

Lecture 16:

**Implementing Locks,
Fine-Grained Synchronization,
and (a short intro to) Lock-Free Programming**

**Parallel Computing
Stanford CS149, Fall 2025**

Today

- Lock implementations
- Using locks
 - Fine-grained locking examples
 - Lock-free data structure designs



Preliminaries: some terminology

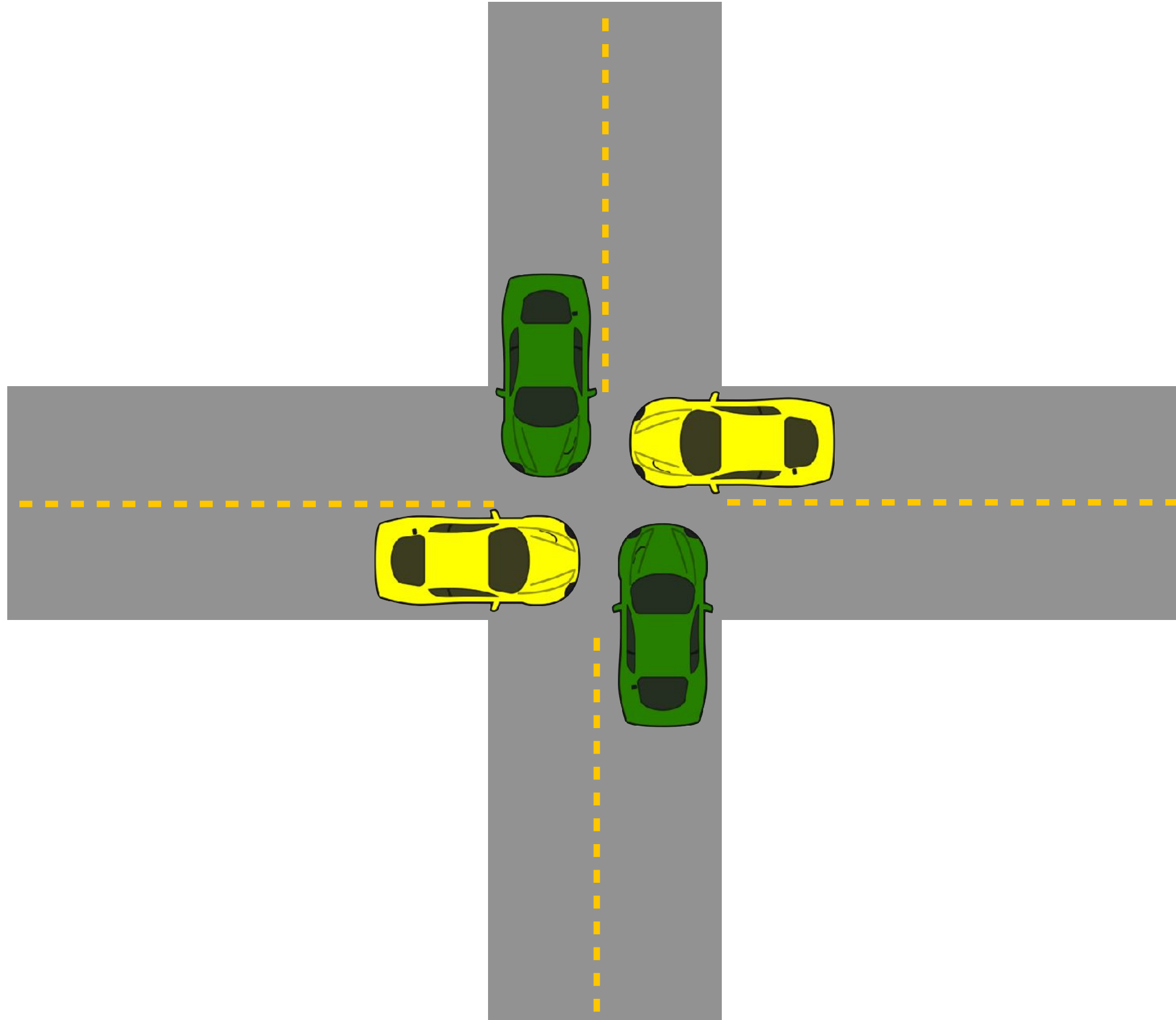
Deadlock

Livelock

Starvation

(Deadlock and livelock concern program correctness. Starvation is really an issue of fairness.)

Deadlock



Deadlock is a state where a system has outstanding operations to complete, but no operation can make progress.

Deadlock can arise when each operation has acquired a shared resource that another operation needs.

In a deadlock situations, there is no way for any thread (or, in this illustration, a car) to make progress unless some thread relinquishes a resource (“backs up”)

Traffic deadlock

**Non-technical side note for car-owning students:
Deadlock happens all the %\$*** time in SF.**

(However, deadlock can be amusing when a bus driver decides to let another driver know they have caused deadlock... “go take cs149 you fool!”)



More illustrations of deadlock

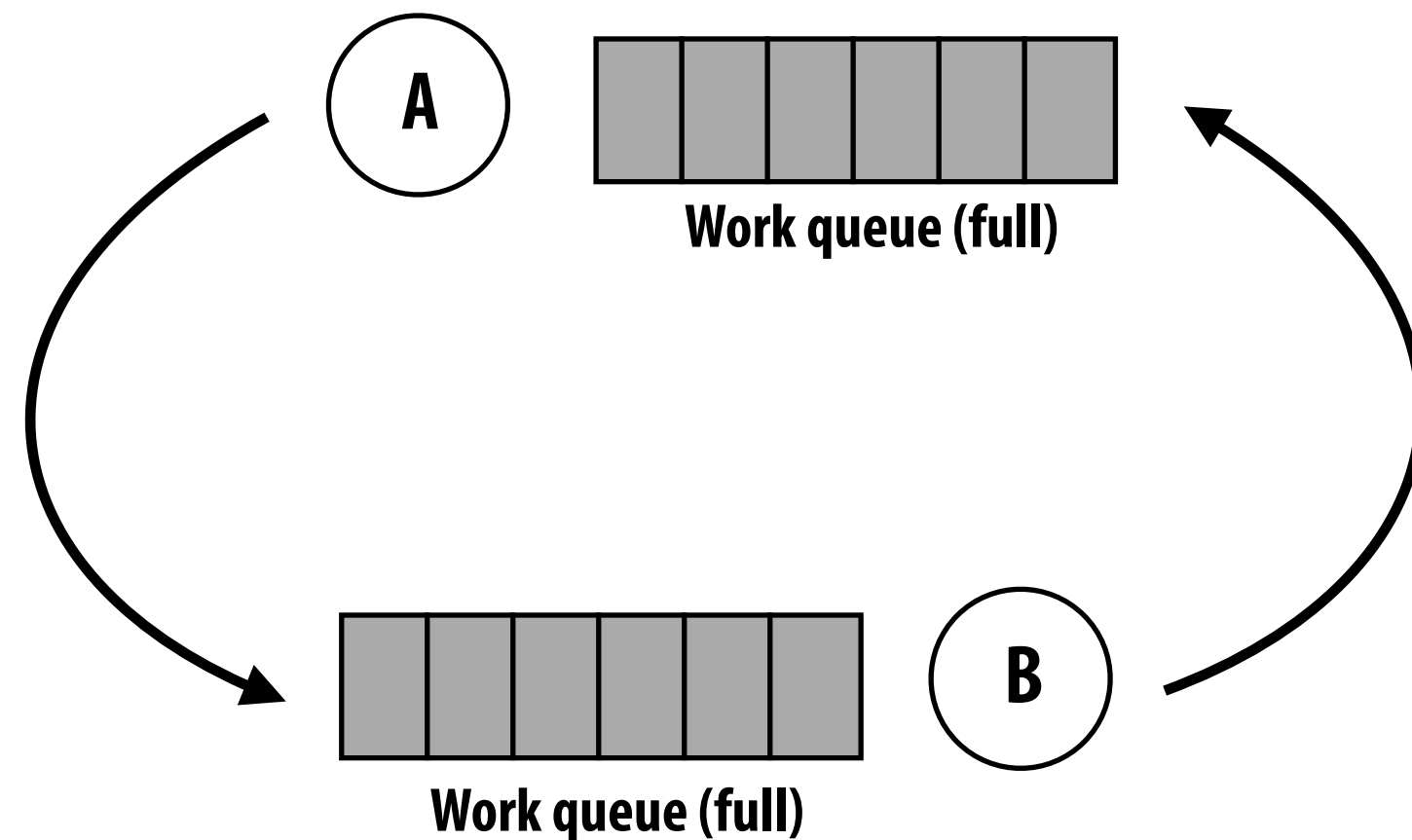


Credit: David Maitland, National Geographic

Why are these examples of deadlock?

Deadlock in computer systems

Example 1:



Thread A produces work for B's work queue

Thread B produces work for A's work queue

**Queues are finite and workers wait if
no output space is available**

Example 2:

```
const int numEl = 1024;
float msgBuf1[numEl];
float msgBuf2[numEl];

int threadId getThreadId();

... do work ...

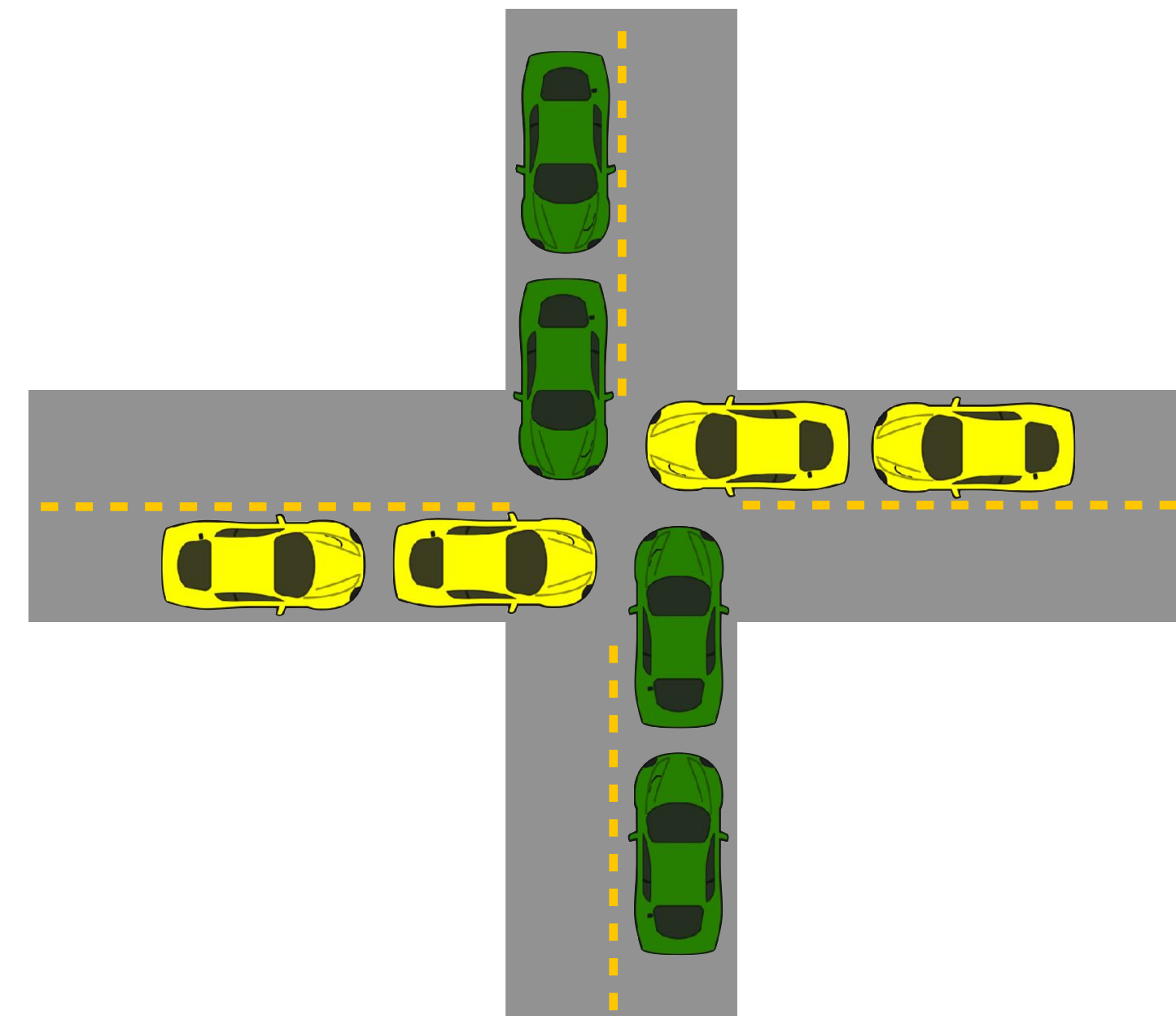
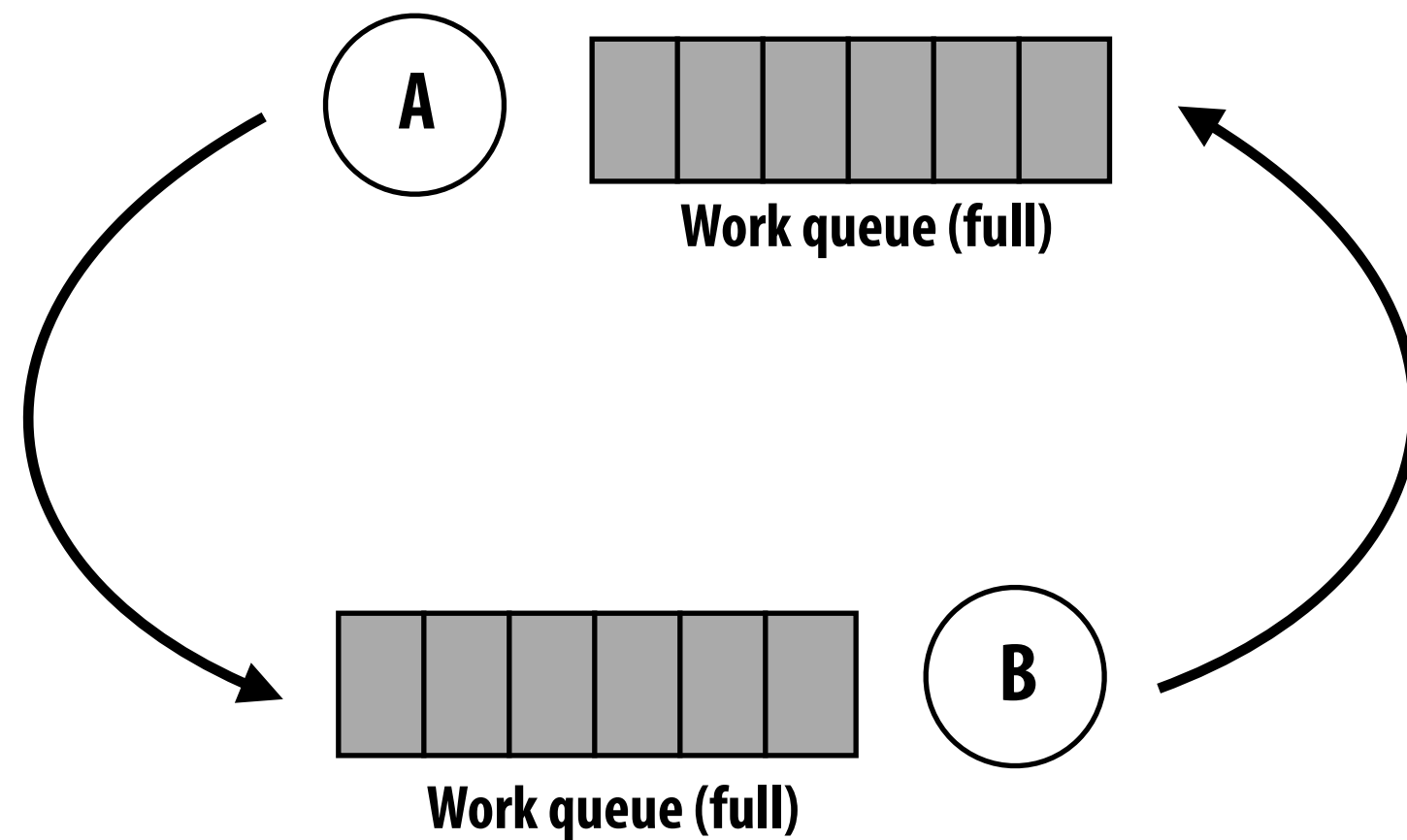
MsgSend(msgBuf1, numEl * sizeof(int), threadId+1, ...
MsgRecv(msgBuf2, numEl * sizeof(int), threadId-1, ...
```

