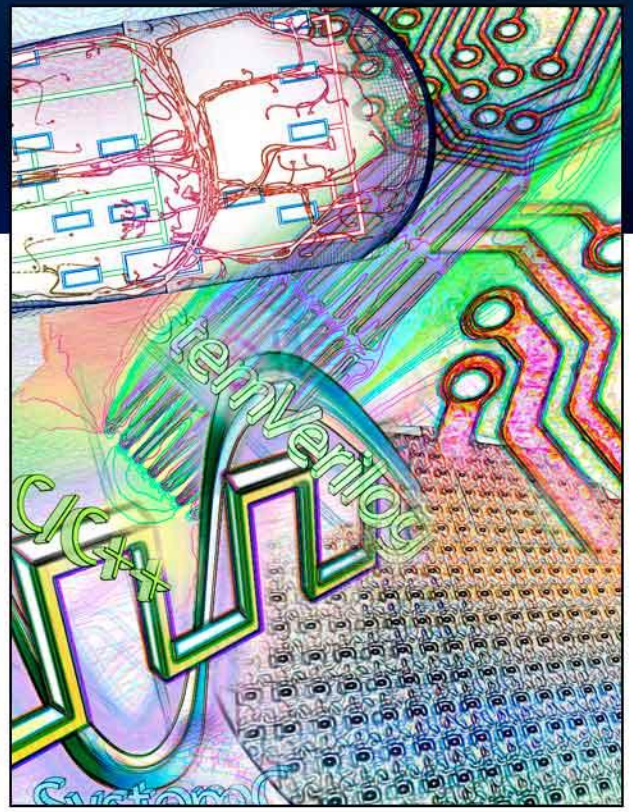


Live Lock-Free
or Deadlock (practical lock-free programming)

Fedor G Pikus

Mentor Graphics, Design2Silicon Division

CPPCon, September 2015

Mentor Graphics®

Roadmap

- Appropriate guru worshiping and supplication
- Simple lock-free example – DCLP
- DCLP in depth, and the hard part of lock-free
- Slightly more complex example – a [kind of] queue
- What is a lock-free program, really?
- What's wrong with locks and what can lock-free get us?
- Locks and lock-free, in real life?
- Back to the [kind of] queue
- Performance of concurrent programs, in depth
- Just what is a concurrent queue? (with real example)
- Back to DCLP

This talk is

- about writing good concurrent programs
  - With C++ examples
- about practical concurrency in everyday applications
- accurate (or strives to be) but not precise
  - Goal is practicality; some things are simplified and not the whole truth but "good enough"
- the one where questions are encouraged

About the author (the lens I look through at software)

- I work in EDA (Electronic Design Automation) industry
  - We write the software used to design and manufacture your phone, your TV, your laptop, and every other integrated circuit
- Very Large Scale Integrated circuits are very large
  - Some tools run for days on 100K CPU cores for one chip
- We write software for sale (we have customers)
  - We support wide variety of hardware and OS platforms
- Customers are very conservative
  - One bad mistake can cost $100M
  - No change for the sake of change
  - Half of our customers are on RHEL5 (i.e. no C++11 for us)

About the subject

- Writing lock-free programs is hard
- Writing correct lock-free programs is even harder
- In practice, most lock-free code is written in attempt to get better performance
- Rule #1 of performance:
- Never guess about performance!
- Lock-free algorithms do not always provide better performance

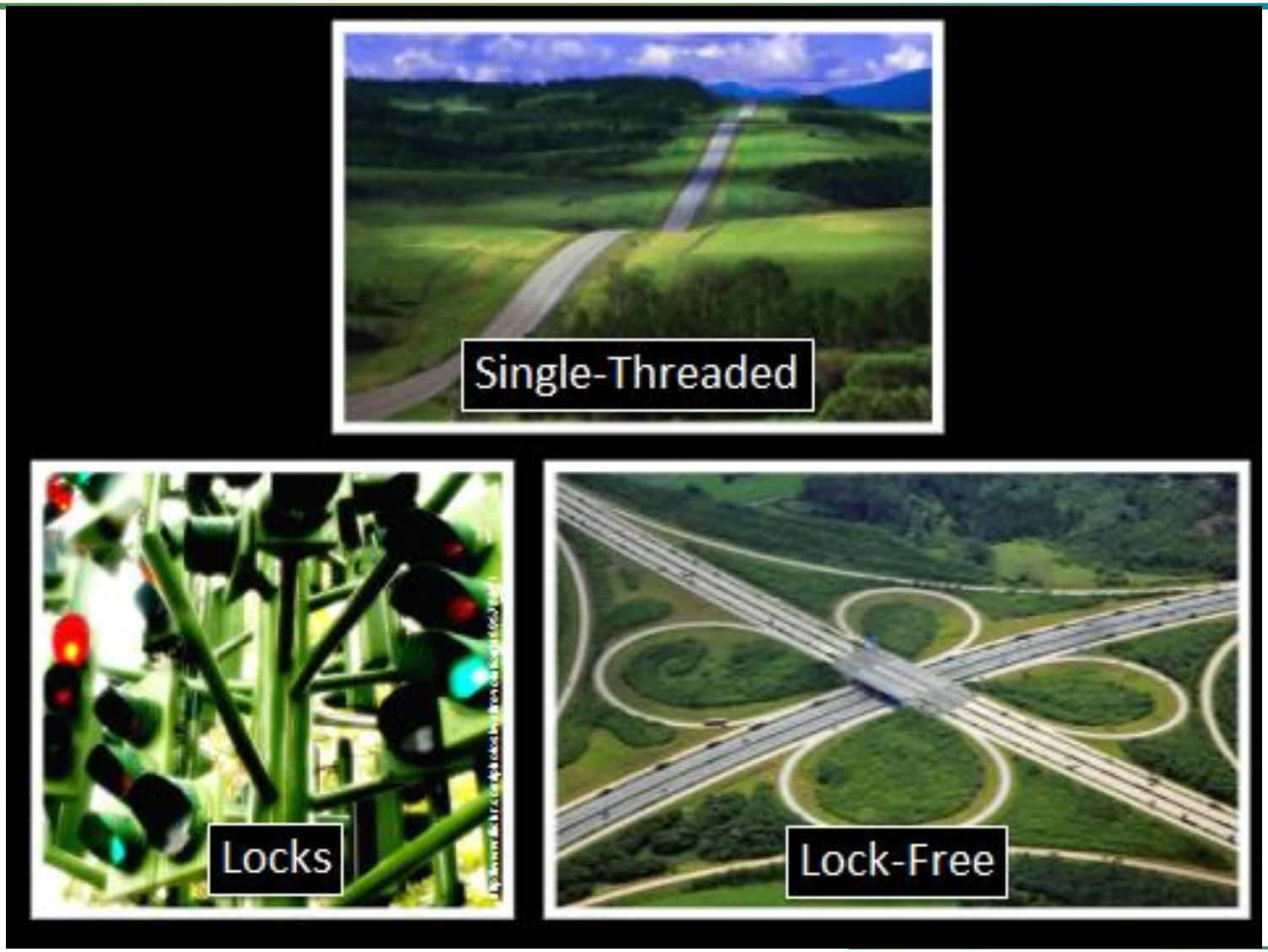
Take-home points

- Lock-free programming is hard, be sure you need it
- Never guess about performance

Note on C++11 atomics support

- Still not 100% available in commercial applications
- More complex than the simplified level we will use
- More simple than I'd like (obscures some details)
  - For teaching purposes – in practical work use C++11 if you can
- Easy to convert from pseudo-code to C++11, Boost, or other atomic library
- We will use (mostly) a C++ API that could be a library API or can trivially map to another API

Herb Sutter says…



Double-Checked locking

- One of the simplest examples of lock-free programming
- Everyone uses it
- But it has some depth that is often ignored
- ~~Thread-safe static initializer in C++11 made double-checked locking largely irrelevant~~
- Thread-safe static initializer in C++11 made one application of double-checked locking largely irrelevant
- Double-checked locking is very useful in real-life lock-free programming (to be continued…)

Singleton Initialization problem

```
class Gizmo { …
  void Transmogrify(); // Thread-safe!
… };
void OperateGizmo() {
  static Gizmo* the_gizmo = new Gizmo;
  the_gizmo->Transmogrify();
}
```

- Is OperateGizmo() thread-safe?
  - Transmogrify() is thread-safe even when called on the same Gizmo by several threads
  - There should be only one Gizmo by design
- Is static initialization thread-safe?
  - In C++11, yes (if the compiler is doing it right)
  - In C++03, may be

Thread-safe initialization

```
static Gizmo* the_gizmo = new Gizmo;
```

- What does static initialization do?

```
static Gizmo* the_gizmo = NULL; // Linker-initialized

if (the_gizmo == NULL) {
  the_gizmo = new Gizmo;
}
```

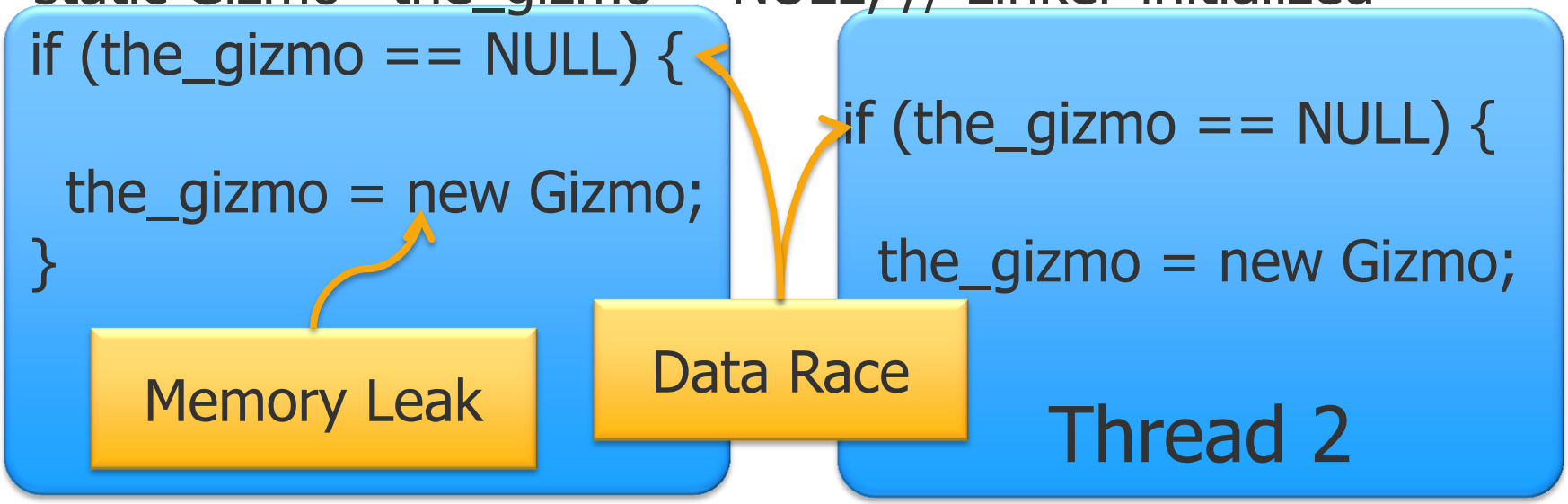
Thread-safe initialization

```
static Gizmo* the_gizmo = new Gizmo;
```

- What does static initialization do?

```
static Gizmo* the_gizmo = NULL; // Linker-initialized
if (the_gizmo == NULL) {

  the_gizmo = new Gizmo;
}
```



- Unprotected static initialization is not thread-safe!
  - — Pointer check and assignments are not done atomically

Thread-safe initialization

```
static Gizmo* the_gizmo = new Gizmo;
```

- What does static initialization do?

```
static Gizmo* the_gizmo = NULL; // Linker-initialized
Lock(&mutex);
if (the_gizmo == NULL) {
  the_gizmo = new Gizmo;
}
Unlock(&mutex);
```

- Now the initialization is thread-safe
- The lock is acquired every time, but the_gizmo is initialized only once!

Double-Checked Locking Pattern

- For all calls except the first one, the_gizmo is not NULL and atomic check-assignment is not needed

```
static Gizmo* the_gizmo = NULL; // Linker-initialized
if (the_gizmo == NULL) {
  Lock(&mutex);
  if (the_gizmo == NULL) {
    the_gizmo = new Gizmo;
  }
 Unlock(&mutex);
}
```

- Now the initialization is thread-safe and the lock is acquired once (may be few times in case of a race)
- One problem – it does not work!

Double-Checked Locking Pattern

the_gizmo = new Gizmo;

- What does initialization really do? This?

```
void* temp = malloc(sizeof(Gizmo));
new (temp) Gizmo;
the_gizmo = temp;
```

the_gizmo == NULL – Lock() and wait

the_gizmo != NULL – just use the_gizmo

Double-Checked Locking Pattern

the_gizmo = new Gizmo;

- What does initialization really do? Or this?

```
void* temp = malloc(sizeof(Gizmo));
the_gizmo = temp;
new (temp) Gizmo;
```

the_gizmo != NULL but object not initialized

- The compiler can implement "new expression" any way it wants
- We can try to force a specific implementation

Double-Checked Locking Pattern

the_gizmo = new Gizmo;

- Make initialization do what we want:

```
void* temp = malloc(sizeof(Gizmo));
new (temp) Gizmo;
the_gizmo = static_cast<Gizmo*>(temp);
```

- Eliminate unnecessary variable – that's what optimizers are for

```
the_gizmo = malloc(sizeof(Gizmo));
new (the_gizmo) Gizmo;
```

- Compiler – 1, programmer – 0.

Double-Checked Locking Pattern

the_gizmo = new Gizmo;

- Make initialization do what we want, harder:

```
volatile void* volatile temp = malloc(sizeof(Gizmo));
new (temp) Gizmo;
the_gizmo = static_cast<Gizmo*>(temp);
```

- Volatile memory accesses cannot be reordered by the compiler – we win.
- This was easy, compiler – 1, programmer – 1.
- "volatile" is a C++-only concept, machine code is not affected
- We probably need some atomic operations here, that's what lock-free programming is all about

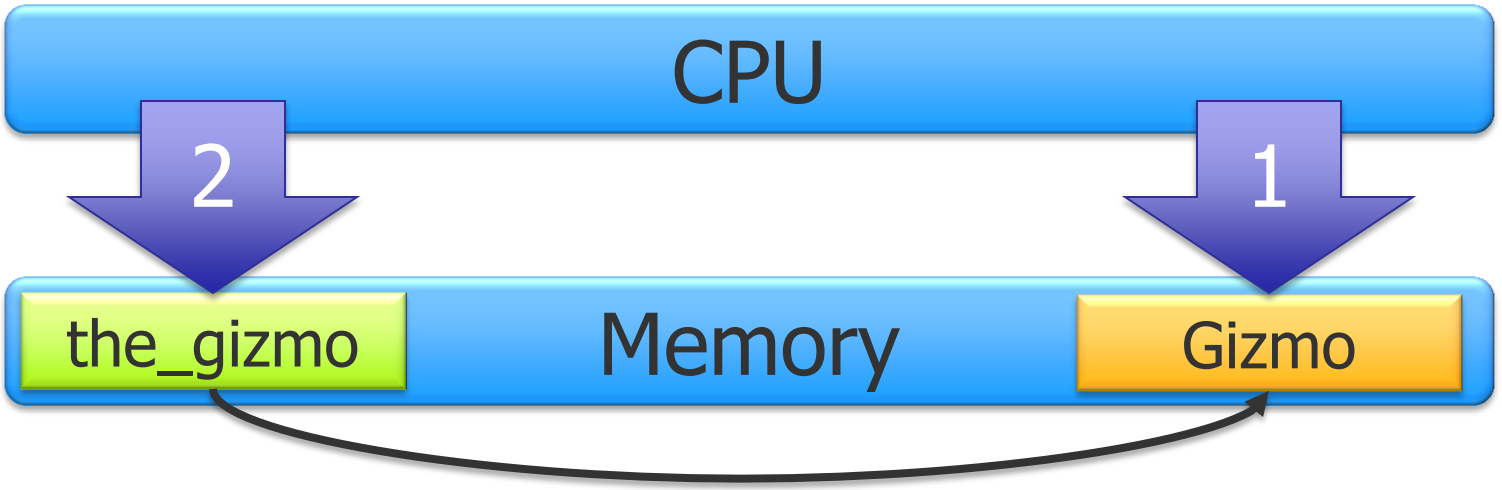
Double-Checked Locking Pattern

```
Atomic<Gizmo*> the_gizmo = NULL;
if (the_gizmo.AtomicLoad() != NULL) {
  Lock();
  if (the_gizmo.NonAtomicLoad() != NULL) {
    Gizmo* temp = new Gizmo;
    the_gizmo.AtomicStore(temp);
  }
  Unlock();
}
```

- Atomic<> is assumed to involve volatile semantics as needed, so the compiler is still prevented from reordering "new Gizmo" and Store()
  - AtomicLoad() is just an atomic load operation, nothing more

Double-Checked Locking Pattern

```
Atomic<Gizmo*> the_gizmo = NULL;
if (the_gizmo.AtomicLoad() != NULL) {
  Lock();
  if (the_gizmo.NonAtomicLoad() != NULL)
    the_gizmo.AtomicStore(new Gizmo);
  Unlock();
}
```

