Section 3
Practice: Spin Locks and Contention

Spin Locks and Contention

- In contrast to uniprocessor programming, in multiprocessor programming, it is crucial to understand the underlying machine architecture.

- What do you do if you cannot acquire the lock (i.e., if you cannot enter the critical section)?
  - Spinning (or busy waiting)
    - Sensible when you expect the lock delay to be short
  - Blocking
    - Sensible when you expect the lock delay to be long
- Many Operating Systems (OS) apply a combination of both techniques
- Here, we turn our attention to spin locks.

Welcome to Real World: Peterson's algorithm

Experiment

- In practice, once both threads have finished the execution of the 500000 accesses to the CS, we may discover that the shared counter's final value may be slightly off from the expected value (1000000).

⇒ **Even if this is a tiny error, why is there any error at all?**

```
shared int count = 0;

Code for Process p_i

while (lcnt++ < 500000) {
      i = ThreadId.get(); // either 0 or 1
      flag[i] = true
      turn = 1-i
      while (flag[1-i] and turn == 1-i)
           noop;

      count++;          // critical section

      flag[i] = false

       remainder section;
}
```

- Compilers re-order instructions to enhance performance!
  - Program order is preserved for each individual variable but not always across multiple variables.
- Due to hardware, writes to multiprocessor memory do not necessarily take effect when they are issued!
  - Writes to shared memory are buffered in a special write (or store) buffer, to be written to memory only when needed.

Welcome to Real World

Memory Fences (or memory barriers)

- A *memory fence* forces outstanding operations to take effect!
- It is usually an expensive operation!
- Stronger primitives, like Get&Set() or Compare&Swap(), as well as reads and writes to volatile vars do not cause such errors (usually because they are implemented using memory fences).

```
ATOMIC boolean Compare&Swap( MemoryWord *pW, Value old, Value new) {
    Value tmp = *pW;
    if (*pW == old) {
        *pW = new;
        return TRUE;
    }
    return FALSE;
}
```

```
ATOMIC boolean
        Test&Set(MemoryByte *pB) {
    boolean tmp = *pB;
    *pB = 1;
    return tmp;
}

ATOMIC void Reset(MemoryByte *pB) {
    *pB = 0;
}
```

```
ATOMIC int Fetch&Inc(MemoryWord *pW) {
    int tmp = *pW;
    *pW = *pW + 1;
    return tmp;
}
```

```
ATOMIC Value
    Get&Set(MemoryWord *pW, Value nv) {
    Value tmp = *pW;
    *pW = nv;
    return tmp;
}
```

⇒ Given that memory fences cost as much as synchronization instructions, it may make sense to design ME algorithms directly from such synchronization primitives!

The TAS and TTAS Locks

THE TAS LOCK

```
void lock(MemoryByte *pB) {
      while (Test&Set(pB)) noop;
            // wait until Test&Set(pB) returns 0
}

void unlock(MemoryByte *pB) {
      reset(pB);
}
```

Theorem
The algorithm above is a correct ME algorithm.

- *Is it possible for some process to starve in this algorithm?*

The TTAS Lock

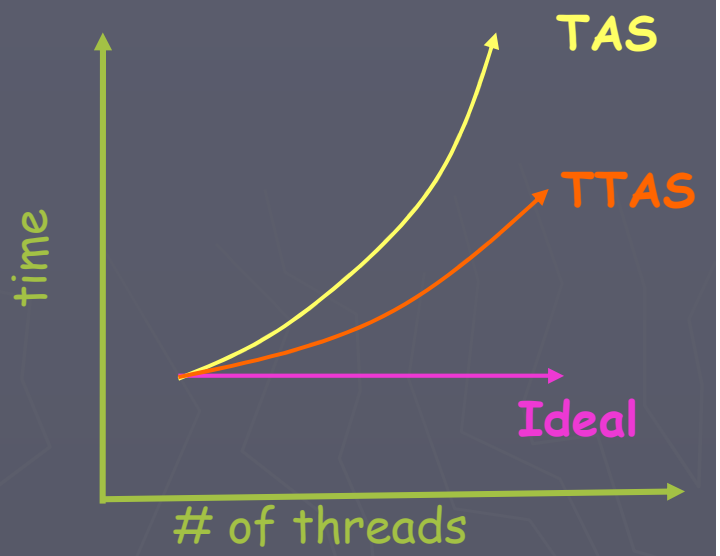
```
void lock(MemoryByte *pB) {
   while (TRUE) {
      while (*pB == TRUE) noop;
      if (Test&Set(pB) == FALSE)
         return;
   }
}

void unlock(MemoryByte *pB) {
      reset(pB);
}
```

Theorem
The algorithm above is a correct ME algorithm.

