CSE 451: Operating Systems

Autumn 2020

Module 8

Synchronization: Performance and Multi-Object

John Zahorjan

Topics

- Readers/Writers Locks
  - Class exercise...
- Performance: Multiprocessor cache coherence
- MCS locks
  - Usual lock semantics
  - Optimized for case that locks are mostly busy
- RCU locks
  - Relaxed semantics (somewhat like readers/writers)
  - Optimized for locks are mostly busy and data is mostly read-only

Module Roadmap

- Condition variables defer to the application all decisions (policy) about when to block…
  - The app executes arbitrary code
- except that the condition variable assumes (insists) that nothing more complicated than a lock is needed to make the decision

- The remaining sync variables in this section impose policy
  - R/W locks: any number of readers and no writers or one writer
  - MCS locks: spinlocks with robust performance at high load
  - RCU locks: efficient, loosely synchronized reads and infrequent, expensive writes

- Plus…
  - Many locks (why?) and deadlock (how not?)

Readers/Writers Locks

Enabling Concurrency

- Imagine you're creating a thread-safe implementation of some data structure
- The interface is read(key) and put(key, value)
- Each instance of the data structure contains a single mutex that is used to restrict concurrent operations

- Does put() need to obtain the mutex?
- Does read() need to obtain the mutex?

Enabling Concurrency

- Imagine you’re creating a thread-safe implementation of some data structure
- The interface is read(key) and put(key, value)
- Each instance of the data structure contains a single mutex that is used to restrict concurrent operations

- Does put() need to obtain the mutex?
- Does read() need to obtain the mutex?

- Usually the answer to both questions is “yes”

Readers/Writers Locks

- Mutex has semantics "one thread at a time"
- Suppose we want semantics
  `any number of readers but no writers`
  OR
  `just one writer`
- Readers/writers locks support this
  - "lock for read" or "lock for write"
- Interface:
  - startRead() ... doneRead
  - startWrite() ... doneWrite()

R/W Locks Implementation

- Take a few minutes and implement them
  - In teams
  - (Heaven help us...)

- The text advocates a “monitor style” programming discipline
  - Implement an abstract data type as a class
  - Each instance contains a lock
  - Every method acquires the lock as the first thing it does
  - Every method releases the lock as the last thing it does
  - What should your code do if it needs to wait?

Lousy Substitution for Class Exercise

This will be very vague, but will be filled in in subsequent slides…

Readers/Writers Locks

- Mutex has semantics “one thread at a time”

- Suppose we want semantics

```
any number of readers but no writers
```

      OR

```
just one writer
```

- Readers/writers locks support this
  - “lock for read” or “lock for write”

- Interface:
  - startRead() ... doneRead
  - startWrite() ... doneWrite()

Readers-Writers Locks

```
int         numWriters = 0;
int         numReaders = 0;

spinlock    lock;

condVar     writeWait;
condVar     readWait;
```

Operations:

startRead() / endRead()

startWrite() / endWrite()

Why a spinlock?

(Why not a mutex?)

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();
}
```

R/W Locks Possible Implementation

```
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 Lock Implementation

- What's good about my implementation?
  - It works!
- What's bad about my implementation?
  - "starvation"
- What alternative semantics might you want?
- How would you know what you want?

```
broadcast(readWaitCV);
signal(writeWaitCV);
```

Module Roadmap

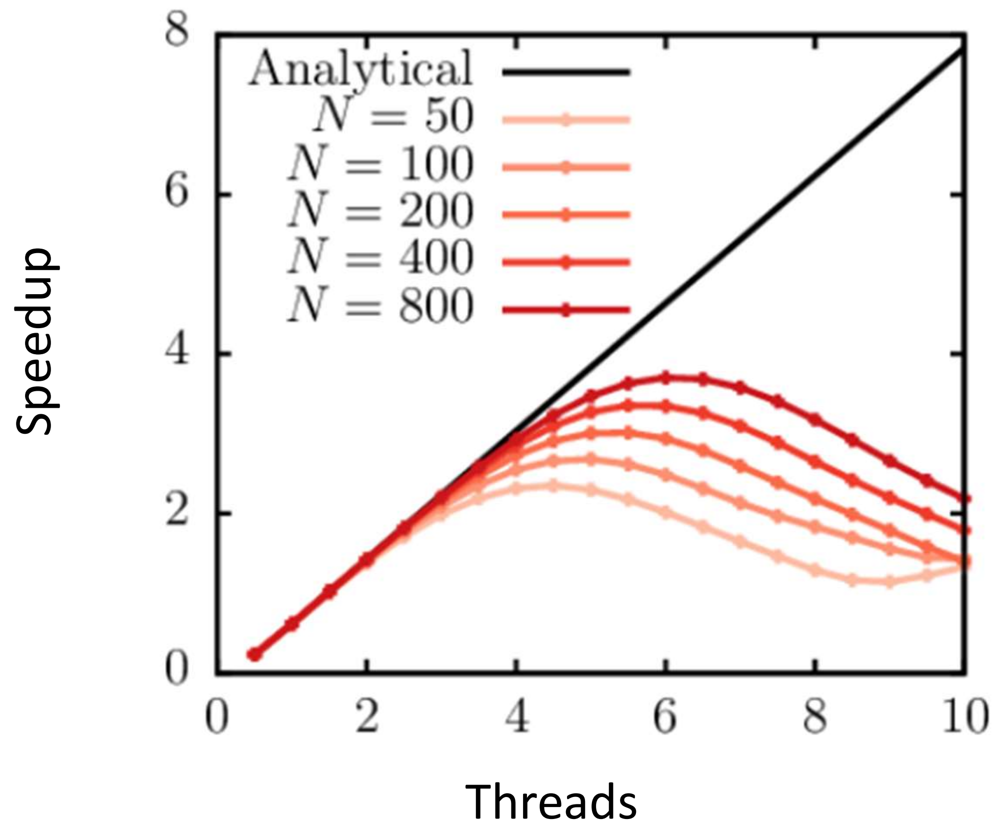
- Condition variables defer to the application all decisions (policy) about when to block…
  - The app executes arbitrary code
- except that the condition variable assumes (insists) that nothing more complicated than a lock is needed to make the decision

- The remaining sync variables in this section impose policy
  - R/W locks: any number of readers and no writers or one writer
  - MCS locks: spinlocks with robust performance at high load
  - RCU locks: efficient, loosely synchronized reads and infrequent, expensive writes

- Plus…
  - Many locks (why?) and deadlock (how not?)

