Synchronization: Performance and Multi-Object

Module 7

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

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 mutex that is used to restrict concurrent operations
- Does put() need to obtain the mutex?
- Does read() need to obtain the mutex?

Readers/Writers Locks

- Normal mutex has semantics “one thread at a time”
- We want semantics “any number of readers but no writers” or “just one writer”
- Readers/writers locks support this
  - Interface: startRead() … doneRead()
    startWrite() … doneWrite()

R/W Locks Implementation

- Take a few minutes and implement them
  - In teams

- 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?

R/W Locks 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 Implementation

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

R/W Lock Implementation

- What’s bad about our implementation?
- What alternative semantics might you want?

Synchronization Performance: Caches

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

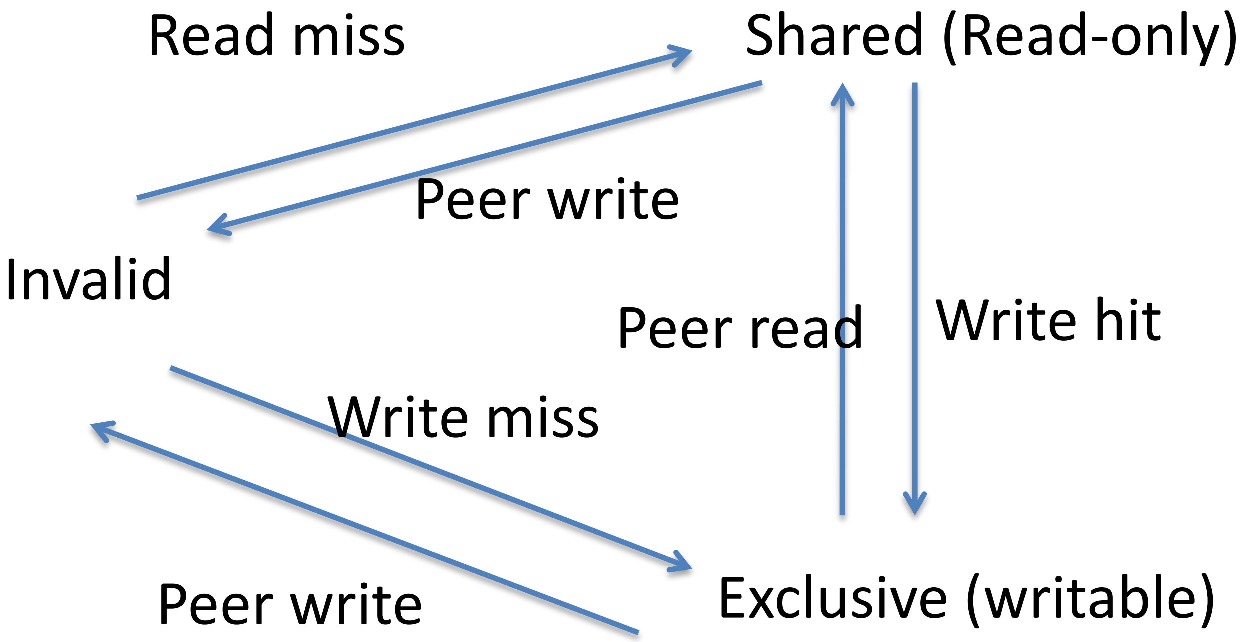
Performance: Multiprocessor Cache Coherence

- 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
- 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



Read miss

Shared (Read-only)

Peer write

Invalid

Peer read

Write hit

Write miss

Peer write

Exclusive (writable)

Cache Coherence

- How do we know which cores have a location cached?
  - Snooping – shared bus; all cores see transactions
  - Directory Based:
    - 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

A Simple Critical Section

```
// A counter protected by a spinlock
Counter::Increment() {
    while (test_and_set(&lock))
        ;
    value++;
    memory_barrier();
    lock = FREE;   // atomic write
}
```

A Simple Test of Cache Behavior

Array 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)

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

time to execute one Increment()