✓ The TAS and TTAS Locks are equivalent in terms of correctness!

The TAS and TTAS Locks

- But are they equivalent in terms of performance?



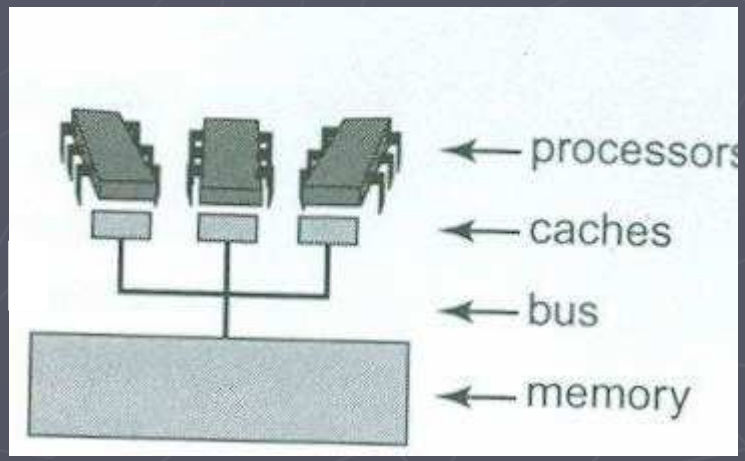
- The TAS Lock performs very poorly!
- The TTAS Lock performs substantially better but still falls far short from the ideal!

The TAS Lock – Some remarks

- Each Get&Set() call causes:
  - a broadcast on the bus, and
  - all other processors to discard their own cached copies of the lock!
- Additionally, when the thread holding the lock tries to release it, it may be delayed due to bus traffic caused by the spinners!

The TTAS Lock – Some important points

- **What happens the first time a thread B reads the lock?**
- **What happens each time B rereads the lock (finding it occupied)?**
- **Is a thread releasing the lock delayed by other threads?**
- **What happens when the lock is released by its holder thread A?**



Exponential Backoff

The idea

- If some other thread acquires the lock between the read step and the Test&Set step in the TTAS algorithm, there is probably high contention for the lock. Therefore, we'll back off for some time, giving competing threads a chance to finish.

How long should the thread backoff before it retries?

- The larger the number of unsuccessful tries, the higher the likely contention and the longer the thread should backoff.

Strategy

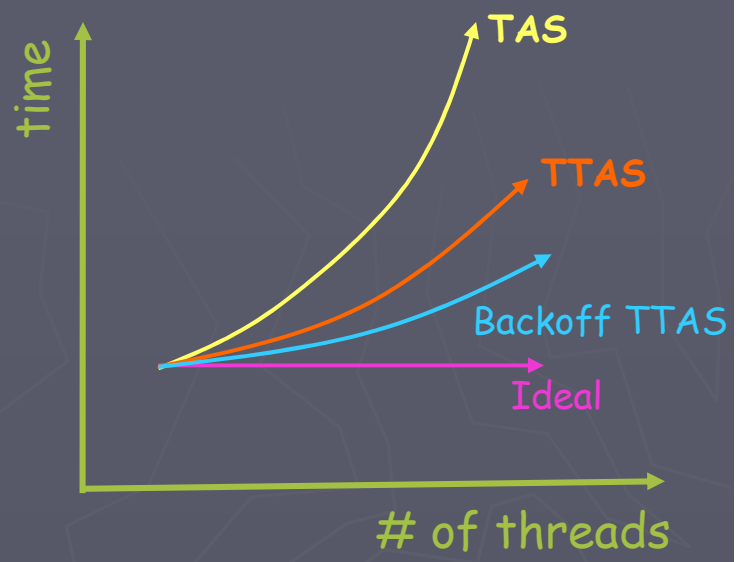
- The thread backoffs for a random duration.
- Each time the thread tries and fails to get the lock, it doubles the expected back-off time, up to a fixed maximum.

```
#define MIN_DELAY …
#define MAX_DELAY …

void BackOff(void) {
    static int limit = MIN_DELAY;
    int delay = rand_next(limit);
     // if 0 < limit < 32, that many low-order bits of the
     // returned value will independently chosen bit values

    limit = min{MAX_DELAY, 2*limit);
    usleep(delay);
}
```

