Utrecht University

B3CC: Concurrency

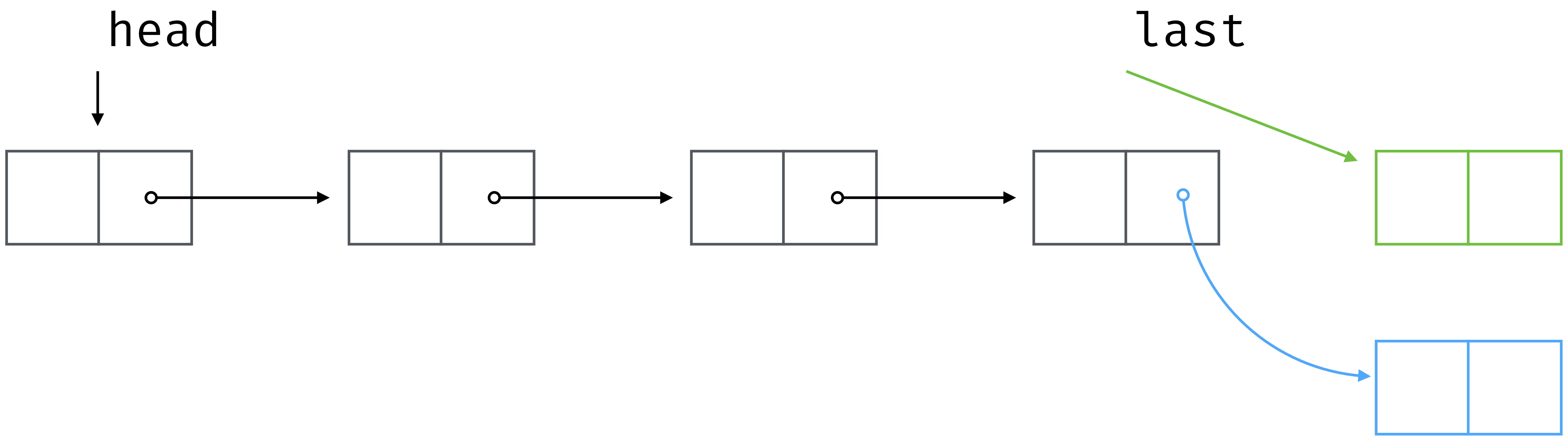
05:Threads (3)

Tom Smeding

Recap

- Concurrency is a way to structure a program using multiple *threads of control*
  - Conceptually threads execute "at the same time": effects are interleaved
  - In *purely functional* code there are no effects to observe, so evaluation order is irrelevant
- Shared (mutable) state is what makes concurrency so challenging
  - Multiple threads can access the same memory location at the same time
  - Concurrency sacrifices determinism

Recap



- Lessons learned
  - Control access to shared resources/variables
    - Control access to the **code** using those shared resources: *critical sections*

Non-blocking algorithms

Non-blocking algorithms

- *Blocking* algorithms use some lock-like technique to synchronise with shared resources
  - When trying to acquire a lock held by another thread: block until lock is free
    - Even if the other thread is not making any progress (e.g. suspended or terminated)
- An algorithm is *non-blocking* if failure or suspension of any thread can not cause failure or suspension of another thread
  - Typically built upon atomic read-modify-write primitives supplied by the hardware (e.g. compare-and-swap)
  - *Software Transactional Memory*: abstraction for writing (almost) non-blocking code (more on that later…)

Non-blocking algorithms

1. Atomic primitives (hardware operations)
2. Progress guarantees (how non-blocking is your code?)
3. Memory models (processors lying to you)
4. Scalability (how to make code slower by adding more cores)

Atomic primitives

- compare-and-swap
  - Perhaps the most common atomic primitive (CMPXCHG LOCK, atomicCasWordAddr#, InterlockedCompareExchange, __atomic_compare_exchange, …)
  - Some architectures (ARM, RISC-V, …) offer an alternative Linked-Load/Store-Conditional (LL/SC)

```
Pair<Bool, T> compare_exchange(T* location, T expected, T replacement) {
  do atomically {
    T old = *location;
    if (old == expected) {
      *location = replacement;
      return {true, old};
    } else {
      return {false, old};
    }
  }
}
```

Atomic primitives

- fetch-and-add
  - Another atomic read-modify-write operation (XADD_LOCK, fetchAddWordAddr#, …)
  - Also variations such as fetch-and-[sub,and,or,xor]

```
T fetch_and_add(T* location, T value) {
  do atomically {
    T old = *location;
    *location = old + value;
    return old;
  }
}
```

Atomic primitives

- exchange
  - Another atomic read-modify-write operation (XCHG, `atomicExchangeWordAddr#`, …)
  - No less useful than the others!

```
T exchange(T* location, T value) {
  do atomically {
    T old = *location;
    *location = value;
    return old;
  }
}
```

Atomic primitives

- Atomic loads and stores
  - These are not read-modify-write operations, they are just independent loads (`atomicReadWordAddr#`) and stores (`atomicWriteWordAddr#`)
  - Generally cheaper/faster than atomic RMW operations
  - Mostly relevant because of *memory access reordering*; see later

Progress guarantees: Wait free

- *Every* thread makes progress regardless of external factors
  - An algorithm is *wait-free* if every operation has a bound on the number of operations it takes to complete
  - Combines guaranteed system-wide throughput with *starvation freedom*
  - Typically implemented using atomic operations that do not contain loops that can be affected by other threads
  - Strongest progress guarantee

```
void increment_reference_count(obj_base* this) {
    atomic_fetch_and_add(&this->count, 1);
}
```

Progress guarantees: Lock free

- The system *as a whole* makes progress, but forward progress of an individual thread is *not* guaranteed
  - At least one thread will finish the operation in a bounded number of steps
  - A blocked/interrupted/terminated thread can not prevent the forward progress of other threads
  - Weaker guarantee than wait-freedom; all wait-free algorithms are lock-free

```
void stack_push(stack* s, node* n) {
  node *top;
  do {
    top     = s->top;
    n->next = top;
  } while (! atomic_compare_exchange(&s->top, top, n) );
}
```

Progress guarantees: Lock free

- The system *as a whole* makes progress, but forward progress of an individual thread is *not* guaranteed
  - At least one thread will finish the operation in a bounded number of steps
  - A blocked/interrupted/terminated thread can not prevent the forward progress of other threads
  - Weaker guarantee than wait-freedom; all wait-free algorithms are lock-free