Lock Performance: The Problem with Test-and-Set

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

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

- Hardware doesn't prioritize FREE

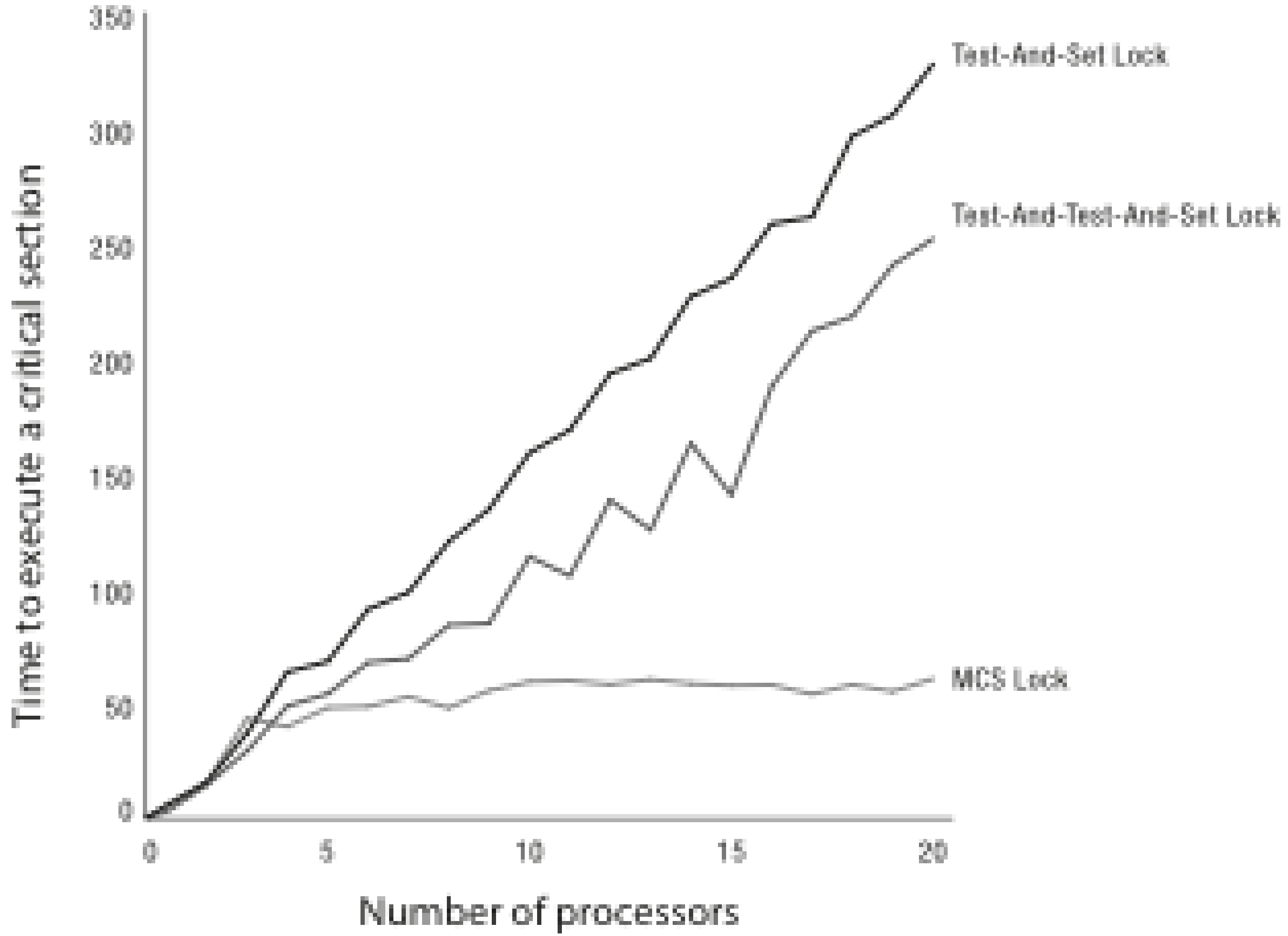
Test-and-Test-and-Set

```
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 between caches

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



Some 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

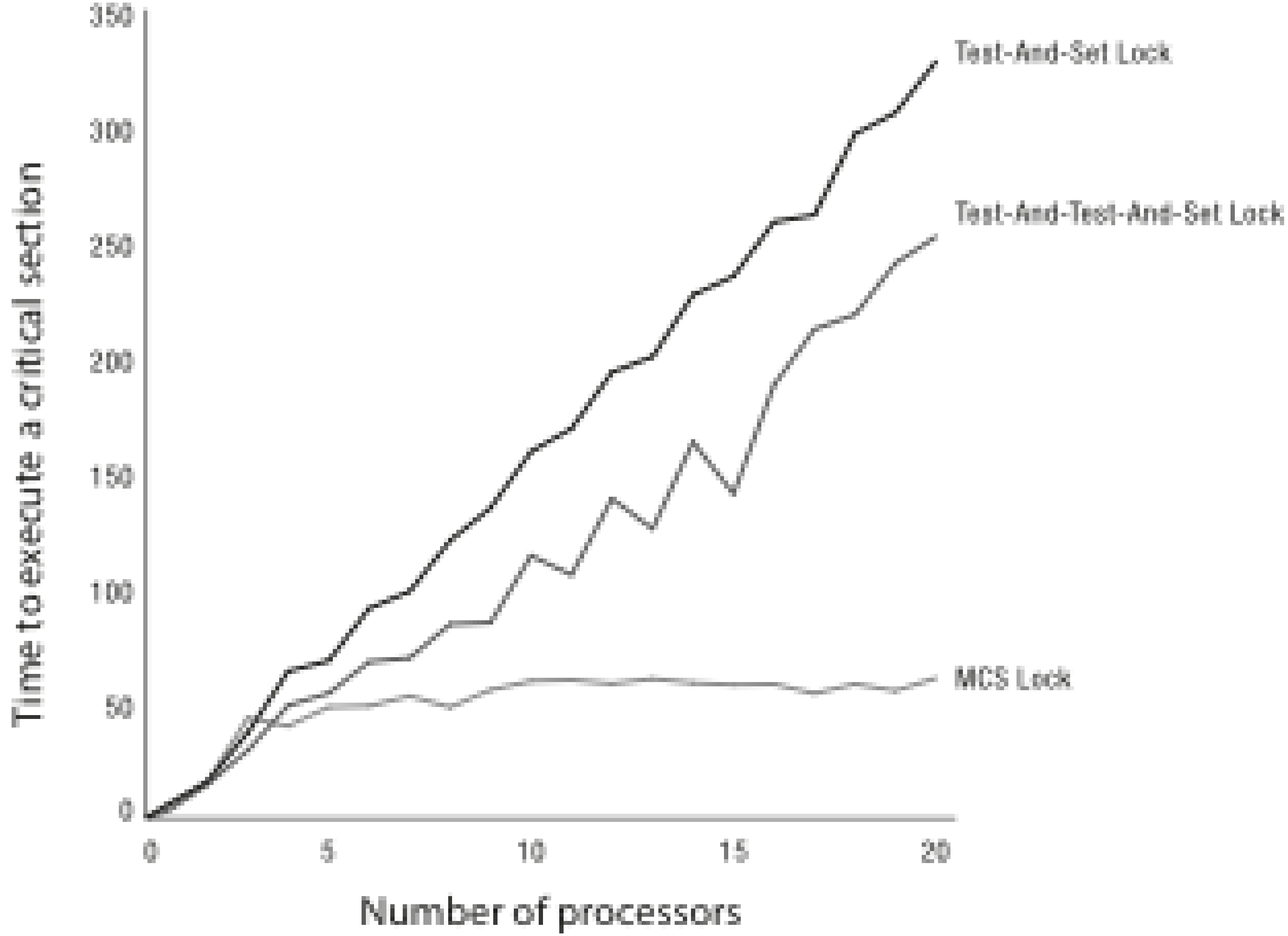
Reducing Lock Contention

- 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