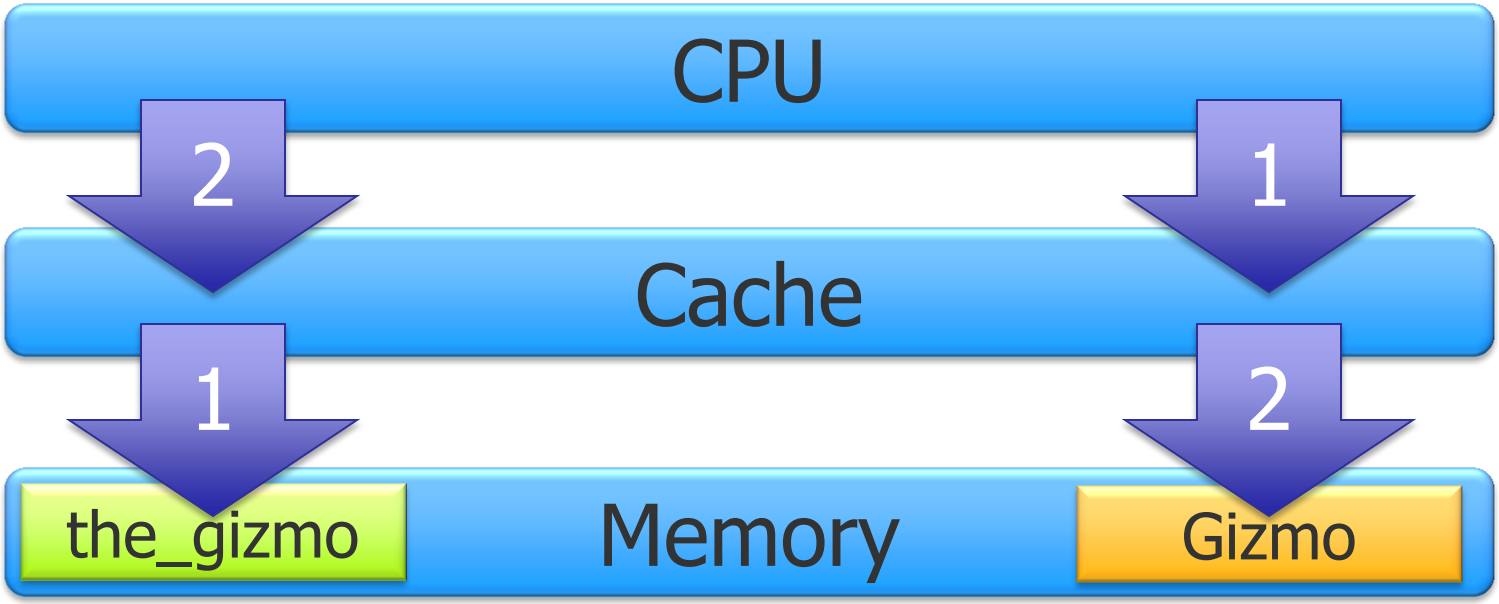


- But that's not how the world works…

Double-Checked Locking Pattern

- Make initialization do what we want, harder:

```
volatile void* volatile temp = malloc(sizeof(Gizmo));
new (temp) Gizmo;
the_gizmo.AtomicStore(static_cast<Gizmo*>(temp));
```

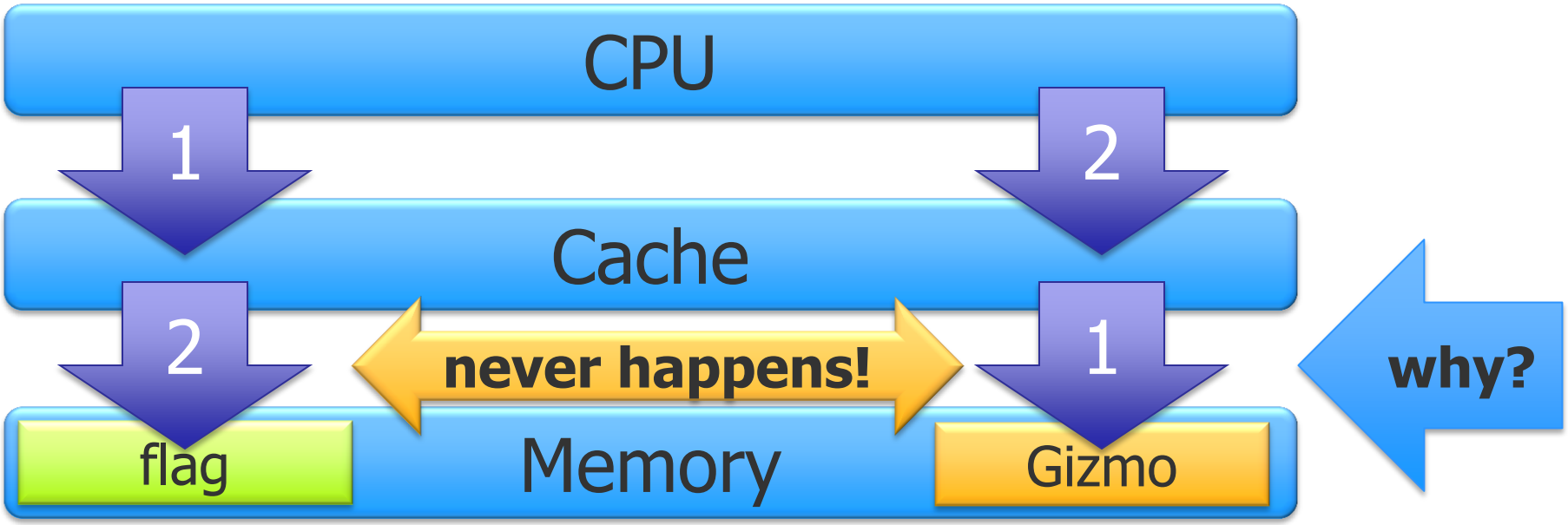


- Hardware – 1, programmer - 0

Back to the Lock

```
Lock(&mutex);
if (the_gizmo == NULL) {
  the_gizmo = new Gizmo;
}
```

- Do locks really work?



Memory Barriers

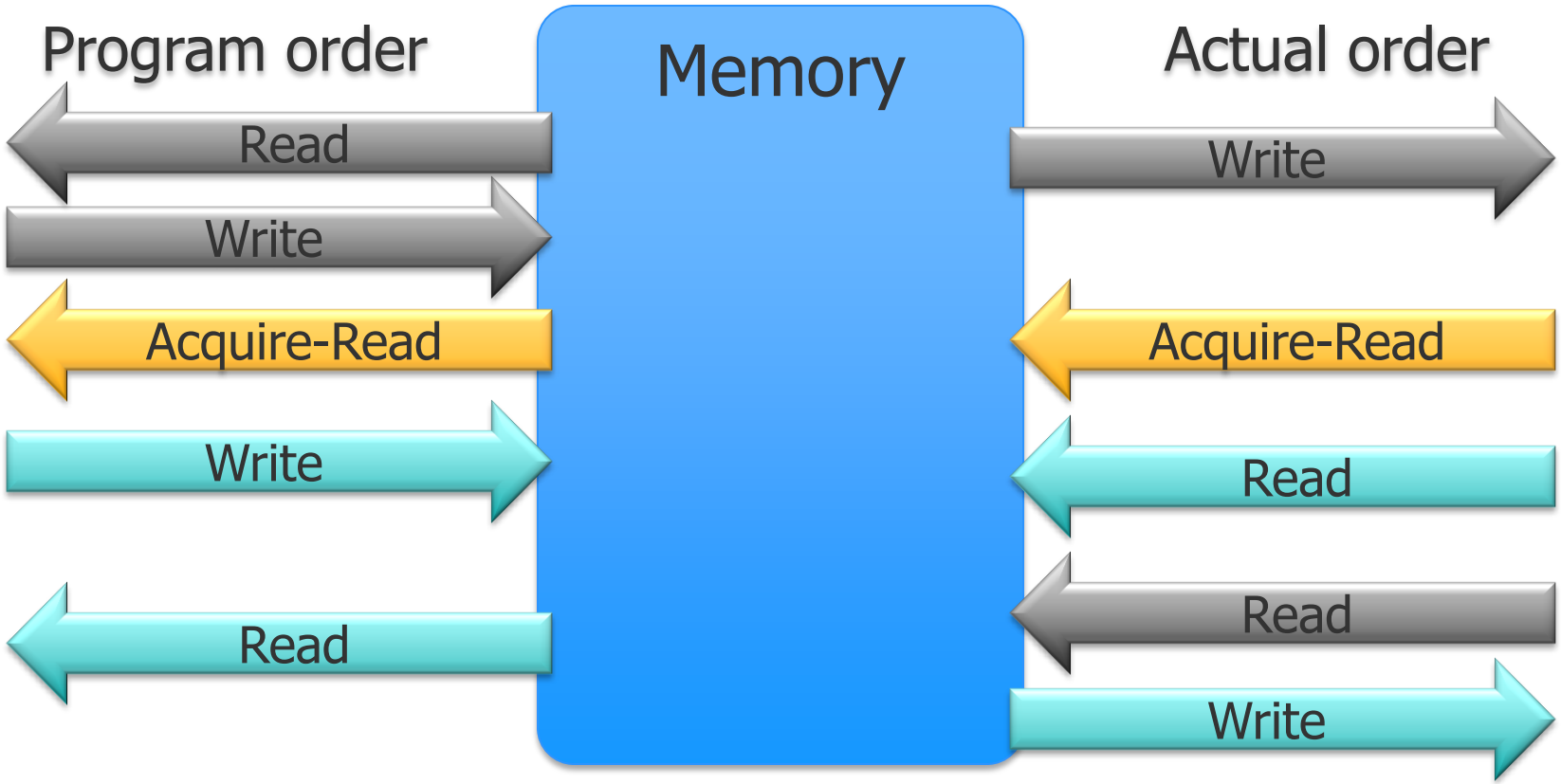
- Synchronization of data access is not possible if we cannot control the order of memory accesses
  - This is global control, across all CPUs
- Such control is provided by memory barriers
  - Memory barriers are implemented by the hardware
  - Memory barriers are invoked through processor-specific instructions (or modifiers on other instructions)
  - Barriers are usually "attributes" on read or write operations, ensuring the specified order of reads and writes
  - ~~There are no portable memory barriers~~
  - C++11 provides portable memory barriers!

Can you handle the truth?

- What follows is a simplified description of memory barriers
- If you want to know all details about memory barriers and C++, go to Michael Wong's talk (Tuesday, 2pm)
- If you want to know really all details about memory barriers, also go to to Paul McKenney's talk (Tuesday, 3:15pm)

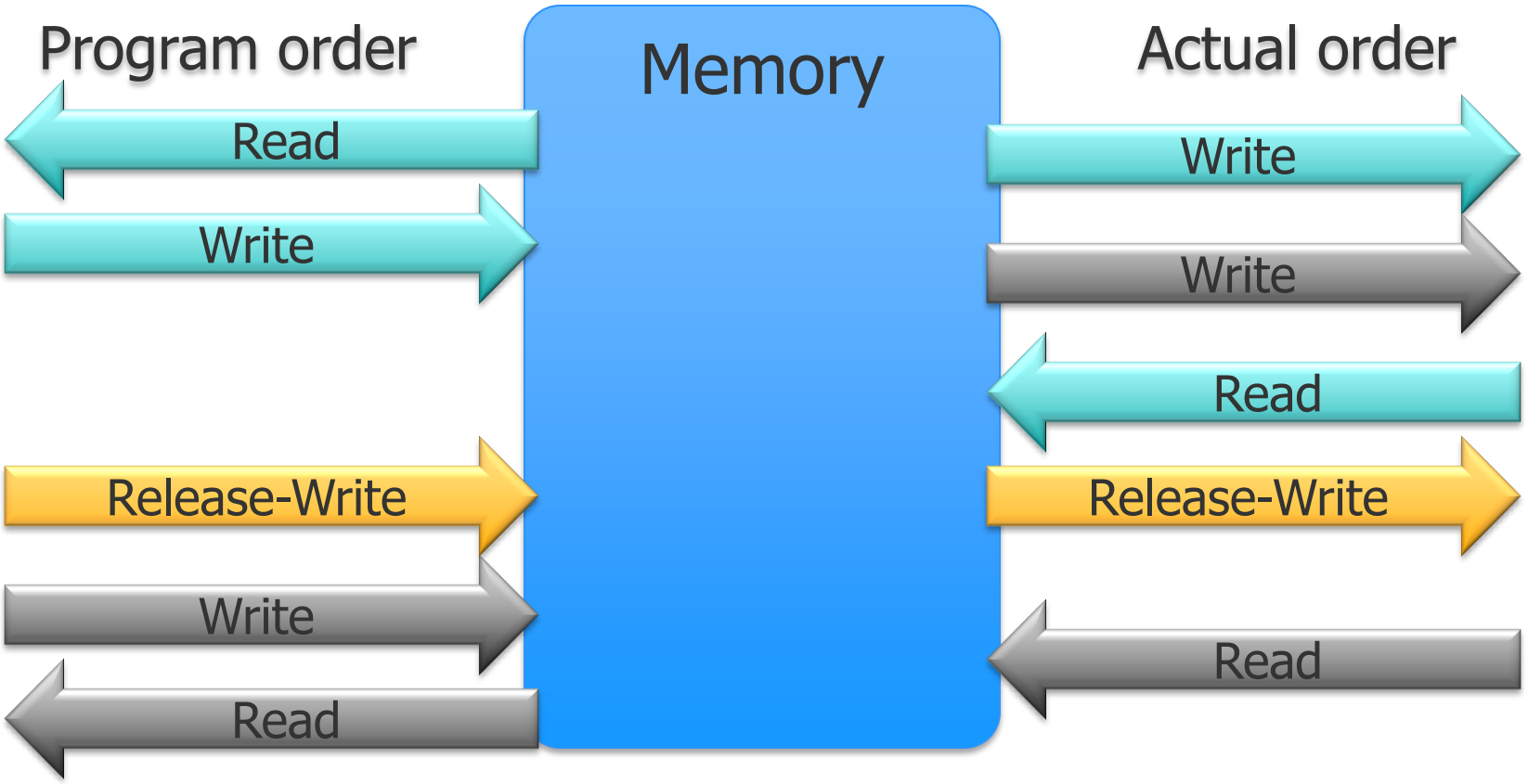
Acquire and Release Memory Barriers

- Acquire barrier guarantees that all memory operations scheduled after the barrier in the program order become visible after the barrier

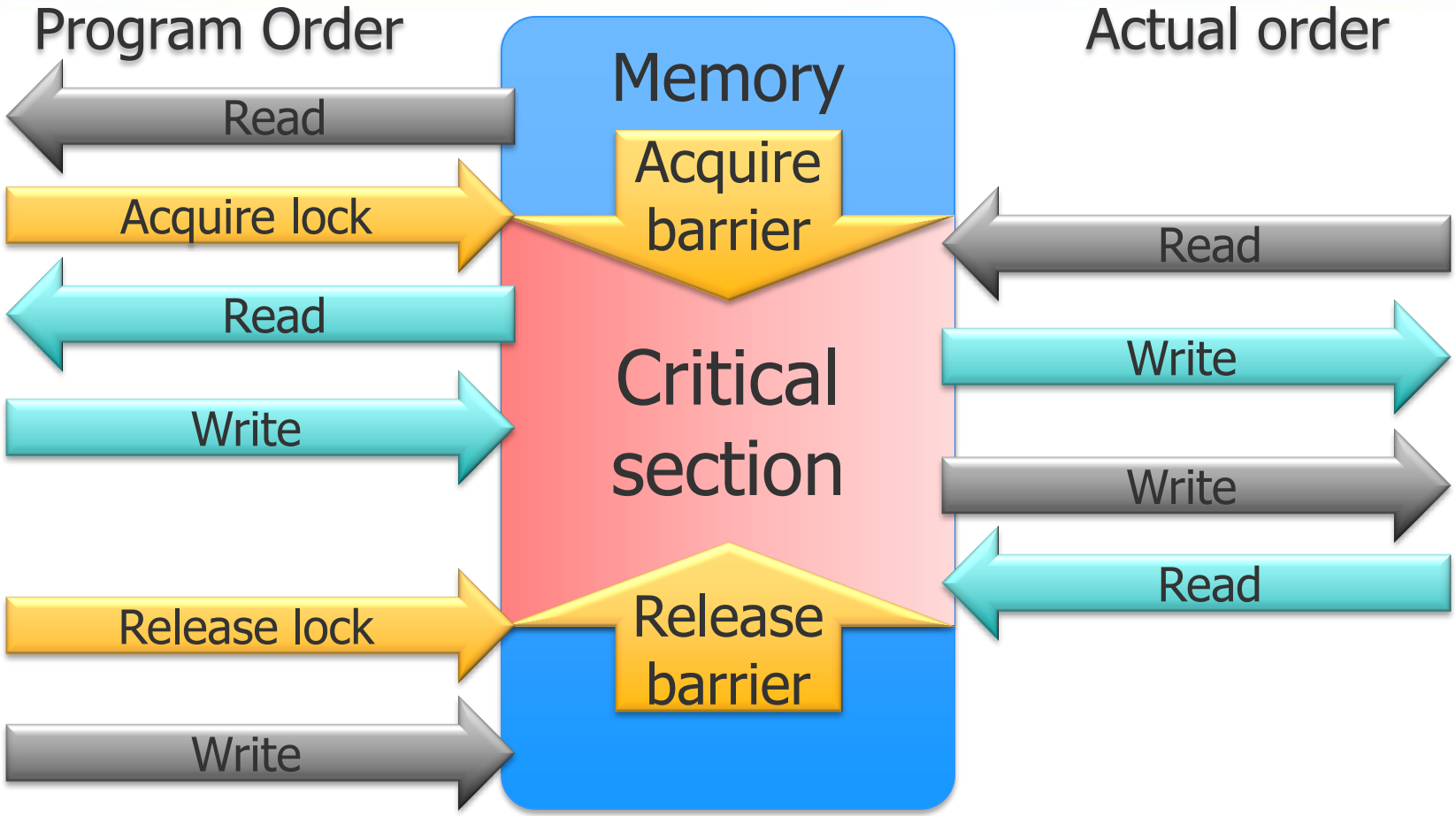


Acquire and Release Memory Barriers

- Release barrier guarantees that all memory operations scheduled before the barrier in the program order become visible before the barrier



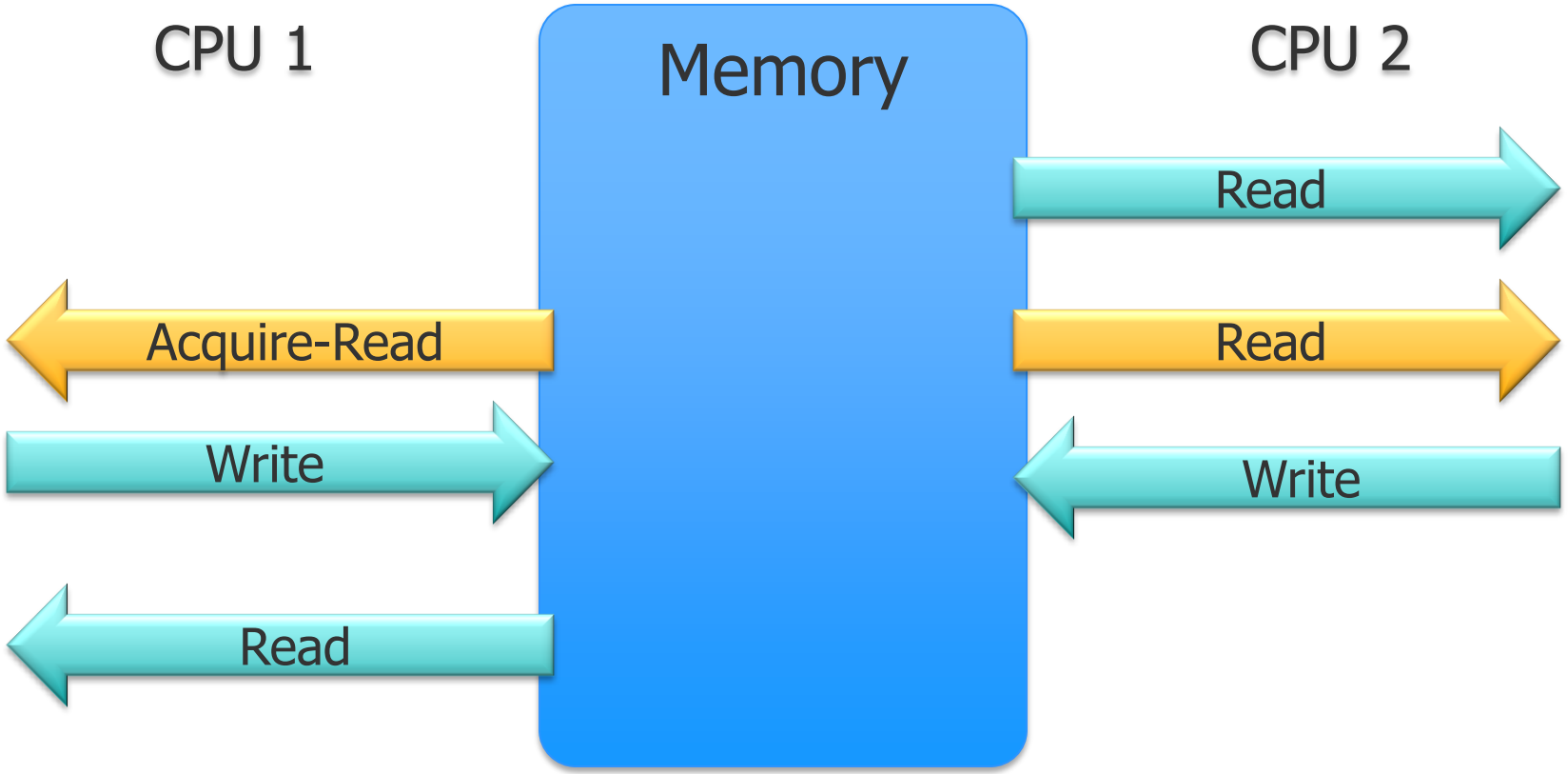
Memory Barriers and Locks



- Memory accesses can move into the critical section but never leave it

Memory Barriers on Many-Core Systems

- Memory barrier guarantees apply only to CPUs that issue memory barrier instructions!