**Every thread sends a message (blocking send)
to the thread with the next higher id**

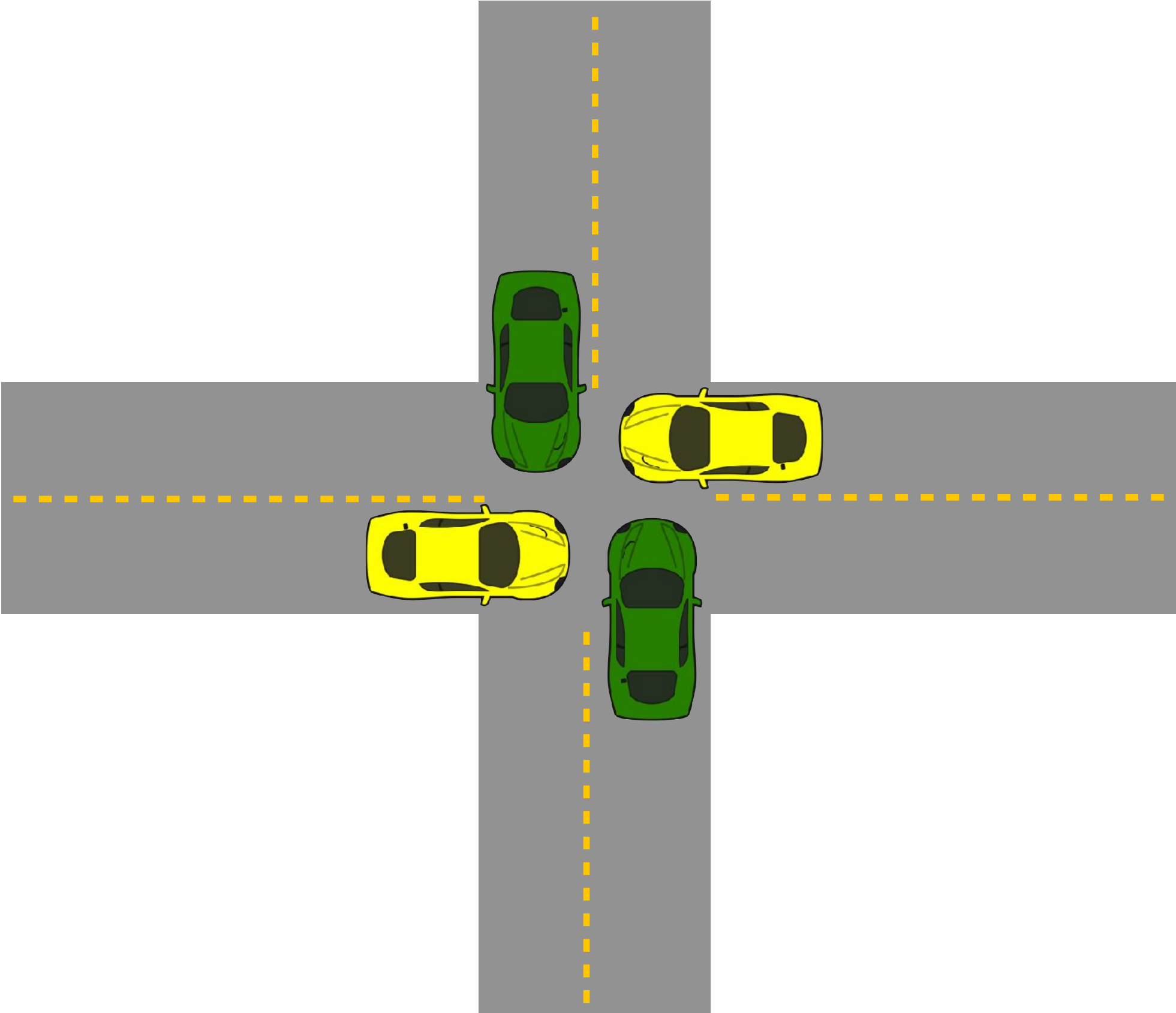
**Then thread receives message from thread with
next lower id.**

Required conditions for deadlock

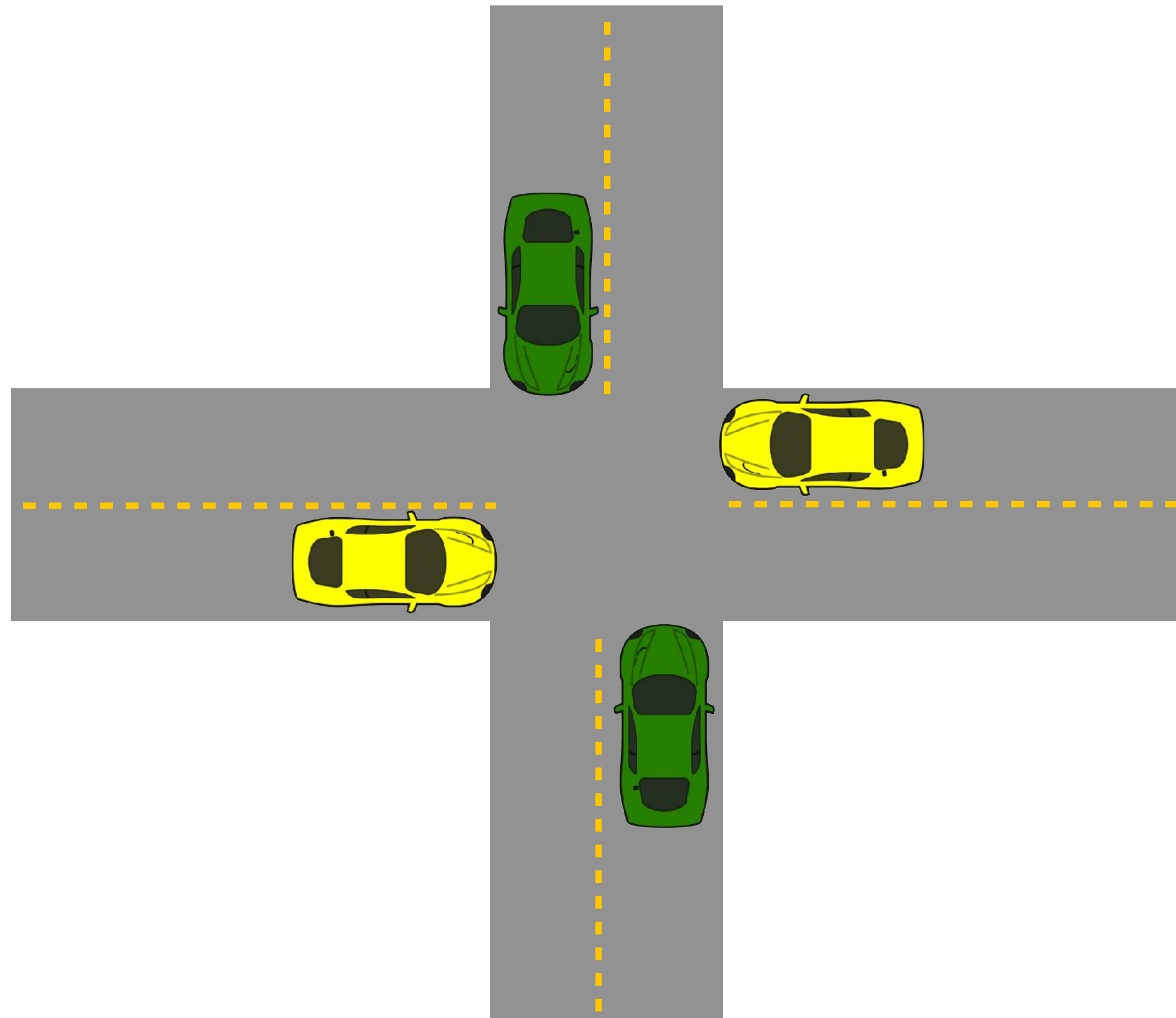
1. **Mutual exclusion:** only one processor can hold a given resource at once
2. **Hold and wait:** processor must hold the resource while waiting for other resources it needs to complete an operation
3. **No preemption:** processors don't give up resources until operation they wish to perform is complete
4. **Circular wait:** waiting processors have mutual dependencies (a cycle exists in the resource dependency graph)