  - Only one thread at a time accesses shared data
  - Example: pipeline of threads

What If Locks are Still Mostly Busy?

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

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



MCS Locks

Background: Atomic CompareAndSwap Instruction

- Operates on a memory word
- Check that the value of the memory word hasn’t changed from what you expect
  - E.g., no other thread did compareAndSwap first
- If it has changed, return an error (and loop)
- If it has not changed, set the memory word to a new value

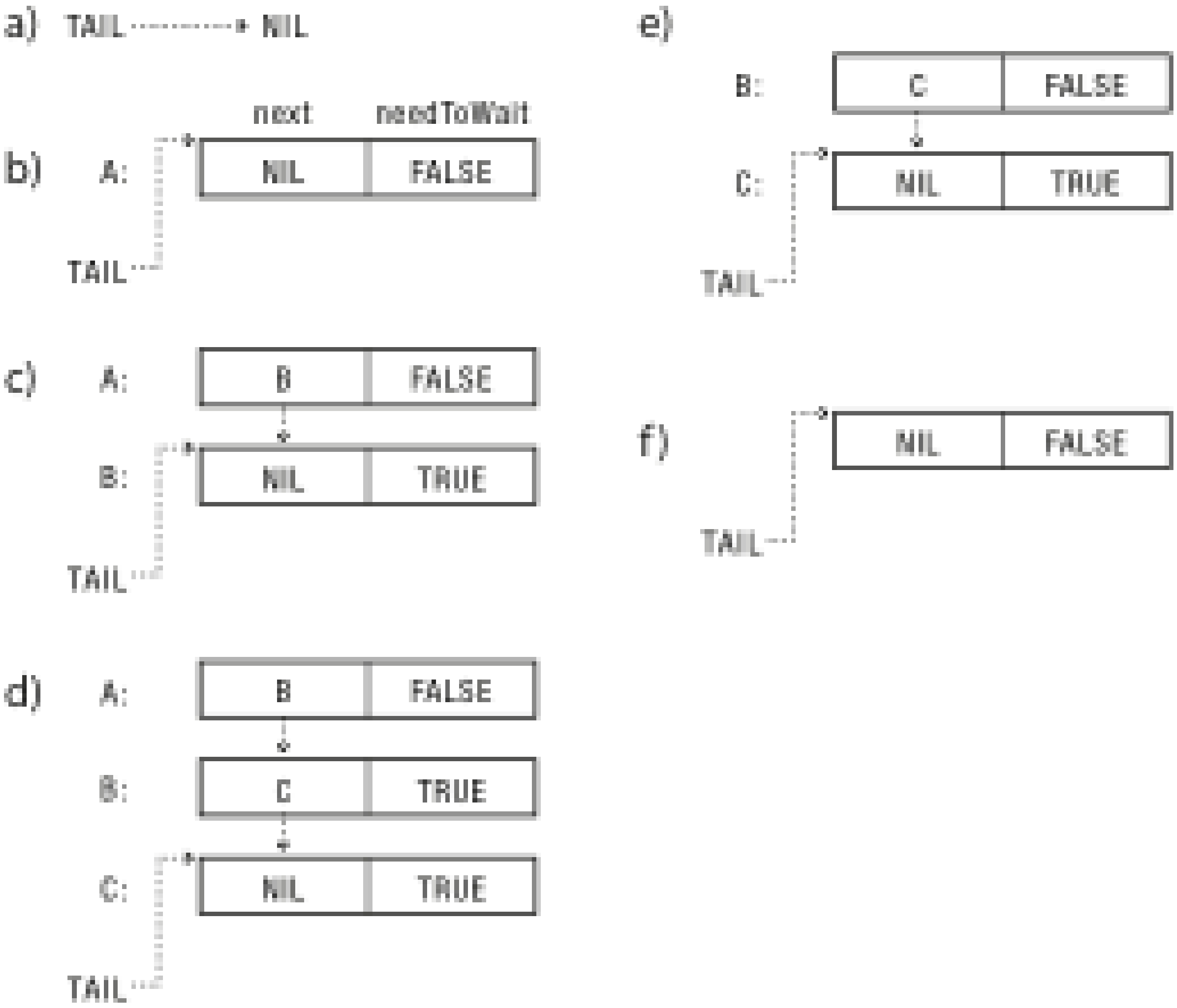
MCS Lock