The Exponential BackOff TTAS Lock

```
void lock(MemoryByte *pB) {
    while (TRUE) {
        while (*pB == TRUE) noop;
        if (Test&Set(pB) == FALSE)
            return;
        else BackOff();
    }
}
void unlock(MemoryByte *pB) {
        reset(pB);      }
```

Exponential Backoff



Advantages

- Easy to implement
- Has better performance than TTAS Lock

Drawbacks

- Must choose parameters carefully
- Not portable across different architectures

Problems

- Cache-Coherence Traffic
  - All threads spin on the same shared location causing cache-coherence traffic on every successful lock access.
- Critical Section Underutilization
  - Threads might back off for too long cause the critical section to be underutilized.

Queue Locks

Idea

- A queue is formed.
  - In a queue, every thread can learn if his turn has arrived by checking whether his predecessor has finished.
  - A queue also has better utilization of the critical section because there is no need to guess when is your turn.

Anderson's Algorithm: The Array-Based Lock

```
#define n <number-of-processes>

shared integer Tail = 0;
shared BOOLEAN flag[n] =
        {TRUE, FALSE, FALSE,... ,FALSE};

/* Code for process $p_i$ */
int slot = -1;  /* global variable for process $p_i$; */

void lock(void) {
    slot = Fetch&Inc(&Tail);
    while (!Flag[slot]) noop;
}

void unlock(void) {
    flag[slot] = FALSE;
    flag[(slot + 1) % n] = TRUE;
}
```

Lemma

The following invariants are maintained concerning the array Flags:

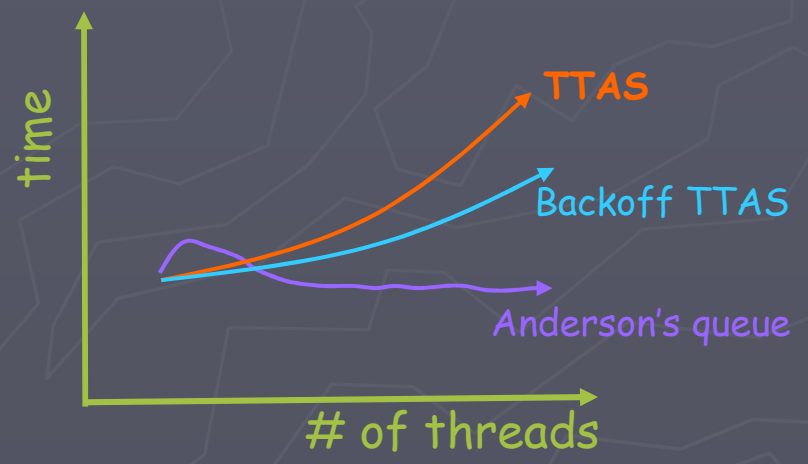
- at most one element is set to TRUE;
- if Flags[k] is set to TRUE then exactly [(k-Tail-1) mod n] processes are in the entry section, each of them spinning on a different entry of array Flags.