- The essence of lock freedom: you fail *only* when somebody else makes progress
  - Compare non-blocking vs. blocking algorithms:
    - CAS loop: loops on progress (by somebody else)
    - Spin-lock: loops on progress *and* non-progress (because another thread took the lock already)

Progress guarantees: Obstruction free

- A thread makes forward progress only if it does not encounter contention from other threads
  - A single thread executed in isolation will complete its operation in a bounded number of steps
  - Weakest progress guarantee; all lock-free algorithms are obstruction free

https://cs.brown.edu/people/mph/HerlihyLM03/main.pdf

Progress guarantees

- Lots of practical programs use locks, of course
- Non-blocking algorithms consider theoretical properties of the program
  - Lock-based program: a thread can make progress (if deadlock-free)
  - Lock-free algorithm: a *running* thread can make progress
  - Non-blocking algorithms work in situations where blocking algorithms cannot (e.g. signal handlers, hard real-time systems)
- Consensus protocols give us forward progress guarantees, but say nothing about performance…

Memory model

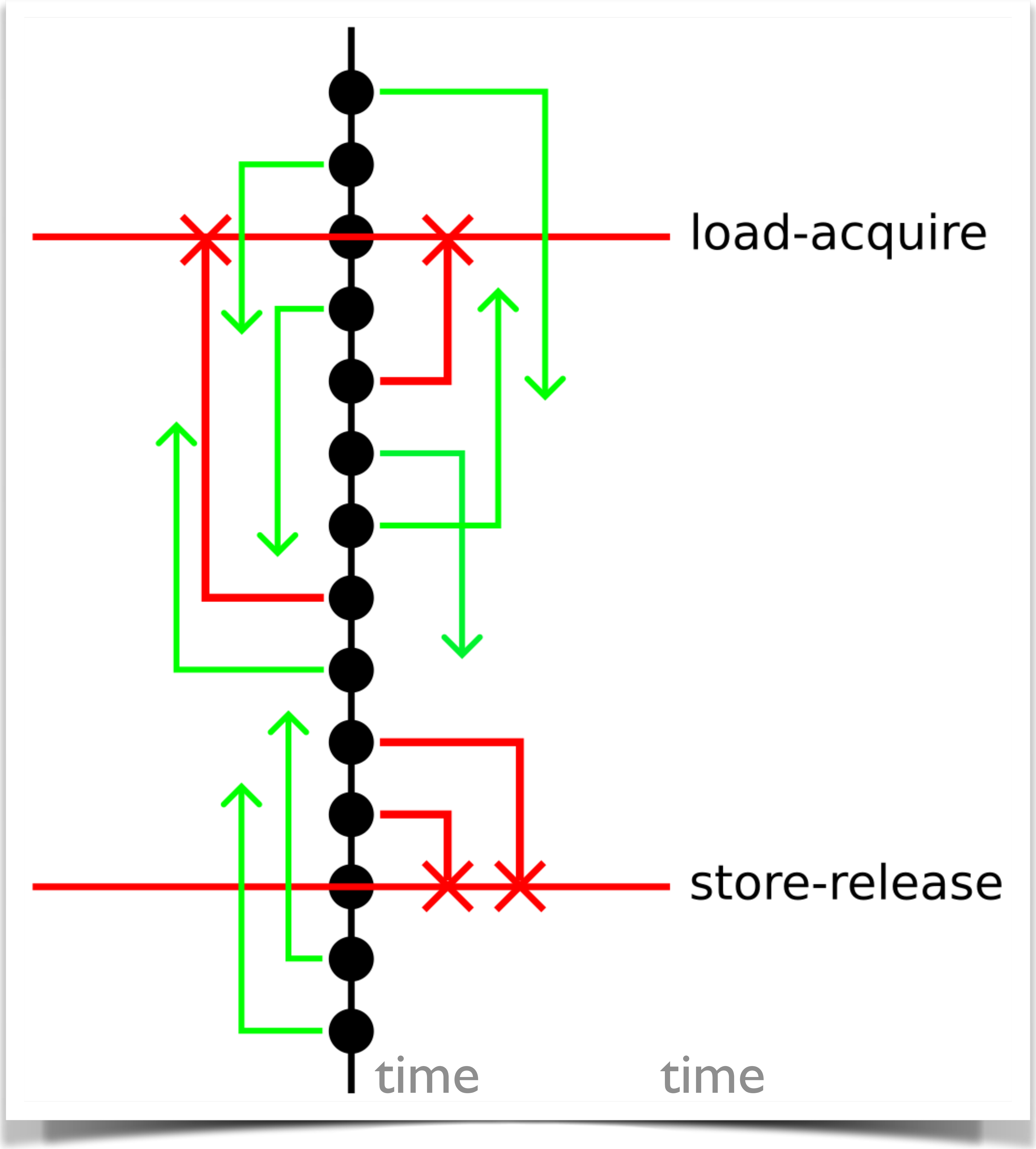
- What is a memory model?
  - Many things: pointer size, paging, cache associativity…
  - For shared-memory concurrency we are concerned with only three things:
    - Atomicity: what operations are atomic? (it completes or it didn’t happen)
    - Visibility: when (or whether) other threads see changes made by the current thread
    - Ordering: what re-ordering of loads and stores are possible relative to program order

Memory model

- For a single threaded program the hardware provides *sequential self-consistency*
  - For the program, everything looks like all memory accesses were done in program order (they weren't)
- For multi-threaded programs, the different threads can see these memory accesses in a weird order
  - The memory model determines which re-orderings are possible relative to program order
  - The hardware provides special instructions to prevent some reorderings