Memory Barriers and Order Guarantees

- There are other types of memory barriers, they vary for different hardware platforms
- Different hardware platforms provide different order guarantees (memory model):
  - Read – Read (can two reads be reordered)?
  - Read – Write
  - Write – Read
  - Write – Write
  - Dependent reads (int* p; can read of p and read of *p be reordered?)
  - Read – Atomic Read
  - Write – Instruction Fetch (prevents self-modifying code)
- X86 Memory Ordering
  - most of the time

Double-Checked Locking Pattern that works

- For all calls except the first one, the_gizmo is not NULL and atomic check-assignment is not needed

```
static Atomic<Gizmo*> the_gizmo = NULL;
if (the_gizmo.AcquireLoad() == NULL) {
  Lock(&mutex);
  if (the_gizmo.NonAtomicLoad() == NULL) {
    the_gizmo.ReleaseStore(new Gizmo);
  }
 Unlock(&mutex);
}
```

- Lock-free programming is 90% about memory visibility and order of memory accesses
- Atomic instructions are the easy part

Take-home points

- Lock-free programming is hard, be sure you need it
- Never guess about performance
- Lock-free programming is mostly about memory order

After C++11, DCLP is no longer needed

- In C++11 singleton can be initialized like this:

```
Gizmo* theGizmo() {
  static Gizmo gizmo;          // thread-safe in C++11
  return &gizmo;
}
```

DCLP reborn

- What general problem does DCLP solve?
- We have an exceptional situation that rarely happens
- Handling the exceptional situation is not thread-safe
  - Must be done under lock
- Accurate test for exception must be atomic with handling the exceptional situation
  - Must be done under the same lock
- There is a fast non-locking test that is usually correct
  - No "false success" – if fast test passes there is no problem
  - "False failure" is possible but rare
- In general, there is no requirement that the exceptional situation, once handled, never reoccurs

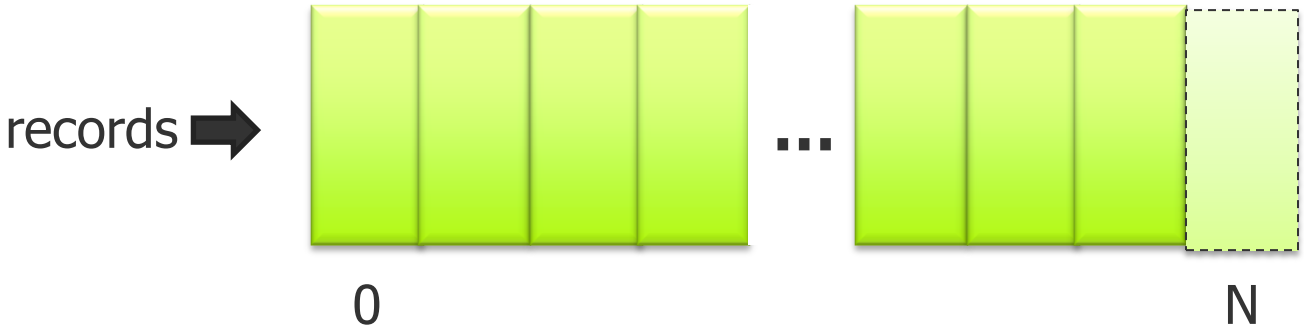
DCLP reborn

- DCLP in general:

```
if (testForProblem_Fast()) {
  Lock(&mutex);
  if (testForProblem_Exact()) {
    HandleProblem();
  }
 Unlock(&mutex);
}
```

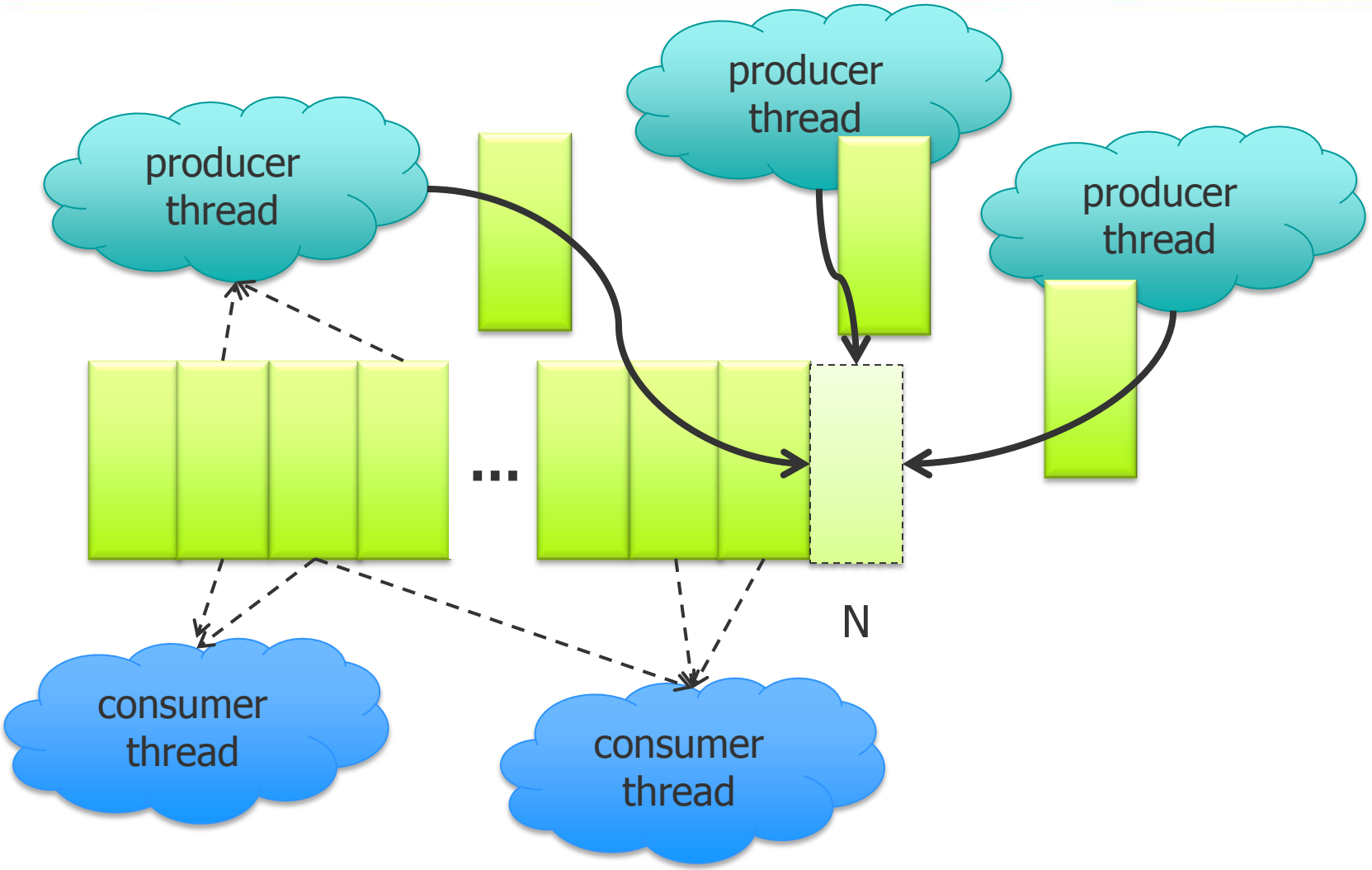
- This assumes that the exceptional situation does not affect anything that was valid before the test
  - out-of-memory handling: anything that was allocated is fine but we can't get any new memory until we deal with the problem
- This will come very useful later

Lock-Free producer queue



- Slots 0-(N-1) are filled ("done")
  - Consumer[s] (readers) are reading records 0-N and using them to compute results
- New record goes into slot N
  - Producer[s] (writers) generate new records using old records and new data
- There are multiple reader threads and writer threads
- Nothing gets deleted

Lock-Free producer queue



Back to basics

- Once upon a time there were these things called “locks”

- Producer:

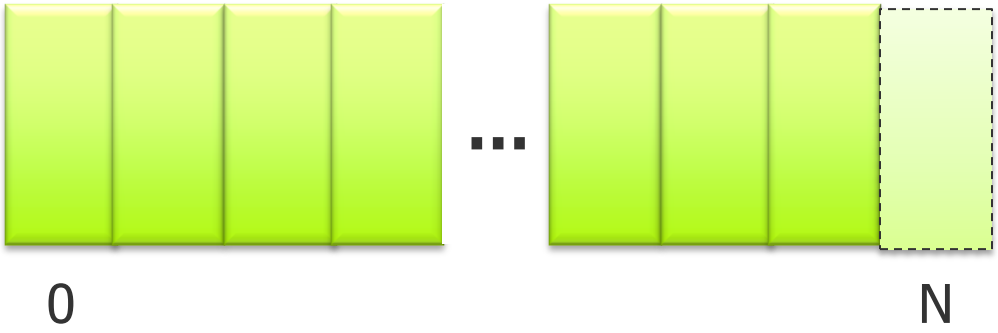
  Lock(); new(records+N) Record(…); ++N; Unlock();

- Consumer:

  Lock(); for(size_t i = 0; i < N; ++i) Consume(records[i]); Unlock();

Lock-Free producer queue

- Lock-free is the same thing but without locks



- Producer:

```
new(records+N) Record(…); ++N;
```

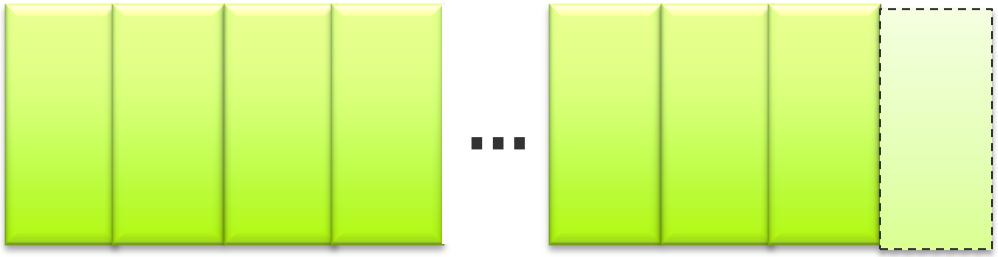
- Consumer:

```
for(size_t i = 0; i < N; ++i) Consume(records[i]);
```

- How many data races can you find?

Lock-Free producer queue with atomics

- Lock-free programming needs atomic variables



- Let's make N atomic:

```
Atomic<size_t> N;
```

- Producer:

```
new(records+N) Record(…); ++N;
```

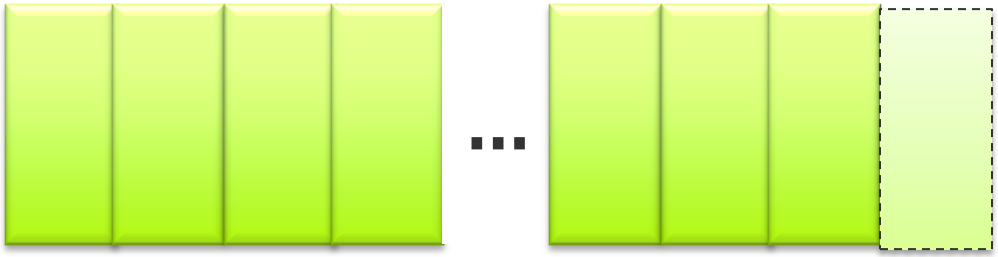
- Consumer:

```
for(size_t i = 0; i < N; ++i) Consume(records[i]);
```

- How many data races can you find?

Lock-Free producer queue with atomics

- Lock-free programming needs atomic variables



- Let's make N atomic:

```
Atomic<size_t> N;
```

- Producer:

```
new(records+N) Record(…); ++N;
```

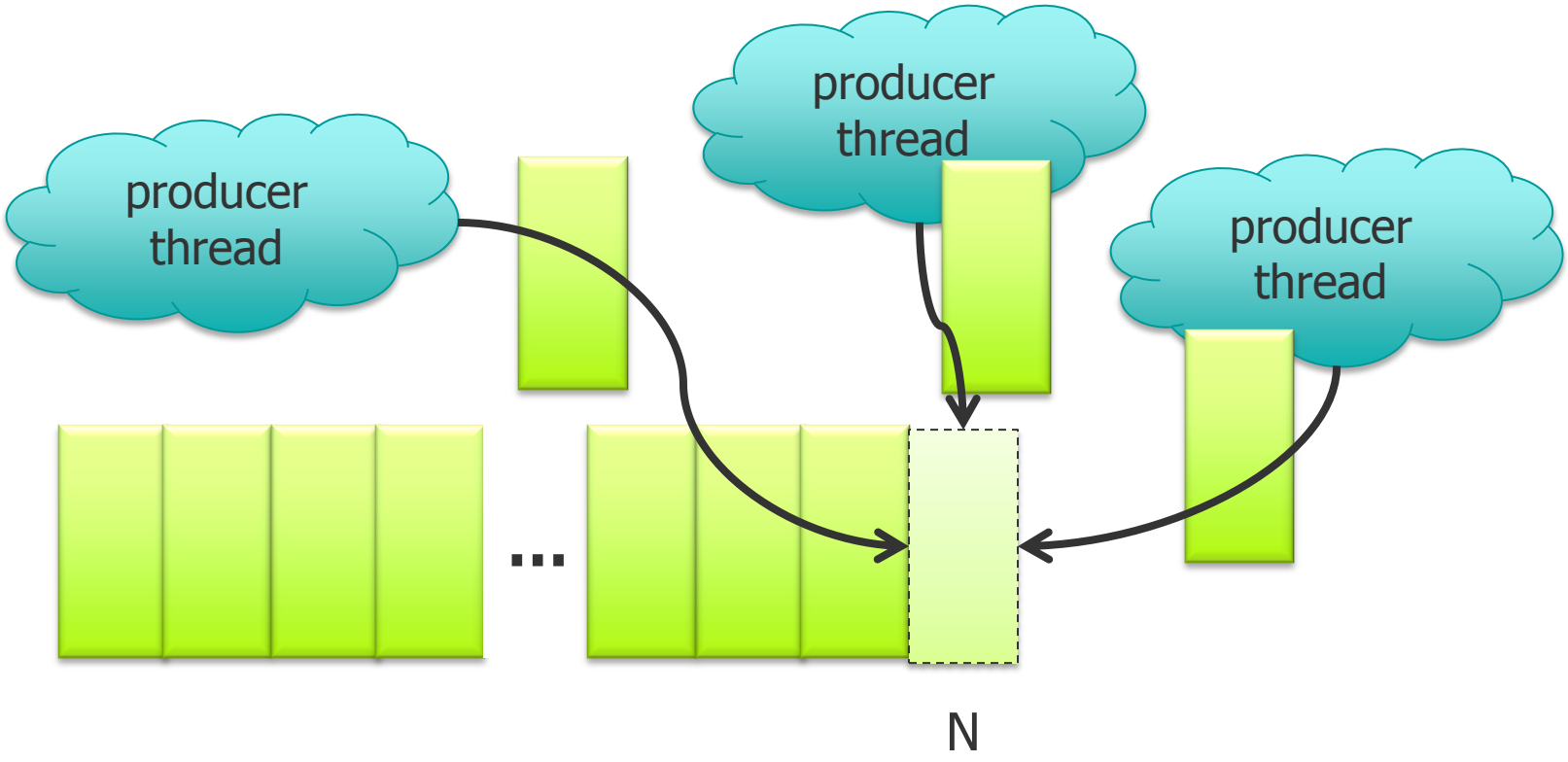
- Consumer:

```
for(size_t i = 0; i < N; ++i) Consume(records[i]);
```

- Concurrent programming in general, but lock-free in particular, requires thinking in terms of transactions