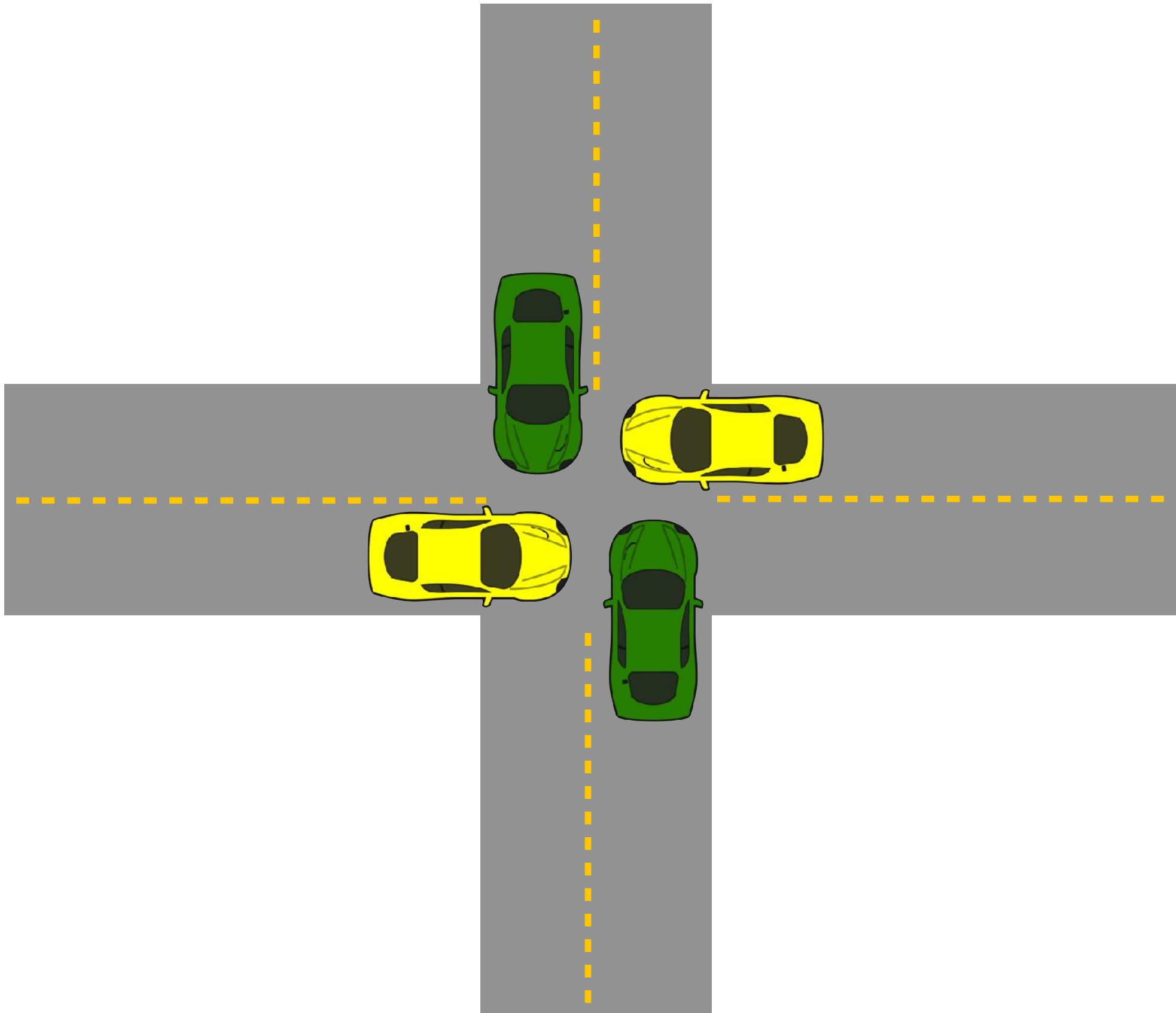
Livelock



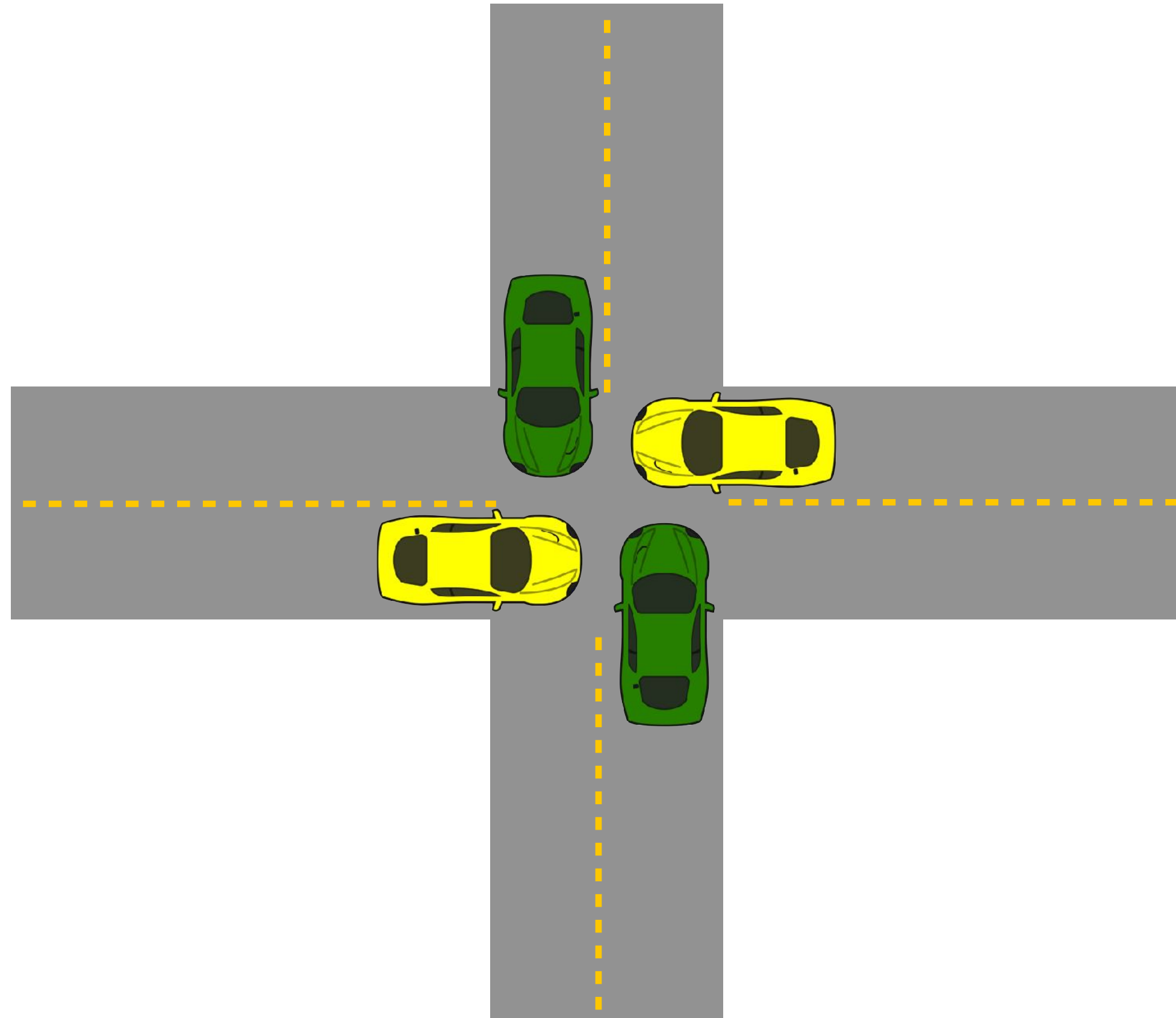
Livelock



Livelock



Livelock



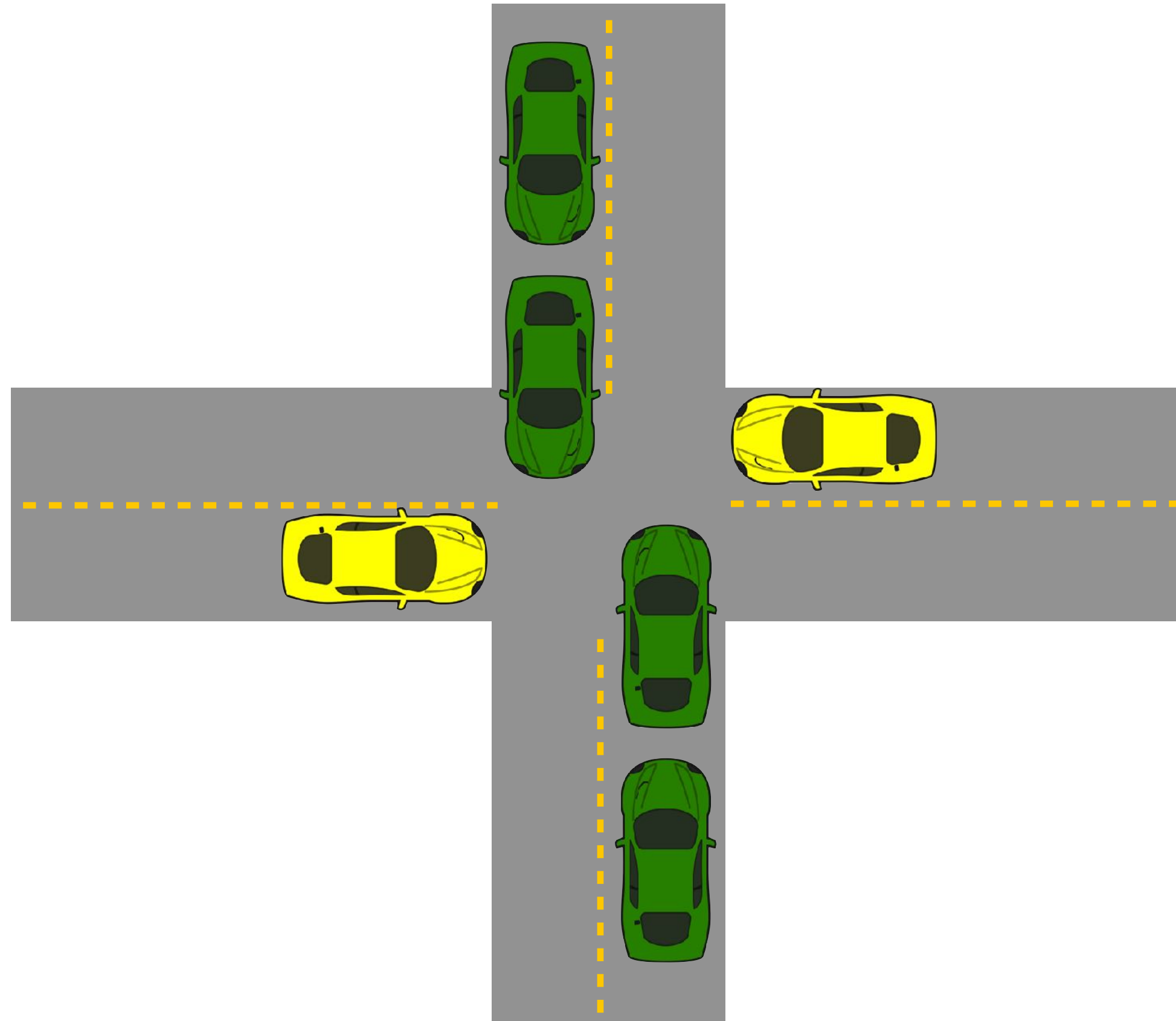
Livelock is a state where a system is executing many operations, but no thread is making meaningful progress.

Can you think of a good daily life example of livelock?

Computer system examples:

Operations continually abort and retry

Starvation



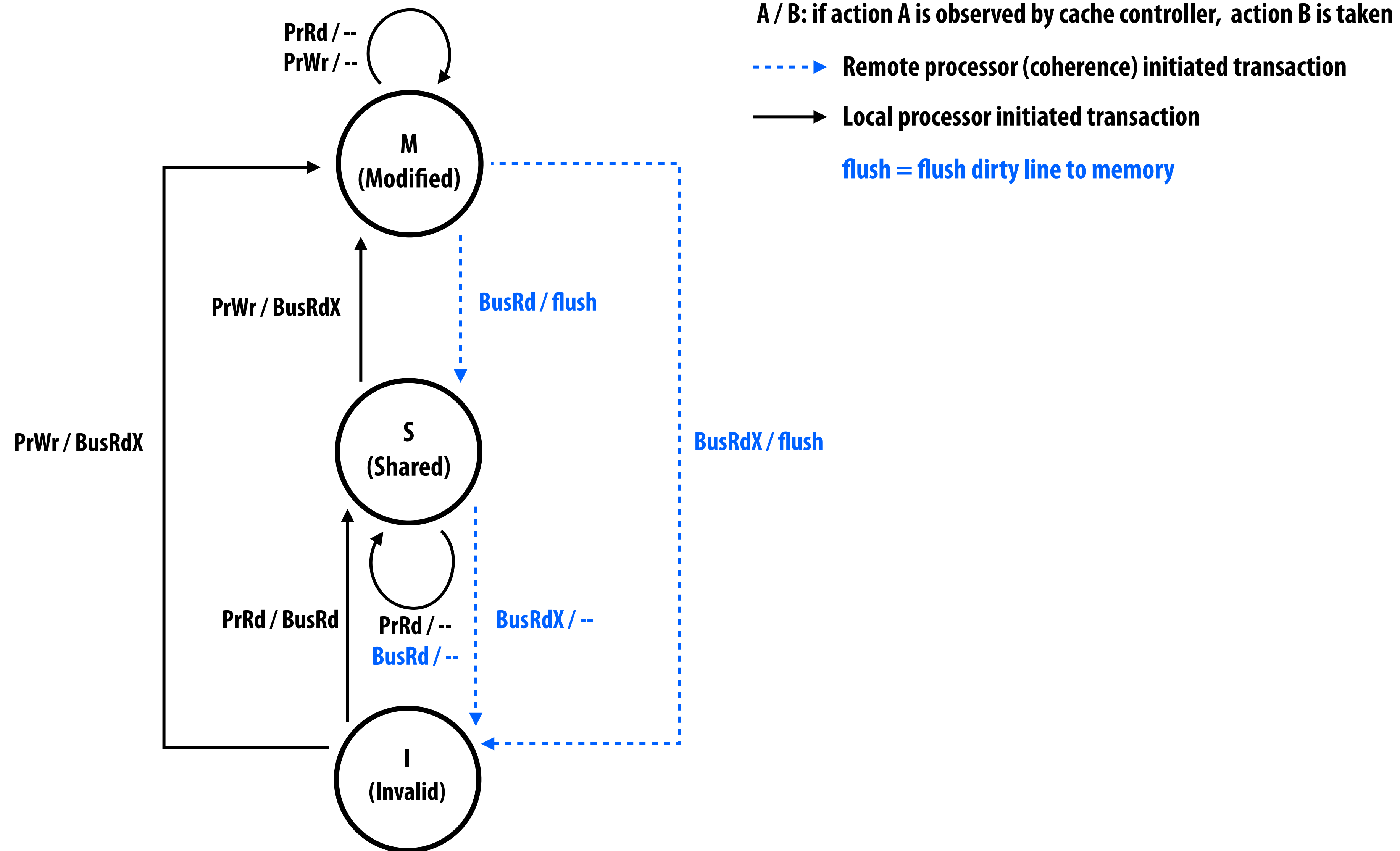
**State where a system is making overall progress,
but some processes make no progress.**
(green cars make progress, but yellow cars are stopped)

Starvation is usually not a permanent state
(as soon as green cars pass, yellow cars can go)

**In this example: assume traffic moving left/right (yellow cars)
must yield to traffic moving up/down (green cars)**

Ok, let's get started...

Review: MSI state transition diagram *



* Remember, all caches are carrying out this logic independently to maintain coherence

Example: testing your understanding

Consider this sequence of loads and stores to addresses X and Y by processors P0 and P1

Assume that X and Y reside on different cache lines, and contain the value 0 at the start of execution.