```
TCB {
        TCB *next;              // next in line
         bool needToWait;
}
MCSLock {
        Queue *tail = NULL; // end of line
}
```

- Maintain a list of threads waiting for the lock
  - Front of list holds the lock
  - MCSLock::tail is last thread in list
  - New thread uses CompareAndSwap to add to the tail

- Lock is passed by thread releasing the lock setting next->needToWait = FALSE;
  - Next thread spins while its needToWait is TRUE

MCS In Operation



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)
          ;
   }
}
```

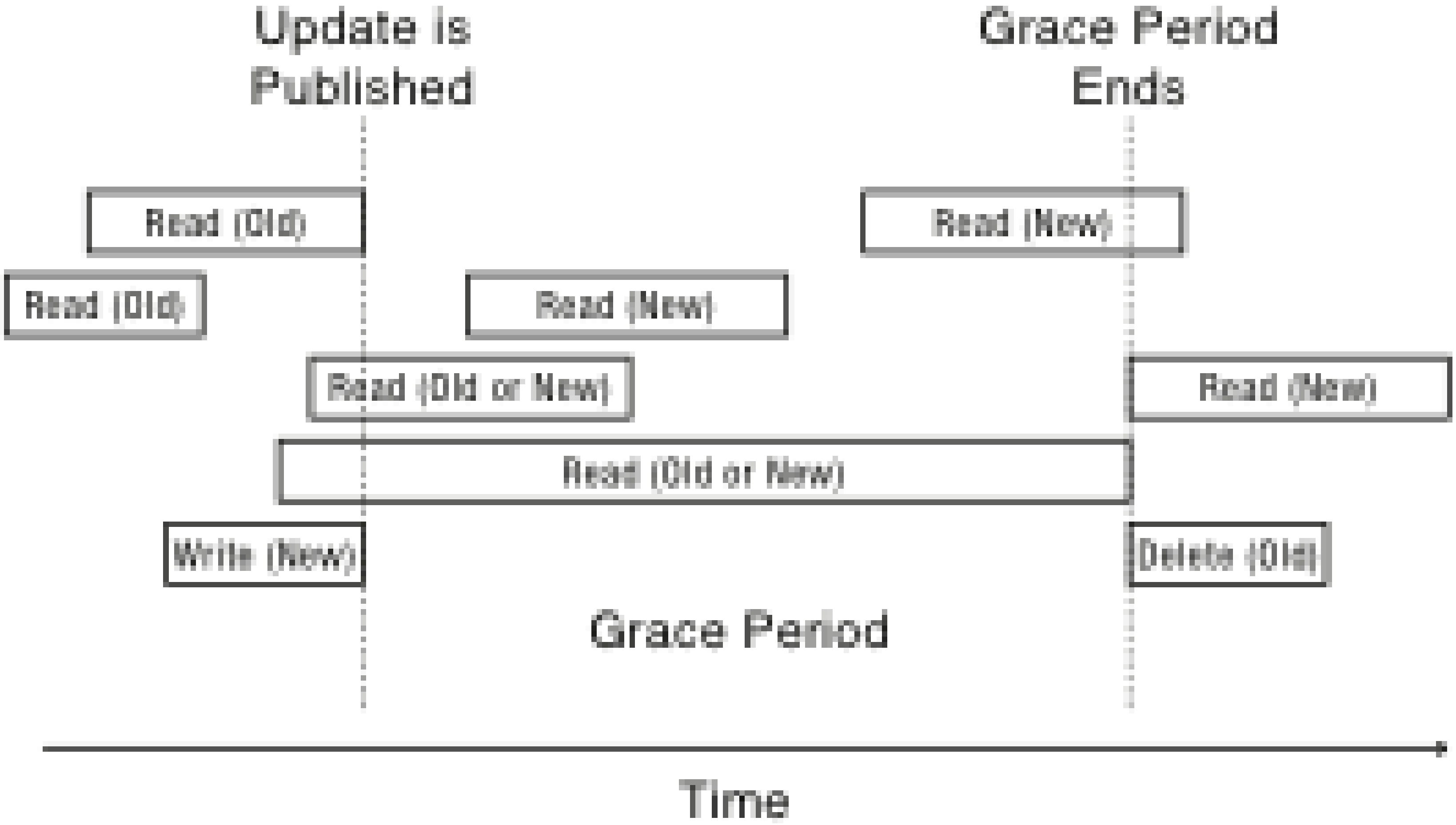
```
MCSLock::release() {
   if (!compareAndSwap(&tail,
                myTCB, NULL)) {
      while (myTCB−>next == NULL)
         ;

    myTCB−>next−>needToWait=FALSE;
   }
}
```

Read-Copy-Update Locks

Read-Copy-Update

- Goal: very fast reads to shared data
  - Reads proceed without first acquiring a lock
  - OK if write is (very) slow and infrequent
- Multiple concurrent versions
  - Readers may see old version for a limited time
- Restricted update
  - Writer computes new version of data structure
  - Publishes new version with a single atomic instruction
- Relies on integration with thread scheduler
  - Guarantee all readers complete within grace period, and then garbage collect old version

Read-Copy-Update



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 read or write lock
- Writer
  - Acquire write lock
    - One writer at a time
  - Compute 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

RCU Lock Implementation

```
void ReadLock() { disableInterrupts(); }
void ReadUnlock() { enableInterrupts(); }
void WriteLock() { writerSpin.lock(); }
void WriteUnlock() { writerSpin.unlock(); }

void publish( void **pp1, void *p2) {
    memory_barrier();
    *pp1 = p2;   // 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
void QuiescentState() {
    memory_barrier();
     PER_PROC_VAR(quiescentCount) = globalCounter;
      memory_barrier();
}
```

Deadlock

Deadlock Definition

- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock => starvation, but not vice versa

Example: 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	Thread B
`buffer1.put(data);`	`buffer2.put(data);`
`buffer1.put(data);`	`buffer2.put(data);`
`buffer2.get();`	`buffer1.get();`
`buffer2.get();`	`buffer1.get();`

Suppose buffer1 and buffer2 both start almost full.