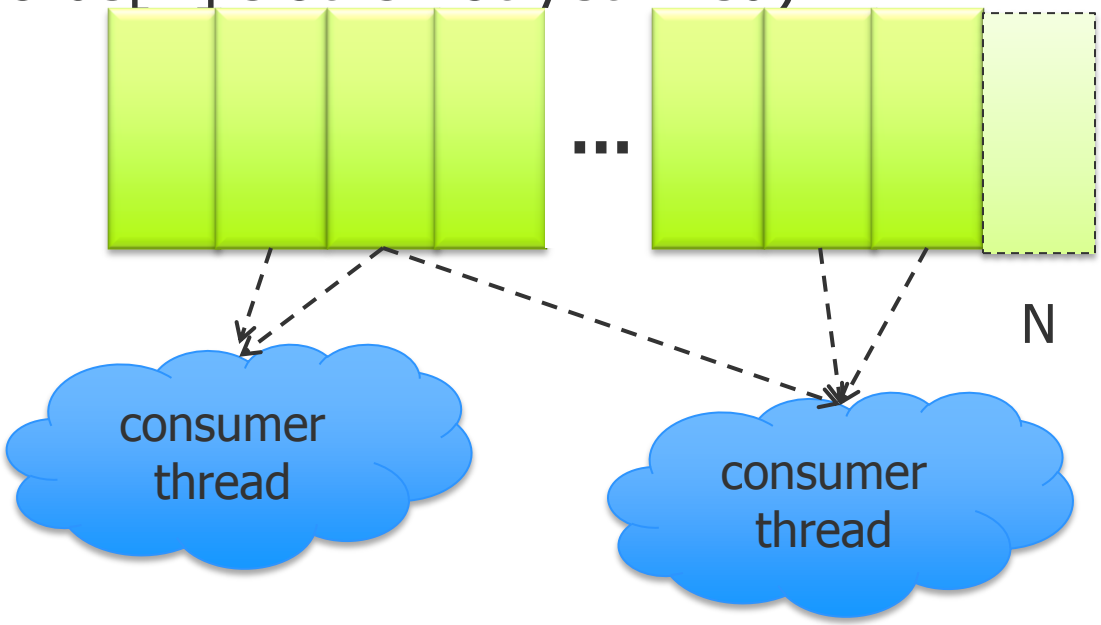
Lock-Free producer queue with atomics

- Two sets of problems:
- Producers race for the slot to insert new records



Lock-Free producer queue with atomics

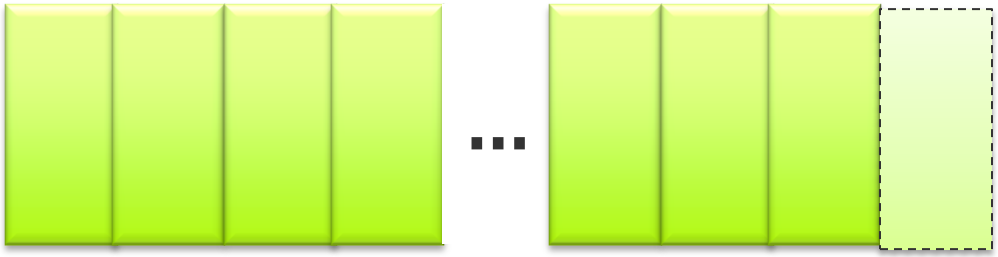
- Two sets of problems:
- Consumers may attempt to read records that are not finished (there is time when N is incremented but records[N] slot is not yet filled)



- Producers act as consumers if they read old records

Lock-Free producer queue with atomics

- Let's handle producers first:



- Let's make N atomic:

```
Atomic<size_t> N;
```

- Producer:

```
new(records+N) Record(…); ++N;
```

- Still has a race…

Compare-and-swap (CAS) – the tool of lock-free programming

- All lock-free algorithms can be implemented using CAS

```
Atomic<size_t> N;
size_t N1 = N.CAS(oldN, newN);
```

- Replace N with newN if and only if N is still equal to oldN
- Return current value of N (before assigning newN)

```
if (N1 == newN) {…}          // CAS succeeded, N is now newN
else {…}                     // CAS failed, N is no longer oldN
```

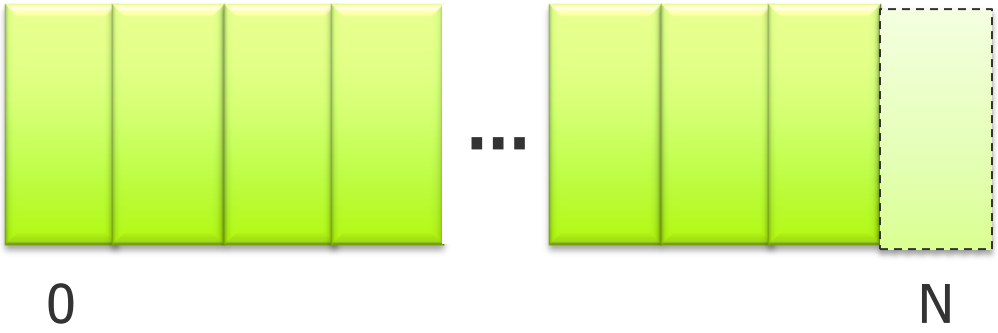
- Standard CAS idiom:

```
size_t oldN;
do {
    oldN = N;
    … compute results using oldN …
} while (N.CAS(oldN, oldN+1) != oldN);
```

- If N changed during computation, start all over

Compare-and-swap (CAS) – the tool of lock-free programming

- All lock-free algorithms can be implemented using CAS



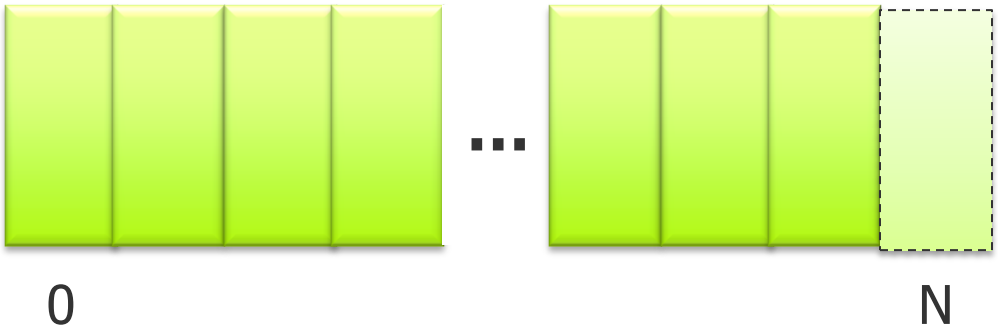

```
Atomic<size_t> N;
size_t oldN; Record R;
do {
    oldN = N;
    BuildRecord(R, records[0], …, records[oldN-1]);
} while (N.CAS(oldN, oldN+1) != oldN);
```

- Now slot [oldN] is ours!

```
new(records+oldN+1) Record(R);
```

What do we have so far?

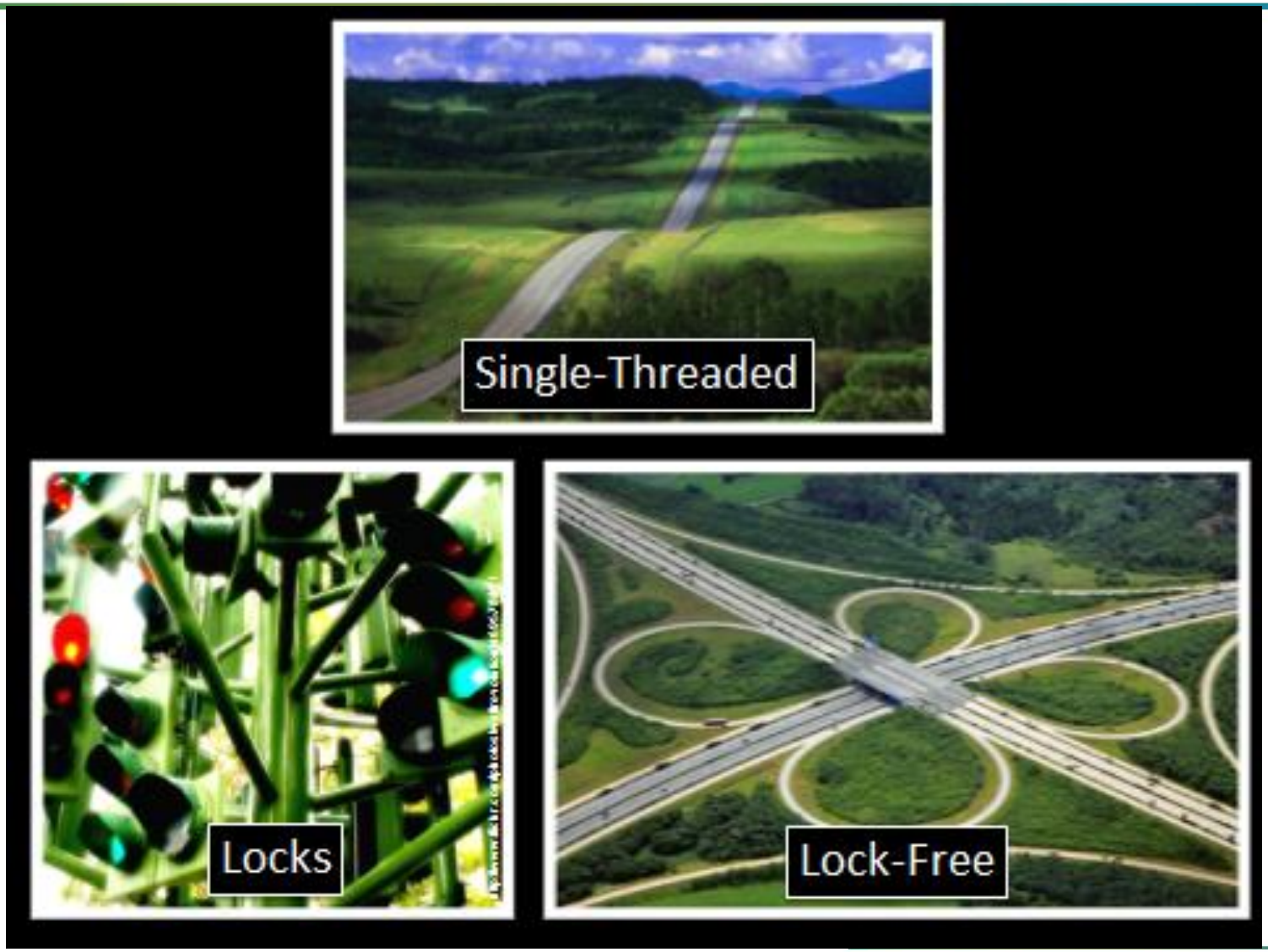
- We simplified the problem quite a bit
  - Perhaps too much?
- A "black hole" queue – records come in, they don't come out



- Producers can write to the array safely but there is no way to know if records[i] is ready to be read
- Also, this looks a lot like a lock…

```
do {…} while (N.CAS(oldN, oldN+1) != oldN);
```

Herb Sutter makes a very wise analogy



Lock-free vs locks

```
size_t oldN;
do {
    oldN = N;
    … compute results using oldN …
} while (N.CAS(oldN, oldN+1) != oldN);
```

- Isn't this a lot like a lock? All threads read N, one gets to proceed, the rest are held back? What's the difference?
- Cars can't enter the same location of the freeway at the same time, so what's the difference vs traffic light?
  - Highway interchange has higher throughput
- Is this all about performance?
- To a degree – lock is coarse granularity compared to CAS
- Is this all the difference?

Lock-Free lock?

```
class SpinLock {
  Atomic<int> lock_;
  public:
  SpinLock() : lock_(0) {}
  void Lock() { while (lock_.AtomicExchange(1) == 1) {} }
  void Unlock() { lock_.AtomicExchange(0); }
};
```

- This is a spin lock
  - Blocked threads are busy-waiting
- L.AtomicExchange(newL) assigns newL to L and returns old L, atomically
  - AtomicExchange is simpler and potentially faster than CAS but can be implemented using CAS

Deeper understanding of lock-free programs

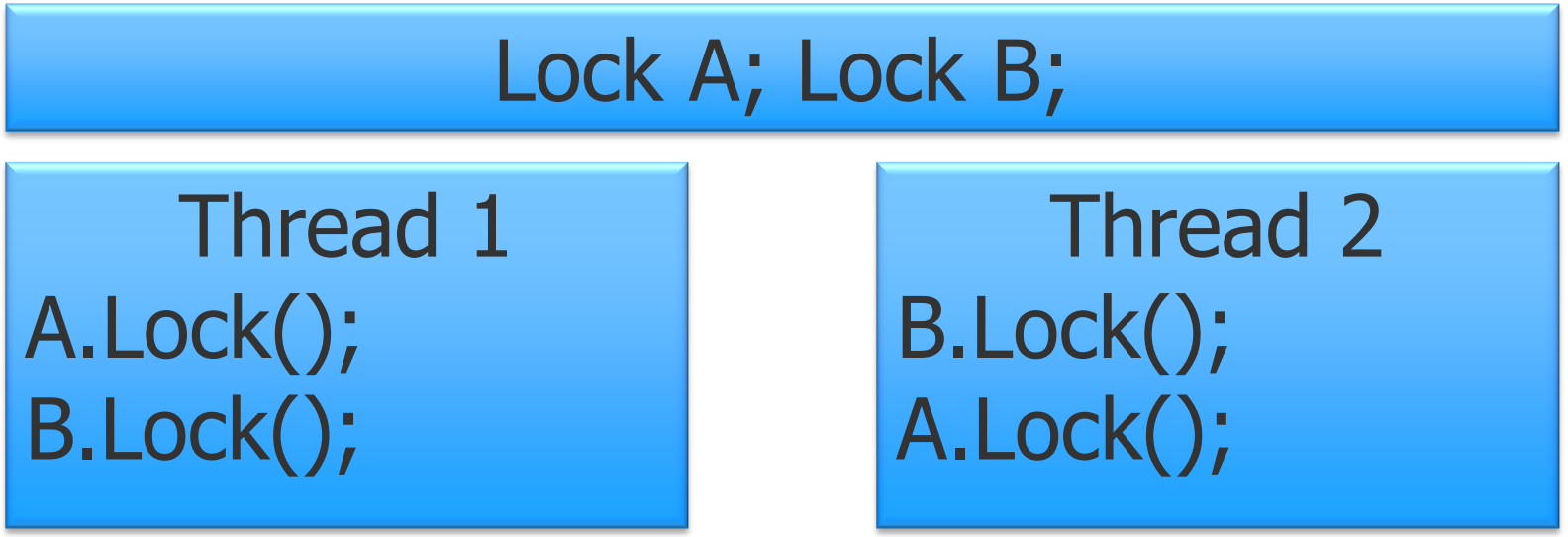
- Wait-free programs: each thread will complete its task in finite number of steps (finite amount of time) no matter what other threads are doing
  - At all times, all threads make progress toward the final result
- Lock-free programs: programs without locks (duh!); at least one thread makes progress no matter what other threads are doing
- Obstruction-free program: thread makes progress as long as no other thread accesses shared data at the same time
- Lock-based programs are not guaranteed to make progress toward the result in all cases
  - In particular, consider what would happen if the thread holding the lock was suspended by the OS?

Problems With Locks

- (Not a problem) Locks are simpler than any alternative
  - Most programs are not lock-free
  - Know problems with locks not just to avoid locks but also to avoid problems when using locks
- Deadlock
- Livelock
- Priority Inversion
- Convoying
- Reentrance and Signal safety
- Shared memory between processes
- Preemption tolerance
- Performance

Problems With Locks

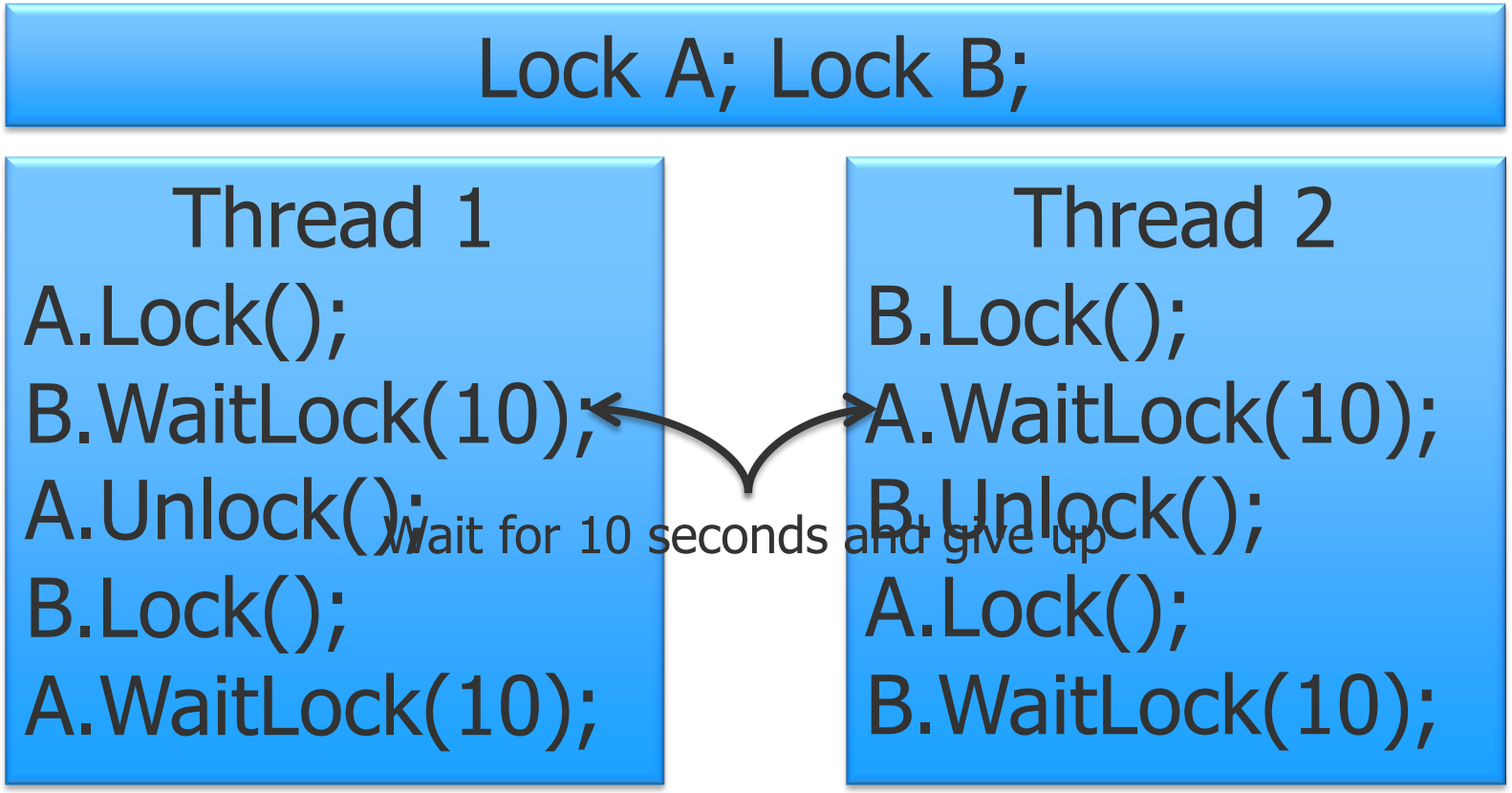
- Deadlock
  - All threads are waiting to access resources (blocked on locks)
  - Locks are held by threads waiting for other locks