	What cache 0 does:	What cache 1 does:
P0: LD X	issue BusRd, load line X in S state	observe BusRd, do nothing (line is in I state)
P0: LD X	cache hit	do nothing
P0: ST X ← 1	issue BusRdX, load line X in M state	observe BusRdX, do nothing (line is in I state)
P0: ST X ← 2	cache hit	do nothing
P1: ST X ← 3	observe BusRdX, flush line X, move line to I state	issue BusRdX, load line X in M state
P1: LD X	Do nothing	cache hit
P0: LD X	issue BusRd, load line X in S state	observe BusRd, flush line X, move to S state
P0: ST X ← 4	issue BusRdX, load line X in M state	observe BusRdX, move to I state
P1: LD X	observe BusRd, flush line X, move to S state	issue BusRd, load line X in S state
P0: LD Y	issue BusRd, load line Y in S state	observe BusRd, do nothing (line Y is in I state)
P0: ST Y ← 1	issue BusRdX, load line Y in M state	observe BusRdX, do nothing (line Y is in I state)
P1: ST Y ← 2	observe BusRdX, flush line Y, move to I state	issue BusRdX, load line Y in M state

Test-and-set based lock

Atomic test-and-set instruction:

```
ts R0, mem[addr]      // load mem[addr] into R0
                       // if mem[addr] is 0, set mem[addr] to 1
```

```
lock:      ts    R0, mem[addr]      // load word into R0
           bnz   R0, lock           // if 0, lock obtained
```

```
unlock:    st    mem[addr], #0      // store 0 to address
```


x86 cmpxchg

■ Compare and exchange (atomic when used with lock prefix)

`lock cmpxchg dst, src`



lock prefix (designates operation is atomic)



often a memory address



x86 register

`if (dst == EAX)`

`ZF = 1`

`dst = src`

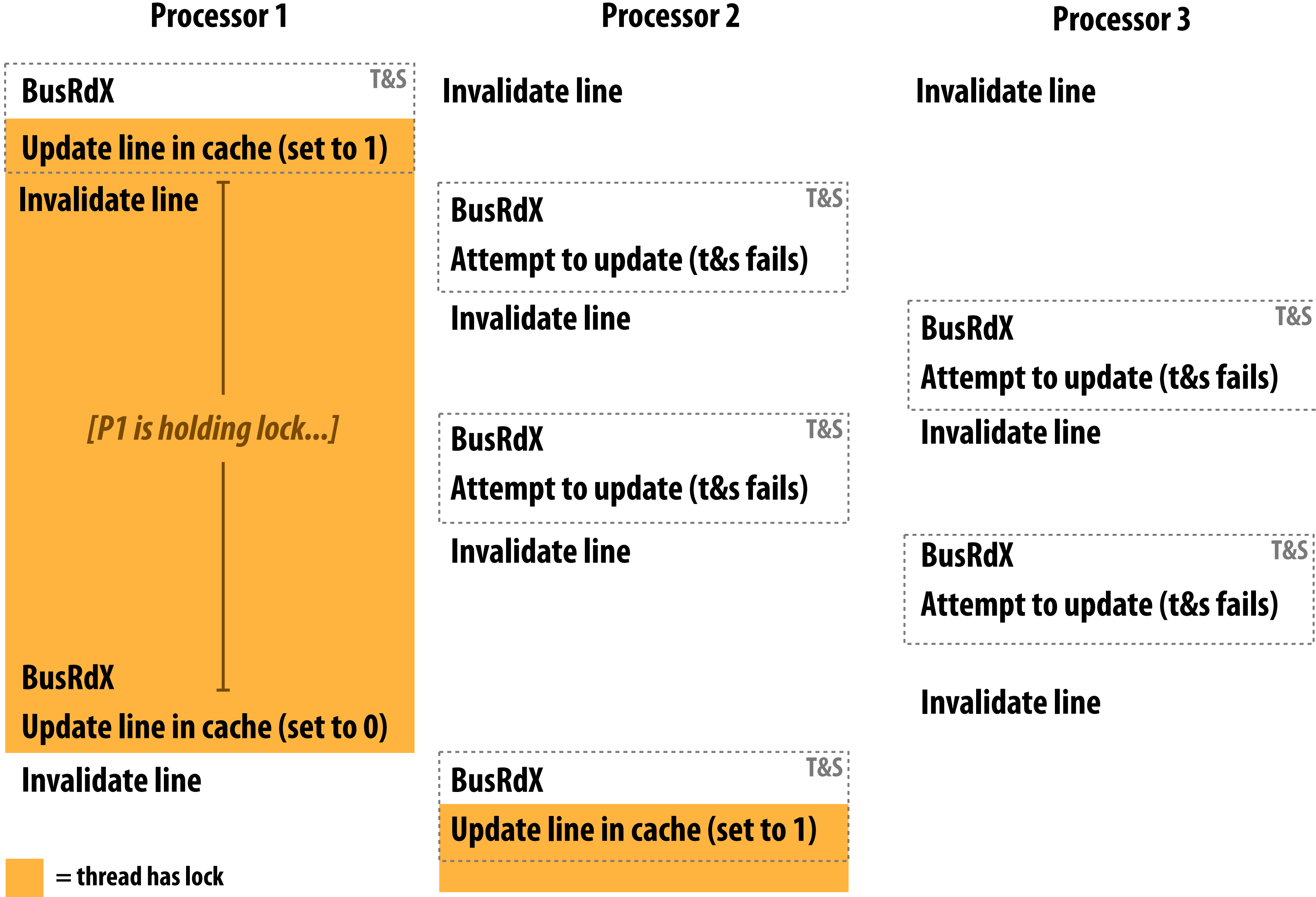
`else`

`ZF = 0`

`EAX = dst`

flag register holds result of check

Test-and-set lock: consider coherence traffic



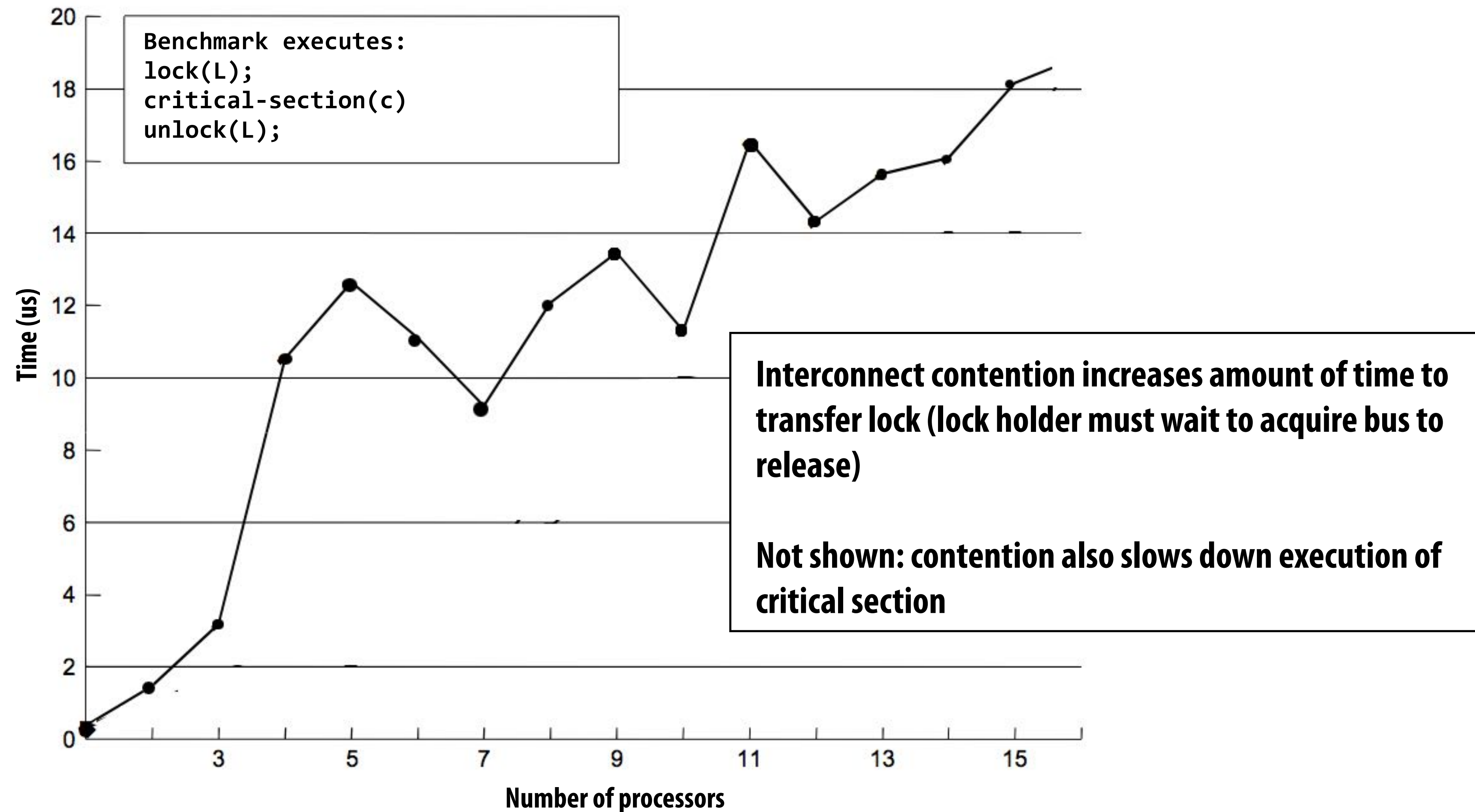
Check your understanding

- **On the previous slide, what is the duration of time the thread running on P1 holds the lock?**
- **At what points in time does P1's cache contain a valid copy of the cache line containing the lock variable?**

Test-and-set lock performance

Benchmark: execute a total of N lock/unlock sequences (in aggregate) by P processors

Critical section time removed so graph plots only time acquiring/releasing the lock



Desirable lock performance characteristics

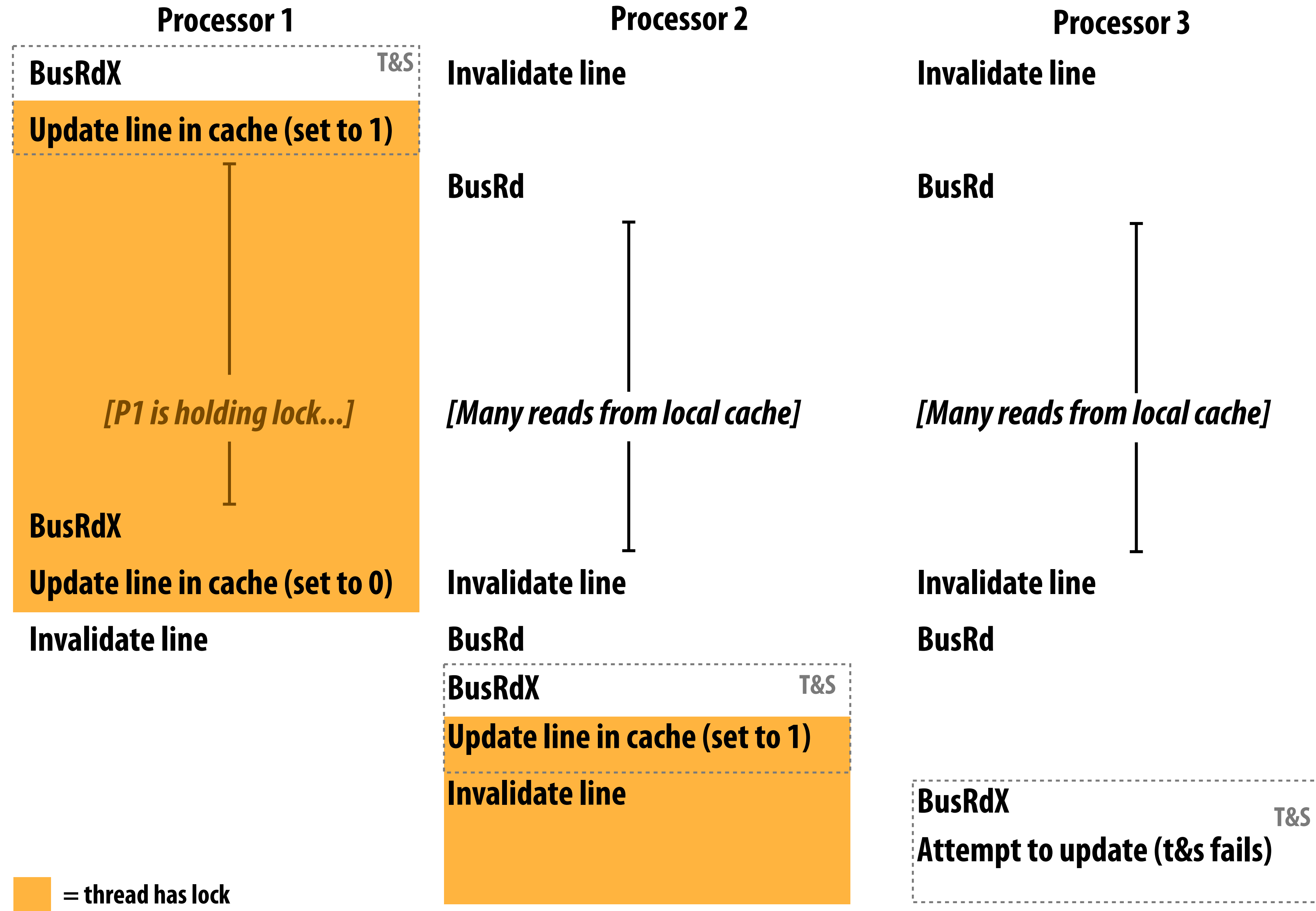
- **Low latency**
 - If lock is free and no other processors are trying to acquire it, a processor should be able to acquire the lock quickly
- **Low interconnect traffic**
 - If all processors are trying to acquire lock at once, they should acquire the lock in succession with as little traffic as possible
- **Scalability**
 - Latency / traffic should scale reasonably with number of processors
- **Low storage cost**
- **Fairness**
 - Avoid starvation or substantial unfairness
 - One ideal: processors should acquire lock in the order they request access to it

Simple test-and-set lock: low latency (under low contention), high traffic, poor scaling, low storage cost (one int), no provisions for fairness

Test-and-test-and-set lock

```
void Lock(int* lock) {  
    while (1) {  
  
        while (*lock != 0);           // while another processor has the lock...  
                                     // (assume *lock is NOT register allocated)  
  
        if (test_and_set(*lock) == 0) // when lock is released, try to acquire it  
            return;  
    }  
}  
  
void Unlock(int* lock) {  
    *lock = 0;  
}
```

Test-and-test-and-set lock: coherence traffic



Test-and-test-and-set characteristics

- **Slightly higher latency than test-and-set in no contention case**
 - Must test... then test-and-set
- **Generates much less interconnect traffic**
 - One invalidation, per waiting processor, per lock release ($O(P)$ invalidations)
 - This is $O(P^2)$ interconnect traffic if all processors have the lock cached
 - Recall: test-and-set lock generated one invalidation per waiting processor per test
- **More scalable (due to less traffic)**
- **Storage cost unchanged (one int)**
- **Still no provisions for fairness**

Another impl: ticket lock

Main problem with test-and-set style locks: upon release, all waiting processors attempt to acquire lock using test-and-set



```
struct lock {  
    int next_ticket;  
    int now_serving;  
};  
  
void Lock(lock* l) {  
    int my_ticket = atomic_increment(&l->next_ticket); // take a "ticket"  
    while (my_ticket != l->now_serving);                // wait for number to be called  
}  
  
void unlock(lock* l) {  
    l->now_serving++;  
}
```

No atomic operation needed to acquire the lock (only a read)

Result: only one invalidation per lock release ($O(P)$ interconnect traffic)

Atomic operations (provided by CUDA)

```
int    atomicAdd(int* address, int val);
float  atomicAdd(float* address, float val);
int    atomicSub(int* address, int val);
int    atomicExch(int* address, int val);
float  atomicExch(float* address, float val);
int    atomicMin(int* address, int val);
int    atomicMax(int* address, int val);
unsigned int atomicInc(unsigned int* address, unsigned int val);
unsigned int atomicDec(unsigned int* address, unsigned int val);
int    atomicCAS(int* address, int compare, int val);
int    atomicAnd(int* address, int val);  // bitwise
int    atomicOr(int* address, int val);   // bitwise
int    atomicXor(int* address, int val);  // bitwise
```

(omitting additional 64 bit and unsigned int versions)

Implementing atomic fetch-and-op

Exercise: how can you build an atomic fetch+op out of atomicCAS()?