The Flag Array

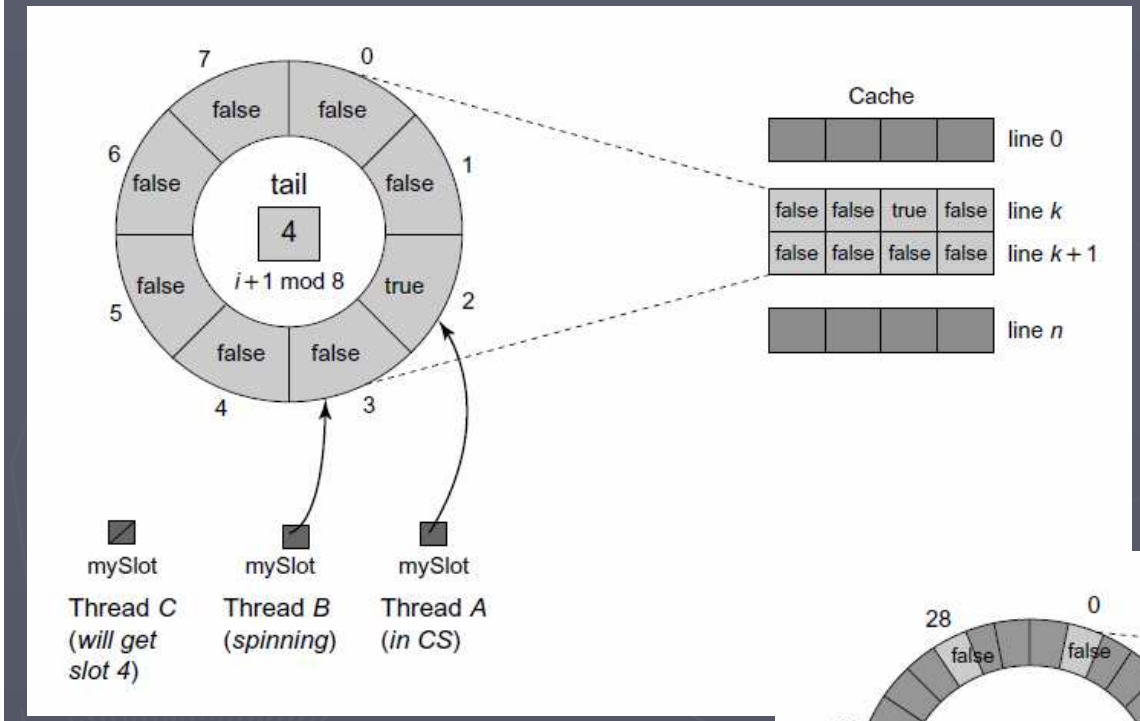
Anderson's Algorithm: The Array-Based Lock

Advantages

- At any given time, each thread spins on its locally cached copy of a single array location
  - Shorter handover than backoff
  - Curve is practically flat
  - Better Scalability
- FIFO Fairness