- Fundamental problem of locks – they are not composable!
  - Consider general case: system has locks A..Z, thread 1 wants to lock A,B,C,D, thread 2 wants to lock D,E,F, thread 3 wants F,G,A

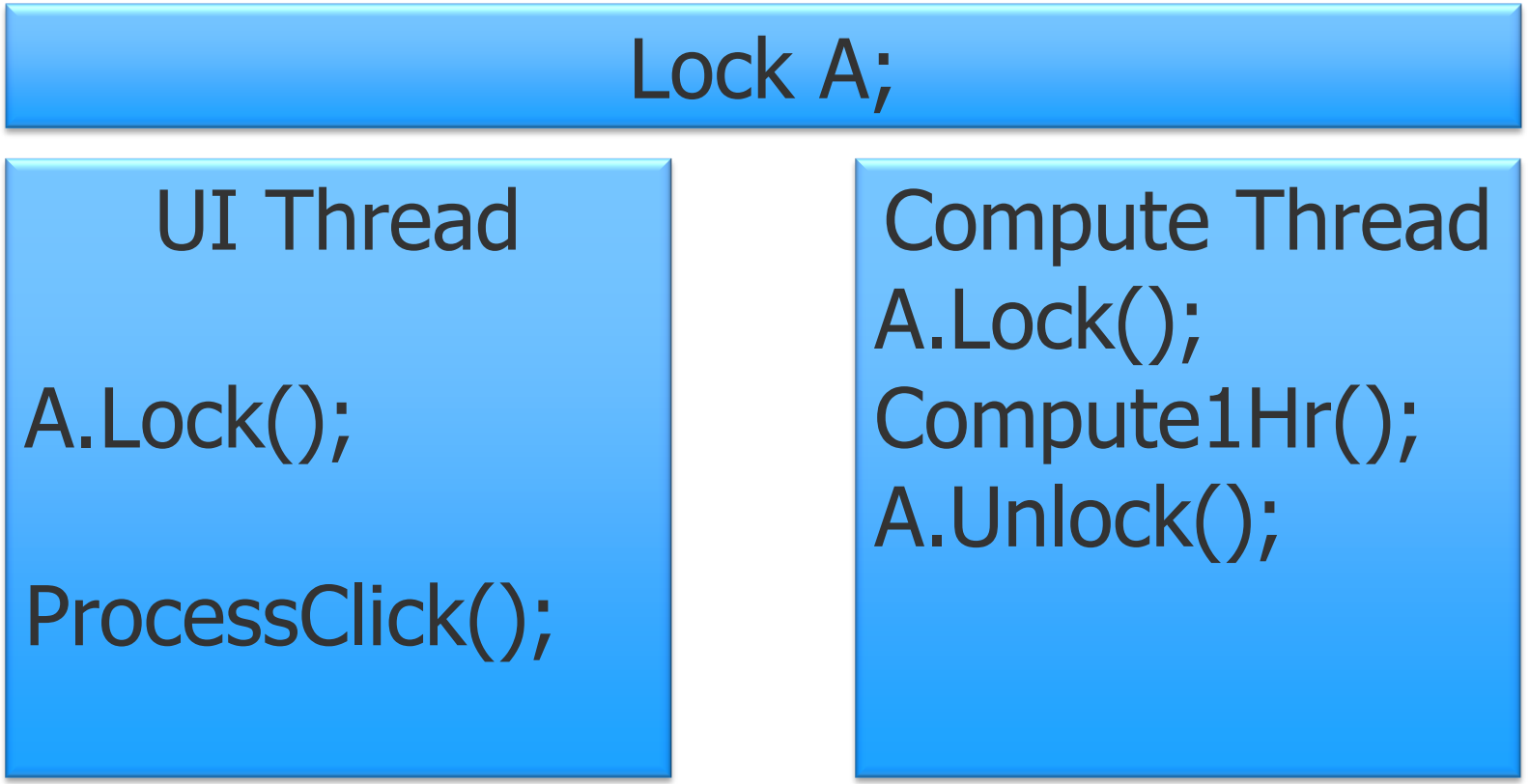
Problems With Locks

- Livelock
  - All threads are doing something but the whole system is not making overall progress



Problems With Locks

- Priority Inversion
  - High-priority thread is waiting for a lock held by a low-priority thread (in real-time or interactive systems)



Problems With Locks

- Convoying
  - One solution to a deadlock is to acquire locks in the same order
  - This may slow down fast threads

```
Lock A; Lock B; Lock C;
```

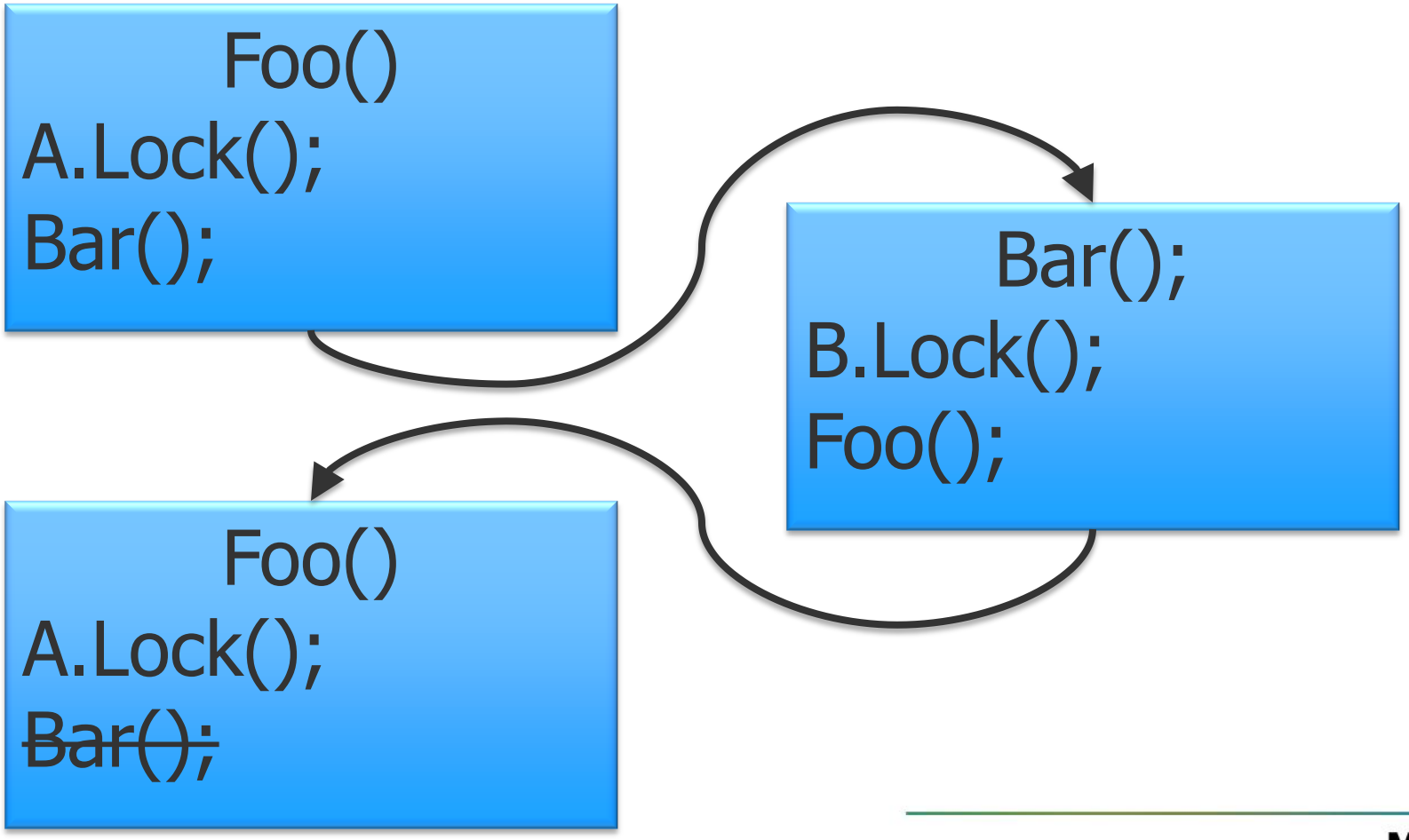
```
Slow Thread
A.Lock();
Compute1Hr();
A.Unlock();B.Lock();
Compute1Hr();
B.Unlock();C.Lock();
```

```
Fast Thread

A.Lock();
Compute1Min();
A.Unlock();B.Lock();
Compute1Min();
```

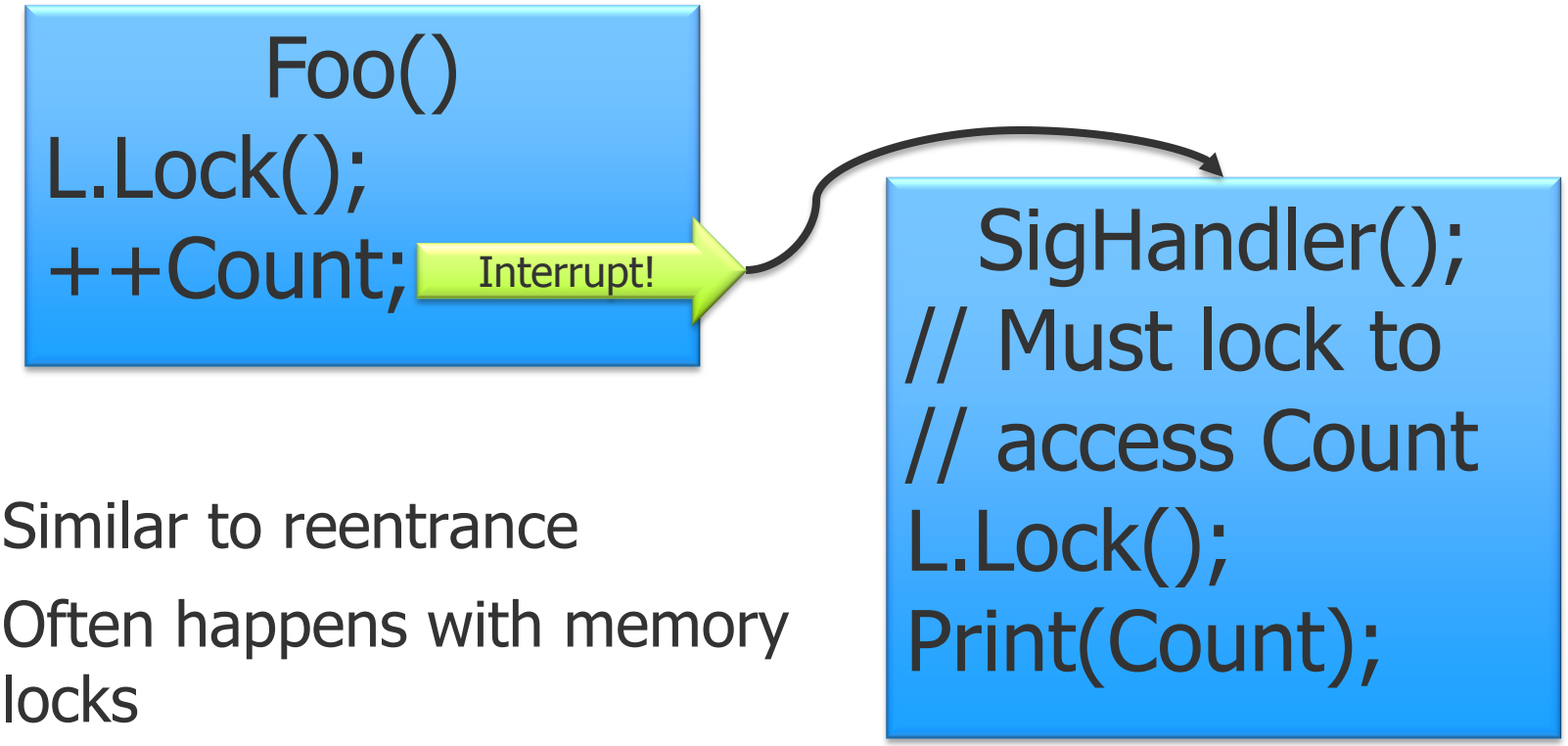
Problems With Locks

- Reentrance
  - Reentrant function may call itself (often indirectly)



Problems With Locks

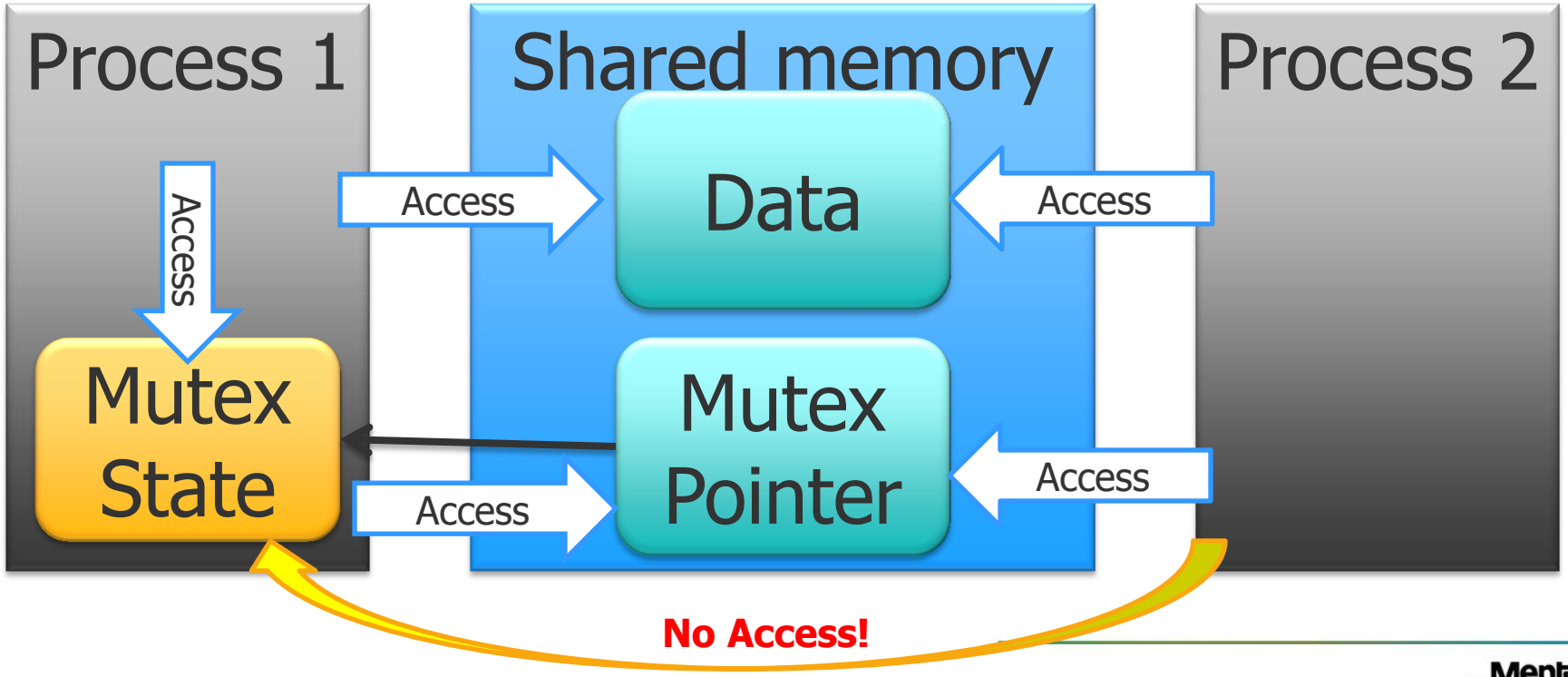
- Signal handling
  - Signal handlers often need to access shared data
  - Signal handlers are asynchronous – you never know which locks might be held when a signal happens



- Similar to reentrance
- Often happens with memory locks

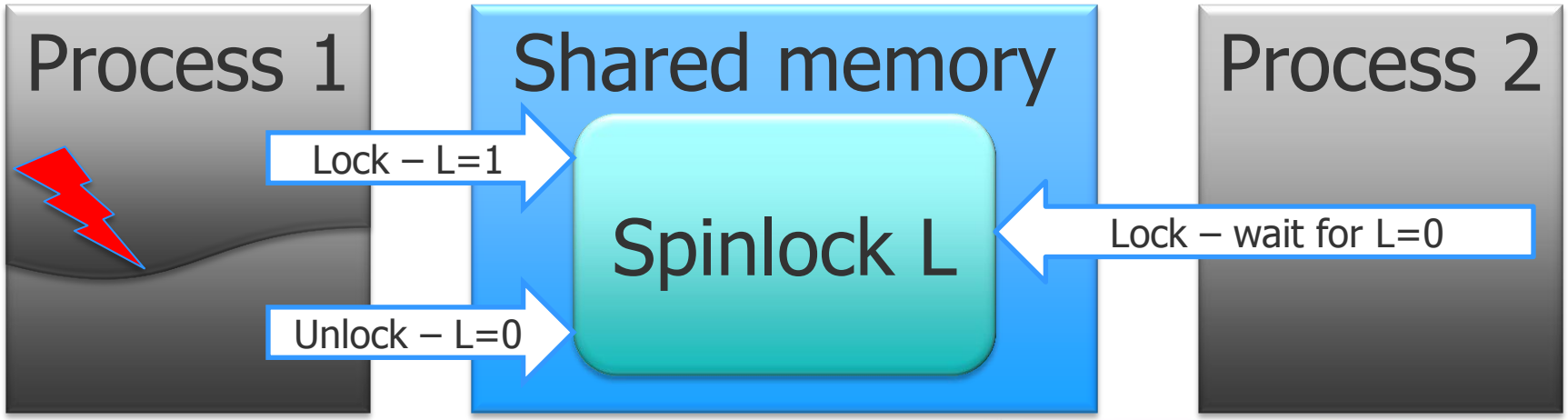
Problems With Locks

- Shared memory used to exchange data between processes (not threads).
- Mutex allocates additional memory and cannot be placed into shared memory



Problems With Locks

- Shared memory used to exchange data between processes (not threads).
- Mutex allocates additional memory and cannot be placed into shared memory.
  - Some OSes offer "shared mutex"
- Spinlock can be placed into shared memory (it's a flag)
- What happens if the process holding the lock dies?