Synchronization Performance: Cache Effects

Multi-threaded/core Performance

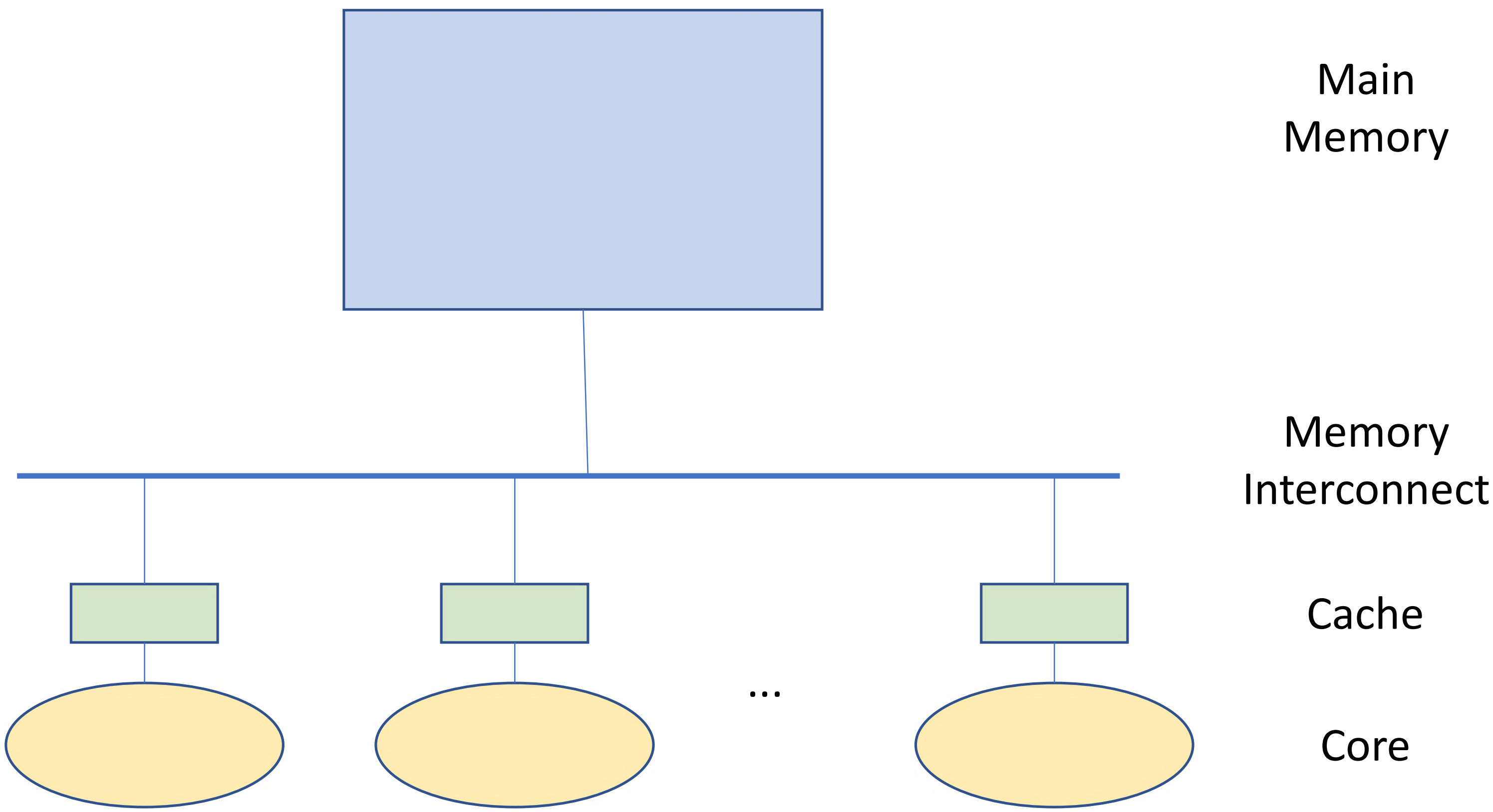


Parallel, numerical application. N reflects size of data (granularity).

Synchronization Performance

- A program with lots of concurrent threads can still have poor performance on a multiprocessor:
  - Overhead of creating threads, if not needed
  - Lock contention: only one thread at a time can hold a given lock
  - Shared data protected by a lock may ping back and forth between cores
  - False sharing: communication between cores even for data that is not shared (but resides in same cache line)

Multicore Caching



Performance: Multiprocessor Cache Coherence

- (Cache) Coherence vs. (Memory) Consistency
  - Consistency: view of values in multiple locations across cores
    - last module (memory barriers)
  - Coherence: view of a single location's value across cores
    - this module

- Scenario:
  - Thread A modifies data inside a critical section and releases lock
  - Thread B acquires lock and reads data

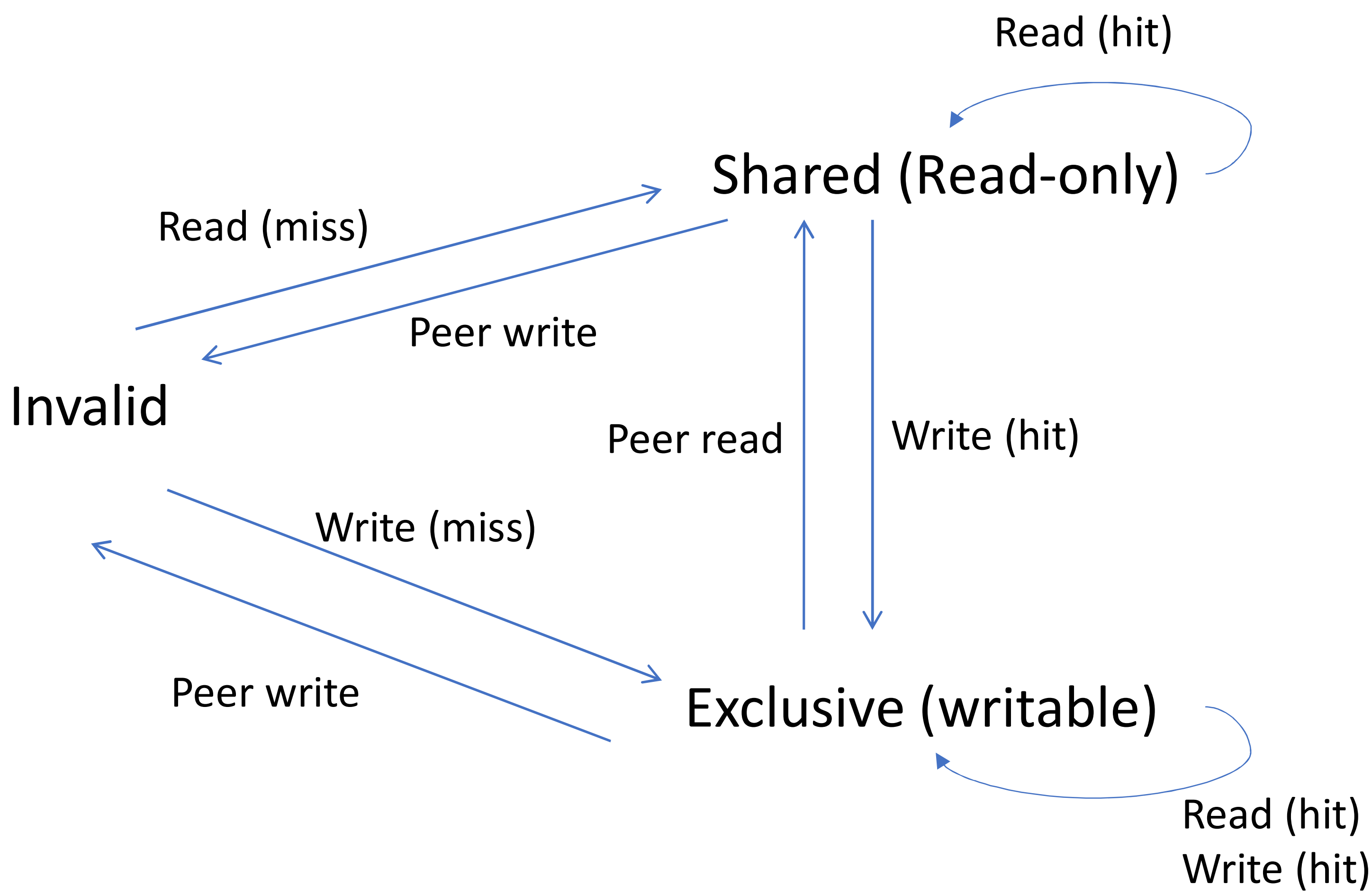
- Easy if all accesses go to main memory
  - Thread A changes main memory; thread B reads it

- With caching
  - What if new data is cached at processor A?
  - What if old data is cached at processor B

Write Back Cache Coherence

- Cache coherence = system behaves as if there is one copy of the data
  - If data is only being read, any number of caches can have a copy
  - If data is being modified, at most one cached copy
- On write: (get ownership)
  - Invalidate all cached copies, before doing write
  - Modified data stays in cache ("write back")
- On read:
  - Fetch value from owner or from memory

Cache State Machine



This is one simple example of a possible state machine.

Cache Coherence

- How do we know which cores have a location cached?
  - Snooping – shared bus; all cores see transactions
  - Directory Based
    - Better scalability than snooping
    - Hardware keeps track of all cached copies
    - On a read miss, if held exclusive, fetch latest copy and invalidate that copy
    - On a write miss, invalidate all copies
- Read-modify-write instructions
  - Atomically fetch cache entry exclusive and update
    - prevents any other cache from reading or writing the data until instruction completes

How Do Caches Affect Multi-thread Performance?

Experiment with a trivial critical section:

```
Class Counter {
    int lock;
    int value;
}
```

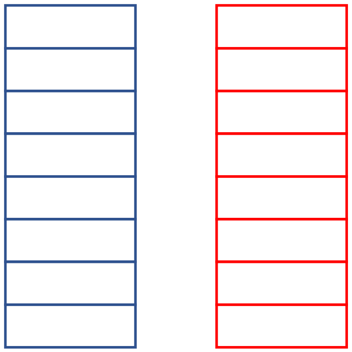
```
// A counter protected by a spinlock
Counter::Increment() {
   while (test_and_set(&lock))  ;        // spinlock acquire
   value++;                              // increment
   memory_barrier();                     // push updated values
   lock = FREE;                          // spinlock release
}
```

This is a very fine-grained critical section

A Simple Test of Cache Effects

Array(s) of 1K counters, each protected by a separate spinlock

- Array small enough to fit in cache



- Test 1: one thread loops over array
- Test 2: two threads loop over different arrays
- Test 3: two threads loop over single array
- Test 4: two threads loop over alternate elements in single array

Results (64 core AMD Opteron)

Time to execute one Increment()

Note: not speedup, just time to execute critical section

One thread, one array	51 cycles
Two threads, two arrays	52
Two threads, one array	197
Two threads, odd/even	127

Lock Performance: The Problem with Test-and-Set

```
Counter::Increment() {
    while (test_and_set(&lock));        // this is a write!
    value++;
    memory_barrier();
   lock = FREE;                         // so is this…
}
```

What happens if many processors try to acquire the lock at the same time?