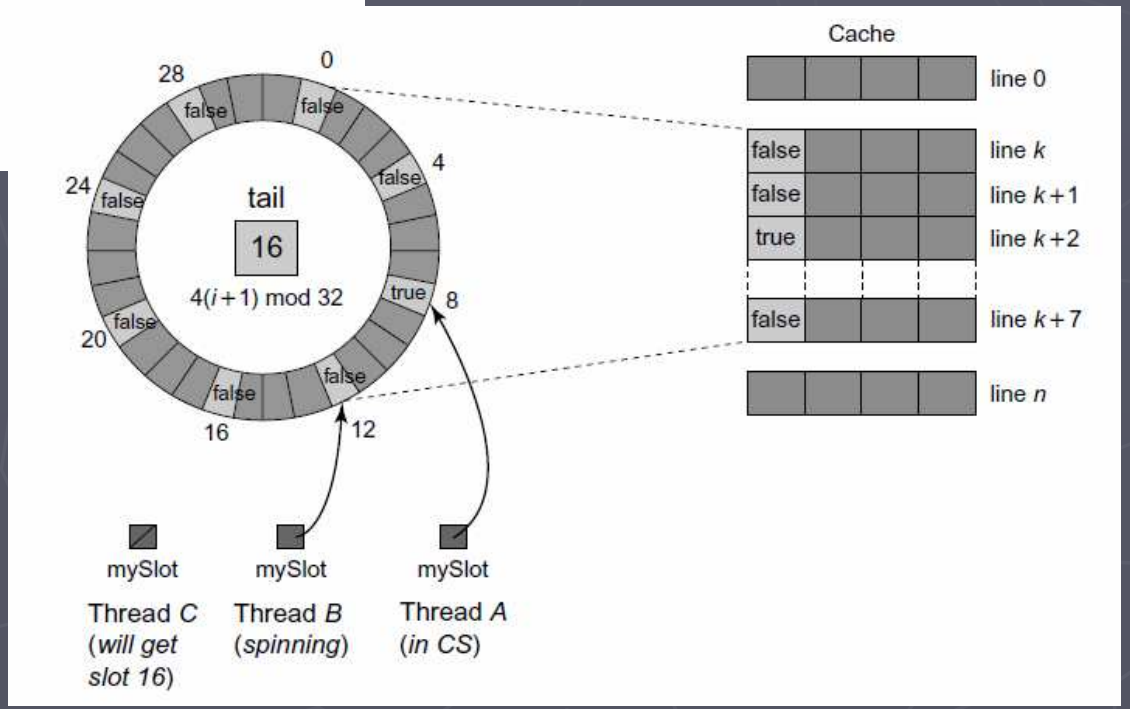


Anderson's Algorithm: The Array-Based Lock



Drawbacks

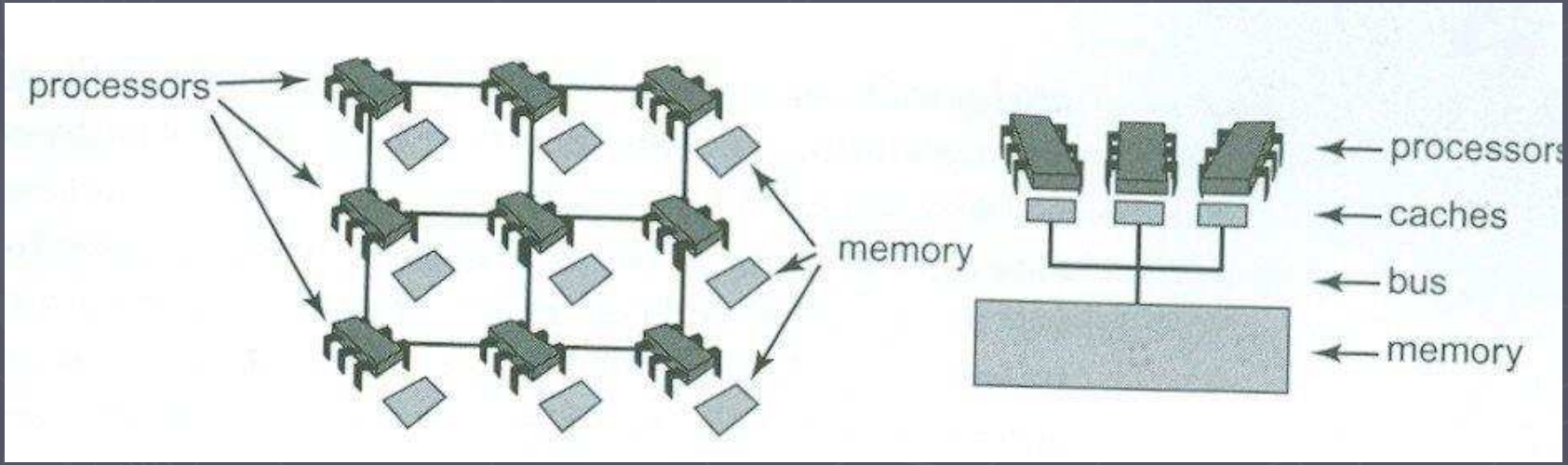
- Padding is required, so that distinct elements are mapped to distinct cache lines in order for false sharing to be avoided. ☹



Anderson's Algorithm: The Array-Based Lock

Drawbacks

- Not very space-efficient. ☹
  - It allocates an array of size $O(n)$ per lock. Synchronizing L distinct objects requires $O(Ln)$ space, even if a thread accesses only one lock at a time ☹
- On cache-less NUMA architectures, the algorithm does not have a good performance since spinning during lock() might be performed on a remote variable ☹