Problems With Locks

- Performance
- It's listed last for a reason
- Performance of a well-written lock program may be no different from that of a lock-free program
- Lock-based program ≠ mutex-based program!
- We will come back to this later

Take-home points

- Lock-free programming is hard, be sure you need it
- Never guess about performance
- Lock-free programming is mostly about memory order
- Know potential dangers of locks and how to avoid them

Practical problems with locks

- In practice a lot of these problems may not matter
  - I write HPC code, I don't have priority threads, I just want the answer as fast as possible.
  - Sure, the thread holding the lock can get suspended, but how often does it really happen?
  - If a lock protects an atomic transaction with no user code inside, this cannot deadlock or livelock.
  - I usually know what can and cannot be in shared memory
- In practice, the main motivation to ret rid of locks is performance
  - Locks can make your program slow
  - Lock-free does not always make it faster
- In some special cases lock-free program may be simpler!

Practical problems with locks

- In practice, the main motivation to ret rid of locks is performance
- Getting rid of locks does not automatically make the program faster
- How to atomically increment two 64-bit integers?
  - Some CPUs have 128-bit CAS (DCAS)
  - There is a really clever algorithm with CAS and a 3rd integer
  - Do it under a spinlock or use atomic exchange and busy-wait
  - On a wide range of x86 systems, even under heavy congestion spinlock is faster than CAS for up to 40-64 cores and always faster than DCAS
- The fastest is to not atomically increment two 64-bit integers

Application vs Library programming

- How to [not] atomically increment two 64-bit integers?
- Do you need to?
  - Can you fit both into one 8-byte word?
  - For counts, is there a practical limit? Can one be imposed?
  - How often is the limit exceeded, can we handle it specially?
  - For a given implementation, are there any bits we can steal?
    - Pointers aligned on 8 bytes have 3 unused bits (library may have to assume char*, in our application we may commit to long*)
    - Pointers on x86 are actually 48-bit wide and have 16 unused bits
- Complexity of lock-free programs increases dramatically as the number of shared variables grows
- In many applications, restrictions can be accepted to pack more data into 64 bits
- These restrictions are usually on the data

Practical Problems with Lock-Free Programs (and practical solutions)

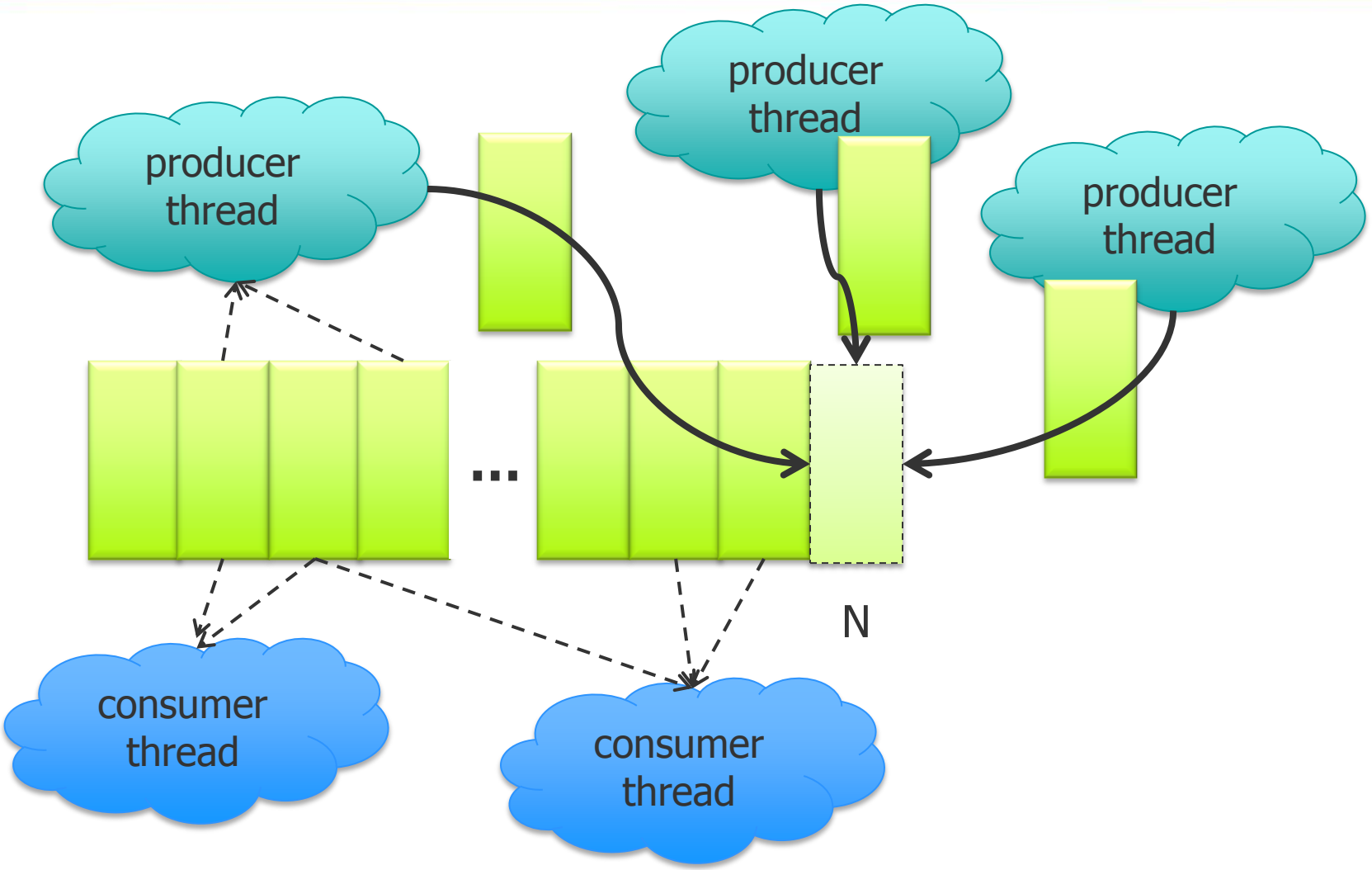
- Wait-Free and Lock-Free programs do not suffer from deadlocks, livelocks, priority-inversion or convoying; they can be reentrant and signal-safe; they can support shared memory.
- Lock-Free and Wait-Free algorithms are usually very hard to write, understand, and maintain.

1. minimize data sharing at the algorithm level.
2. use spinlock (or equivalent) for accessing shared data.
3. avoid designing lock-free algorithms, design lock-free data structures, it's easier and usually is what you want.
4. design only what you need, avoid generic designs
   — Does not work if you are writing STL

Take-home points

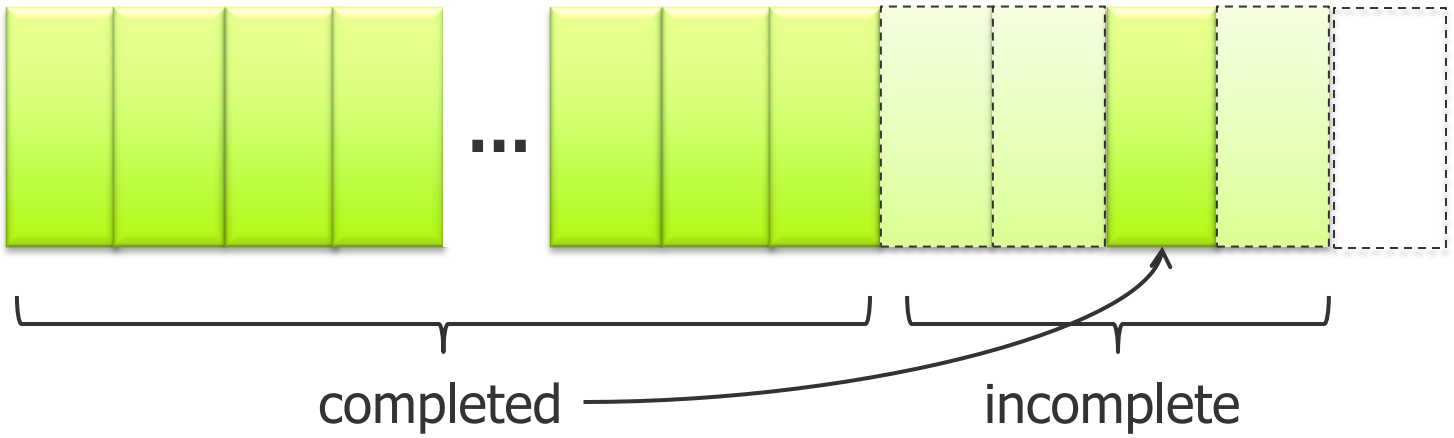
- Lock-free programming is hard, be sure you need it
- Never guess about performance
- Lock-free programming is mostly about memory order
- Know potential dangers of locks and how to avoid them
- Design lock-free data structures

Lock-Free producer queue



Lock-Free producer queue

- We have half of the problem solved: allocating next slot to producers
- We still don't know when records[i] is safe to read



- May be we can mark records when they are ready?

What's in the record?

- Simplest case – record is a number (unsigned long)
- May be 0 is not valid? Or ULONG_MAX?
  - Not an option for a library designer
  - Quite possible in a specific application
- Now we have a solution:
  - Make sure memory is initialized to 0
  - After producers takes control of the slot N, atomically write non-zero value into it
  - Consumers read records from 0 to N, stop at the first zero value

Lock-free producer queue of numbers

```
Atomic<size_t> N = 0;
Atomic<unsigned long> records[maxN]; // Initialized to 0
```

- Producer:

```
size_t myN = N.ReleaseAtomicIncrement(1); // Could use CAS too
unsigned long x = ComputeNewX(); // Can do what consumers do
records[myN].NoBarrierStore(x);
```

- Consumer:

```
size_t currentN = N.AcquireLoad(); // Safe to use at least records[N-1]
for (size_t i = 0; i < currentN; ++i) {
  unsigned long x = records[i].NoBarrierLoad();
  if (x == 0) break; // Not ready yet
  ProcessData(x);
}
```

- Simple, easy to reason about and understand

What's in the record?

- Let's get real – how often to do get to store just numbers

```
class Record {
  size_t count_;
  Data data_[maxCount];
    …
};
```

- How would we know if this is ready to read?
- When data_ is ready, count_ becomes non-zero!
  - As long as we can guarantee the memory order
- We've seen this before:
  1. Make a number non-zero when it's ready
  2. Make sure the pointer (count, index, etc) is written after data

Lock-free producer queue of numbers

```
Atomic<size_t> N = 0;
class Record {
  Atomic<size_t> count_;
  Record(…) : {
    InitializeData();
    count_.ReleaseStore(n);                    // Must be done LAST!
  }
  bool ready() const { return count_.AcquireLoad(); }
};
Atomic<unsigned long> records[maxN]; // Initialized to 0 for count_
```

- Producer:

```
size_t myN = N.ReleaseAtomicIncrement(1);
new (records + myN) Record(…);      // setting count_ "publishes" record
```

Lock-free producer queue of numbers

```
Atomic<size_t> N = 0;
class Record {
  Atomic<size_t> count_;
  Record(…) : {
    InitializeData();
    count_.ReleaseStore(n);                    // Must be done LAST!
  }
  bool ready() const { return count_.AcquireLoad(); }
};
Atomic<unsigned long> records[maxN]; // Initialized to 0 for count_
```

- Consumer:

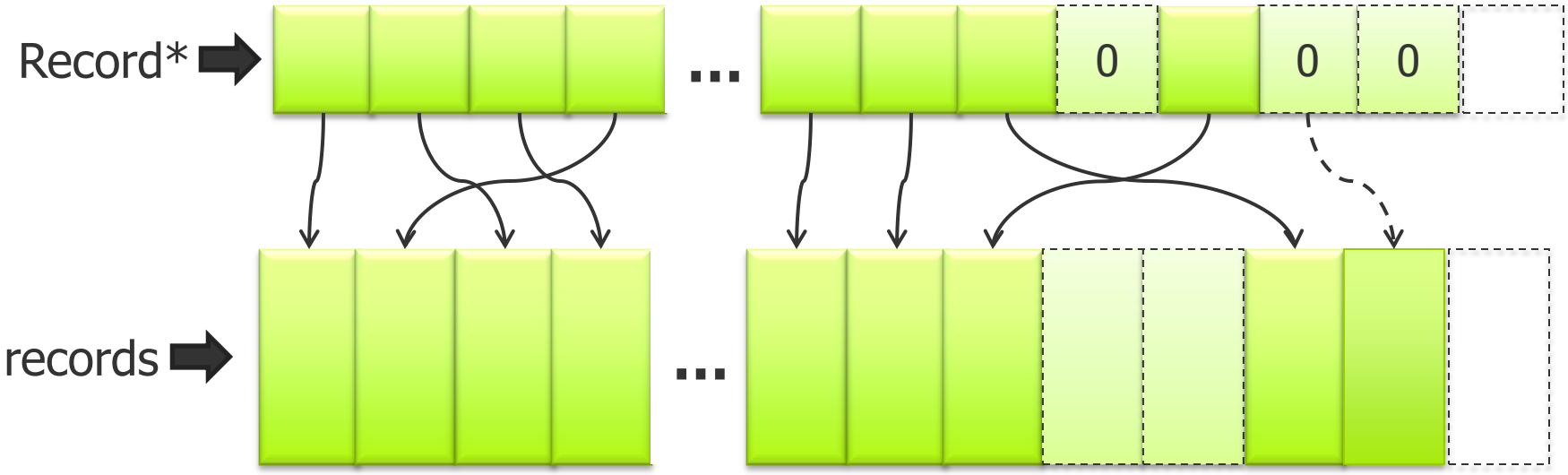
```
size_t currentN = N.AcquireLoad(); // Safe to use at least records[N-1]
for (size_t i = 0; i < currentN && records[i].ready(); ++i) {
  ProcessData(records[i]);
}
```

What's in the record?

- Ok, this is more realistic
- What if there is no natural "ready flag" in the record?
  - Or no "invalid state"?
- We can always add a field…

Any problem in computing can be solved by

- Another layer of indirection:



- records[] is the "black hole queue", append-only
- Pointers are "published" with Release barrier after records are built
- To consume a record, Acquire the pointer

Take-home points

- Lock-free programming is hard, be sure you need it
- Never guess about performance
- Lock-free programming is mostly about memory order
- Know potential dangers of locks and how to avoid them
- Know why you are going lock-free
- Design lock-free data structures
- Avoid writing generic lock-free code

Not quite a queue…

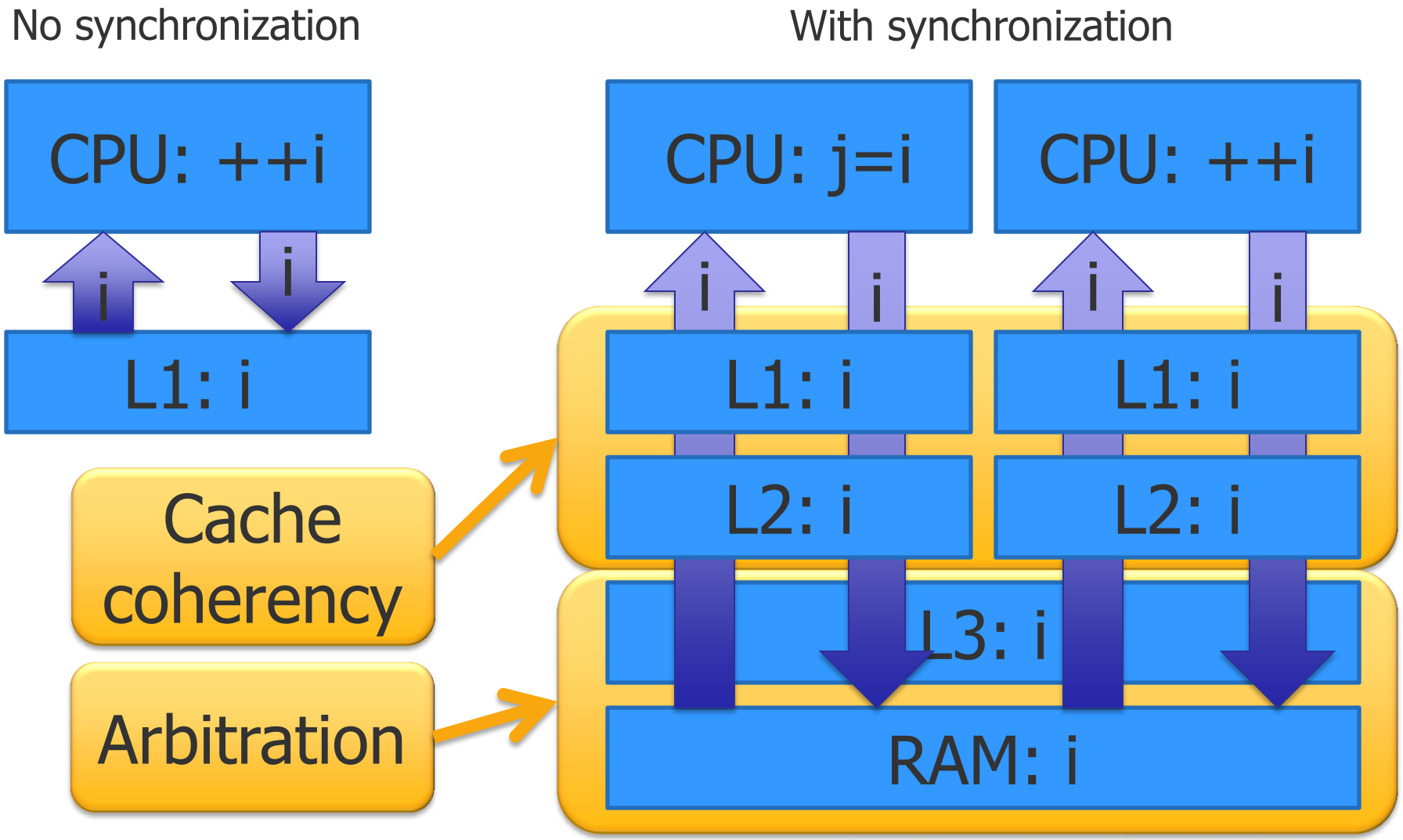
- Generic lock-free queue is very hard to write
- Problems:
  - Another atomically incremented counter for "front"
  - What to do when queue is empty
  - What to do when queue runs out of memory
- Generic lock-free queue is not always faster than a well-written non-thread-safe queue with a spinlock