- Threads trying to get lock acquire cache line ownership
- Hardware doesn't prioritize FREE

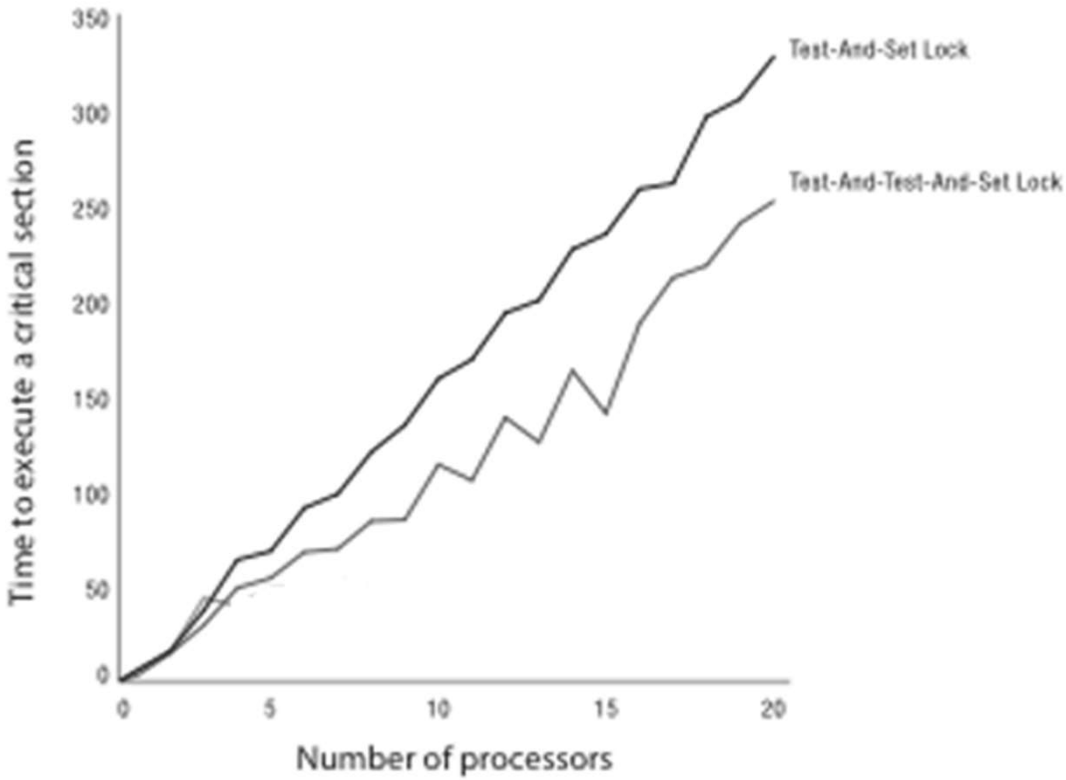
Test-and-Test-and-Set

```
Counter::Increment() {
   while (lock == BUSY || test_and_set(&lock)) ;
   value++;
   memory_barrier();
   lock = FREE;
}
```

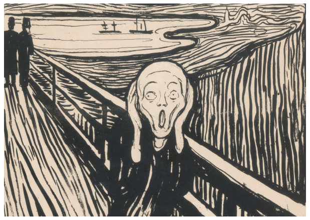
What happens if many processors try to acquire the lock?

- Lock value pings among caches

Test(-and-Test)-and-Set Performance



Some Possible Approaches



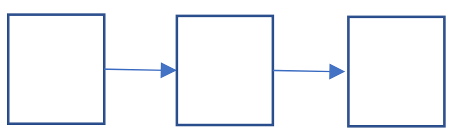
- Insert a delay in the spin loop
  - Helps but acquire is slow when not much contention
- Spin adaptively
  - No delay if few waiting
  - Longer delay if many waiting
  - Guess number of waiters by how long you wait
- Reduce Lock Contention
- Build a better lock

Reducing Lock Contention

- Fine-grained locking
  - Partition object into subsets, each protected by its own lock
    - Example: hash table buckets
  - vs. coarse-grained locking

- Per-processor data structures
  - Partition object so that most/all accesses are made by one processor
  - Example: per-processor heap

- Ownership/Staged architecture
  - Mostly only one thread at a time accesses shared data
  - Example: pipeline of threads

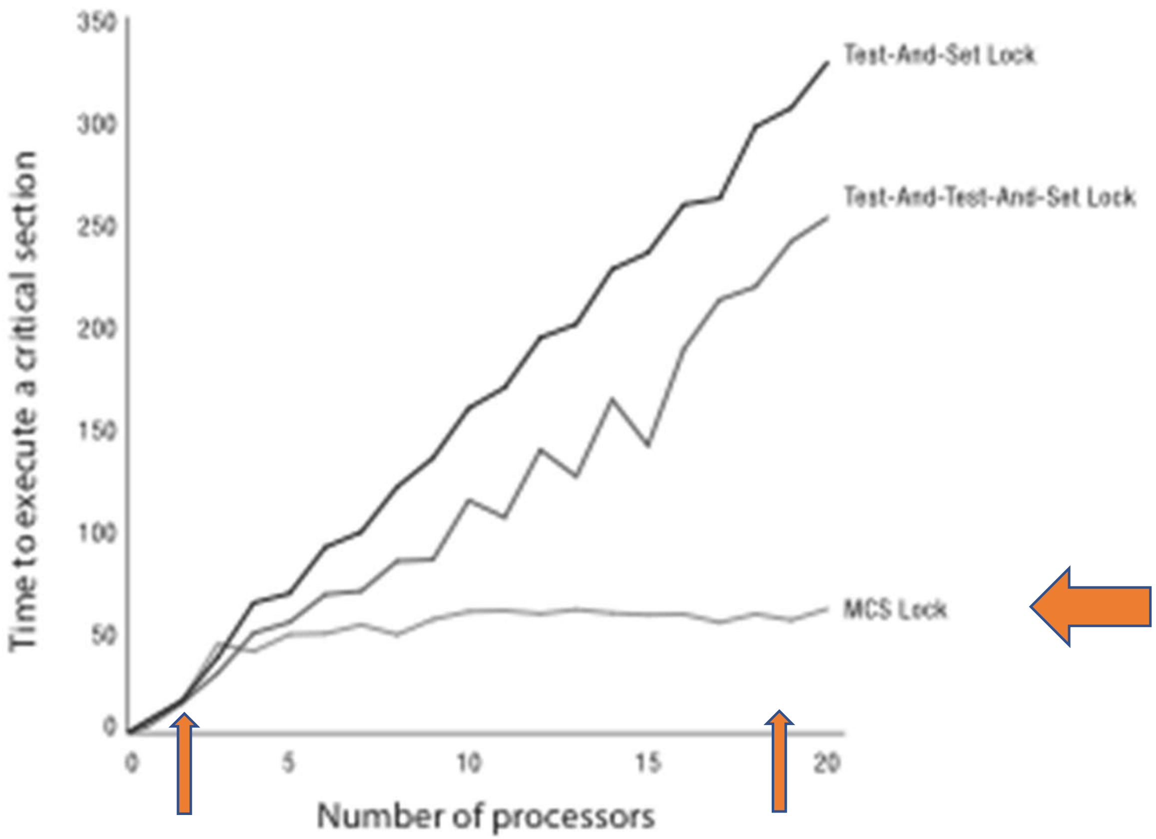


But what If Locks are Still Mostly Busy?

Reminder: Linux fastpath mutex implementation is an example of optimizing when the common case is locks are overwhelmingly idle

- MCS Locks (Mellor-Crummey and Scott)
  - Memory system-aware, optimized lock implementation for when lock is contended
- RCU Locks (read-copy-update)
  - Efficient readers/writers lock used in Linux kernel
  - Readers never block
  - Writer updates while readers operate (!), but at a cost…
- Both rely on atomic read-modify-write hardware instructions

More Robust Lock Performance



MCS Locks

Background: Atomic CompareAndSwap Instruction

- CompareAndSwap(memory address, comparison value, update value);

- Atomically:

```
if ( value at memory address == comparison value ) {
        value at memory address = update value;
        return true;
}
else return false;
```

Obviously, CompareAndSwap can be used to implement test-and-set semantics, although it costs an extra register. CompareAndSwap is more powerful, though, and MCS locks use the additional capability.