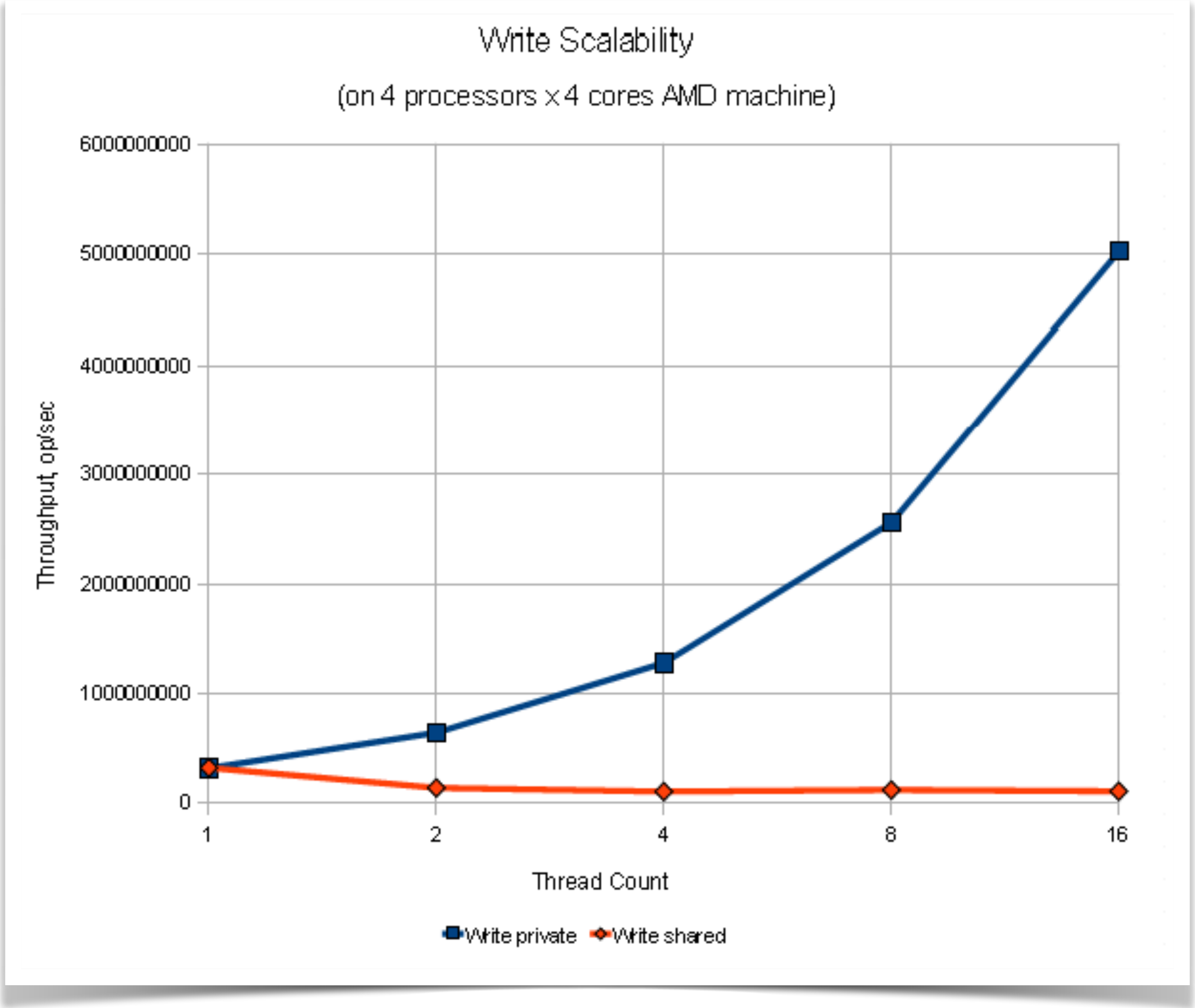
Memory model

- Typical usage: Fence tied to a memory access
- Examples:
  - load-acquire
    - Prevents memory accesses from hoisting above it
    - Allows memory accesses to sink below it
  - store-release
    - Allows memory accesses to hoist above it
    - Prevents memory accesses from sinking below it