Cache-less NUMA architecture SMP architecture with caches

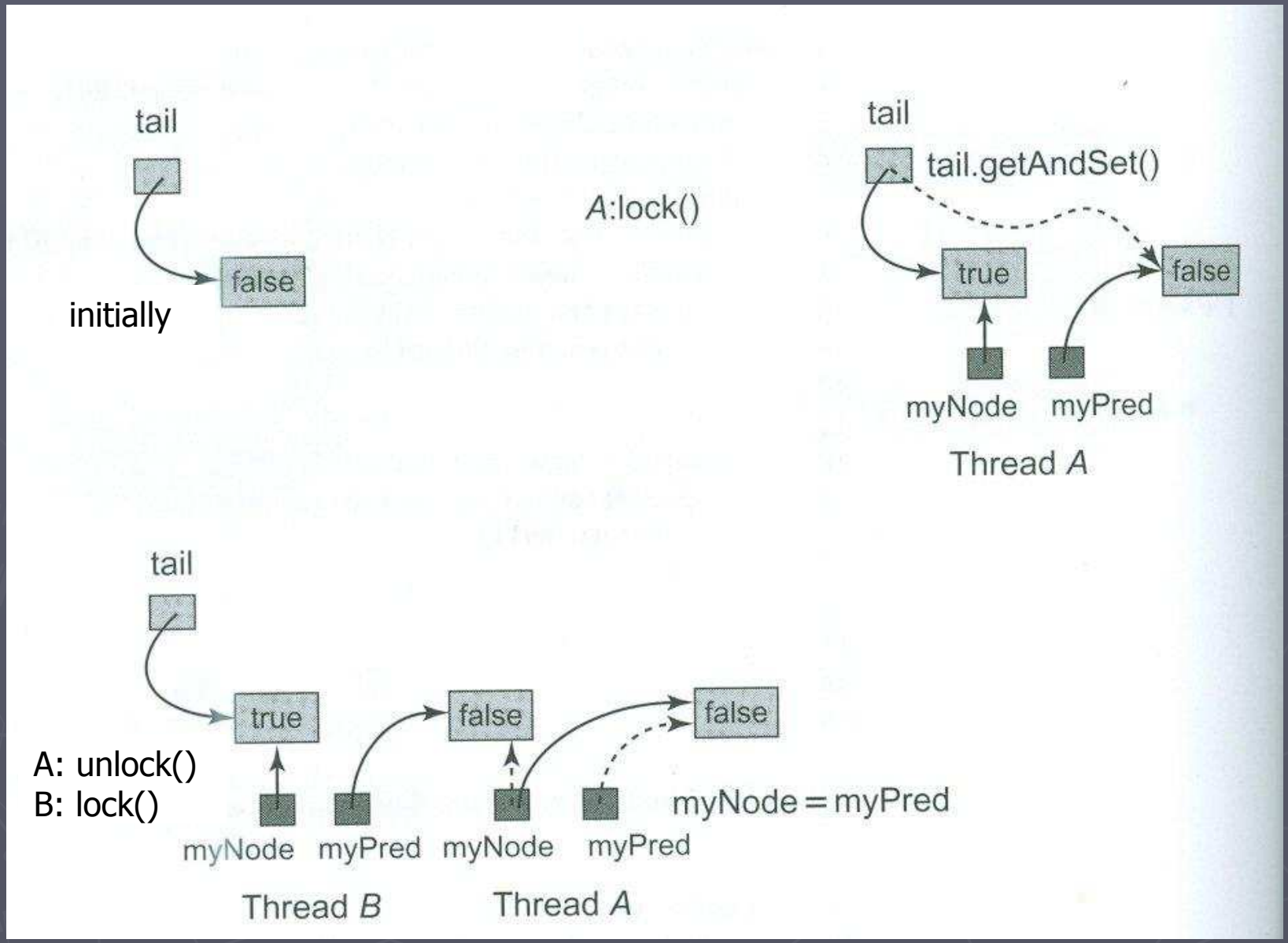
The CLH Lock
A more space-efficient Algorithm

```
typedef struct node {
        BOOLEAN locked;
} NODE;

shared NODE *Tail;
/* initially, points to a NODE n with n.locked == FALSE */

/* Section of global variables for process $p_i$ */
NODE *MyNode, *MyPred = NULL;
        /* Variable MyNode initially points to a struct NODE

void lock(void) {
        MyNode.locked = TRUE;
        MyPred = Get&Set(&Tail, MyNode);
        while (MyPred->locked == TRUE) noop;
}

void unlock(void) {
        MyNode->locked = FALSE;
        MyNode = MyPred;
                /* recycling of nodes */
}
```