MCS Lock Object

```
// thread control block (per thread)
TCB {
   TCB *next;              // next in line
    bool needToWait;   // per-thread flag
}
```

```
MCSLock {
      Queue *tail = NULL; // end of line
}
```

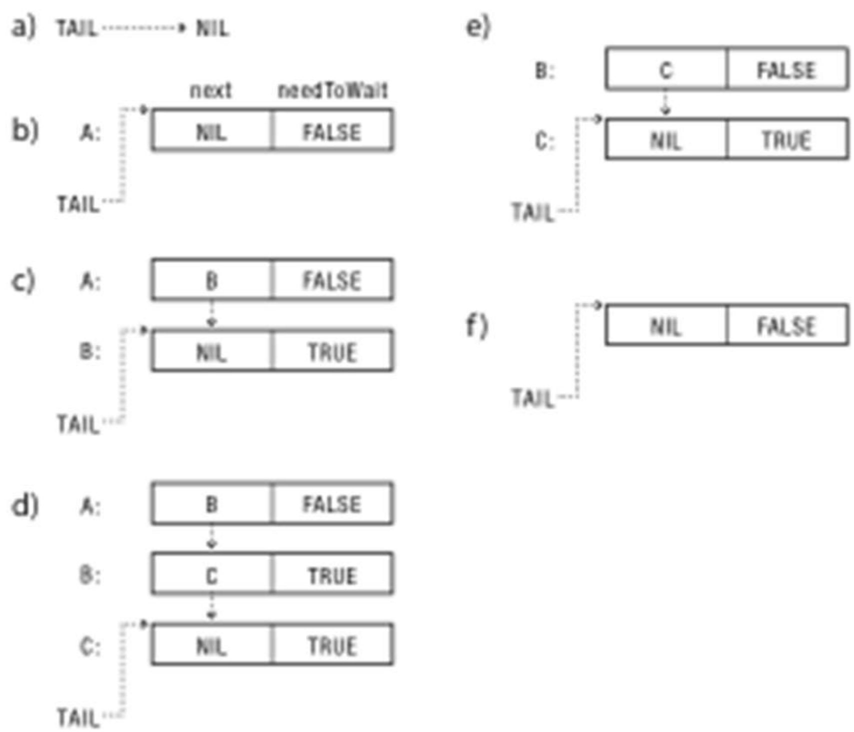
- MCSLock maintains a list of threads waiting for the lock
  - MCSLock::tail is reference to the last thread in list
  - The lock is free if tail is NULL
  - Otherwise, the lock is in use
    - the thread at the head of the list holds the lock
  - New thread uses CompareAndSwap to add to the tail

- Threads spin on their private needToWait flags

- Lock is handed off by the thread releasing the lock:

  next->needToWait = FALSE;

MCS In Operation



For this to work, must be able to do each required operation in one (atomic) CompareAndSwap instruction

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)   ;
    }
}
```

race

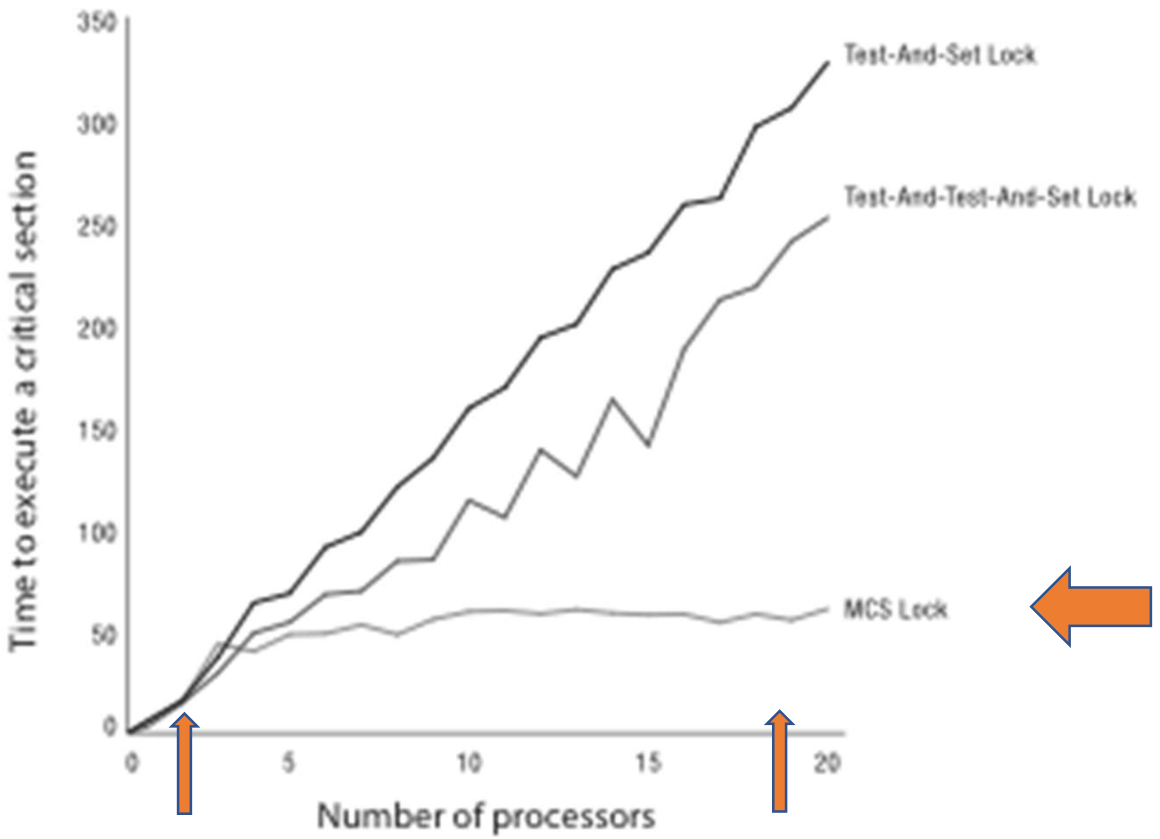
```
MCSLock::release() {
    if (!compareAndSwap(&tail,
                        myTCB, NULL)) {
        while (myTCB->next == NULL)  ;
        myTCB->next->needToWait=FALSE;
    }
}
```

Why is this fast?

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

Spin on thread-specific location

More Robust Lock Performance

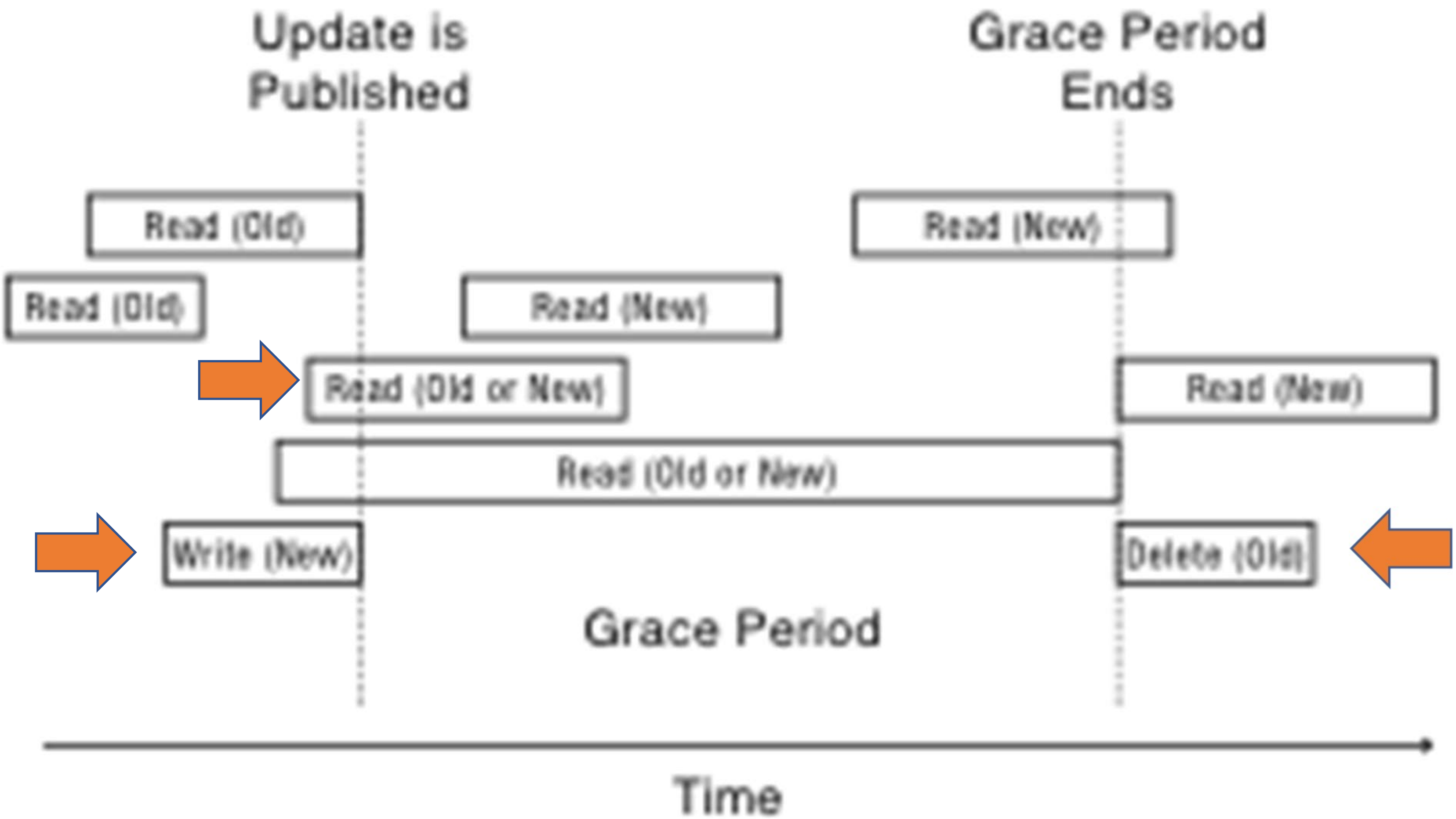


Read-Copy-Update Locks

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

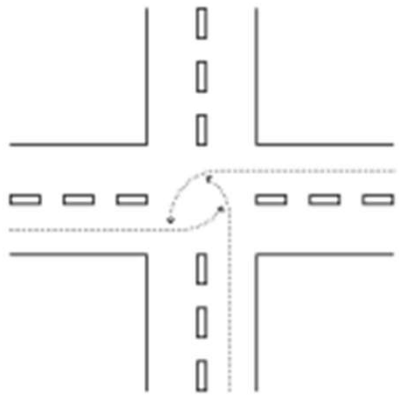
```
// called after each modification (after releasing write lock)
void synchronize() {
    c = atomicIncrement(globalCounter);
    for (p=0; p<NUM_CORES; p++ )
        while (PER_PROC_VAR(quiescentCount,p) < c)
            sleep(10);    // about a scheduling quantum
}
// called by scheduler  (if scheduler 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