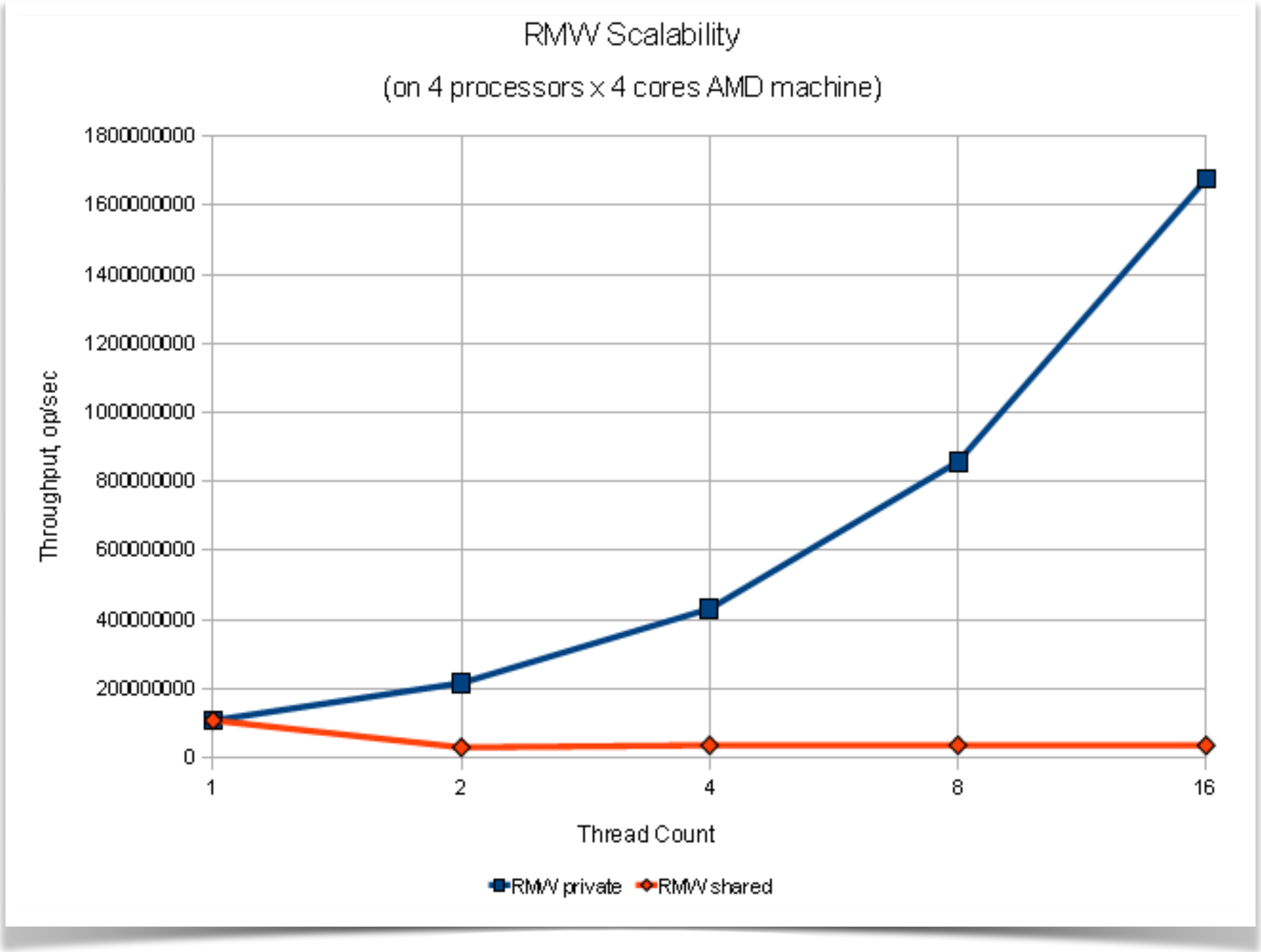
Scalability

- Scalability of write operations (x86 MOV instruction)



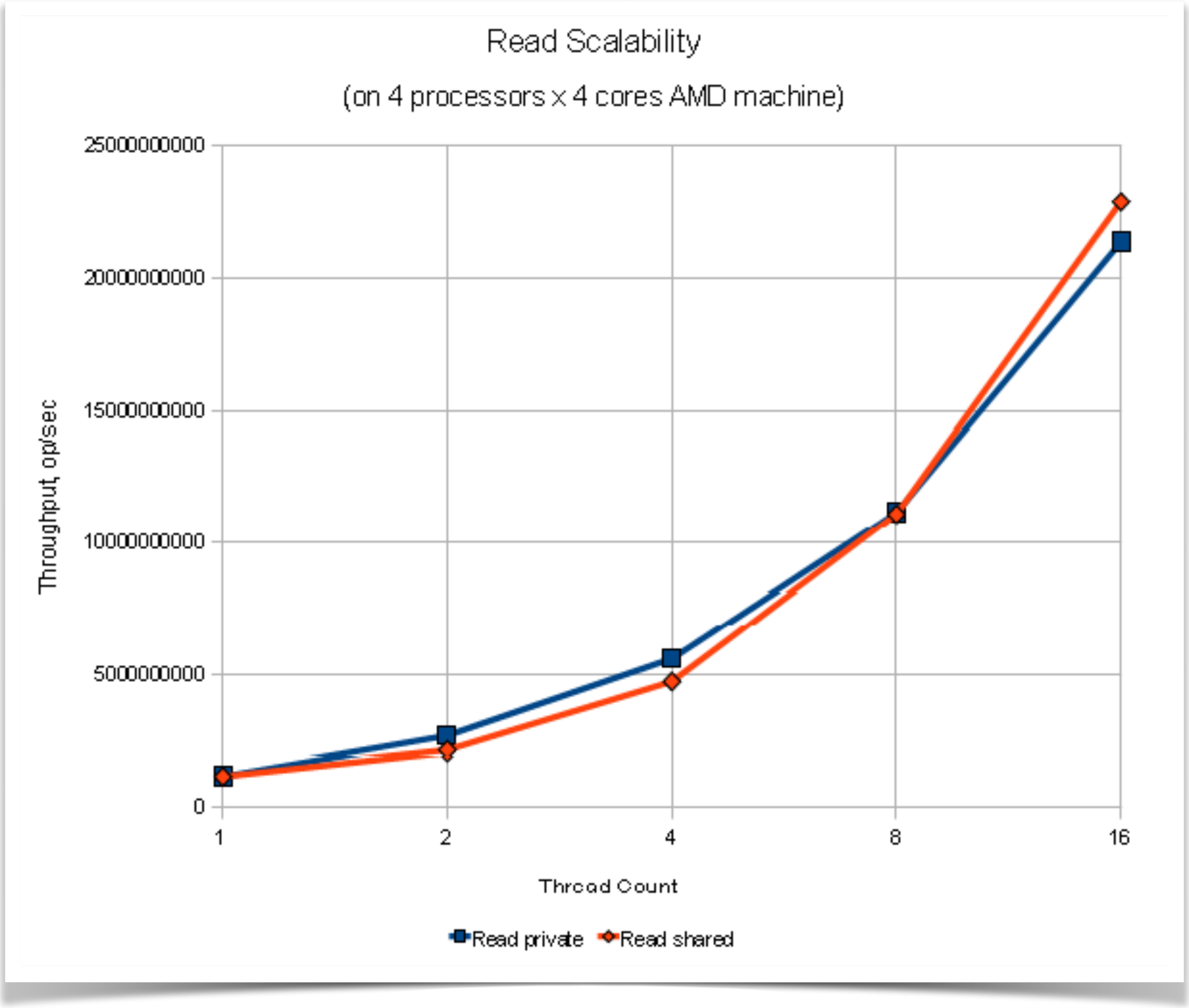
Scalability

- Scalability of atomic read-modify-write operations (x86 XADD LOCK instruction)