Two locks and a condition variable

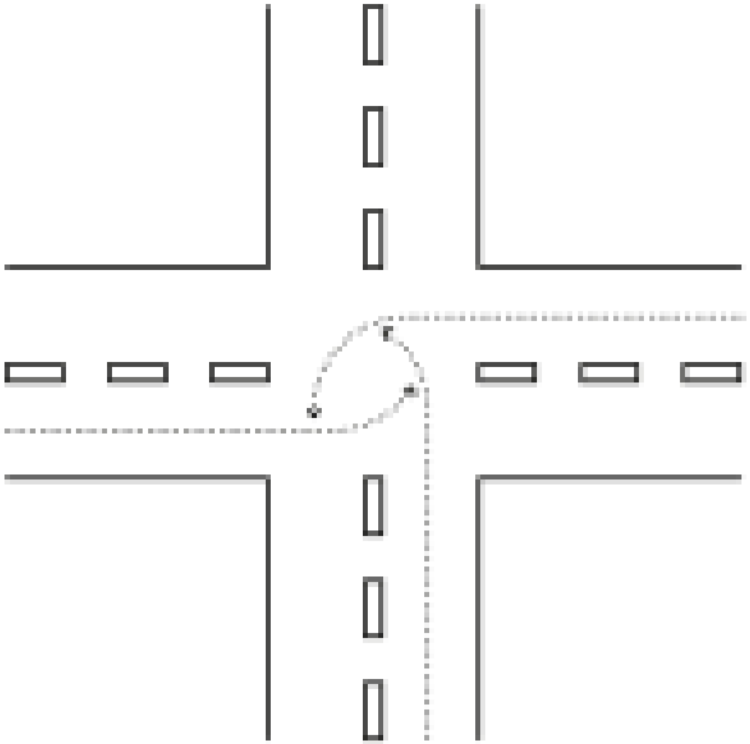
Thread A

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

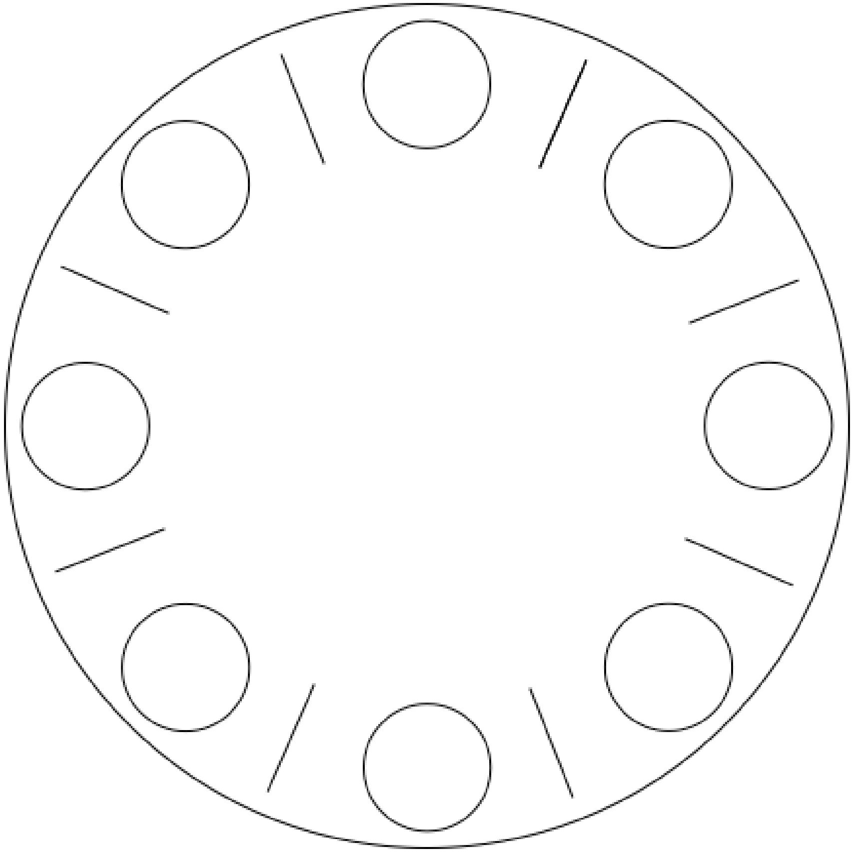
Thread B

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

Yet another Example



Dining Lawyers



Each lawyer needs two chopsticks to eat.
Each grabs chopstick on the right first.

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

Question

- How does Dining Lawyers meet the necessary conditions for deadlock?
  - Limited access to resources
  - No preemption
  - Hold and wait
  - Circular chain of requests
- How can we modify Dining Lawyers to prevent deadlock?

Preventing and Avoiding Deadlock

Preventing Deadlock

- Make sure at least one of the four conditions can't hold by
  - Exploit or limit program behavior
    - Limit program from doing anything that might lead to deadlock
  - Predict the future
    - If we know what program will do, we can tell if granting a resource might lead to deadlock
  - Detect and recover
    - If we can rollback a thread, we can fix a deadlock once it occurs

Exploit or Limit Behavior

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

Example

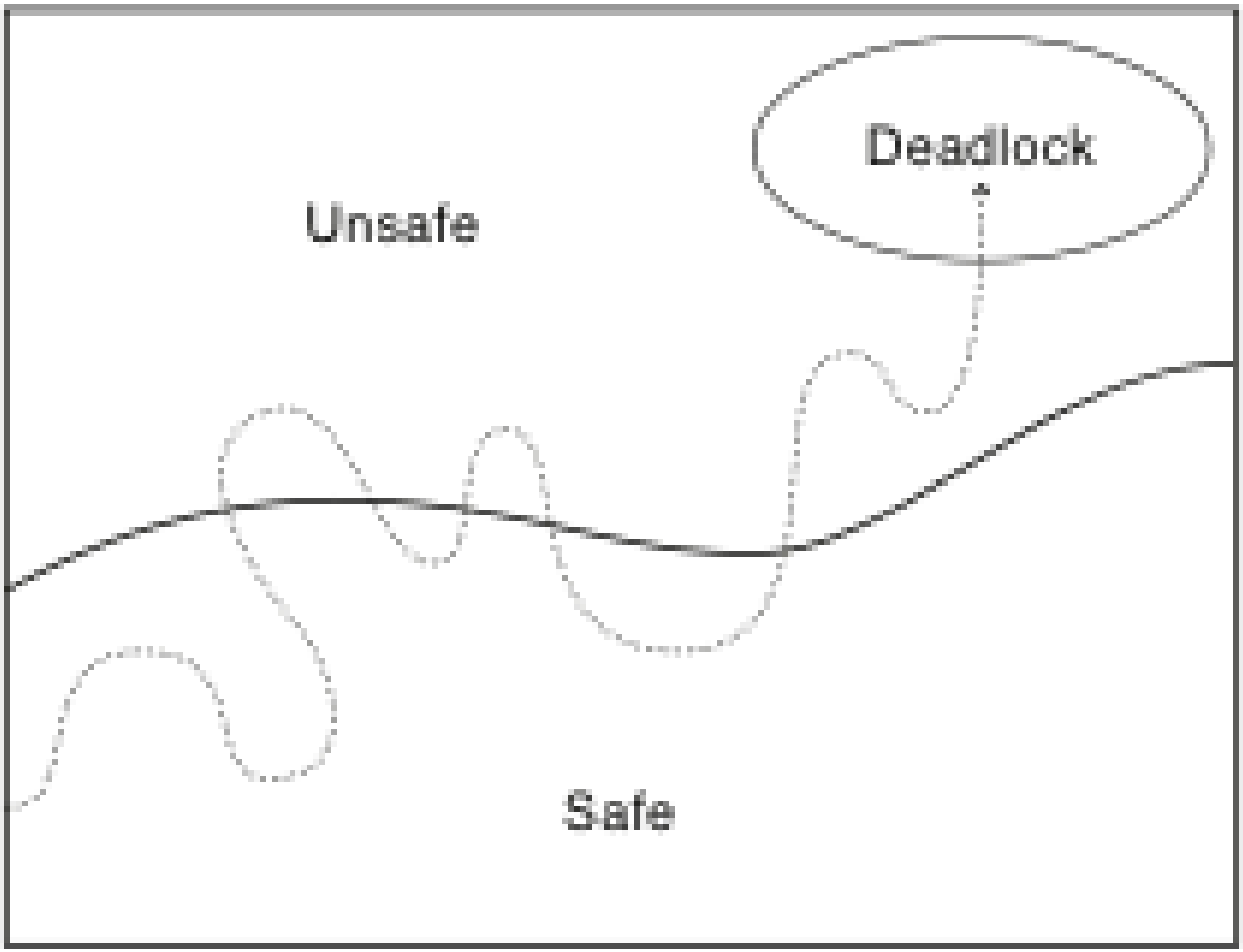
Thread 1	Thread 2
1. Acquire A	1.
2.	2. Acquire B
3. Acquire C	3.
4.	4. Wait for A
5. If (cond) Acquire B	