Brief Description

- An implicit list of NODES is created. Each thread creates just one NODE.
- NODEs are not connected to each other. Rather, each thread refers to its predecessor thread through a (non-shared) global variable, called MyPred.
- NODEs are re-cycled by having each thread using the NODE pointed to by its MyPred Variable as its current NODE the next time it requires the lock.
- An initial NODE is placed in the queue implementing each lock.

The CLH Lock


tail
false
initially
A:lock()
tail
tail.getAndSet()
true
false
myNode
myPred
Thread A
tail
A: unlock()
B: lock()
true
false
false
myNode
myPred
myNode
myPred
myNode = myPred
Thread B
Thread A

The CLH Lock

Advantages

- Requires much less space than the Anderson's algorithm
  - Synchronizing L distinct objects requires $O(L+n)$ space (much better than the Array-based Lock).
- Does not require knowledge of the number of processes that might access the lock.
- Has all the performance advantages of the Anderson's algorithm.

Drawbacks

- Like Anderson's algorithm, on cache-less NUMA architectures, the CLH algorithm does not have a good performance since it causes a lot of Remote Memory References (RMRs); i.e., spinning is performed on a remote variable.

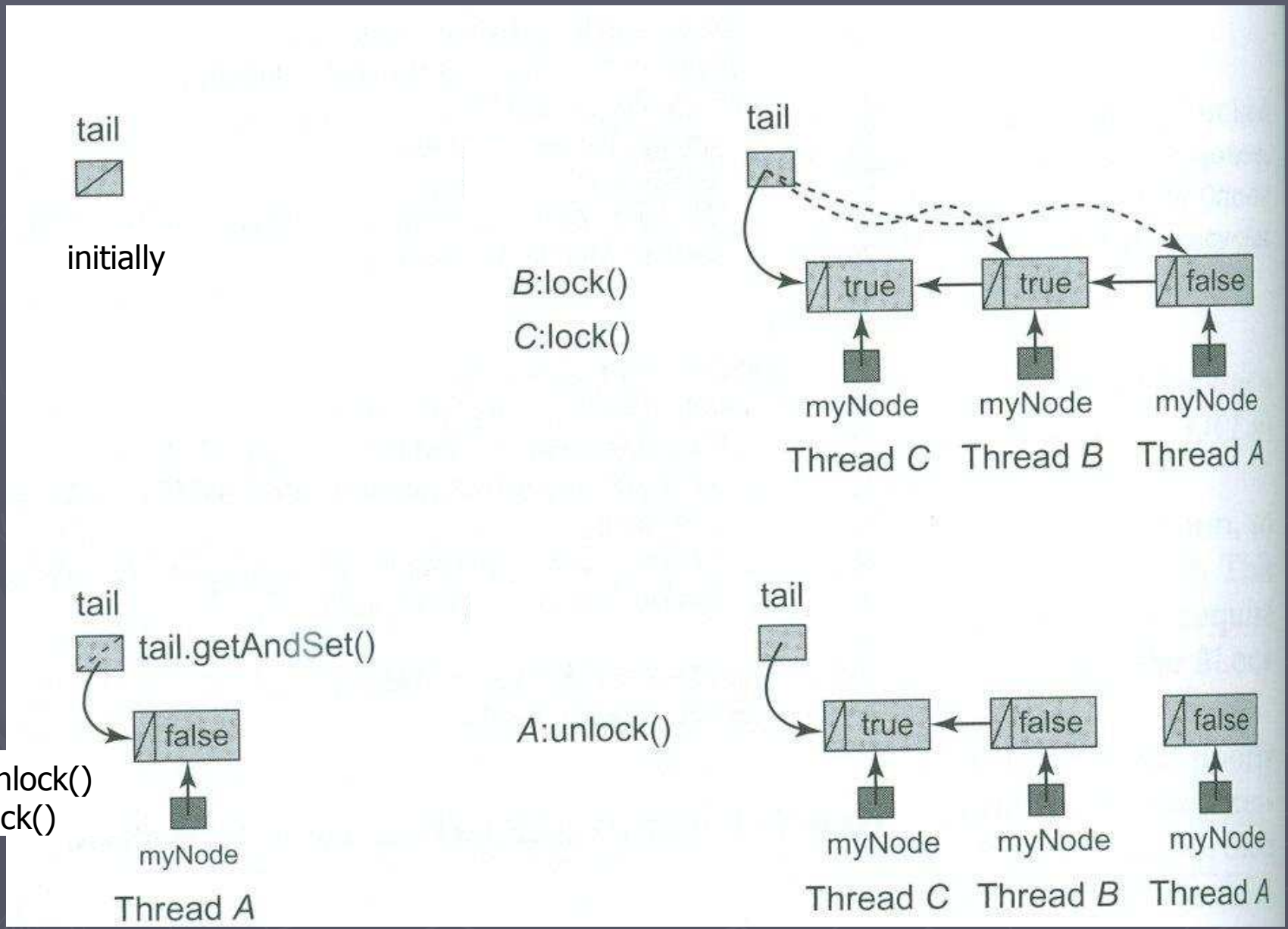
The MCS Lock

```
typedef struct node {
        BOOLEAN locked;
        struct node *next;
} NODE;

/* shared variables section */
shared NODE *Tail = NULL;

/* Section of global variables for process $p_i$ */
NODE *MyNode, *MyPred = NULL;
/* Variable MyNode initially points to a NODE */

void lock {
        MyPred = Get&Set(&Tail, MyNode);
        if (MyPred != NULL) {
                MyNode->locked = TRUE;
                MyPred->next = MyNode;
                while (MyNode->locked) noop;
        }
}

void unlock {
        if (MyNode->next == NULL) {
                if (Compare&Swap(&Tail, MyNode, NULL) == TRUE) return;
                 while (MyNode->next == NULL) noop;
        }
        MyNode->next->locked = FALSE;
        MyNode->next = NULL;
}
```