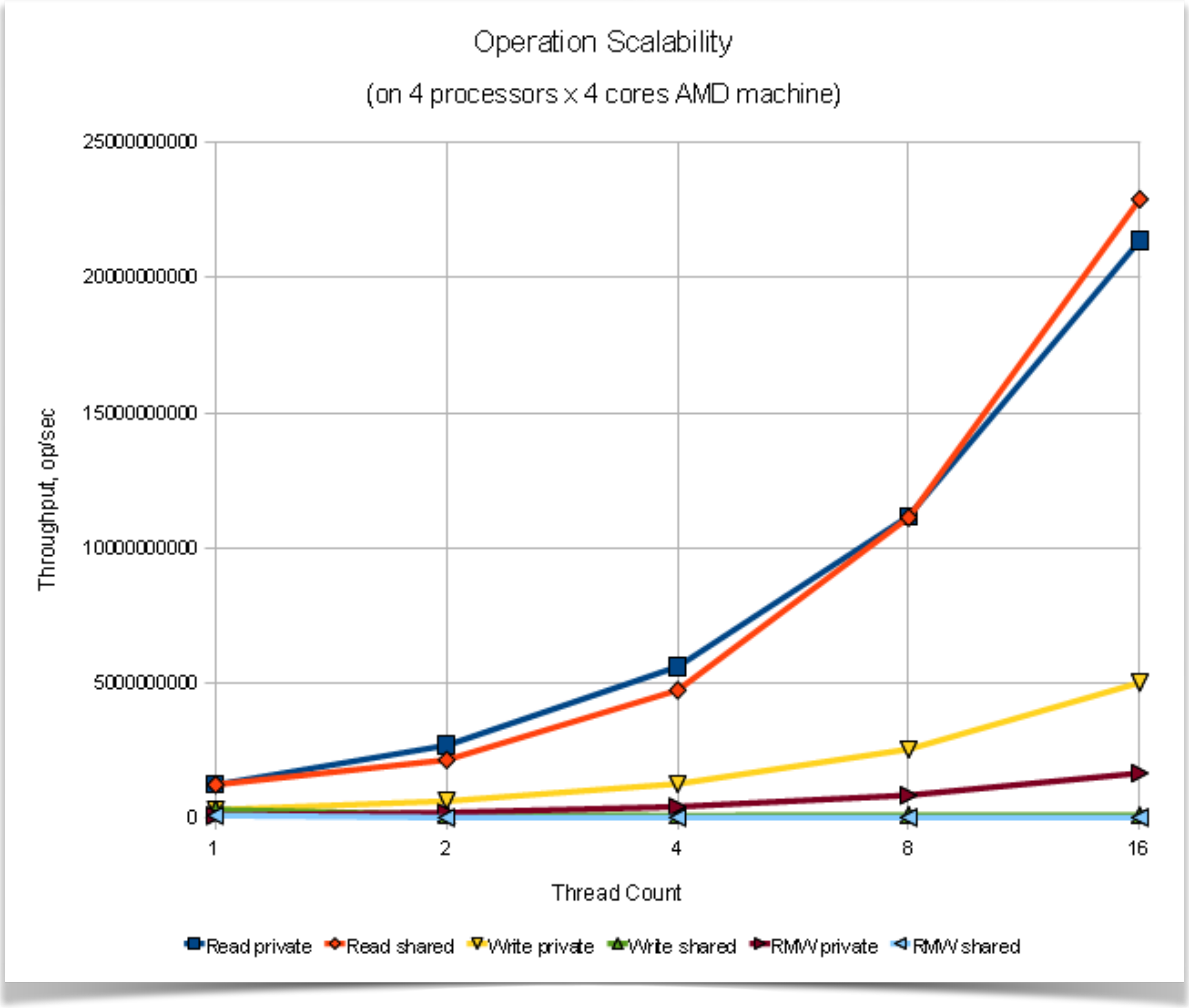
Scalability

- Scalability of read operations (x86 MOV instruction)



Scalability

- All together now:



← read

← private write

← private RMW

← shared (RM)W

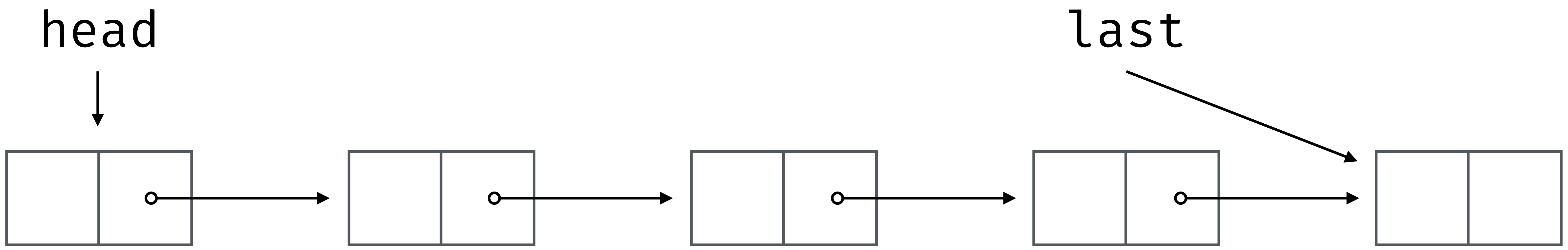
Scalability

- If there is write sharing, performance of the system *will degrade*
  - The more threads we add, the slower it becomes
- If there is no write sharing, the system scales linearly
  - Atomic RMW operations are slower than plain load+store, but scale in the same way
- Loads are always scalable
  - Several threads are able to read the same memory location simultaneously
  - Read-only access is your best friend in a concurrent environment!
- Be aware of *false sharing*
  - For performance reasons cache-coherence protocols work with whole cache lines, not bytes/words

MVars as a building block (II)

Concurrent queue

Recall: Example: access to a global queue



- Inserting:
  - Create new object
  - Set `last->next` to `&new`
  - Set `last` to `&new`