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 delay allocating resources
- Doomed state:
  - All possible computations lead to deadlock

Possible System States



Question

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

Communal Dining Lawyers

- n chopsticks in middle of table
- n lawyers, each can take one chopstick at a time
- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Communal Mutant Dining Lawyers

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

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

Avoiding Deadlock: Predict the Future

- Banker's algorithm
  1. Threads state maximum resource needs in advance

- Aside (from Banker's Alg)
  - If the app knows the maximum resources it can possibly want going forward, how could we prevent deadlock?

Avoiding Deadlock: Predict the Future

- Banker's algorithm
  1. Threads state maximum resource needs in advance
  2. Allocate resources dynamically when resource is needed
     1. wait if granting request could lead to deadlock
  - Request can be granted if some sequential ordering of threads is deadlock free

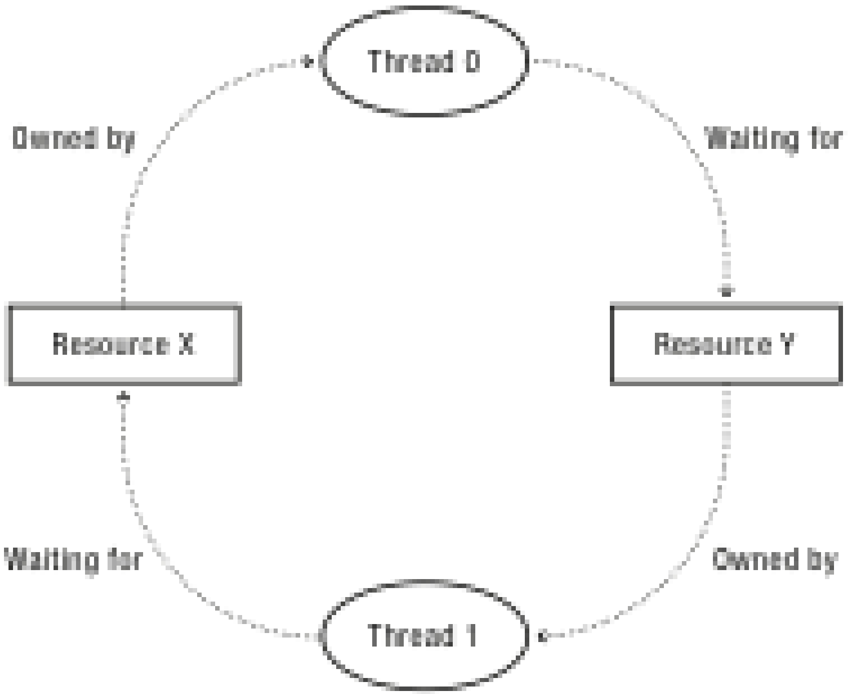
Banker's Algorithm

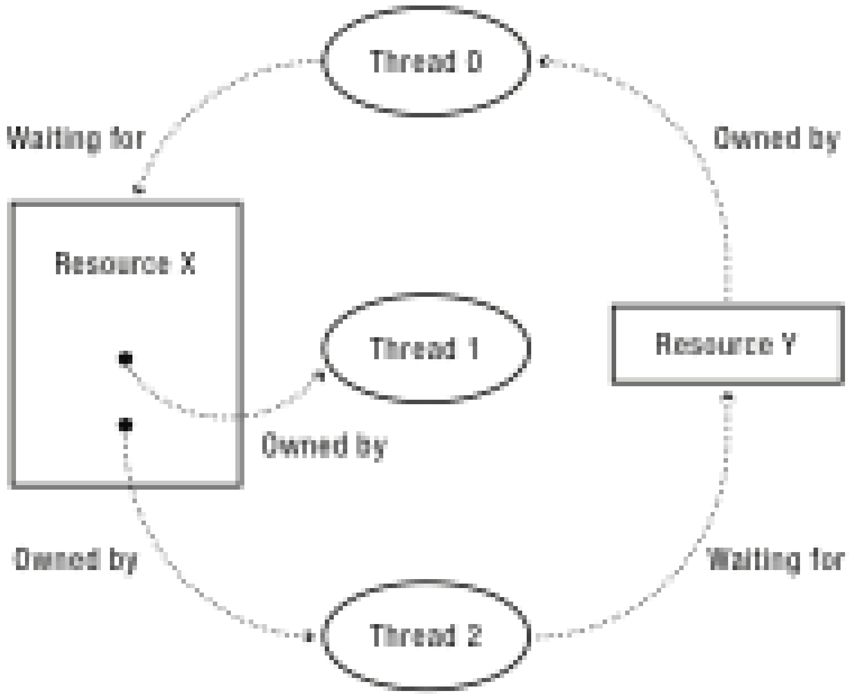
- Grant request iff result is a safe state
  - i.e., not an unsafe state
- Sum of maximum resource needs of current threads can be greater than the total resources
  - Provided there is some way for all the threads to finish without getting into deadlock even if all request their maximum
- Example: proceed if
  - total available resources - # allocated >= max remaining that might be needed by this thread in order to finish
  - Guarantees this thread can finish
    - Is this condition necessary?