Example: atomic_min()

```
// atomicCAS: ("compare and swap")
// performs the following logic atomically
int atomicCAS(int* addr, int compare, int val) {
    int old = *addr;
    *addr = (old == compare) ? val : old;
    return old;
}
```

```
void atomic_min(int* addr, int x) {
    int old = *addr;
    int new = min(old, x);
    while (atomicCAS(addr, old, new) != old) {
        old = *addr;
        new = min(old, x);
    }
}
```

What about these operations?

```
int atomic_increment(int* addr, int x);    // for signed values of x
void lock(int* addr);
```

Another exercise: build a lock

Let's build a lock using compare and swap:

```
// atomicCAS:  
// atomic compare and swap performs the following logic atomically  
int atomicCAS(int* addr, int compare, int val) {  
    int old = *addr;  
    *addr = (old == compare) ? val : old;  
    return old;  
}
```

```
typedef int lock;  
  
void lock(Lock* l) {  
    while (atomicCAS(l, 0, 1) == 1);  
}  
  
void unlock(Lock* l) {  
    *l = 0;  
}
```

The following is potentially more efficient under contention: Why?

```
void lock(Lock* l) {  
    while (1) {  
        while(*l == 1);  
        if (atomicCAS(l, 0, 1) == 0)  
            return;  
    }  
}
```


Load-linked, store conditional (LL/SC)

- **Pair of corresponding instructions (not a single atomic instruction like compare-and-swap)**
 - **load_linked(x):** load value from address
 - **store_conditional(x, value):** store value to x, if x hasn't been written to by any processor since the corresponding load linked operation
- **Corresponding ARM instructions: LDREX and STREX**
- **How might LL/SC be implemented on a cache coherent processor?**

C++ 11 `atomic<T>`

- Provides atomic read, write, read-modify-write of entire objects
 - Atomicity may be implemented by mutex or efficiently by processor-supported atomic instructions (if T is a basic type)
- Provides memory ordering semantics for operations before and after atomic operations
 - By default: sequential consistency
 - See `std::memory_order` or more detail

```
atomic<int> i;  
i++; // atomically increment i  
  
int a = i;  
// do stuff  
i.compare_exchange_strong(a, 10); // if i has same value as a, set i to 10  
bool b = i.is_lock_free();        // true if implementation of atomicity  
                                   // is lock free
```

Using locks

Example: a sorted linked list

What can go wrong if multiple threads operate on the linked list simultaneously?

```
struct Node {
    int value;
    Node* next;
};

struct List {
    Node* head;
};
```

```
void insert(List* list, int value) {

    Node* n = new Node;
    n->value = value;

    // assume case of inserting before head of
    // of list is handled here (to keep slide simple)

    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value > value)
            break;

        prev = cur;
        cur = cur->next;
    }

    n->next = cur;
    prev->next = n;
}
```

```
void delete(List* list, int value) {

    // assume case of deleting first node in list
    // is handled here (to keep slide simple)

    Node* prev = list->head;
    Node* cur = list->head->next;

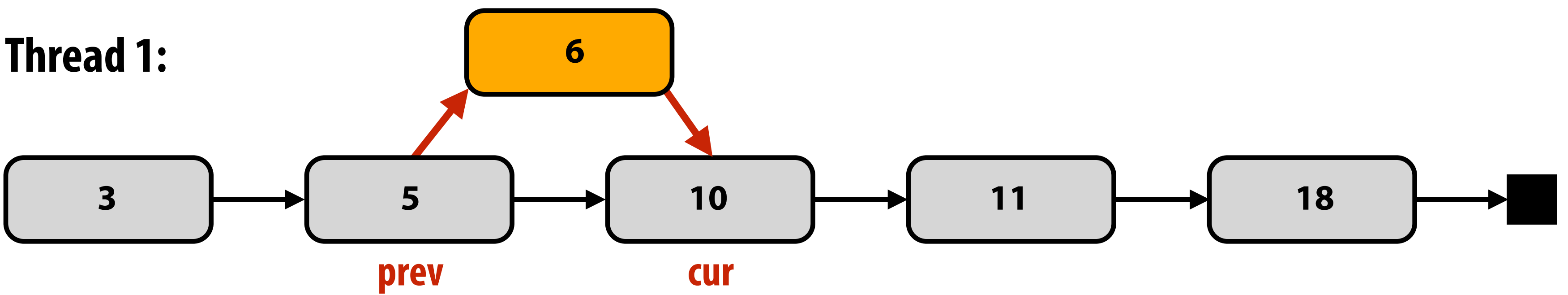
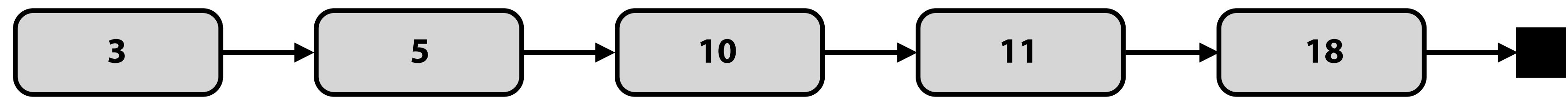
    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            delete cur;
            return;
        }

        prev = cur;
        cur = cur->next;
    }
}
```

Example: simultaneous insertion

Thread 1 attempts to insert 6

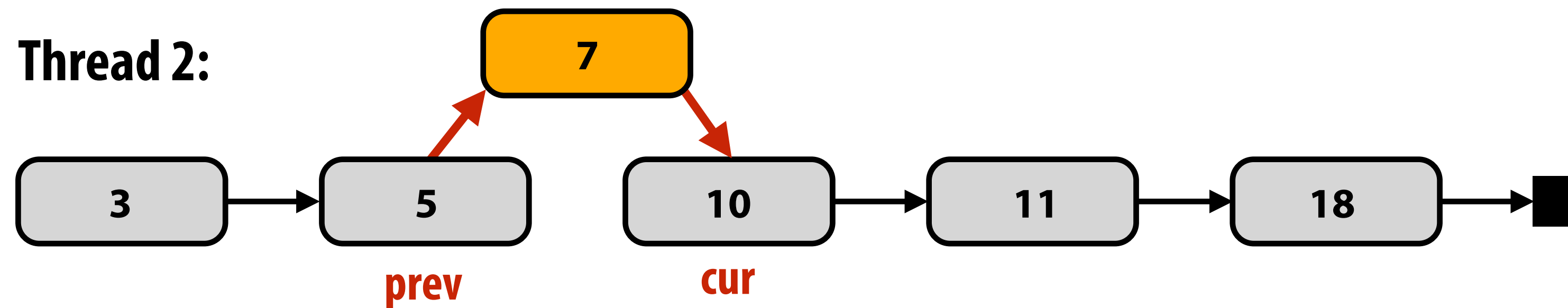
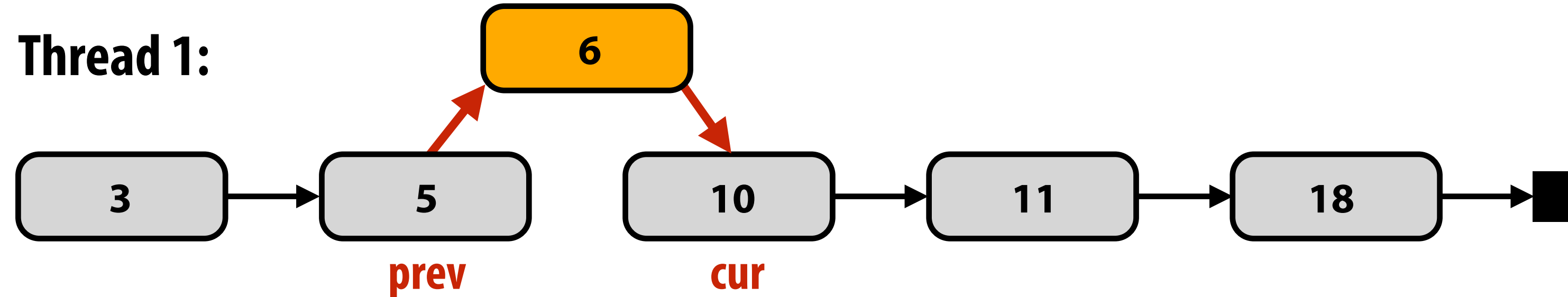
Thread 2 attempts to insert 7



Example: simultaneous insertion

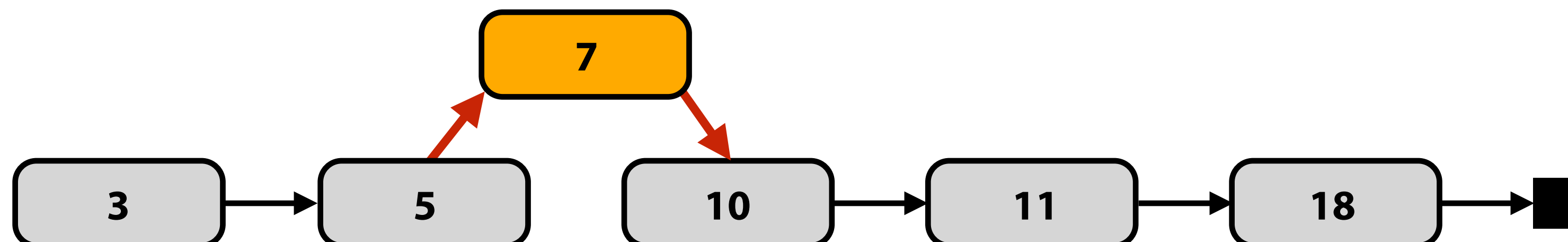
Thread 1 attempts to insert 6

Thread 2 attempts to insert 7



Thread 1 and thread 2 both compute same prev and cur.
Result: one of the insertions gets lost!

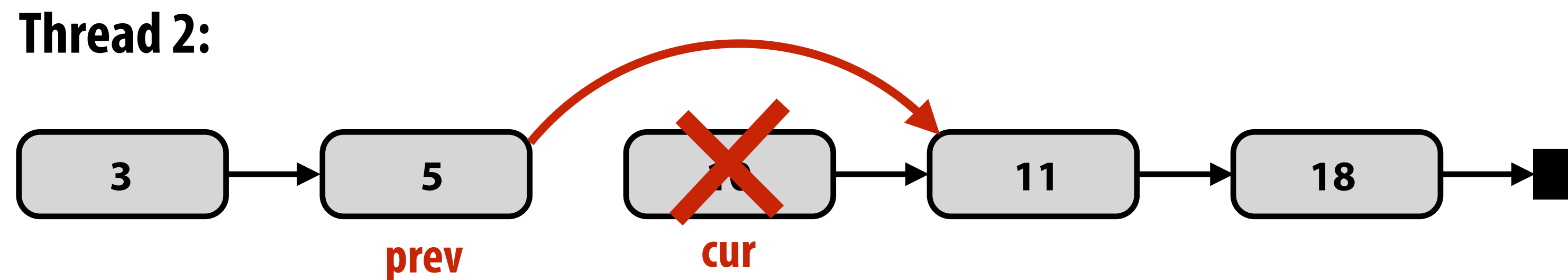
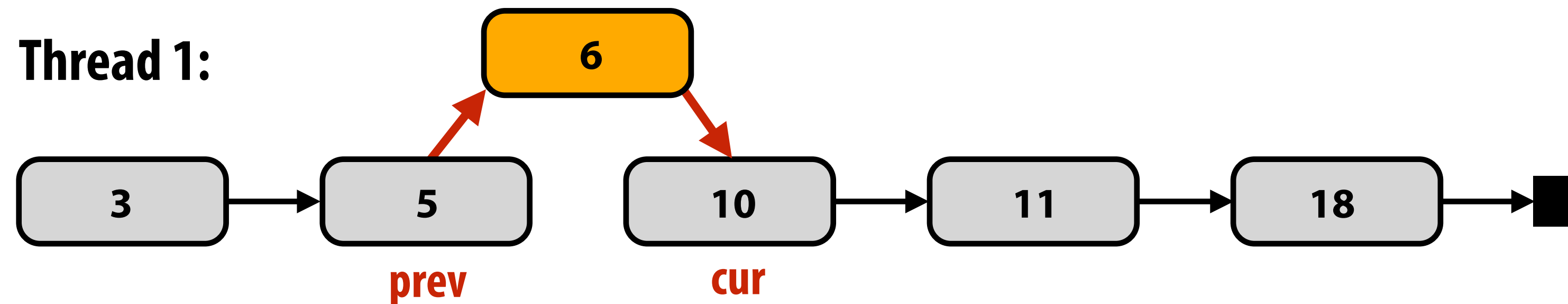
Result: (assuming thread 1 updates prev->next before thread 2)



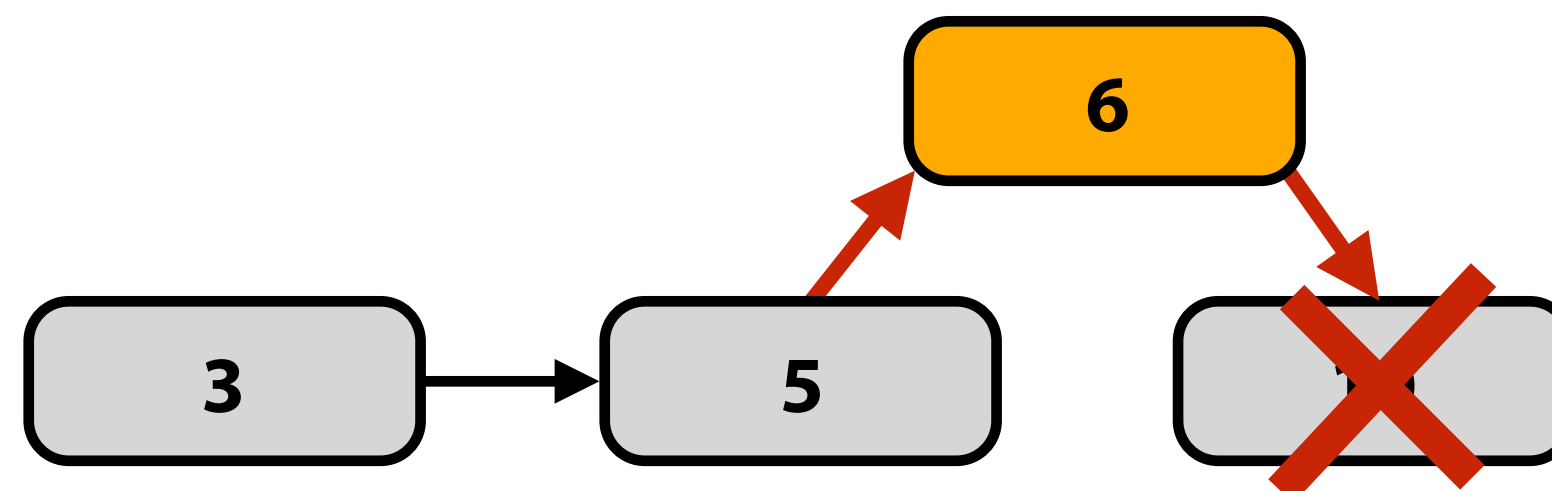
Example: simultaneous insertion/deletion