LAST WEEK

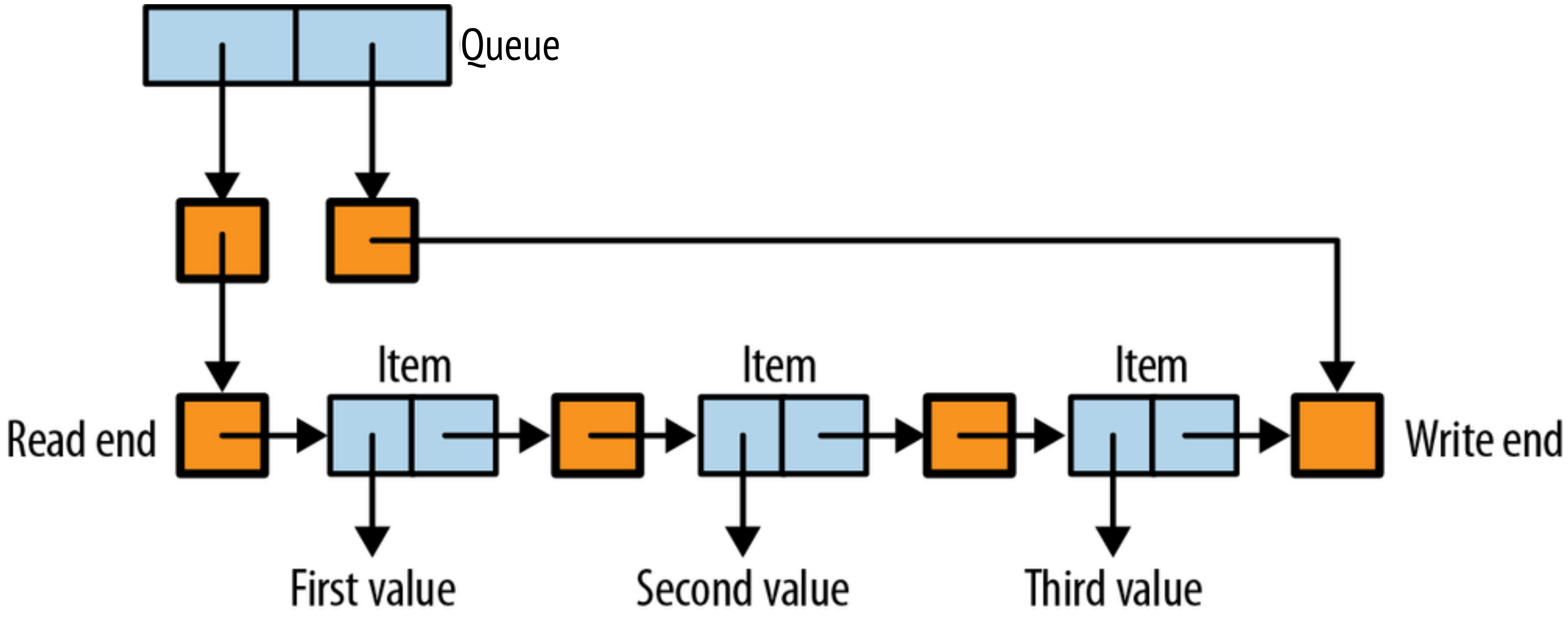
Unbounded queue

- The goal:
  - An unbounded multi-producer multi-consumer concurrent queue
  - Writers and readers do not conflict with each other (for queues of length ≥ 2)
  - Basic interface:

```
data Queue a

newQueue :: IO (Queue a)
enqueue  :: Queue a -> a -> IO ()
dequeue  :: Queue a -> IO a
```

Structure of the queue

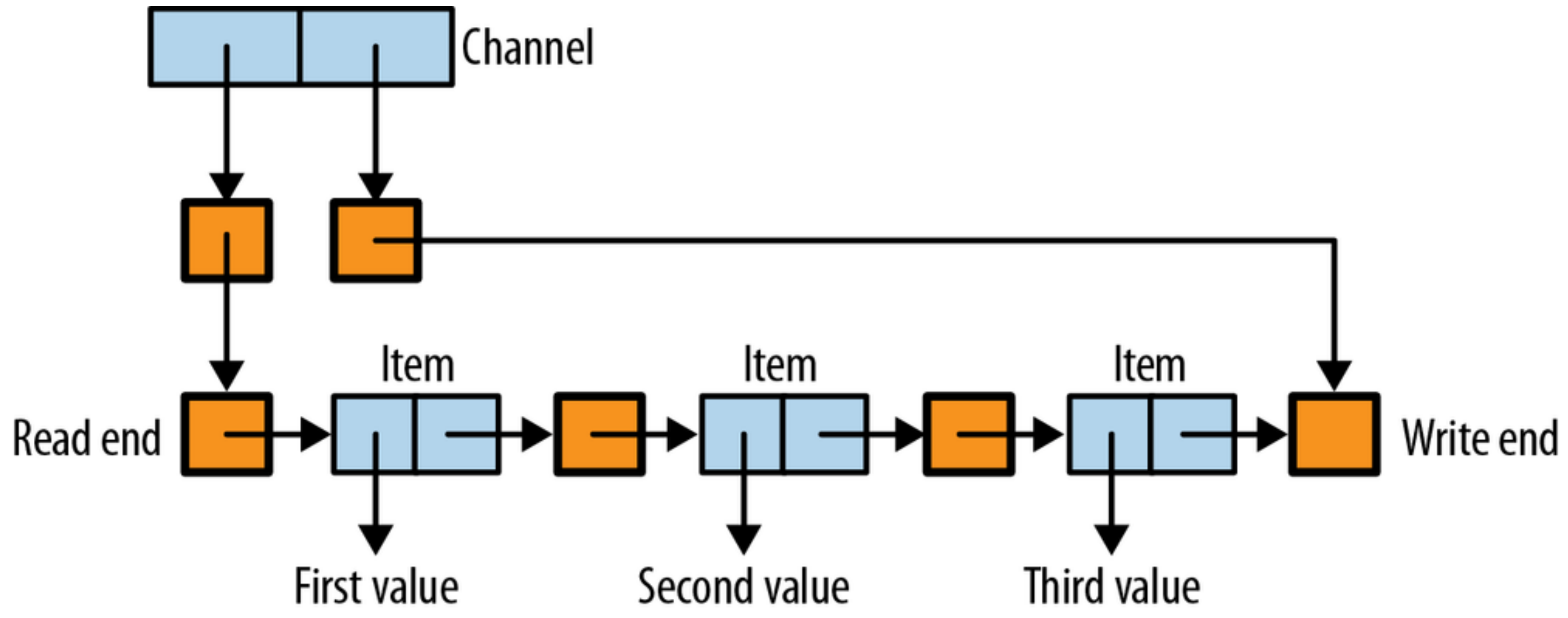


```
data Queue a =
  Queue (MVar (List a))
        (MVar (List a))

type List a = MVar (Item a)
data Item a = Item a (List a)
```

newQueue

- Create a new empty queue
  - Both locks point to *the empty stream*: the place to read/write the next value



```
newQueue :: IO (Queue a)
newQueue = do
  hole      <- newEmptyMVar
  readLock  <- newMVar hole
  writeLock <- newMVar hole
  return (Queue readLock writeLock)
```