Deadlock: Classic Example



Deadlock: <u>circular waiting</u> for resources

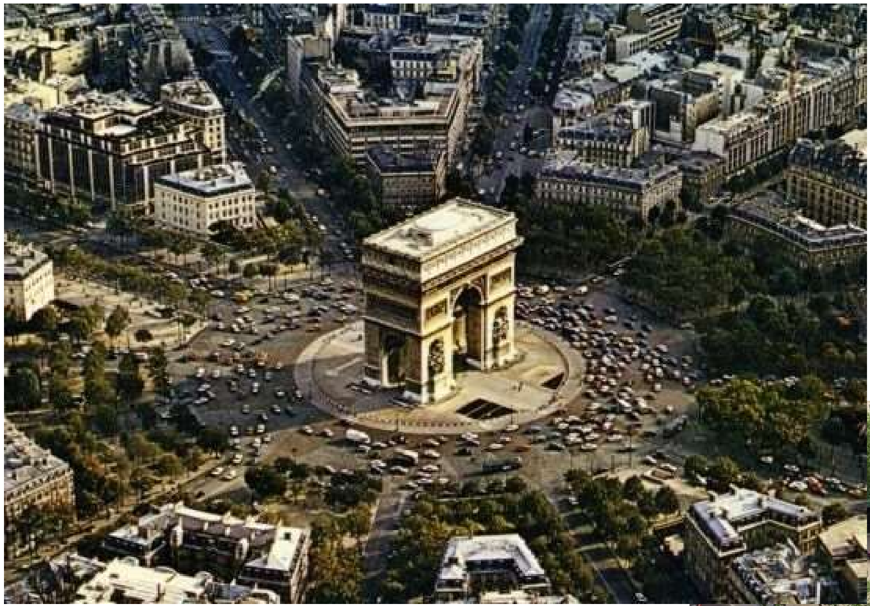
Deadlock is about the ***dynamic state*** of the computation

- The execution ***may*** deadlock (more likely than it ***will*** deadlock)

Static solution

Dynamic solution

Computer Science and Life



Deadlock Terminology

- **Deadlock**
  - circular waiting for resources
- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock, another thread's progress)
  - Preemptable: can be taken away (by OS)
    - *Releasable: can be given up by thread*
  - Non-preemptable: must leave with thread
- ***Starvation***
  - Although a thread can in theory make progress, in practice it waits indefinitely
- *Livelock*
  - *failure to progress due to repeated conflicting actions taken by multiple threads*

[Livelock]

MIND & BODY

Researchers explain that awkward sidewalk dance we all do – and hate

As you approach someone on the street, you try to move out of the way but she steps in the same direction. Why do we engage in this sidewalk dance?

Sept. 11, 2018, 2:07 PM PDT / Source: TODAY

By Meghan Holohan

Walking down the sidewalk, you notice a person coming directly toward you. As you near each other, you realize you're going to collide.

You step to the right.

The other person steps to the left.

Now you're in each other's way – again. Then you shift to the left and she moves right and you're in a standoff. After moving back and forth, you finally stop and let her pass.

This phenomenon, a sidewalk dance of sorts, seems to happen all the time.

https://www.today.com/health/sidewalk-dance-when-pedestrians-keep-stepping-way-t137227

Deadlock Example with Two Locks

Thread A	Thread B
lock1.acquire();	lock2.acquire();
lock2.acquire();	lock1.acquire();
lock2.release();	lock1.release();
lock1.release();	lock2.release();

Bidirectional Bounded Buffer

Thread A

```
buffer1.put(data);
buffer1.put(data);

buffer2.get();
buffer2.get();
```

Thread B

```
buffer2.put(data);
buffer2.put(data);

buffer1.get();
buffer1.get();
```

Suppose buffer1 and buffer2 both start almost full.

Two locks and a condition variable

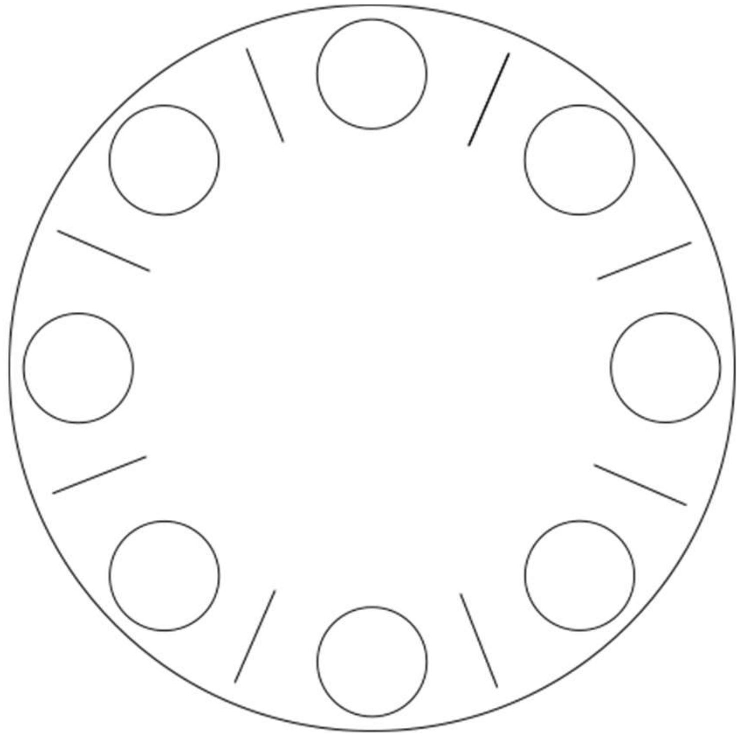
Thread A

```
lock1.acquire();
…
lock2.acquire();
while (need to wait) {
    condition.wait(lock2);
}
lock2.release();
…
lock1.release();
```

Thread B

```
lock1.acquire();
…
lock2.acquire();
…
condition.signal(lock2);
…
lock2.release();
…
lock1.release();
```

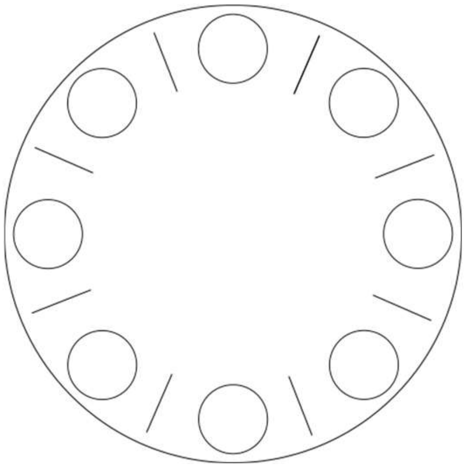
Dining Lawyers



Lawyers alternate talking on their phones and eating.
Each lawyer needs two forks to eat.
Each grabs fork on the right then fork on the left.
They're lawyers…

