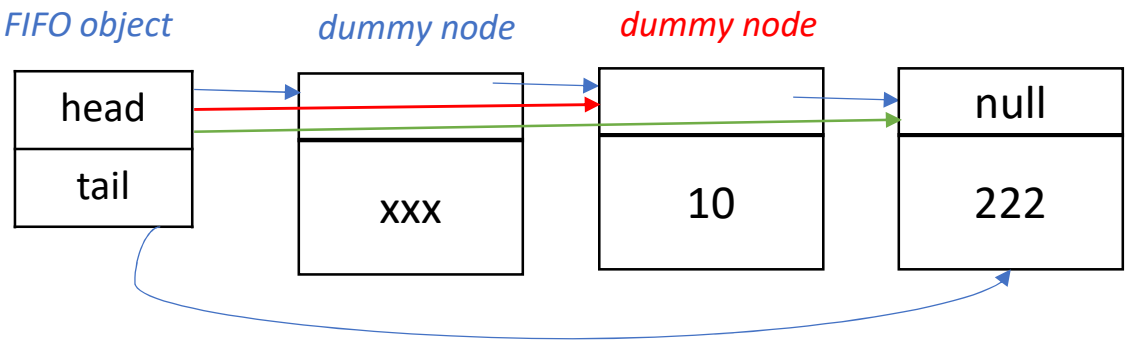
1. Update tail->next to point to new node
2. Update tail to point to new node

But other inserts might be going on at same time…

In general, the tail pointer might “fall behind” the actual tail of the FIFO.
Think of the tail pointer as a performance hint

- it’s better to start looking for the tail from where it points than from where the head pointer points

Non-blocking FIFO: dequeue



1. Return failure if head pointer is null
2. Update tail->head to point to next node
3. Free previous dummy node
4. Return 10

But other dequeues might be going on at same time…
The first of them might free the node that contains the value I need (10)!

1.5 So, grab the value optimistically, then return it only if you manage to move the head pointer to that node (making it the new dummy node).

Non-blocking FIFO: enqueue()

```
enqueue(Q: pointer to queue_t, value: data type)
E1:     node = new_node()                                   # Allocate a new node from the free list
E2:     node->value = value                                 # Copy enqueued value into node
E3:     node->next.ptr = NULL                               # Set next pointer of node to NULL
E4:     loop                                                # Keep trying until Enqueue is done
E5:         tail = Q->Tail                                  # Read Tail.ptr and Tail.count together
E6:         next = tail.ptr->next                           # Read next ptr and count fields together
E7:         if tail == Q->Tail                              # Are tail and next consistent?
E8:             if next.ptr == NULL                         # Was Tail pointing to the last node?
E9:                 if CAS(&tail.ptr->next, next, <node, next.count+1>)  # Try to link node at the end of the linked list
E10:                    break                               # Enqueue is done. Exit loop
E11:                endif
E12:            else                                        # Tail was not pointing to the last node
E13:                CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)  # Try to swing Tail to the next node
E14:            endif
E15:        endif
E16:    endloop
E17:    CAS(&Q->Tail, tail, <node, tail.count+1>)           # Enqueue is done. Try to swing Tail to the inserted node
```

Non-blocking FIFO: dequeue

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:     loop                                                  # Keep trying until Dequeue is done
D2:         head = Q->Head                                    # Read Head
D3:         tail = Q->Tail                                    # Read Tail
D4:         next = head->next                                 # Read Head.ptr->next
D5:         if head == Q->Head                                # Are head, tail, and next consistent?
D6:             if head.ptr == tail.ptr                       # Is queue empty or Tail falling behind?
D7:                 if next.ptr == NULL                       # Is queue empty?
D8:                     return FALSE                          # Queue is empty, couldn't dequeue
D9:                 endif
D10:                CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)   # Tail is falling behind. Try to advance it
D11:            else                                          # No need to deal with Tail
                    # Read value before CAS, otherwise another dequeue might free the next node
D12:                *pvalue = next.ptr->value
D13:                if CAS(&Q->Head, head, <next.ptr, head.count+1>)  # Try to swing Head to the next node
D14:                    break                                 # Dequeue is done. Exit loop
D15:                endif
D16:            endif
D17:        endif
D18:    endloop
D19:    free(head.ptr)                                        # It is safe now to free the old dummy node
D20:    return TRUE                                           # Queue was not empty, dequeue succeeded
```

Performance Results

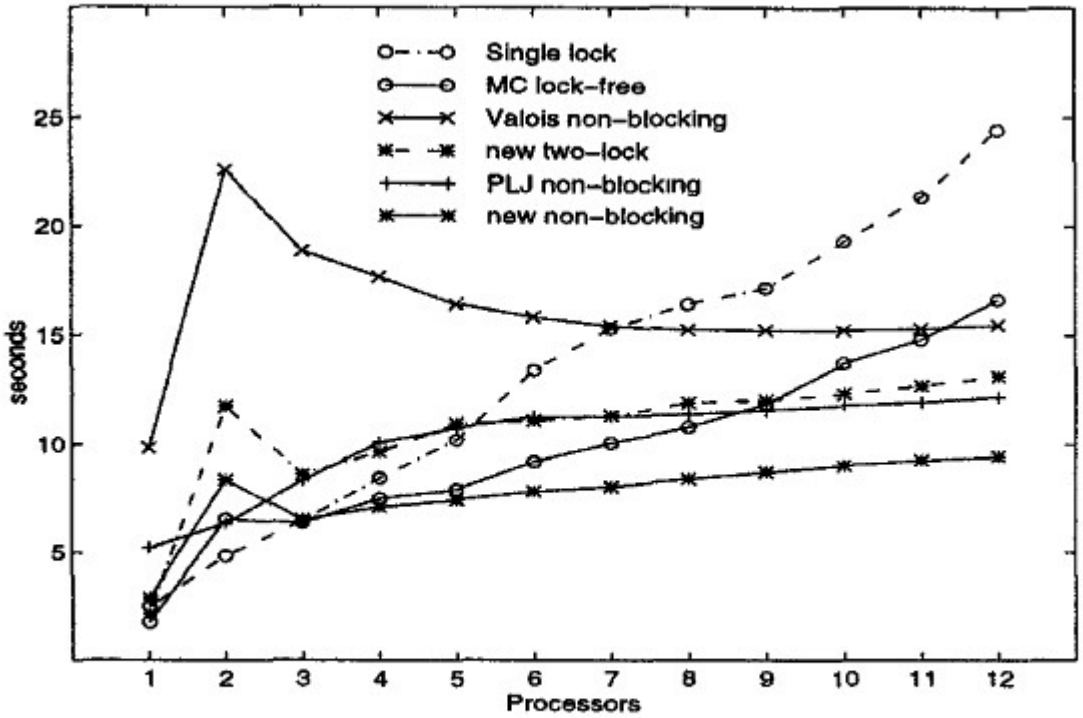


Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

12 processor Silicon Graphics Challenge

Looking Forward: Storage

- The OS wants to do for storage what it did for CPU and memory. What is that (at this very general level of description)?

- Outline:
  - What is the file system abstraction?
  - What are the basic mechanisms for implementation?
  - What are the performance characteristics of the physical storage devices?
  - What is fast on them, what is slow?
  - Maintaining file system integrity in the face of fail-stop errors