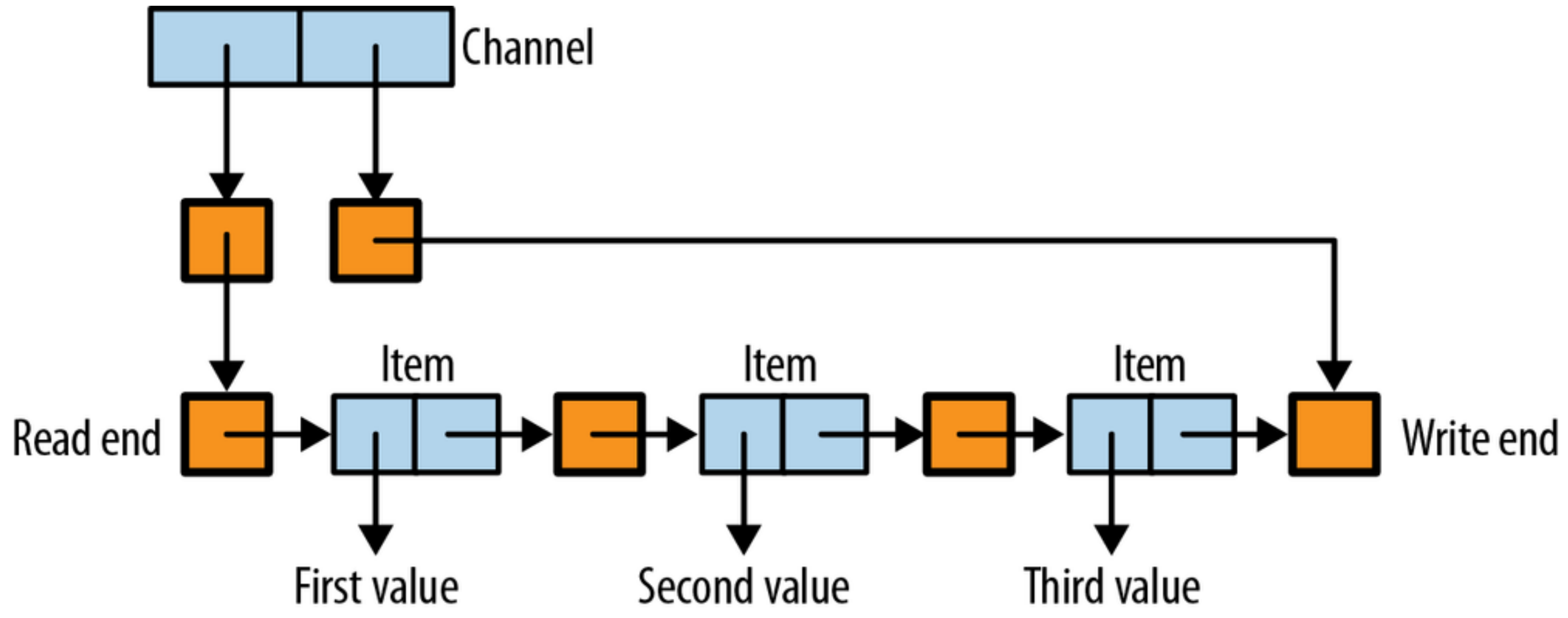
enqueue

- To add an element to the queue

  1. Make an item with a new hole

  2. Fill in the current hole to point to the new item

  3. Update the write end of the queue to point to the new item

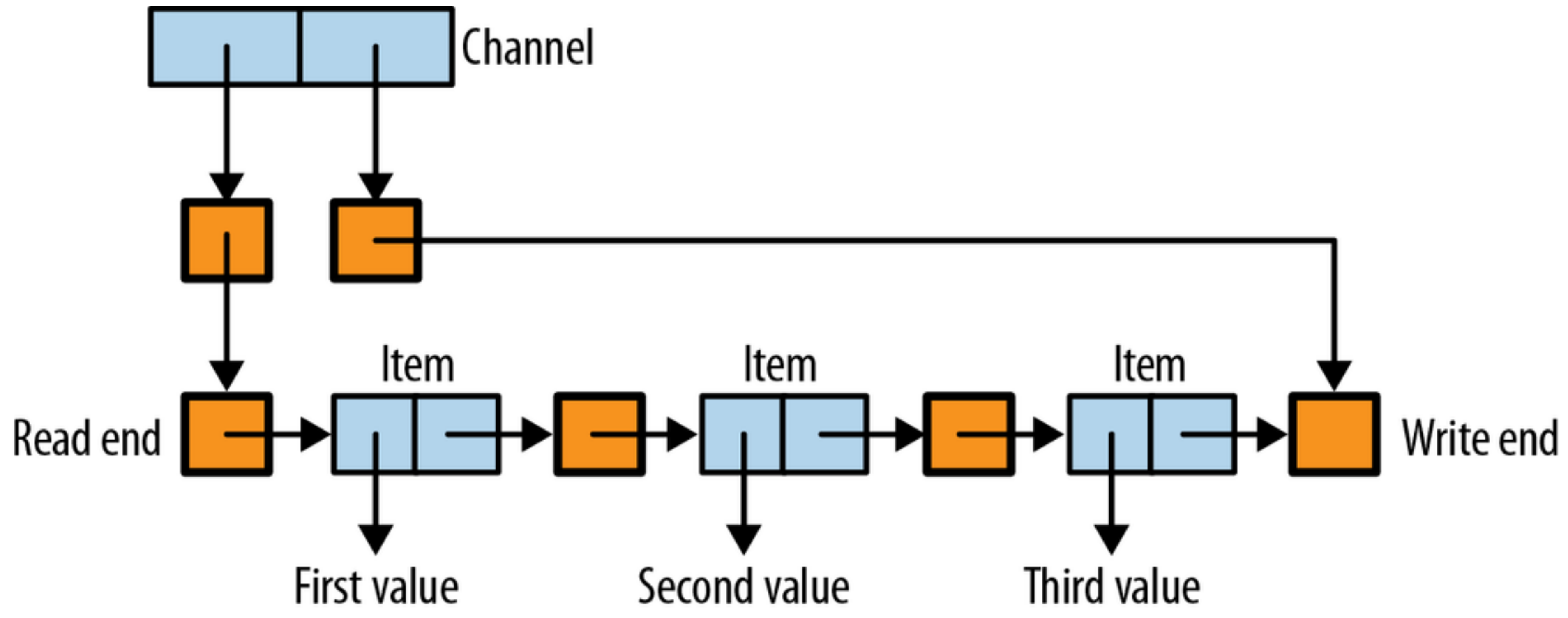


```haskell
enqueue :: Queue a -> a -> IO ()
enqueue (Queue _ writeLock) val = do
  newHole <- newEmptyMVar
  let item = Item val newHole
  oldHole <- takeMVar writeLock
  putMVar oldHole item
  putMVar writeLock newHole
```

dequeue

- To remove an element from the queue

    1. Follow the read end of the queue to the first item of the stream
    2. Get the first item
    3. Update the read end to point to the next item in the queue
    4. Return the value



```haskell
dequeue :: Queue a -> IO a
dequeue (Queue readLock _) = do
  -- try it yourself!
```

Ask yourself

- What is the behaviour for…
  - Multiple readers?
  - Multiple writers?
  - Concurrent reads and writes?