Thread 1 attempts to insert 6

Thread 2 attempts to delete 10



Possible result: (thread 2 finishes delete first)



Solution 1: protect the list with a single lock

```
struct Node {  
    int value;  
    Node* next;  
};
```

```
struct List {  
    Node* head;  
    Lock lock;  
};
```

Per-list lock



```
void insert(List* list, int value) {  
  
    Node* n = new Node;  
    n->value = value;  
  
    lock(list->lock);  
  
    // assume case of inserting before head of  
    // of list is handled here (to keep slide simple)  
  
    Node* prev = list->head;  
    Node* cur = list->head->next;  
  
    while (cur) {  
        if (cur->value > value)  
            break;  
  
        prev = cur;  
        cur = cur->next;  
    }  
    n->next = cur;  
    prev->next = n;  
    unlock(list->lock);  
}
```

```
void delete(List* list, int value) {  
  
    lock(list->lock);  
  
    // assume case of deleting first element is  
    // handled here (to keep slide simple)  
  
    Node* prev = list->head;  
    Node* cur = list->head->next;  
  
    while (cur) {  
        if (cur->value == value) {  
            prev->next = cur->next;  
            delete cur;  
            unlock(list->lock);  
            return;  
        }  
  
        prev = cur;  
        cur = cur->next;  
    }  
    unlock(list->lock);  
}
```

Single global lock per data structure

- **Good:**

- **It is relatively simple to implement correct mutual exclusion for data structure operations (we just did it!)**

- **Bad:**

- **Operations on the data structure are serialized**
- **May limit parallel application performance**

Challenge: who can do better?

```
struct Node {  
    int value;  
    Node* next;  
};
```

```
struct List {  
    Node* head;  
};
```

```
void insert(List* list, int value) {
```

```
    Node* n = new Node;  
    n->value = value;
```

```
    // assume case of inserting before head of  
    // of list is handled here (to keep slide simple)
```

```
    Node* prev = list->head;  
    Node* cur = list->head->next;
```

```
    while (cur) {  
        if (cur->value > value)  
            break;
```

```
        prev = cur;  
        cur = cur->next;
```

```
    }
```

```
    prev->next = n;  
    n->next = cur;
```

```
}
```

```
void delete(List* list, int value) {
```

```
    // assume case of deleting first element is  
    // handled here (to keep slide simple)
```

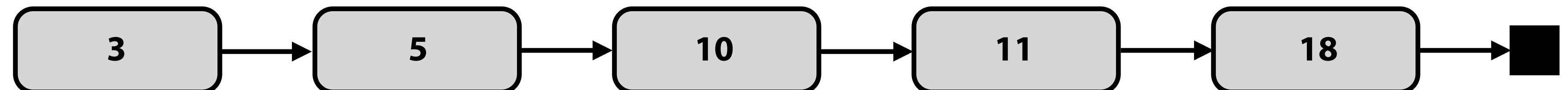
```
    Node* prev = list->head;  
    Node* cur = list->head->next;
```

```
    while (cur) {  
        if (cur->value == value) {  
            prev->next = cur->next;  
            delete cur;  
            return;  
        }
```

```
        prev = cur;  
        cur = cur->next;
```

```
    }
```

```
}
```



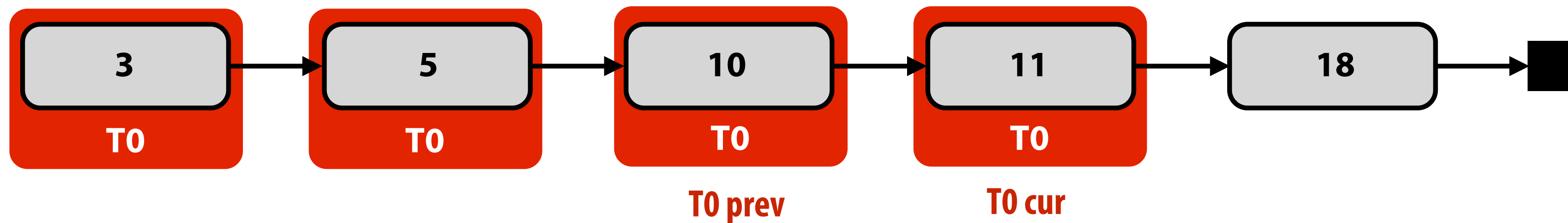
Hand-over-hand traversal



Credit: (Hal Boedeker, Orlando Sentinel) American Ninja Warrior

Solution 2: “hand-over-hand” locking

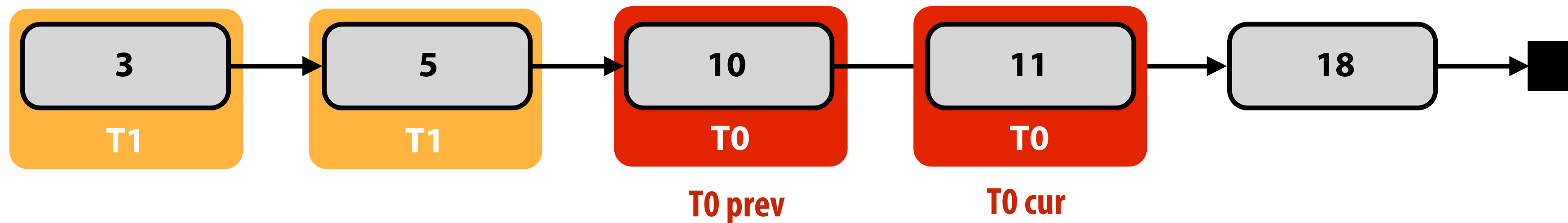
Thread 0: delete(11)



Solution 2: “hand-over-hand” locking

Thread 0: delete(11)

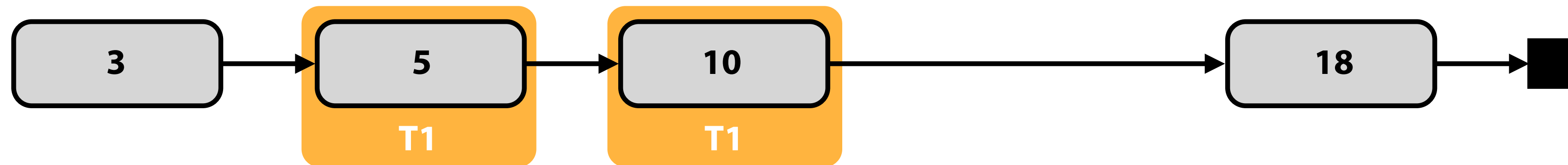
Thread 1: delete(10)



Solution 2: “hand-over-hand” locking

Thread 0: delete(11)

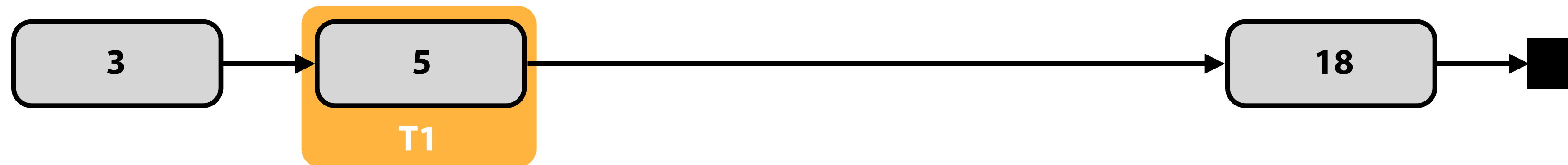
Thread 1: delete(10)



Solution 2: “hand-over-hand” locking

Thread 0: delete(11)

Thread 1: delete(10)



Solution 2: fine-grained locking

```
struct Node {
    int value;
    Node* next;
    Lock* lock;
};

struct List {
    Node* head;
    Lock* lock;
};

void insert(List* list, int value) {

    Node* n = new Node;
    n->value = value;

    // assume case of insert before head handled
    // here (to keep slide simple)

    Node* prev, *cur;

    lock(list->lock);
    prev = list->head;

    lock(prev->lock);
    unlock(list->lock);

    cur = prev->next;
    if (cur) lock(cur->lock);

    while (cur) {
        if (cur->value > value)
            break;

        Node* old_prev = prev;
        prev = cur;
        cur = cur->next;
        unlock(old_prev->lock);
        if (cur) lock(cur->lock);
    }

    n->next = cur;
    prev->next = n;

    unlock(prev->lock);
    if (cur) unlock(cur->lock);
}
```

Challenge to students: there is way to further improve the implementation of insert(). What is it?

```
void delete(List* list, int value) {

    // assume case of delete head handled here
    // (to keep slide simple)

    Node* prev, *cur;

    lock(list->lock);
    prev = list->head;

    lock(prev->lock);
    unlock(list->lock);

    cur = prev->next;
    if (cur) lock(cur->lock)

    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            unlock(prev->lock);
            unlock(cur->lock);
            delete cur;
            return;
        }

        Node* old_prev = prev;
        prev = cur;
        cur = cur->next;
        unlock(old_prev->lock);
        if (cur) lock(cur->lock);
    }
    unlock(prev->lock);
}
```

Fine-grained locking

■ Goal: enable parallelism in data structure operations

- Reduces contention for global data structure lock
- In the linked-list example: a single monolithic lock is overly conservative (operations on different parts of the linked list can proceed in parallel)

■ Challenge: tricky to ensure correctness

- Determining when mutual exclusion is required
- Deadlock? (Self-check: in the linked-list example, why do you immediately know that the code is deadlock free?)
- Livelock?

■ Costs?

- Overhead of taking a lock each traversal step (extra instructions + traversal now involves memory writes)
- Extra storage cost (a lock per node)
- What is a middle-ground solution that trades off some parallelism for reduced overhead? (hint: similar issue to selection of task granularity)

Practice exercise (on your own time)

- **Implement a fine-grained locking implementation of a binary search tree supporting insert and delete**

```
struct Tree {  
    Node* root;  
};
```

```
struct Node {  
    int value;  
    Node* left;  
    Node* right;  
};
```

```
void insert(Tree* tree, int value);  
void delete(Tree* tree, int value);
```

A quick introduction to lock-free data structures

Blocking algorithms/data structures

- A blocking algorithm allows one thread to prevent other threads from completing operations on a shared data structure indefinitely
- Example:
 - Thread 0 takes a lock on a node in our linked list
 - Thread 0 is swapped out by the OS, or crashes, or is just really slow (takes a page fault), etc.
 - Now, no other threads can complete operations on the data structure (although thread 0 is not actively making progress modifying it)
- An algorithm that uses locks is blocking regardless of whether the lock implementation uses spinning or pre-emption

Lock-free algorithms

- **Non-blocking algorithms are lock-free if some thread is guaranteed to make progress (“systemwide progress”)**
 - **In lock-free case, it is not possible to preempt one of the threads at an inopportune time and prevent progress by rest of system**
 - **Note: this definition does not prevent starvation of any one thread**

Single reader, single writer bounded queue *

```
struct Queue {
    int data[N];
    int head;    // head of queue
    int tail;    // next free element
};

void init(Queue* q) {
    q->head = q->tail = 0;
}
```

- Only two threads (one producer, one consumer) accessing queue at the same time
- Threads never synchronize or wait on each other
 - When queue is empty (pop fails), when it is full (push fails)

```
// return false if queue is full
bool push(Queue* q, int value) {

    // queue is full if tail is element before head
    if (q->tail == MOD_N(q->head - 1))
        return false;

    q->data[q->tail] = value;
    q->tail = MOD_N(q->tail + 1);
    return true;
}

// returns false if queue is empty
bool pop(Queue* q, int* value) {

    // if not empty
    if (q->head != q->tail) {
        *value = q->data[q->head];
        q->head = MOD_N(q->head + 1);
        return true;
    }
    return false;
}
```

* Assume a sequentially consistent memory system for now
(or the presence of appropriate memory fences, or C++ 11 `atomic<>`)

Single reader, single writer unbounded queue *

Source: Dr. Dobbs Journal

```
struct Node {
    Node* next;
    int    value;
};

struct Queue {
    Node* head;
    Node* tail;
    Node* reclaim;
};

void init(Queue* q) {
    q->head = q->tail = q->reclaim = new Node;
}
```