Necessary Conditions for Deadlock

1. Limited access to resources
   - If infinite resources, no deadlock!
2. No preemption
   - If resources are preemptable, can break deadlock
3. Hold and Wait
   - Threads don't voluntarily give up resources
4. Circular chain of requests



Dealing with Deadlock

Preventing and Avoiding Deadlock

1. Prevent deadlock **purely statically** by exploiting or limiting program behavior to make sure at least one of the four conditions can't ever hold
   - Limit program from doing anything that might lead to deadlock – can never deadlock
2. Avoid deadlock **by dynamically** monitoring program state and steering clear of "bad states"
   - Program can sometimes deadlock. Detect when deadlock could develop and intervene.
   - Requires knowing something about possible future thread behavior → some static analysis
3. Detect and recover **purely dynamically**
   - If we can rollback a thread's changes to process state, we can fix a deadlock once it occurs
   - (So, okay, the ability to rollback has to be provided statically…)

Exploit or Limit Behavior

- Provide enough resources
  - How many forks are enough?
- Eliminate wait while holding
  - Acquire all locks at once, or none
  - Release lock when calling out of module
- Eliminate circular waiting
  - Lock ordering: always acquire locks in a fixed order
    - Example: move file from one directory to another

[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.

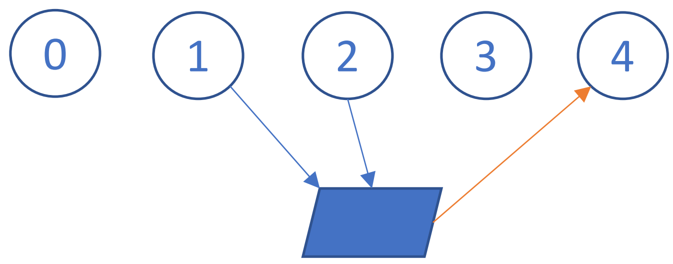
The function locks the objects using an unspecified sequence of calls to their members lock, try_lock and unlock that ensures that all arguments are locked on return (without producing any deadlocks).

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/

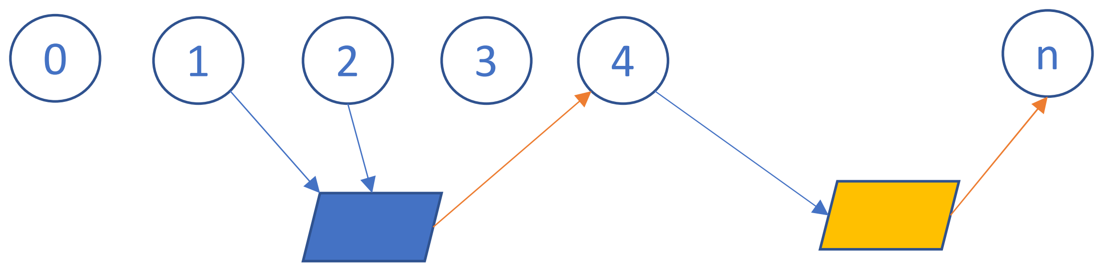
See code sample there for clearer connection to deadlock issues.

Static Resource Ordering



- Thread shown holds some resources and is trying to acquire another one
  - The one labelled 4
- What can happen?
  - If 4 is free, no problem
  - If 4 is busy, possible deadlock?
    - Deadlock if whoever holds 4 wants or will want 1 or 2 (or transitively…)

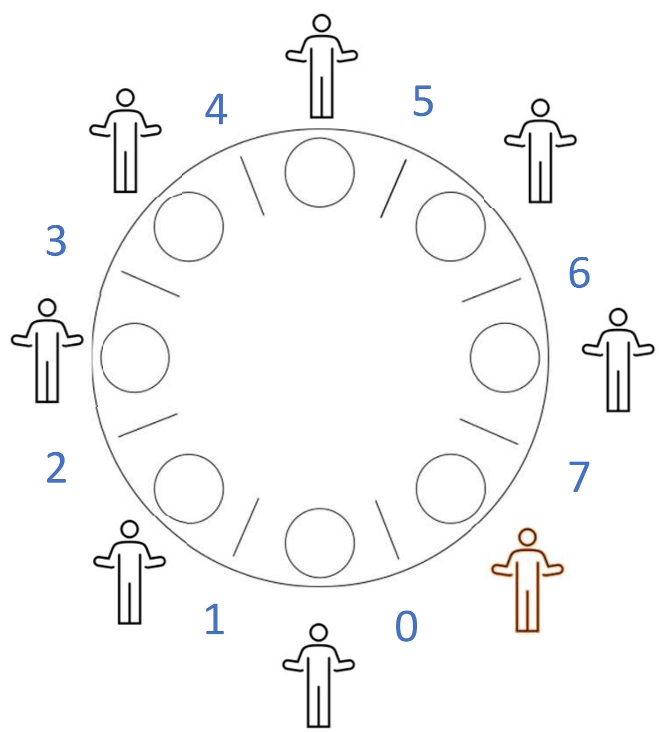
Static Resource Ordering



- If 4 is held, and if the thread holding it (eventually) is blocked waiting for some resource, that resource must have index greater than 4
- Similarly, whoever is holding resource n can only wait on resources with indices greater than n
- So, not cycle of wait-for is possible

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



Dynamic Monitoring Example

Thread 1

1. Acquire A
2. Acquire C
3. If (cond) Acquire B

Thread 2

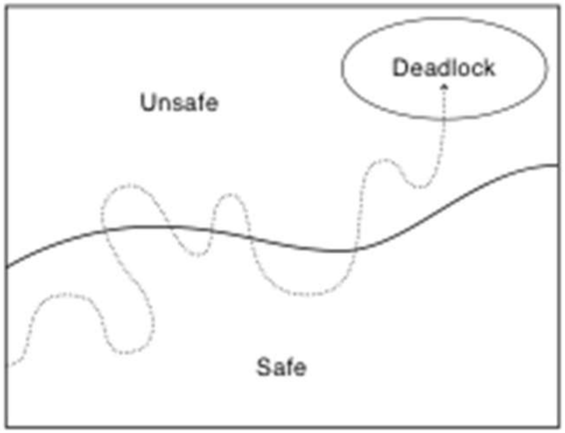
1. Acquire B
2. Acquire A

How can we "pause" thread execution to make sure to avoid deadlock?

Deadlock Dynamics

- Safe state:
  - For any possible sequence of **future** resource requests, it is possible to eventually grant all requests (perhaps by delaying some requests)
- Unsafe state:
  - Some sequence of resource requests can result in deadlock, even if you reserve all remaining resources for one chosen thread (in case it will want them)
- Doomed state:
  - All possible computations lead to deadlock

Possible System States



Dining Lawyers

- What are the unsafe states?
- What are the safe states?
- What are the doomed states for Dining Lawyers?

- Note: In Dining Lawyers we know **exactly** what each thread will do. This dynamic approach to deadlock prevention is oriented toward situations where threads conditionally acquire resources
  - *"I need up to two forks (but sometimes I use just one fork and I'm done)"*

Communal Dining Lawyers

- n forks in middle of table
- n lawyers, each can take one fork at a time

- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Communal Mutant Dining Lawyers

- N forks in the middle of the table
- N lawyers, each takes one forks at a time
- Lawyers need k forks to eat, k > 1

- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Maybe 1, Maybe 2 Forks Lawyers

- Lawyers in a circle with a fork between each adjacent pair
- “Nobody’s going to tell me in what order I have to pick up my forks!”
- If a lawyer is holding one fork, the next thing s/he might do is
  - Try to get the other fork, or
  - Use the one fork and then put it down

- Note that the situation where every lawyer is holding a fork is **not** (necessarily) deadlock
- Note that the situation where every lawyer is holding one fork and waiting to get a second **is** deadlock

General Method: Banker's Algorithm
(Pessimistic)

Basic Setup

- There is a resource manager that "owns" all the resources
  - There can be many types of resources, all controlled by one manager
- Threads request resources from the manager and return them to the manager
  - Requests/allocations are one resource at a time

Dynamic Operation

- Threads state maximum resource needs to manager when they start
- Threads request resources dynamically as needed
- Manager delays granting request if doing so could lead to deadlock
- Manager grants request if some sequential ordering of threads is deadlock free