A note on fairness

- Is our queue *fair*?
  - i.e. no thread is starved of CPU time indefinitely

- Threads blocked on an `MVar` are woken up in FIFO order: single wakeup

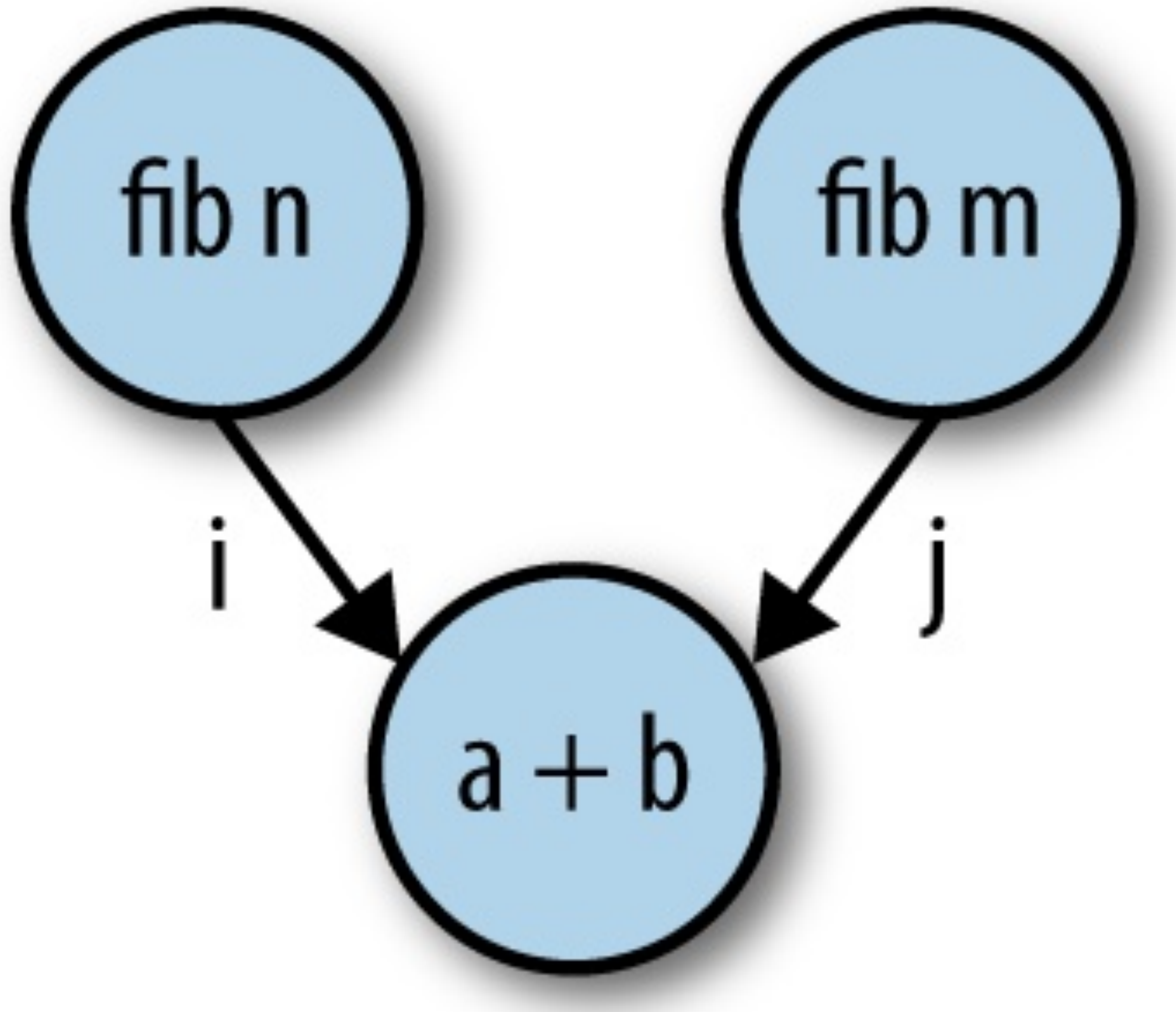
IORefs as a building block (1)

Dataflow computations

Regaining determinism

- The goal:
  - Compose a computation by specifying data-flow dependencies
  - Result should be deterministic
  - Example:

```
fib n | n < 2 = return 1
fib n         = do
  i <- new
  j <- new
  fork (fib (n-1) >>= put i)
  fork (fib (n-2) >>= put j)
  a <- get i
  b <- get j
  return (a + b)
```



Regaining determinism

- Data flow
  - Key idea: a non-deterministic result can only arise from a *choice* between multiple `puts`, so make that an error
  - Basic interface:

```
data IVar a = IVar (IORef (IVarContents a))
data IVarContents a
  = Empty
  | Full a
  | Blocked [a -> IO ()]

new   :: Par (IVar a)
fork  :: Par () -> Par ()
put   :: IVar a -> a -> Par ()
get   :: IVar a -> Par a
```

About `Par`:

- A monad, kind of like `IO` (it's built on `IO`)
- In `get`, we can "capture" the remainder of the computation in an `a -> IO ()`
- User can only use *our* chosen methods (`new`, `fork`, `put`, `get`)

Regaining determinism

- Non-determinism can only arise from a *choice* between multiple `puts`
  - Trying to `put` a value into a full `IVar` results in a runtime error
  - Reschedules any threads that were blocked waiting on this value

```haskell
data IVar a = IVar (IORef (IVarContents a))
data IVarContents a
  = Empty
  | Full a
  | Blocked [a -> IO ()]
```

```haskell
put :: IVar a -> a -> Par ()
put (IVar ref) !v = do
  liftIO $ do
    ks <- atomicModifyIORef' ref $ \old -> case old of
            Empty      -> (Full v, [])
            Blocked ks -> (Full v, ks)
            Full _     -> error "multiple put!"
    forM_ ks $ \k ->
      forkIO $ k v
```