- Tail points to last element added (if non-empty)
- Head points to element BEFORE head of queue
- Node allocation and deletion performed by the same thread (producer thread)

```
void push(Queue* q, int value) {

    Node* n = new Node;
    n->next = NULL;
    n->value = value;

    q->tail->next = n;
    q->tail = q->tail->next;

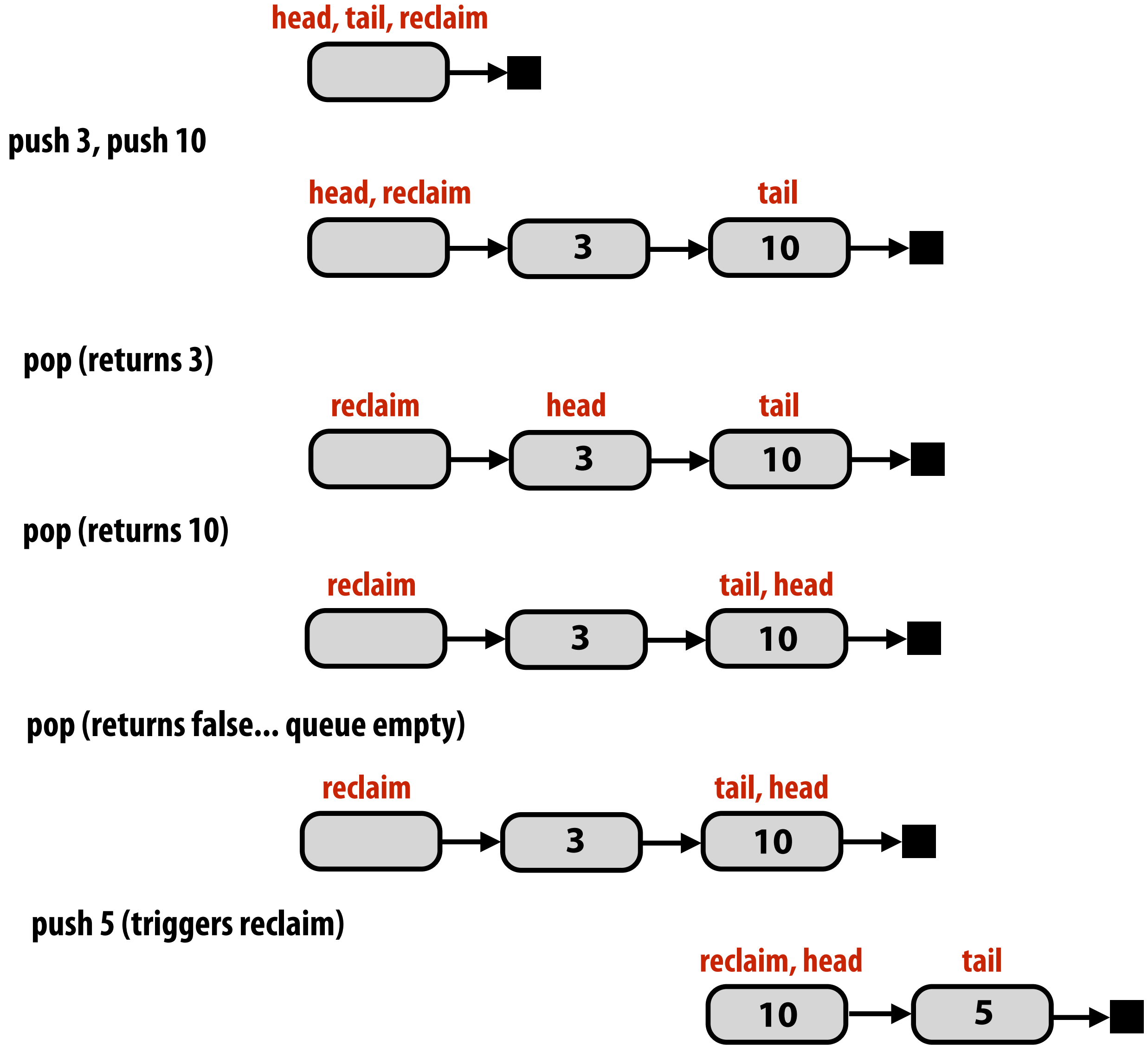
    while (q->reclaim != q->head) {
        Node* tmp = q->reclaim;
        q->reclaim = q->reclaim->next;
        delete tmp;
    }

    // returns false if queue is empty
    bool pop(Queue* q, int* value) {

        if (q->head != q->tail) {
            *value = q->head->next->value;
            q->head = q->head->next;
            return true;
        }
        return false;
    }
}
```

* Assume a sequentially consistent memory system for now
(or the presence of appropriate memory fences, or C++ 11 atomic<>)

Single reader, single writer unbounded queue



Lock-free stack (first try)

```
struct Node {  
    Node* next;  
    int    value;  
};
```

```
struct Stack {  
    Node* top;  
};
```

Main idea: as long as no other thread has modified the stack, a thread's modification can proceed.

Note difference from fine-grained locking: In fine-grained locking, the implementation locked a part of a data structure. Here, threads do not hold lock on data structure at all.

```
void init(Stack* s) {  
    s->top = NULL;  
}
```

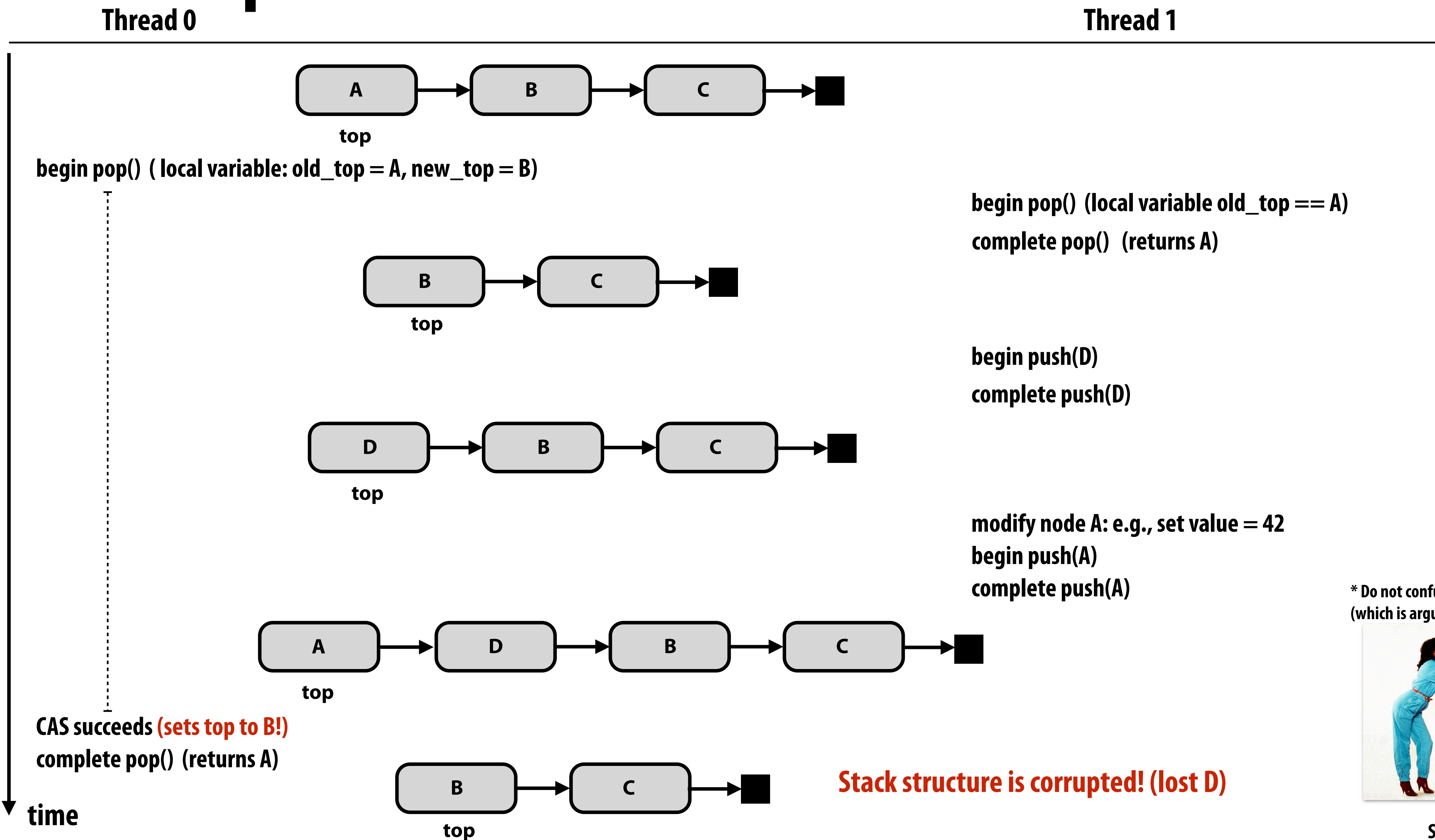
```
void push(Stack* s, Node* n) {  
    while (1) {  
        Node* old_top = s->top;  
        n->next = old_top;  
        if (compare_and_swap(&s->top, old_top, n) == old_top)  
            return;  
    }  
}
```

```
Node* pop(Stack* s) {  
    while (1) {  
        Node* old_top = s->top;  
        if (old_top == NULL)  
            return NULL;  
        Node* new_top = old_top->next;  
        if (compare_and_swap(&s->top, old_top, new_top) == old_top)  
            return old_top;  
    }  
}
```

* Assume a sequentially consistent memory system for now
(or the presence of appropriate memory fences, or C++ 11 `atomic<>`)

The ABA problem *

Careful: On this slide A, B, C, and D are addresses of nodes, not the value stored by the nodes!



* Do not confuse with the ABBA problem (which is arguably larger)



Lock-free stack using counter for ABA soln

```
struct Node {  
    Node* next;  
    int    value;  
};  
  
struct Stack {  
    Node* top;  
    int    pop_count;  
};
```

```
void init(Stack* s) {  
    s->top = NULL;  
}  
  
void push(Stack* s, Node* n) {  
    while (1) {  
        Node* old_top = s->top;  
        n->next = old_top;  
        if (compare_and_swap(&s->top, old_top, n) == old_top)  
            return;  
    }  
}
```

```
Node* pop(Stack* s) {  
    while (1) {  
        int pop_count = s->pop_count;  
        Node* top = s->top;  
        if (top == NULL)  
            return NULL;  
        Node* new_top = top->next;  
        if (double_compare_and_swap(&s->top, top, new_top,  
                                    &s->pop_count, pop_count, pop_count+1))  
            return top;  
    }  
}
```

test to see if either have changed
(assume function returns true if no changes)

- Maintain counter of pop operations
- Requires machine to support “double compare and swap” (DCAS) or doubleword CAS
- Could also solve ABA problem with careful node allocation and/or element reuse policies

Compare and swap on x86

- **x86 supports a “double-wide” compare-and-swap instruction**
 - Not quite the “double compare-and-swap” used on the previous slide
 - But could simply ensure the stack’s count and top fields are contiguous in memory to use the 64-bit wide single compare-and-swap instruction below.
- **cmpxchg8b**
 - “compare and exchange eight bytes”
 - Can be used for compare-and-swap of two 32-bit values
- **cmpxchg16b**
 - “compare and exchange 16 bytes”
 - Can be used for compare-and-swap of two 64-bit values

Another problem: referencing freed memory

```
struct Node {
    Node* next;
    int    value;
};

struct Stack {
    Node* top;
    int    pop_count;
};
```

```
void init(Stack* s) {
    s->top = NULL;
}

void push(Stack* s, int value) {
    Node* n = new Node;
    n->value = value;
    while (1) {
        Node* old_top = s->top;
        n->next = old_top;
        if (compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}
```

```
int pop(Stack* s) {
    while (1) {
        Stack old;
        old.pop_count = s->pop_count;
        old.top = s->top;

        if (old.top == NULL)
            return NULL;

        Stack new_stack;
        new_stack.top = old.top->next;
        new_stack.pop_count = old.pop_count+1;

        if (doubleword_compare_and_swap(s, old, new_stack))
            int value = old.top->value;
            delete old.top;
            return value;
    }
}
```

old top might have been freed at this point
(by some other thread that popped it)



[Advanced topic] Hazard pointer: avoid freeing a node until it's known that all other threads do not hold reference to it

```
struct Node {
    Node* next;
    int value;
};

struct Stack {
    Node* top;
    int pop_count;
};

// per thread ptr (node that cannot
// be deleted since the thread is
// accessing it)
Node* hazard;

// list of nodes this thread must
// delete (this is a per thread list)
Node* retireList;
int  retireListSize;

// delete nodes if possible
void retire(Node* ptr) {
    push(retireList, ptr);
    retireListSize++;

    if (retireListSize > THRESHOLD)
        for (each node n in retireList) {
            if (n not pointed to by any
                thread's hazard pointer) {
                remove n from list
                delete n;
            }
        }
}
```

```
void init(Stack* s) {
    s->top = NULL;
}

void push(Stack* s, int value) {
    Node* n = new Node;
    n->value = value;
    while (1) {
        Node* old_top = s->top;
        n->next = old_top;
        if (compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}

int pop(Stack* s) {
    while (1) {
        Stack old;
        old.pop_count = s->pop_count;
        old.top = hazard = s->top;

        if (old.top == NULL) {
            return NULL;
        }

        Stack new_stack;
        new_stack.top = old.top->next;
        new_stack.pop_count = old.pop_count+1;

        if (doubleword_compare_and_swap(s, old, new_stack)) {
            int value = old.top->value;
            retire(old.top);
            return value;
        }
        hazard = NULL;
    }
}
```

Lock-free linked list insertion *

```
struct Node {
    int value;
    Node* next;
};

struct List {
    Node* head;
};

// insert new node after specified node
void insert_after(List* list, Node* after, int value) {

    Node* n = new Node;
    n->value = value;

    // assume case of insert into empty list handled
    // here (keep code on slide simple for class discussion)

    Node* prev = list->head;

    while (prev->next) {
        if (prev == after) {
            while (1) {
                Node* old_next = prev->next;
                n->next = old_next;
                if (compare_and_swap(&prev->next, old_next, n) == old_next)
                    return;
            }
        }
        prev = prev->next;
    }
}
```

Compared to fine-grained locking implementation:

No overhead of taking locks

No per-node storage overhead

* For simplicity, this slide assumes the **only** operation on the list is insert. Delete is more complex.

Lock-free linked list deletion

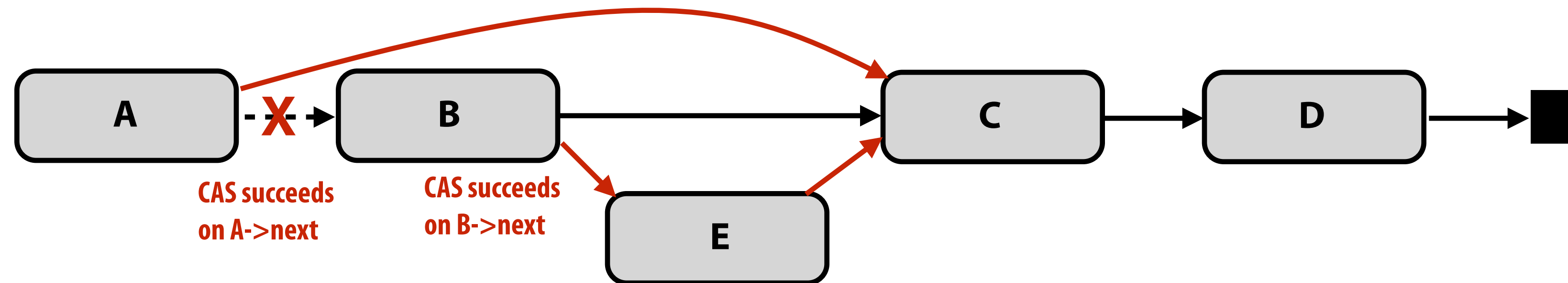
Supporting lock-free deletion significantly complicates data-structure

Consider case where B is deleted simultaneously with insertion of E after B.