Detect and Repair

- 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





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;
}
```

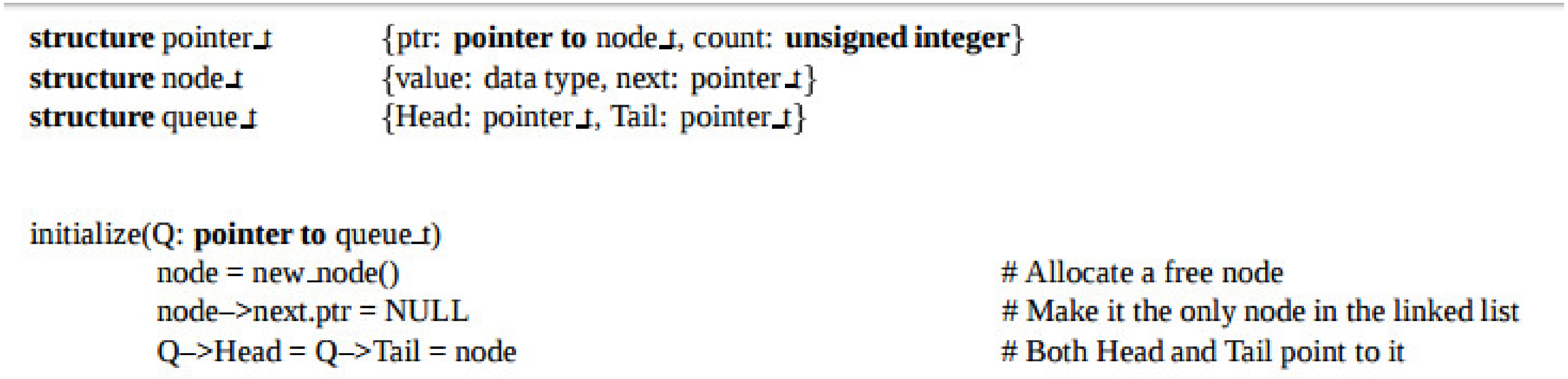
Non-blocking atomic integer

-
  ```
  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?

Why non-blocking

- What if a thread is pre-empted while holding a lock?
- If there are no locks, can there be deadlock?
- Priority inversion
  - Suppose a low priority thread holds a lock needed by a high priority thread
  - (Alternative solution: priority inheritance)

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

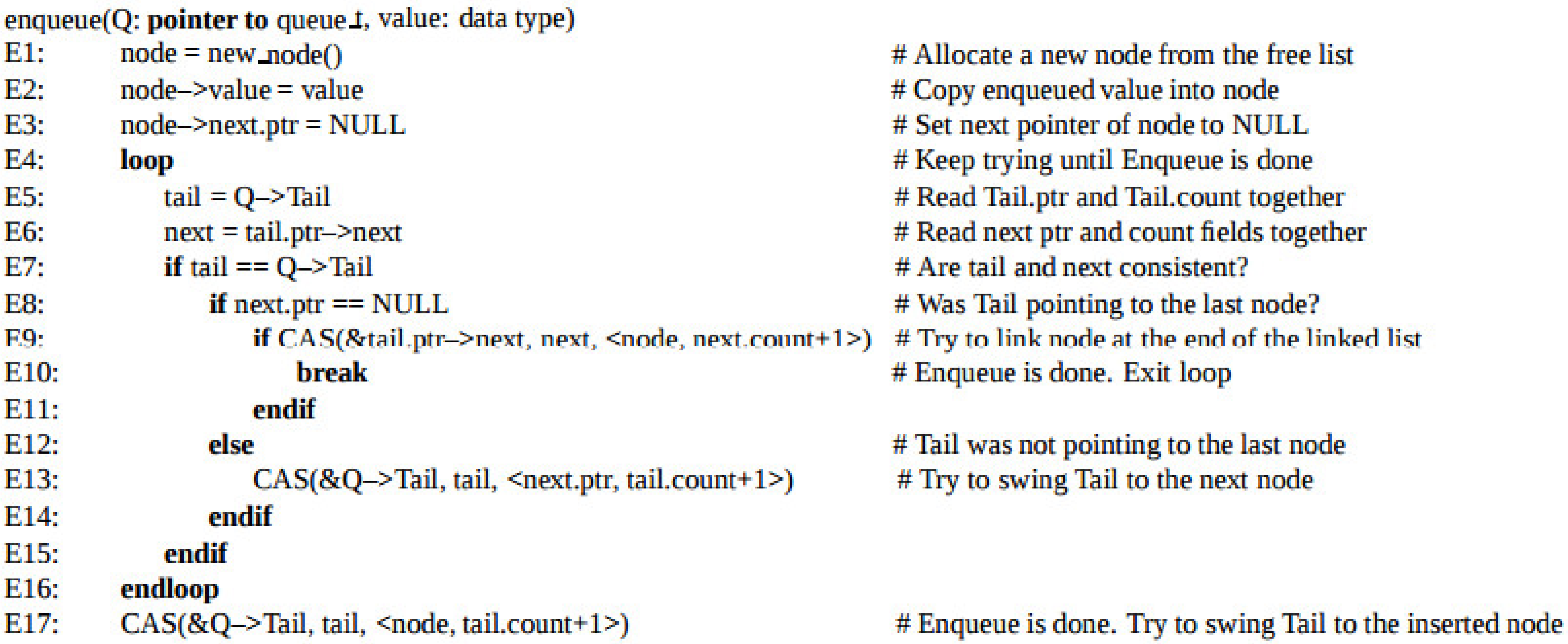


```
structure pointer_t    {ptr: pointer to node_t, count: unsigned integer}
structure node_t       {value: data type, next: pointer_t}
structure queue_t      {Head: pointer_t, Tail: pointer_t}

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)

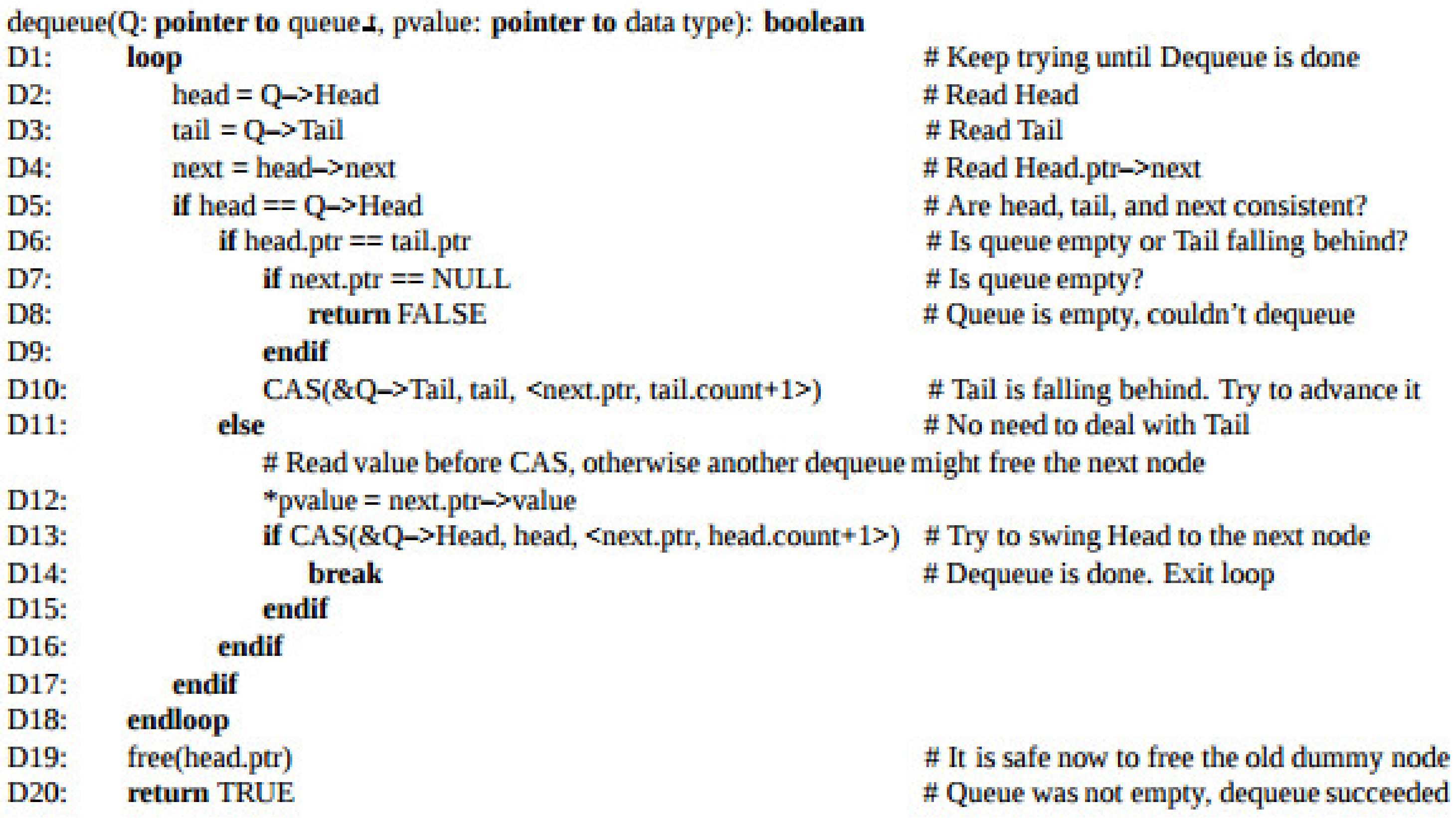
From Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
by Michael & Scott.

Non-blocking FIFO: enqueue()