What limits performance of concurrent programs

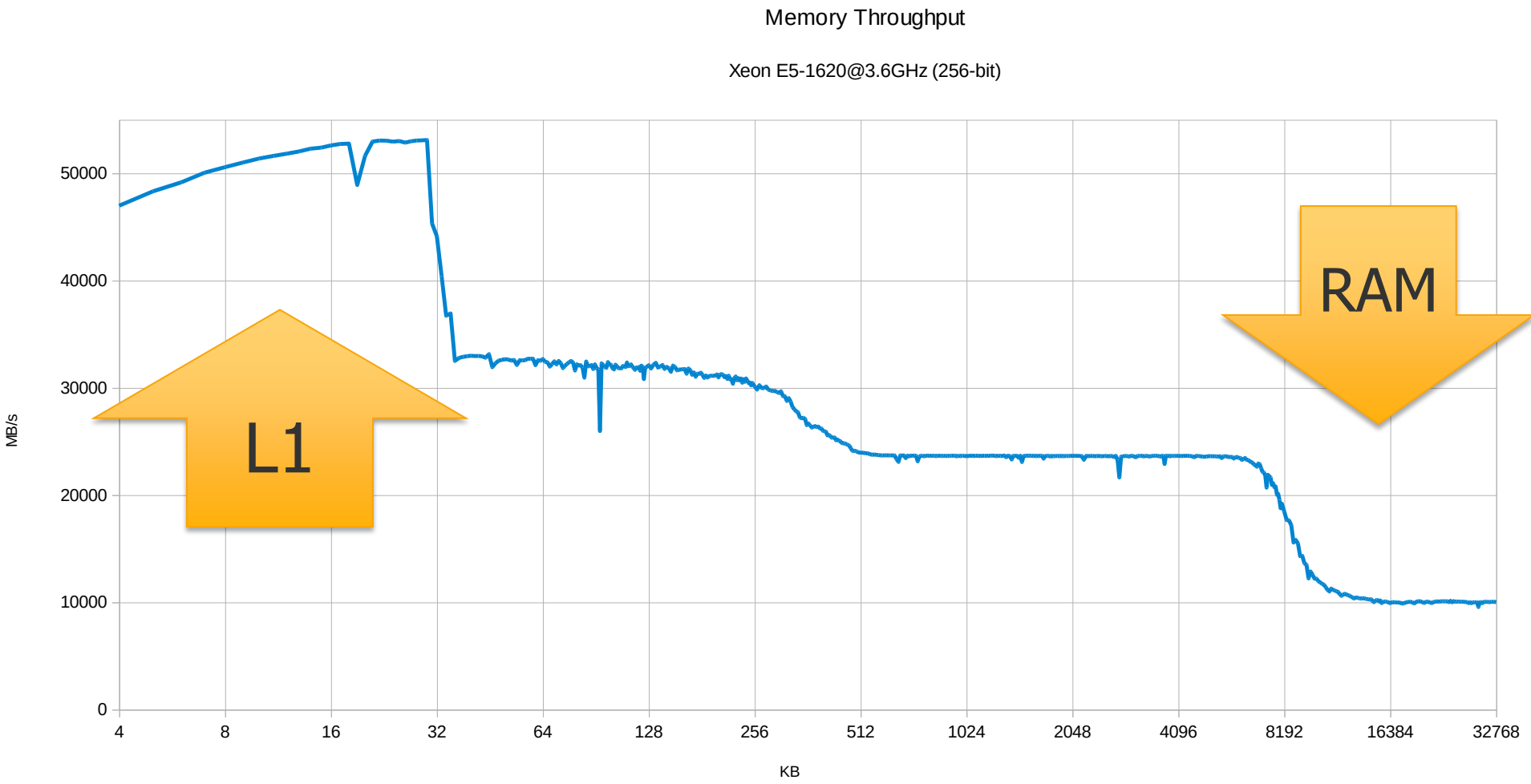
- Measure time it takes to increment a 64-bit integer:
  1. Shared variable (atomic, no locks)
  2. Not shared variable (each thread has its own)

Number of threads	Nanoseconds per computation - shared	Nanoseconds per computation – not shared
1	15	5
2	27	4
4	48	2.3
8	63	1.5
16	63	0.6
32	63	0.3
64	72	0.4
128	68	0.5

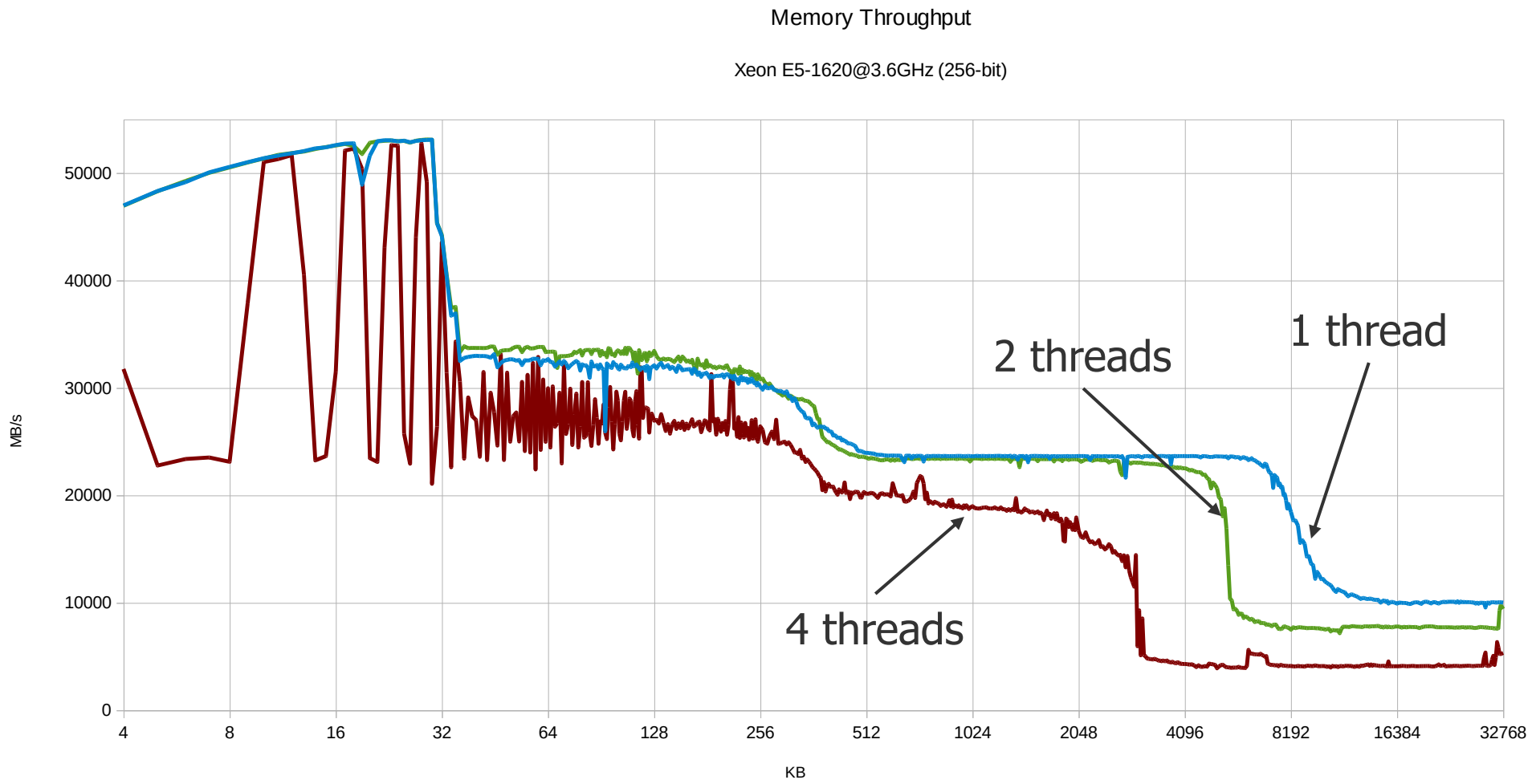
Synchronized vs Non-Synchronized Access



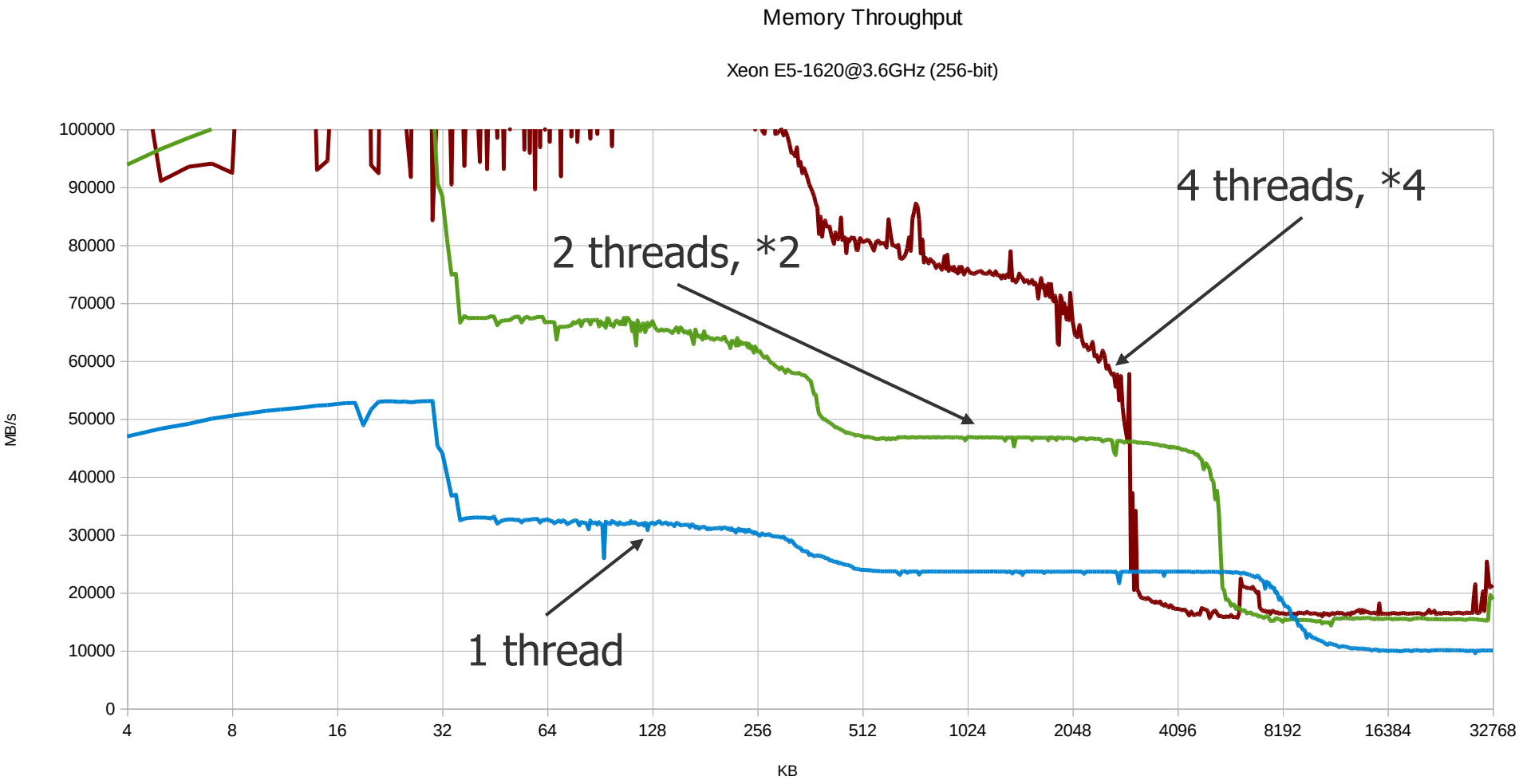
Memory Bandwidth for One Thread



Memory Bandwidth for Multiple Threads



Total Memory Bandwidth for Multiple Threads



Data Sharing is Expensive

- Non-thread-safe data access is faster than synchronized data access.
  - Simplest case: non-atomic vs atomic access
  - Avoids the unknown overhead of locking
- Why is synchronized access more expensive?
- Cache coherency must be maintained in hardware.
- Side note: circuits for caches and their maintenance are the largest part of modern CPUs and the limiting factor in increasing the number of cores.
  - Some CPU architectures give up on cache coherency altogether for that reason (Cell processor)

Take-home points

- Lock-free programming is hard, be sure you need it
- Never guess about performance
- Lock-free programming is mostly about memory order
- Know potential dangers of locks and how to avoid them
- Design lock-free data structures
- Avoid writing generic lock-free code
- Minimize data sharing, know exactly what is shared

Problems With Locks

- Performance
- It's listed last for a reason
- Performance of a well-written lock program may be no different from that of a lock-free program
- Lock-based program ≠ mutex-based program!
- Overhead of data sharing depends on the amount of data
- Mutexes have high overhead – a lot of shared data
- Spinlocks have one shared word
  - Plus the shared data they protect
- If a lock-free program needs to share several pieces of data to arrange lock-free synchronization, it may end up being slower than a program with spinlocks

Not quite a queue…

- Generic lock-free queue is very hard to write
- Problems:
  - Another atomically incremented counter for "front"
  - What to do when queue is empty
  - What to do when queue runs out of memory
- Generic lock-free queue is not always faster than a well-written non-thread-safe queue with a spinlock
- But a well-written lock-free queue is always faster
- Right?

The performance mess

- Comparing general lock-free queue with spinlocked wrapper around std::queue
  - Results are inconsistent, varies a lot between CPU architectures (Haswell, Nehalem, Sandy Bridge) and machine configurations (number of CPUs, number of sockets)
  - Results depend strongly on access patterns and congestion
  - If there is a trend, it's slightly in favor of std::queue (on the set of machines I have access to and set of access patterns I tested)
  - Subtle implementation details matter, such as having top and bottom pointers on separate cache lines
- Special-case queues can be much faster
  - Maximum number of elements known in advance
  - One producer thread, many consumer threads
  - One consumer thread, many producer threads
  - Very different implementations but well worth it

Not quite a queue…

- Generic lock-free queue is very hard to write
- Problems:
  - Another atomically incremented counter for "front"
  - What to do when queue is empty
  - What to do when queue runs out of memory
- Generic lock-free queue is not always faster than a well-written non-thread-safe queue with a spinlock
- But a specialized lock-free queue often is!
- What exactly is a thread-safe queue, anyway?

A queue is a queue is a queue

- We know what a queue is! std::queue:
  - front(), push(), pop(), empty()…
- Of course, std::queue is not thread-safe
- What if we wrote a thread-safe implementation of std::queue (lock-free, of course)?
- Several threads can call push() at once (producers)
- What about consumers?

```
std::queue<T> q;
if (!q.empty()) {
  T x = q.front();
  q.pop();
}
```

- Each member function is thread-safe

Thread-safe data structures require special interfaces

- A thread-safe data structure can, at most, guarantee that each member function call is a transaction
  - Whether lock-free or guarded by a lock
- That transaction must be useful to the caller
- Thread-safe queue:

void push(const T&) – thread-safe, always works

bool pop(T&) – atomically dequeues the first element, if any stores it in the argument and returns true, or returns false and does not change argument

std::pair<T,bool> pop() – on failure, returns T() and false

- Still, what is a thread-safe queue?

Take-home points

- Lock-free programming is hard, be sure you need it
- Never guess about performance
- Lock-free programming is mostly about memory order
- Know potential dangers of locks and how to avoid them
- Design lock-free data structures
- Avoid writing generic lock-free code
- Minimize data sharing, know exactly what is shared
- Design in terms of atomic transactions
- Lock-free data structures usually need special interfaces

Thread-safe data structures require special interfaces

- A thread-safe data structure can, at most, guarantee that each member function call is a transaction
  - Whether lock-free or guarded by a lock
- That transaction must be useful to the caller
- Thread-safe queue:

void push(const T&) – thread-safe, always works

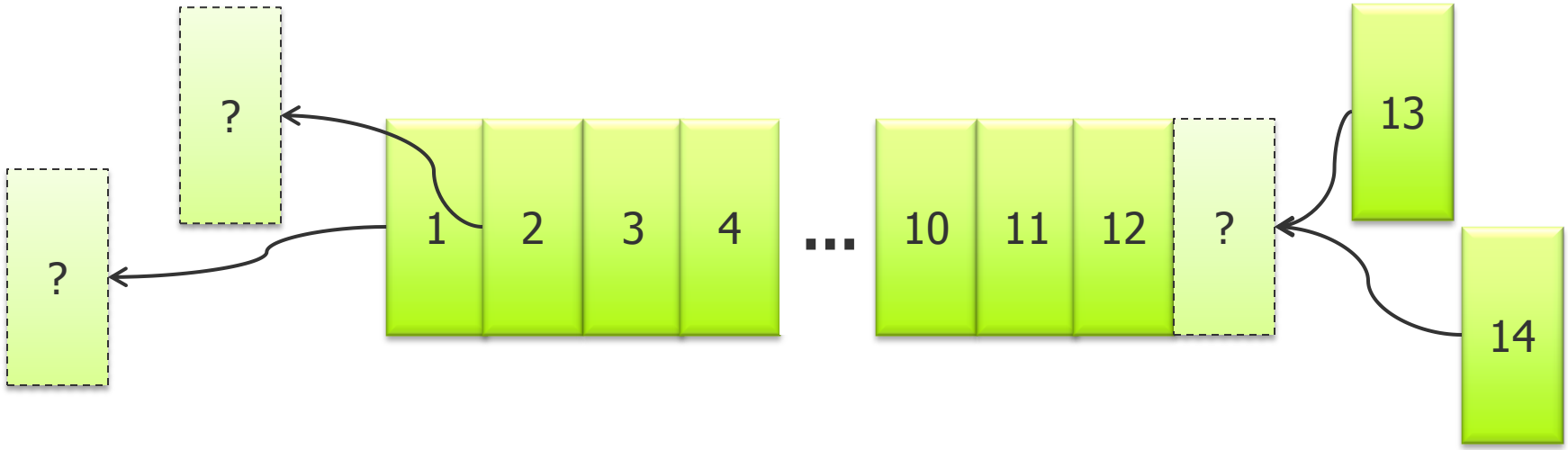
bool pop(T&) – atomically dequeues the first element, if any stores it in the argument and returns true, or returns false and does not change argument

std::pair<T,bool> pop() – on failure, returns T() and false

- Still, what is a thread-safe queue?