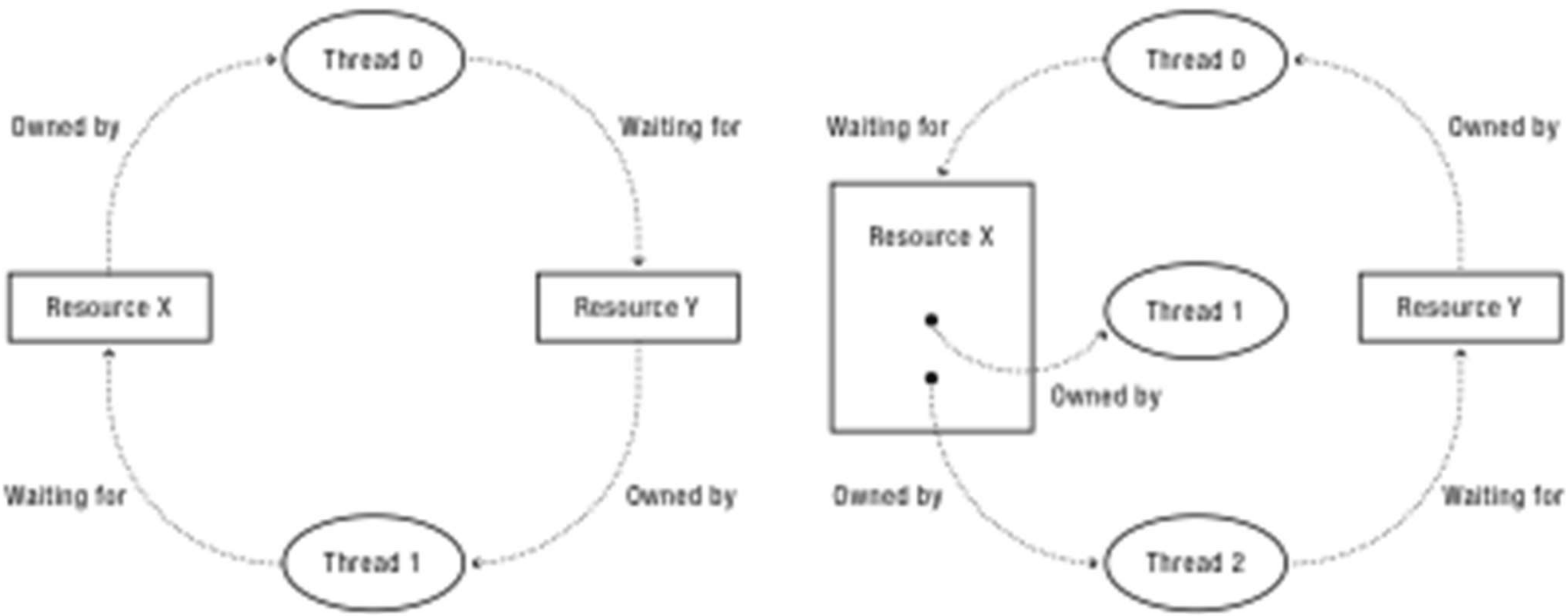
Banker's Algorithm

- Grant request iff result is a safe state
  - Simple Example: proceed if total available resources - # allocated >= max remaining that might be needed by this thread in order to finish
- More generally, allocate resource if manager can find a way for all threads to eventually finish, even if each asks for its maximum request, even if the requested resource is allocated
  - Otherwise, don't allocate and block thread until it's safe to grant its request
- Why would you want to go through all this trouble?
  - Sum of maximum resource needs of threads can be greater than the total resources
  - No static ordering (or any other) constraints on acquiring resources that have to be respected in the code

Another Approach: Detect and Repair
(Optimistic)

- Possible Algorithm
  - Scan wait for graph
  - Detect cycles
  - Fix cycles
- Proceed without the resource
  - Requires robust exception handling code
- Roll back and retry
  - Transaction: all operations are provisional until have all required resources to complete operation

Detecting Deadlock



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)

```
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?

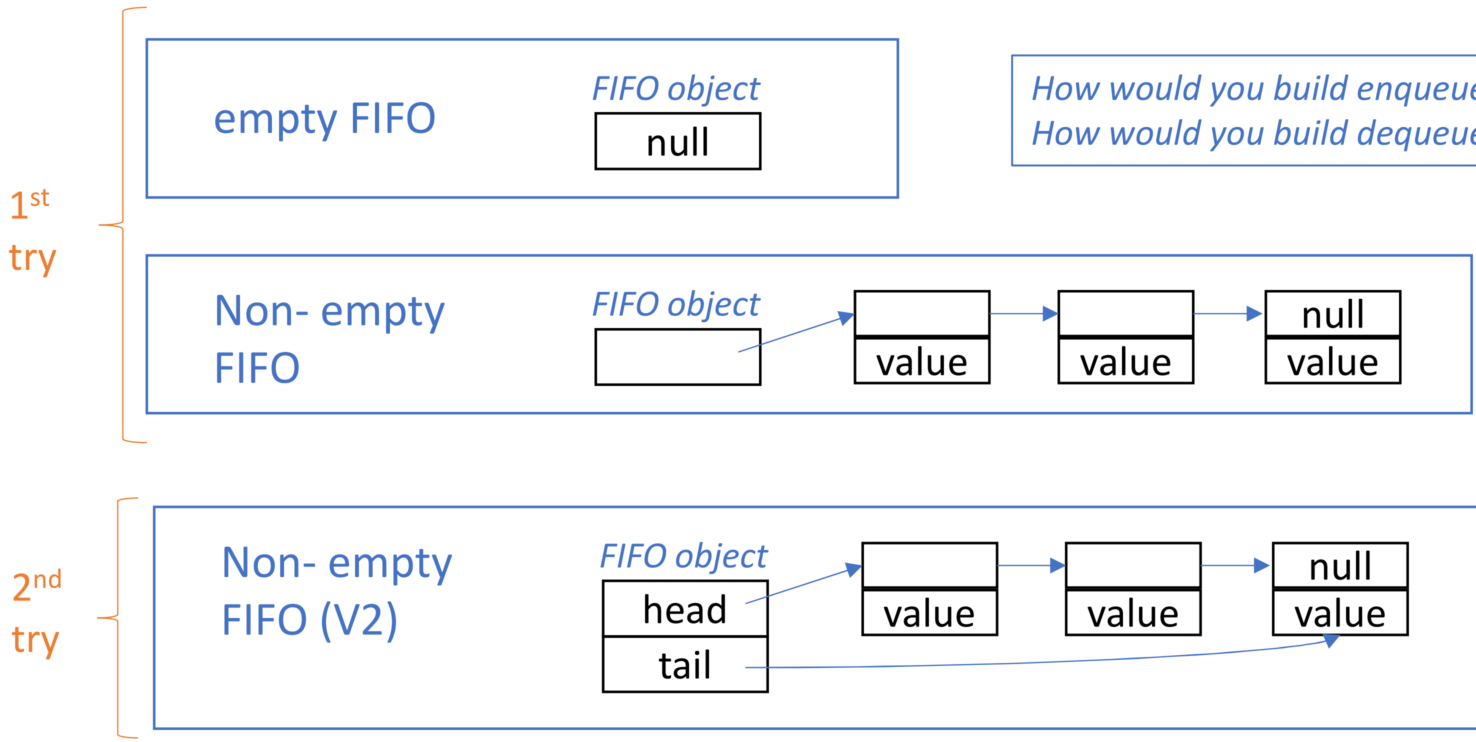
- Priority inversion and locks
  - Assume threads have been assigned priorities, and we'd like to preferentially allocate cores to the highest priority runnable threads
  - Now suppose a low priority thread holds a lock needed by a high priority thread
  - Medium priority threads might steal the core 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?

How would you build enqueue using CAS?
How would you build dequeue using CAS?



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

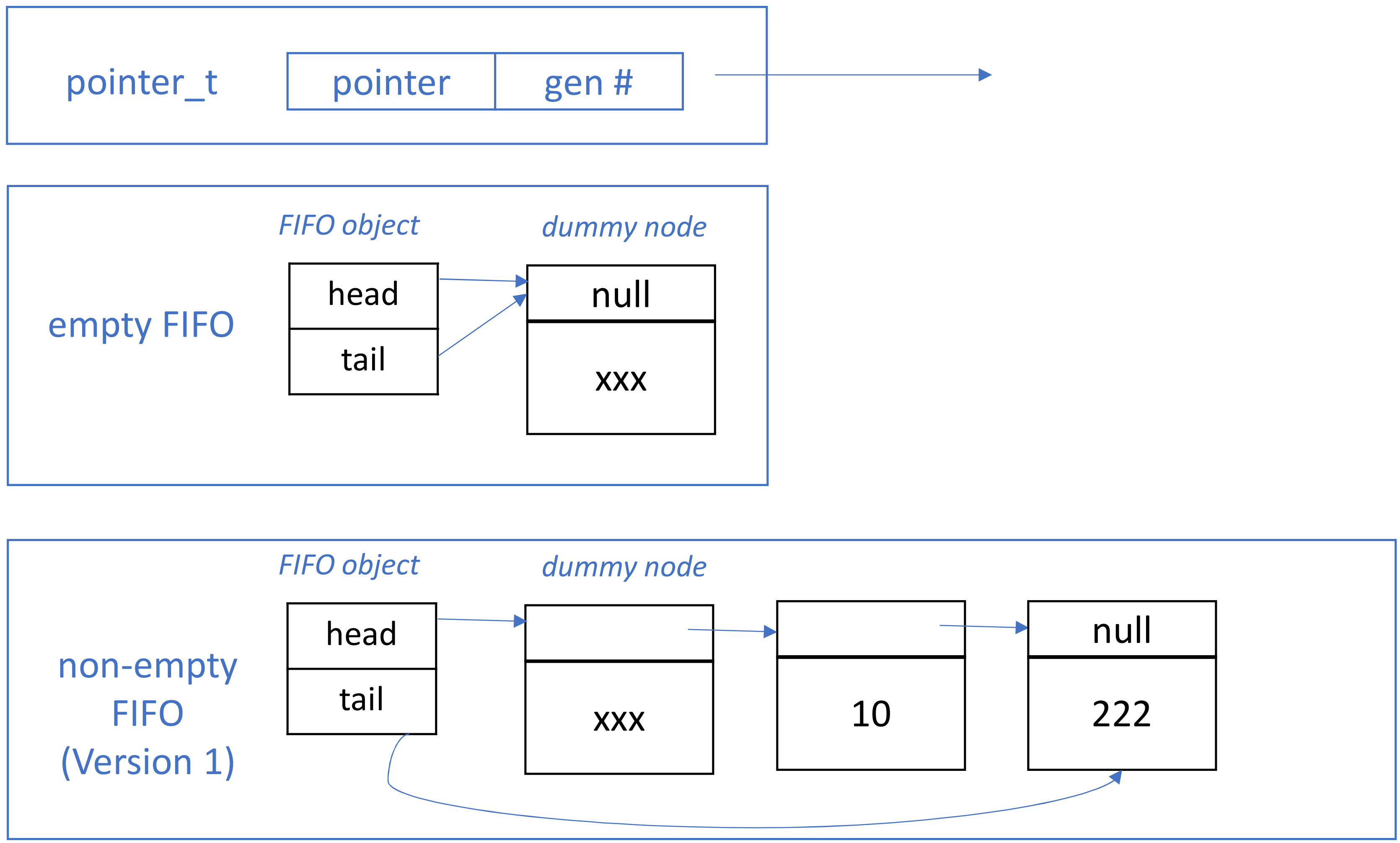
```
structure pointer_t     {ptr: pointer to node_t, count: unsigned integer}
structure node_t        {value: data type, next: pointer_t}
structure queue_t       {Head: pointer_t, Tail: pointer_t}