```
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>)                  # 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
    loop                                                   # Keep trying until Dequeue is done
        head = Q–>Head                                     # Read Head
        tail = Q–>Tail                                     # Read Tail
        next = head–>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
```

Performance Results

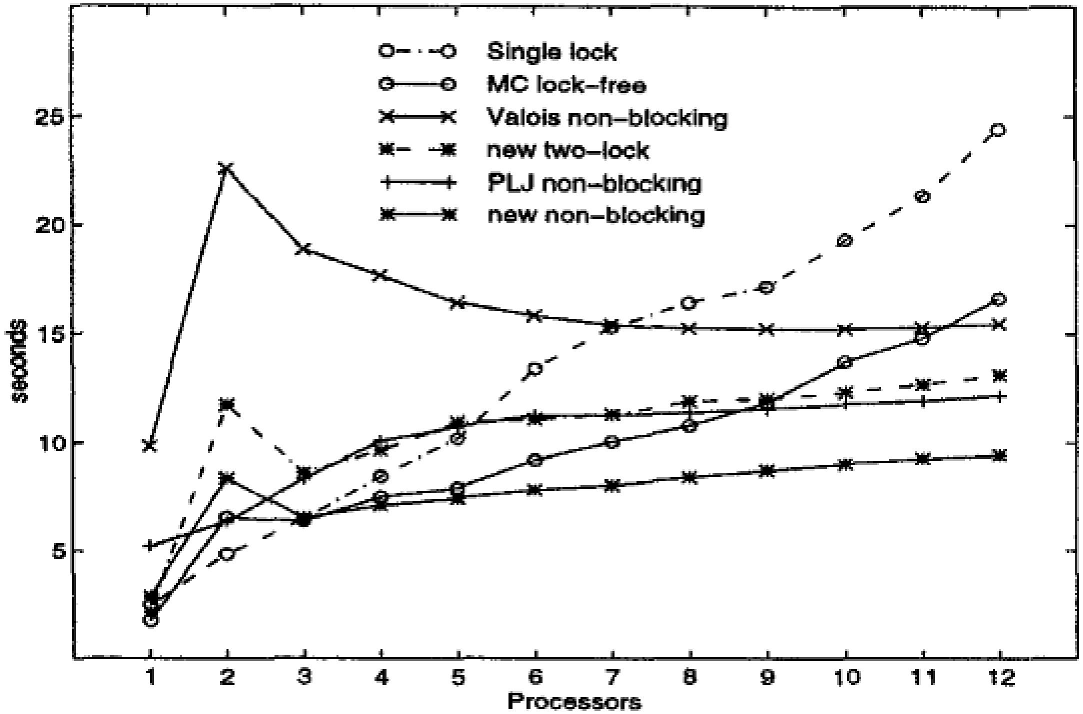


Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

12 processor Silicon Graphics Challenge