Major Ideas

- The created list of NODEs is now explicit.
- Each thread does not spin on the NODE pointed to by MyPred but on that pointed to by MyNode (which is a variable in the thread's local memory)

The MCS Lock



A: unlock()
B: lock()

The MCS Lock

Advantages

- Each unlock causes just one invalidation (as with CLH)
- Lock/Unlock causes just O(1) RMRs on a NUMA architecture
- Recycling can be applied (as in CLH) to obtain space overhead O(n)

Drawbacks

- Releasing a lock requires spinning.
- The algorithm executes more reads and writes to execute lock/unlock, and it requires Compare&Swap() for unlock.

J. Mellor-Crumney and M. Scott, “Algorithms for Scalable Synchronization on shared-memory multiprocessors”, ACM Transactions on Computer Systems, 9(1):21-65, 1991.

Lower Bounds

Fetch&Φ Primitives

```
atomic Value fetch&φ(MemoryWord *pW, InputStruct input) {
    old = *pW;
    *pW = φ(old, input);
    return(old);
}
```

- A comparison primitive conditionally updates a shared variable after first testing that its value meets some condition.
  - Compare&Swap()
  - Test&Set()
- Non-comparison primitives update variables unconditionally
  - Fetch&Increment, Fetch&Add
  - Fetch&Store
- **Lower Bound [Anderson & Kim, J. of Parallel and Distributed Computing]**
  Any n-process mutual exclusion algorithm based on reads, writes and comparison primitives causes Ω(logn / log log n) remote memory references.
- Several algorithms with constant RMR complexity exist when non-comparison primitives are used. A generic algorithm using (any non-comparison) fetch&φ primitive is presented by Anderson and Kim.

One Lock To Rule Them All?

- TTAS+Backoff, CLH, MCS, TOLock.
- Each better than others in some way
- There is no one solution
- Lock we pick really depends on:
  - the application
  - the hardware
  - which properties are important