B now points to E, but B is not in the list!

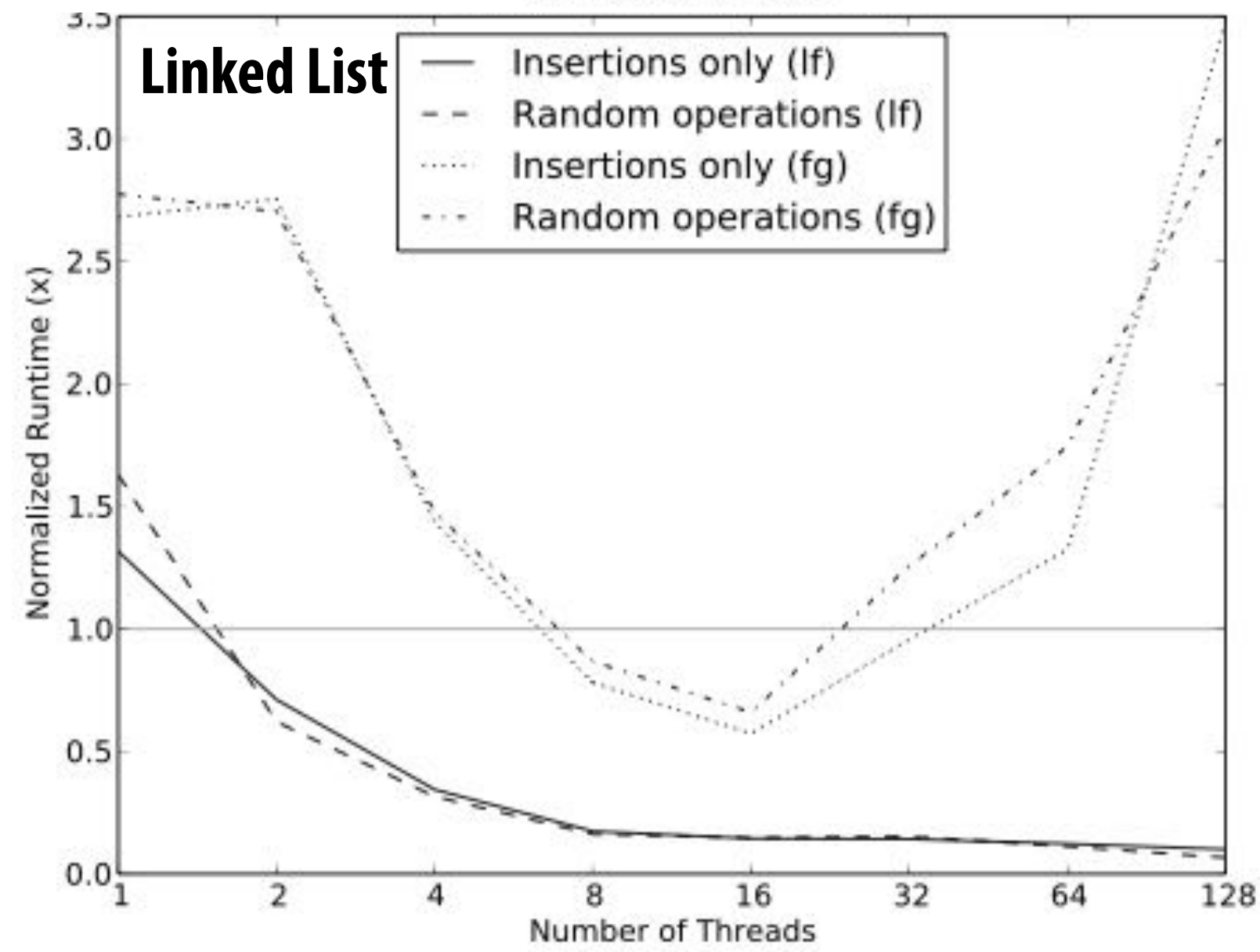
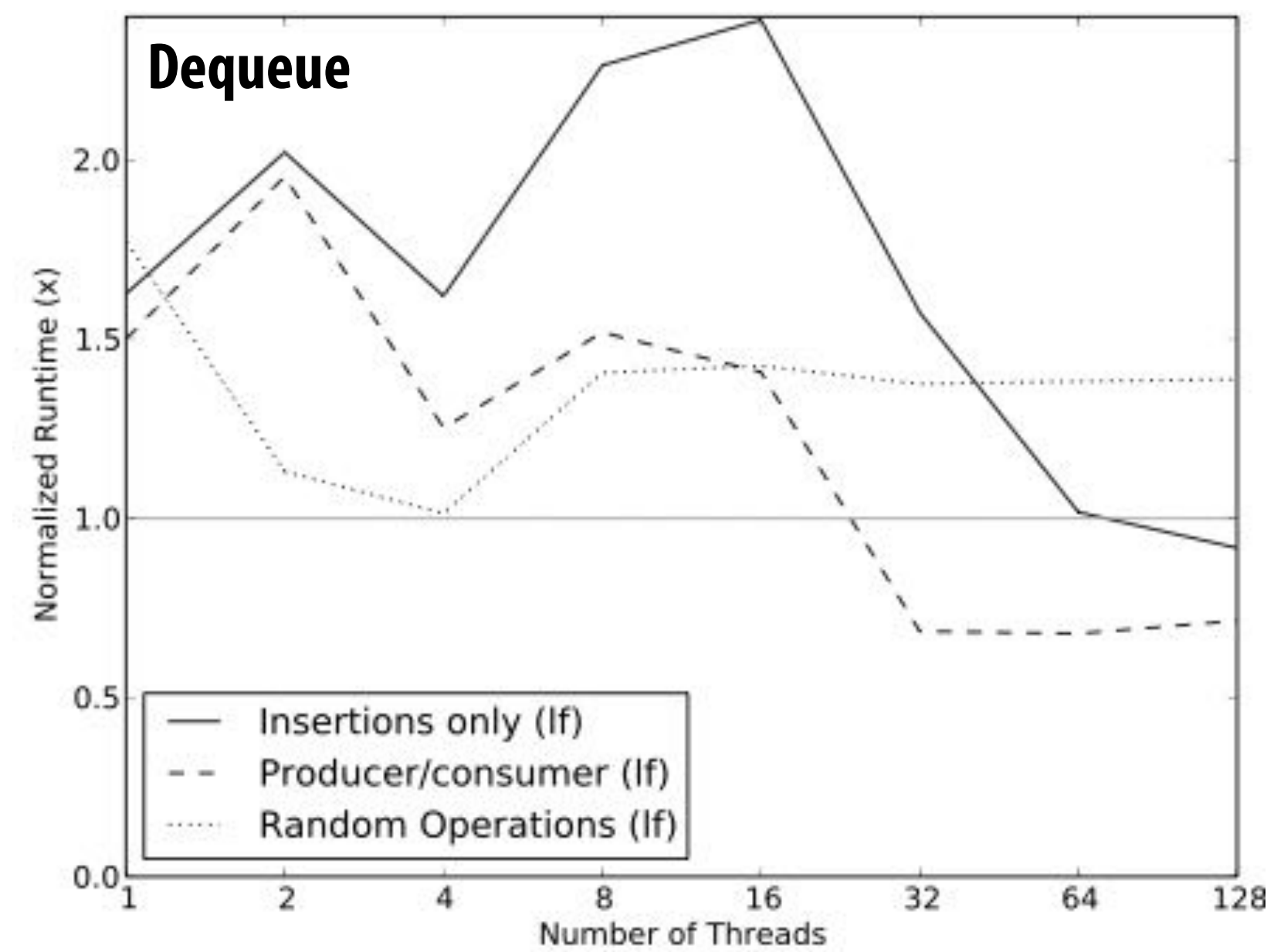
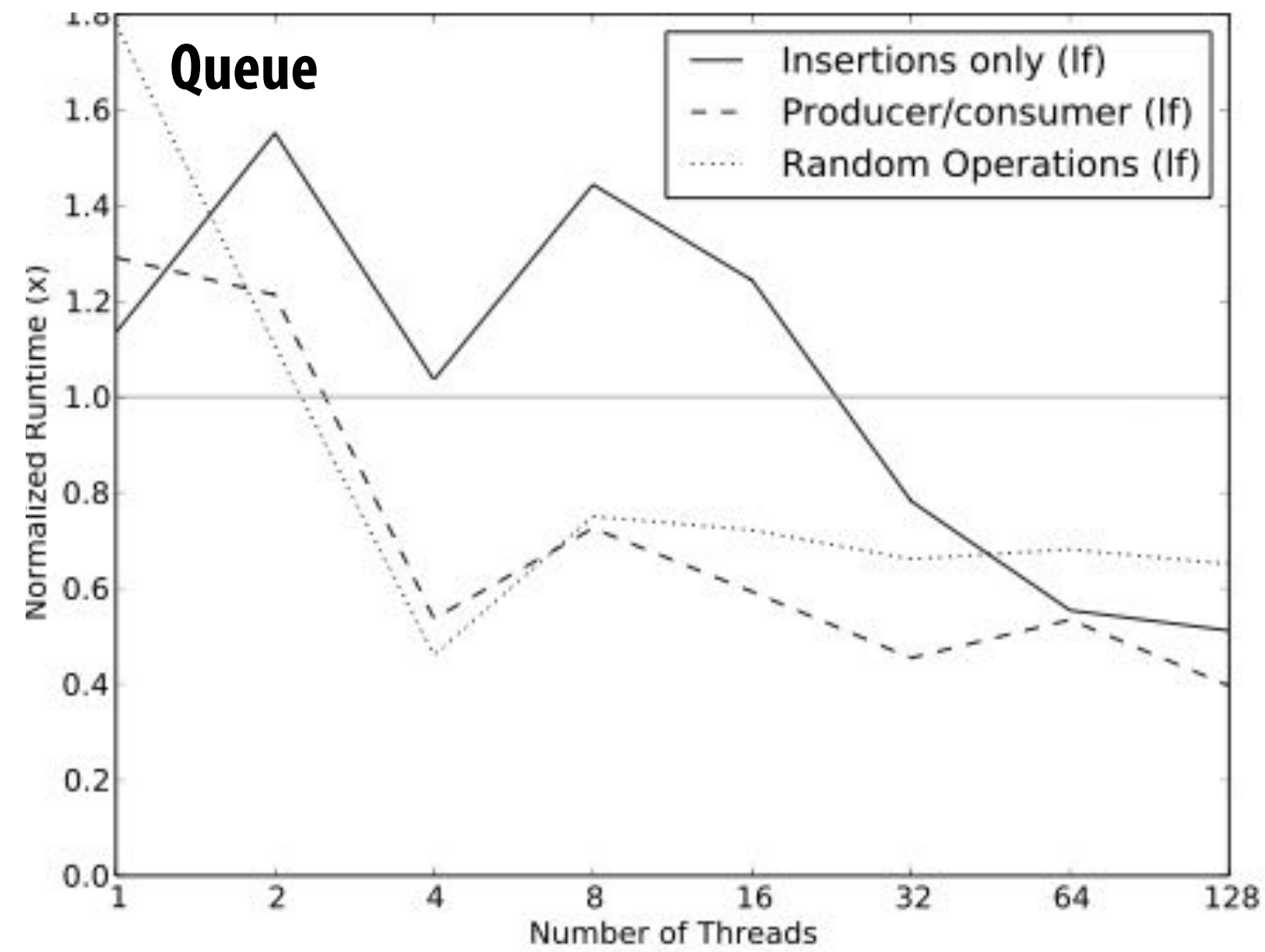
For the curious:

- Harris 2001. "A Pragmatic Implementation of Non-blocking Linked-Lists"
- Fomitchev 2004. "Lock-free linked lists and skip lists"



Lock-free vs. locks performance comparison

Lock-free algorithm run time normalized to run time of using pthread mutex locks



If = "lock free"

fg = "fine grained lock"

Source: Hunt 2011. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures

In practice: why lock free data structures?

- **When optimizing parallel programs in this class you often assume that only your program is using the machine**
 - Because you care about performance
 - Typical assumption in scientific computing, graphics, machine learning, data analytics, etc.
- **In these cases, well-written code with locks can be as fast (or faster) than lock-free code**
 - And is often much simpler to implement
- **But there are situations where code with locks can suffer from tricky performance problems**
 - Situations where a program features many threads (e.g., database, webserver) and page faults, pre-emption, etc. can occur while a thread is in a critical section
 - Locks create problems like priority inversion, convoying, crashing in critical section, etc. that are often discussed in OS classes

Summary

- **Use fine-grained locking to reduce contention (maximize parallelism) in operations on shared data structures**
 - But fine-granularity can increase code complexity (errors) and increase execution overhead
- **Lock-free data structures: non-blocking solution to avoid some overheads/pitfalls of locks**
 - But can be tricky to implement (and ensuring correctness in a lock-free setting has its own overheads)
 - Still requires appropriate memory fences on modern relaxed consistency hardware
- **Note: a lock-free design does not eliminate contention**
 - Compare-and-swap can fail under heavy contention, requiring spins

Preview: transactional memory

- **Q. What was the role of the compare and swap in our lock-free implementations?**
- **A. Determining if another thread had modified the data structure while the calling thread was in the middle of an operation.**
- **Next time... transactional memory**
 - **A more general mechanism to allow a system to speculate that an operation will be successfully completed before another thread attempts to modify the structure**
 - **With mechanisms to “abort” an operation in the event another thread does.**

More reading on lock-free structures

- **Michael and Scott 1996. Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms**
 - **Multiple reader/writer lock-free queue**
- **Harris 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists**
- **Michael Sullivan's Relaxed Memory Calculus (RMC) compiler**
 - **<https://github.com/msullivan/rmc-compiler>**
- **Many good blog posts and articles on the web:**
 - **<http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279>**
 - **<http://developers.memsql.com/blog/common-pitfalls-in-writing-lock-free-algorithms/>**