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
```

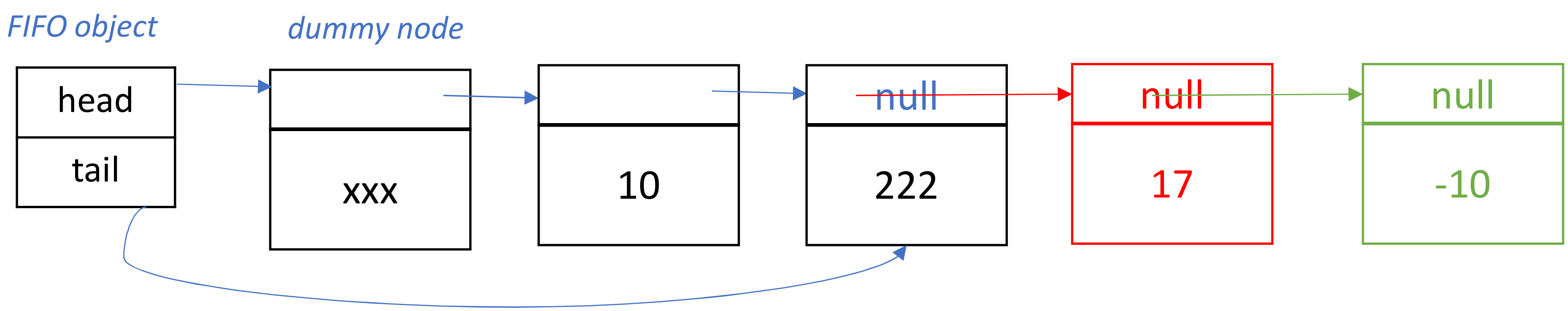
Pointers are stored with a generation number in one 8-byte quantity
(32-bit pointer + 32-bit generation number)

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

Non-blocking FIFO



Non-blocking FIFO: enqueue value 17



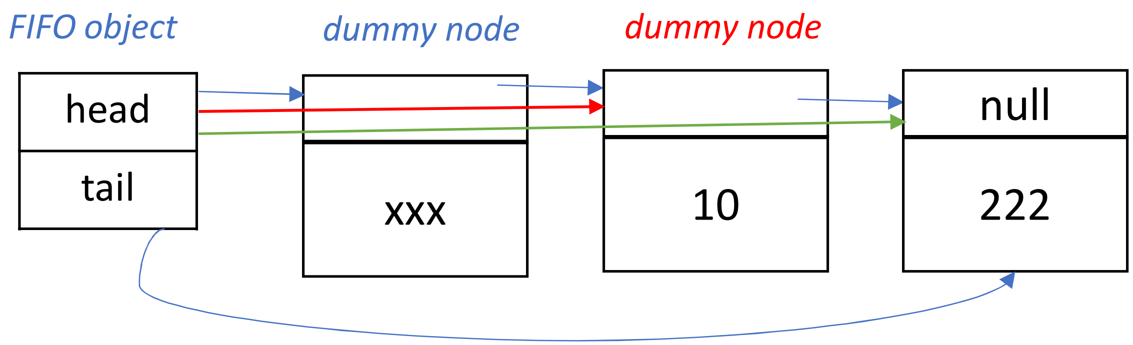
1. Update tail->next to point to new node
2. Update tail to point to new node

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

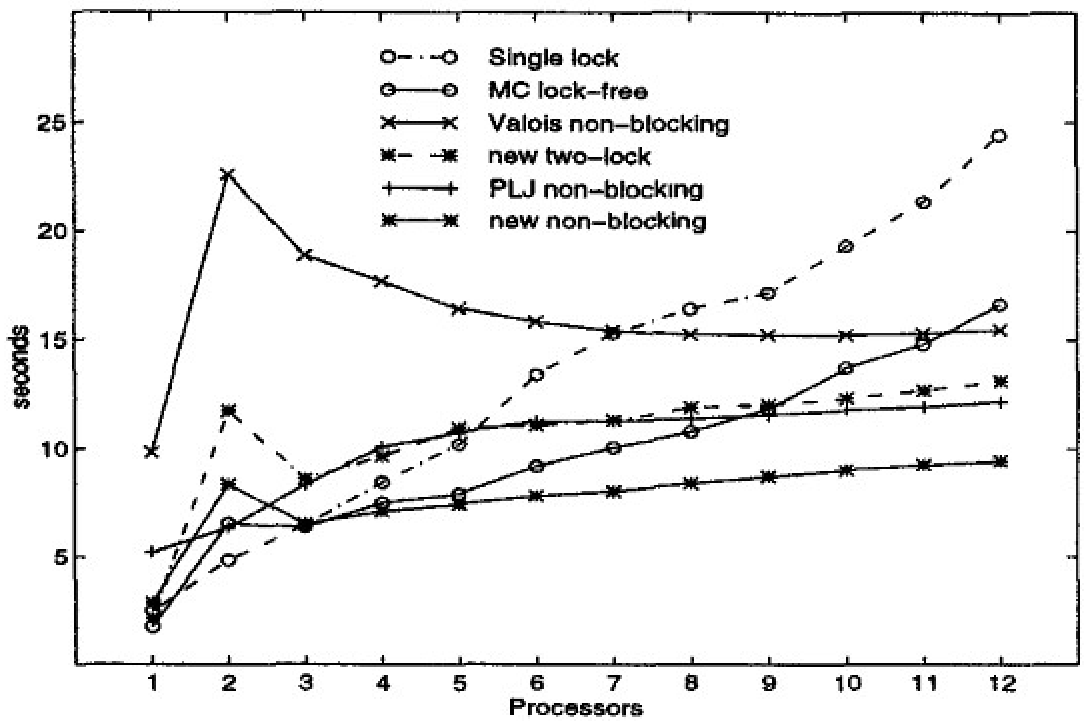


Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

12 processor Silicon Graphics Challenge