When a queue is not a queue

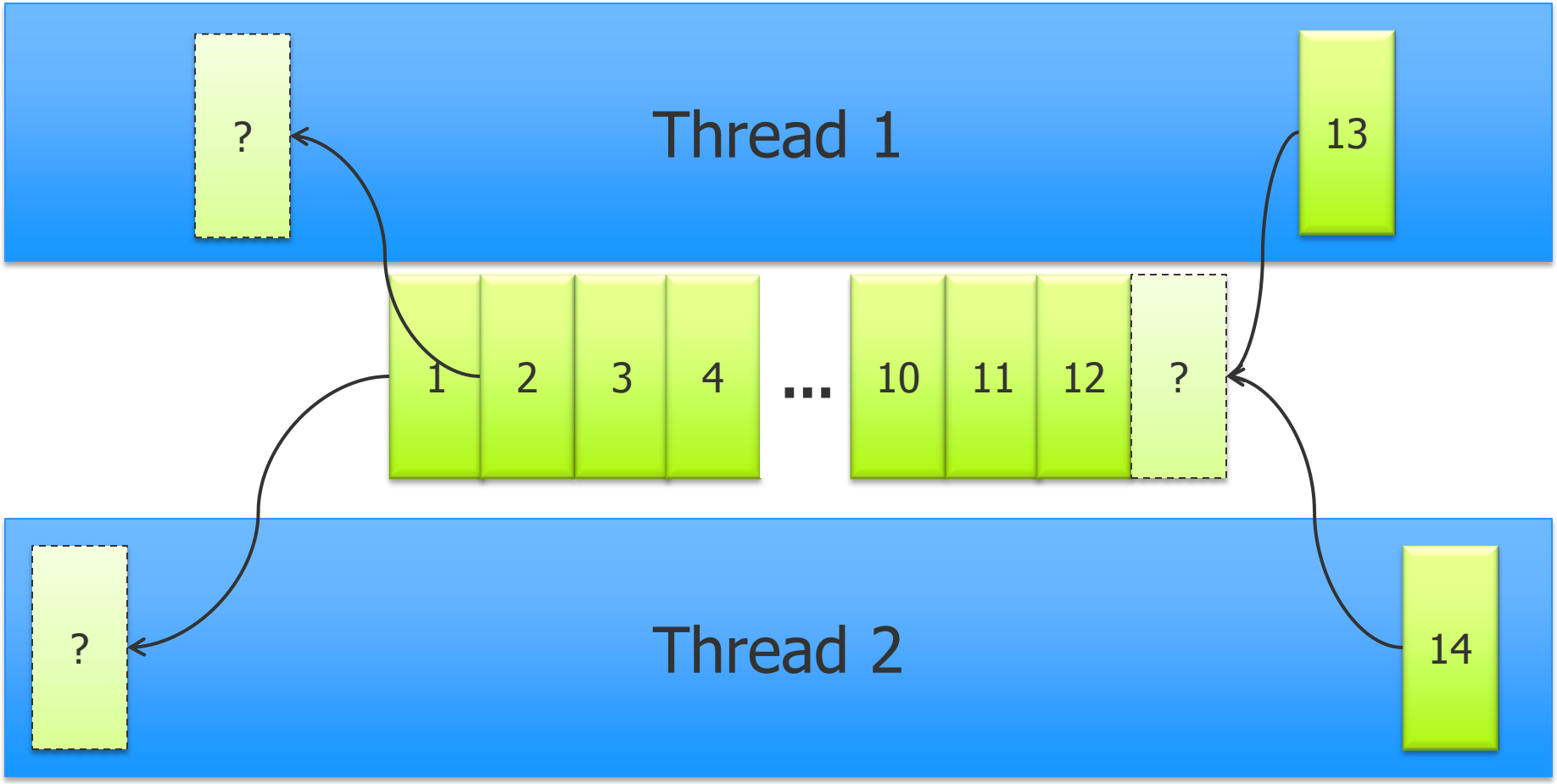
- A queue is a data collection in which the items are kept in the order in which they were inserted, and the primary operations are enqueue (insert an item at the end) and dequeue (remove the item at the front). (chegg.com)



- 1 will be dequeued before 2
- If 13 is enqueued before 14, 13 will come out before 14

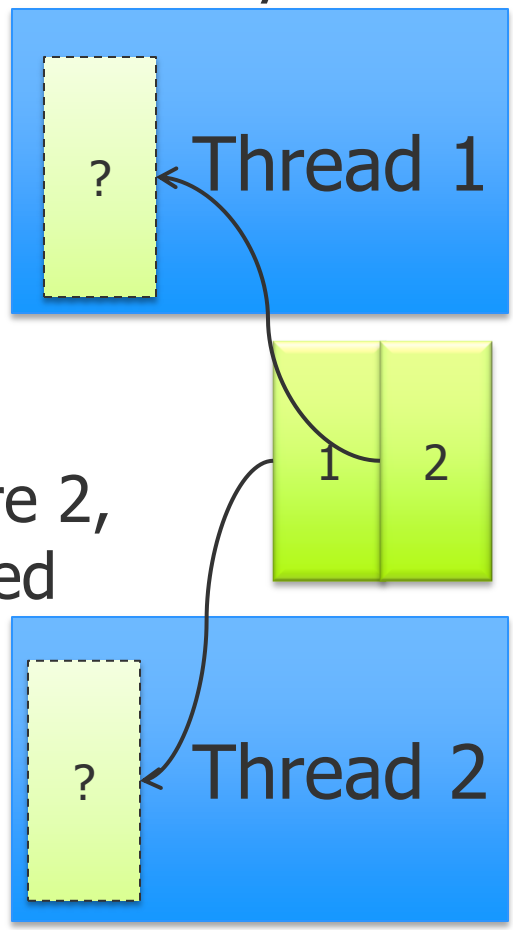
When a queue is not a queue

- Does the order guarantee still hold with concurrency?



When a queue is not a queue

- Does the order guarantee still hold with concurrency?
- Both threads call dequeue()
- Which one completes "first"?
- How would you know?
  - The only way to define "first" and "second" is to serialize the execution
- Even if 1 is removed from the queue before 2, there is no guarantee that they are returned to the caller in that order




```
T& dequeue() {
  T x = … get first element …
  return x;
}
```

1 first

2 first

When a queue is not a queue

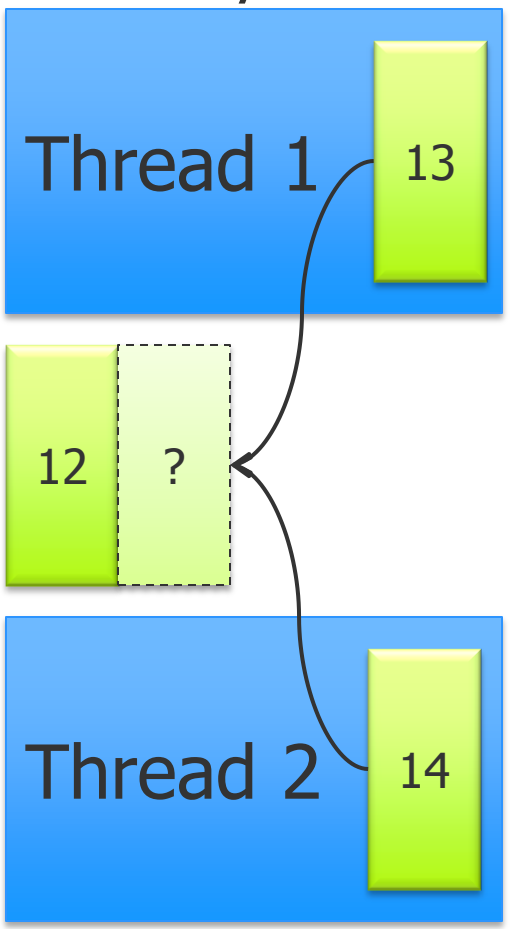
- Does the order guarantee still hold with concurrency?
- Both threads call enqueue()
- Which one completes "first"?
- The only way to know which was "first" is to serialize the execution
- Even if enqueue(13) is called before enqueue(14), there is no guarantee that they access queue in that order

```
enqueue(const T& x) {

  new(… get queue slot…) T(X);
}
```

13 first

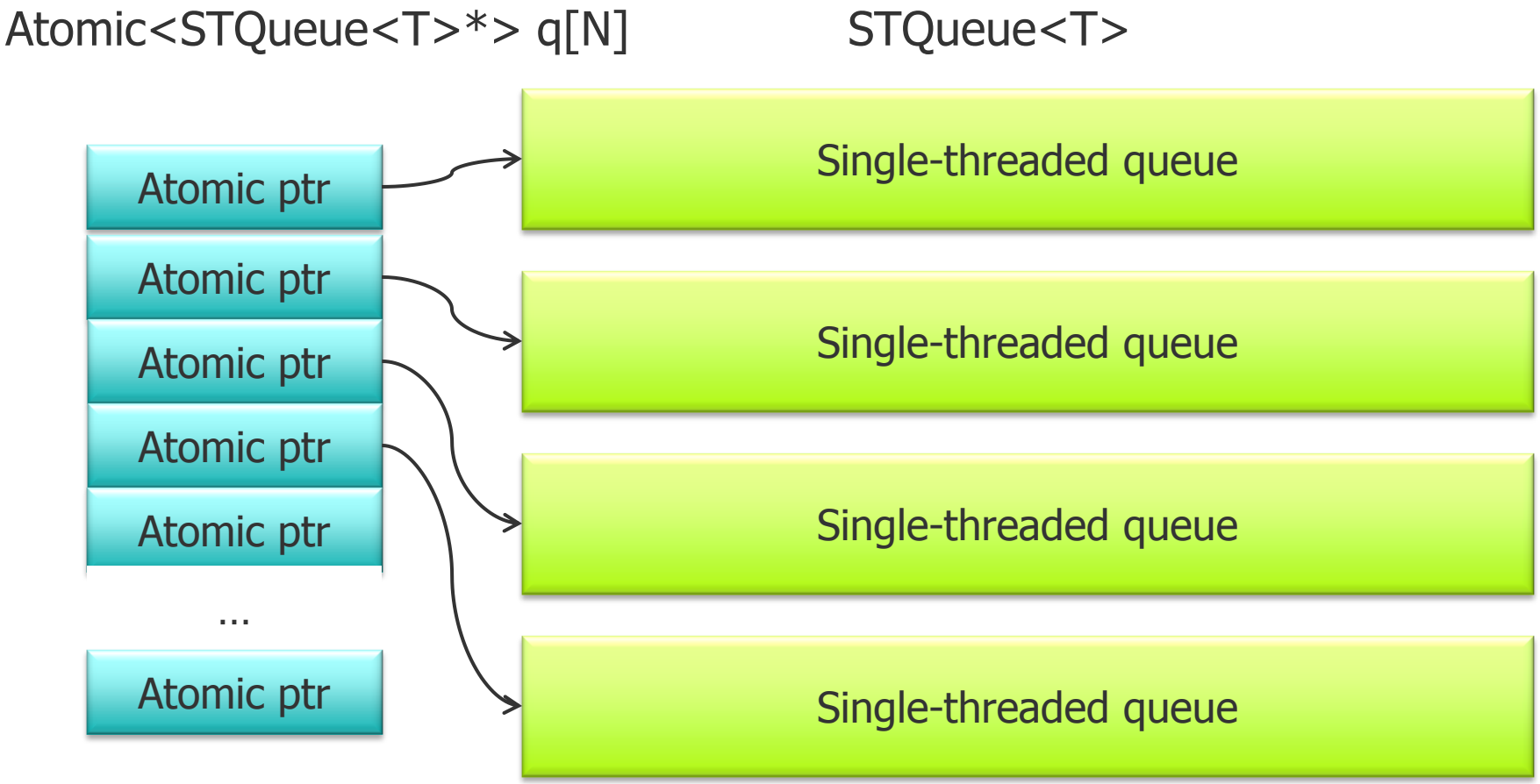
14 first



What is a concurrent queue?

- If x and y were enqueued by one thread, and x was enqueued before y, then, if x and y are also dequeued by one thread, x will come out before y.
- If enqueue(x) completed before enqueue(y) was called, then, if dequeue()->x completes before another call to dequeue(), y will come out after x.
- Together, these conditions define sequential consistency
- These conditions are expensive to assert in practice
  - Expensive for the program, not just for the queue itself
  - A program that is sensitive to these conditions is likely to scale poorly (even if the queue scales well)
- Practical guarantee: Elements mostly come out in the right order. Typically. Most of the time. Usually.

Fast concurrent queue



Fast concurrent queue


```
template <typename T, size_t num_queues> class Queue {
  typedef STQueue<T>* qptr_t;
  Atomic<qptr_t> queues_[num_queues];
  public:
  Queue(size_t max_queue_size);
  void Enqueue(const T& x);
  bool Dequeue(T& x);
}
```

Fast concurrent queue

```
void Enqueue(const T& x) {
  GetSTQueue q;
  q->enqueue(x);
}
bool Dequeue(T& x) {
  GetSTQueue q;
  if (q->empty()) return false;
  x = q->dequeue();
  return true;
}
```

- GetSTQueue must atomically take ownership of one of STQueue queues in constructor, and release it in destructor

Fast concurrent queue

```
class GetSTQueue {
  qptr_t q_; size_t i;
  GetSTQueue() {
    for (i = 0; i < num_queues; ++i) {
      q_ = queues_[i].BarrierAtomicExchange(NULL);
      if (q_ != NULL) return;          // We own queues_[i]
    }
  }
  …
};
```

- What to do if we did not get a queue?
  - Give up, return failure to caller
  - Keep trying, we'll get one eventually

Can enqueue fail?

- Best approach depends on the application
- What would the caller do if enqueue() failed?

```
bool Enqueue(const T& x) {
  GetSTQueue q;
  if (!q || !q->enqueue(x)) return false;
  return true;
}
```

- Example: producer threads prepare records to process and enqueue them; if queue is full have producer thread process its own data to balance producers vs consumers
- Take every advantage of the specific requirements of the application, avoid generic solutions

Fast concurrent queue

```
class GetSTQueue {
 qptr_t q_; size_t i;
  GetSTQueue() { … }
  ~GetSTQueue() {
    queues_[i].ReleaseStore(q_);
  }
  operator qptr_t() { return q_; }
};
```

Can dequeue fail?

- dequeue() fails if the queue is empty
- If enqueue() adds to the queue at the same time, dequeue() may still fail (there is no "same time")
- Eventually dequeue() must find all elements

```
bool Dequeue(T& x) {
  GetSTQueue q(true);
  return q->dequeue(x);
}
```

- GetSTQueue(true) finds next non-empty queue
  - Gives up after trying every queue

Performance considerations

- Is it better to avoid false sharing of queue pointers?
  - False sharing occurs when independent variables share a cache line, concurrent accesses cost similar to shared variables
  - My measurements did not show any difference
  - Do your own measurements – your results may differ!
- Should search for the next queue start from slot 0, from the last slot, or every thread starts from its own place?
  - Spreading the starting slot probably reduces contention
  - To get thread-specific index we need thread id (~40 usec)
  - To use "next slot" always, we need to store last accessed slot, this adds a shared variable
  - Measurements are needed to determine best approach

Take-home points

- Lock-free programming is hard, be sure you need it
- Never guess about performance
- Lock-free programming is mostly about memory order
- Know potential dangers of locks and how to avoid them
- Design lock-free data structures
- Avoid writing generic lock-free code
- Minimize data sharing, know exactly what is shared
- Design in terms of atomic transactions
- Lock-free data structures usually need special interfaces
- Take every advantage of practical limitations

It's all fun and games until someone runs out of memory

- What to do when queue is full?
- In some applications enqueue() can fail
- What if we need to handle higher load and want to increase the number of queues?
  - Very hard to do safely while the queue operates
  - Does not happen often
- A problem that is both hard and rare – use DCLP

```
GetSTQueue() {
  if (wait.AcquireLoad() == 0) …  // Proceed normally
    total_lock.Lock();            // Everything is halted
    …
```

DCLP reborn

- Most of the time, we can just take a queue as usual
- Sometimes, a thread wants to add queues
- Threads that already own a queue can proceed
- Threads that want a queue should be paused
- If a thread is in the process of taking a queue, the global modification should wait

```
GetSTQueue() {
  if (wait.AcquireLoad() == 0) …  // Proceed normally
  total_lock.Lock();              // Everything is halted
    … we got the lock, proceed normally …
  total_lock.Unlock();            // Allow global mods
}
```

DCLP reborn

- Most of the time, we can just take a queue as usual
- Sometimes, a thread wants to add queues
- Threads that already own a queue can proceed
- Threads that want a queue should be paused
- If a thread is in the process of taking a queue, the global modification should wait

```
AddMoreQueues() {
  wait.ReleaseStore(1);              // Request threads to wait
  total_lock.Lock();                 // Everything is halted
    … we got the lock, add queues …
  total_lock.Unlock();               // Resume paused threads
  wait.ReleaseStore(0);              // Revolution is over
}
```

Take-home points

- Lock-free programming is hard, be sure you need it
- Never guess about performance
- Lock-free programming is mostly about memory order
- Know potential dangers of locks and how to avoid them
- Design lock-free data structures
- Avoid writing generic lock-free code
- Minimize data sharing, know exactly what is shared
- Design in terms of atomic transactions
- Lock-free data structures usually need special interfaces
- Take every advantage of practical limitations
- Use DCLP to handle special cases

Take-home points

- Lock-free programming is hard, be sure you need it
- Never guess about performance
- Lock-free programming is mostly about memory order
- Know potential dangers of locks and how to avoid them
- Design lock-free data structures
- Avoid writing generic lock-free code
- Minimize data sharing, know exactly what is shared
- Design in terms of atomic transactions
- Lock-free data structures usually need special interfaces
- Take every advantage of practical limitations
- Use DCLP to handle special cases

Mentor Graphics®

www.mentor.com