ORACLE

# Reciprocating Locks

**Dave Dice & Alex Kogan**

Oracle Labs

# Background

- Topic : Mutual exclusion for cache-coherent (CC) systems
- Motivation
  - Throughput under contention - reduced coherence traffic
  - Practical – easy to integrate into existing systems  (vs CLH)
  - Gracefully hold many locks ; imbalanced lock sites
  - Willing to forego strict FIFO
  - Starvation avoidance - thwart DoS attacks on contended locks

# Related

- MCS : gold standard queue-based lock
- CLH :  simple ; no explicit linked list – no “Next” pointers
- HemLock in SPAA 2021
- Chen & Huang TPDS 2009
  - “Bounded-Bypass Mutual Exclusion with Minimum Number of Registers”
  - Forms and detaches stack of waiting threads with atomic exchange
  - Global spinning
  - “Large” atomic variables

# Key Primitive

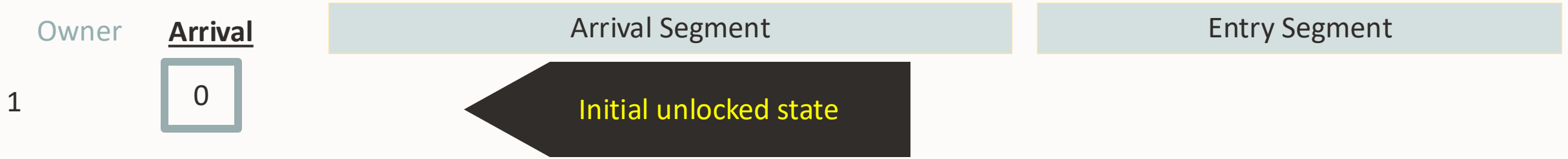
- Concurrent ***pop-stack***
- Push individual item with atomic exchange
- Detach entire stack with atomic exchange
- “items” are addresses of MCS-like queue elements
- Elements represent waiting threads – local spinning
- Each thread knows only immediate predecessor in stack
  Returned from exchange operator

# Informally

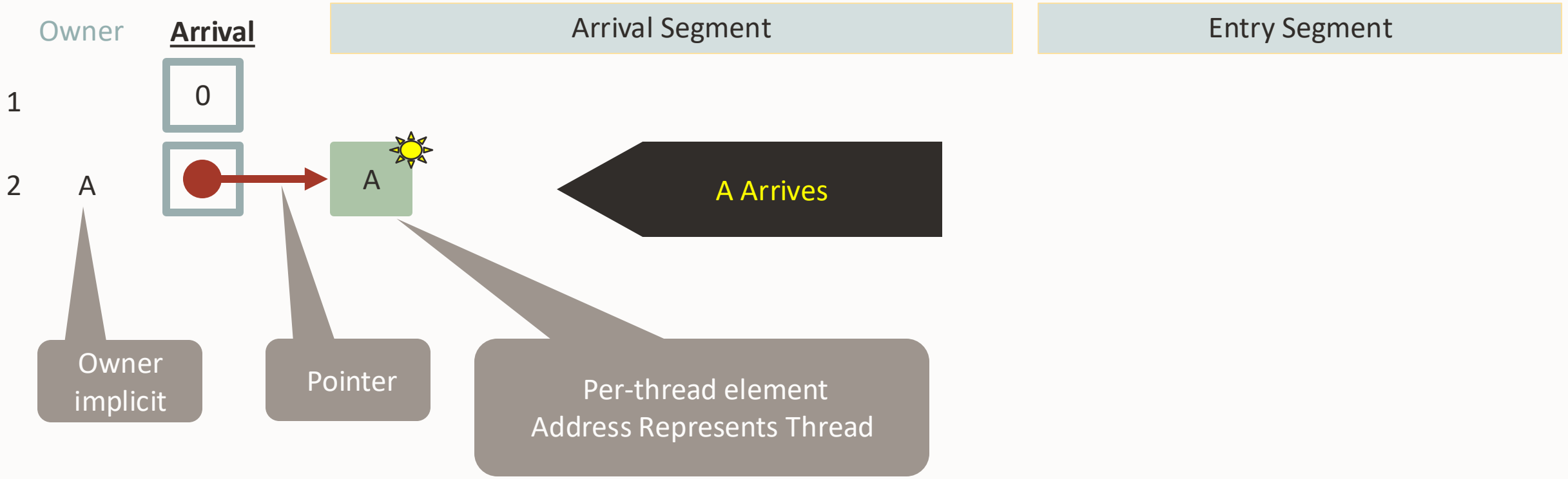
- Arrival segment and Entry segment – lists of waiting threads
- Arriving threads push onto Arrival Segment with exchange
- Lock instance consists of Arrival Word :
  - 0 = Unlocked
  - 1 = Locked with empty Arrival Segment
  - T = Locked where T is most recent arrival (Arrival Segment – ToS)

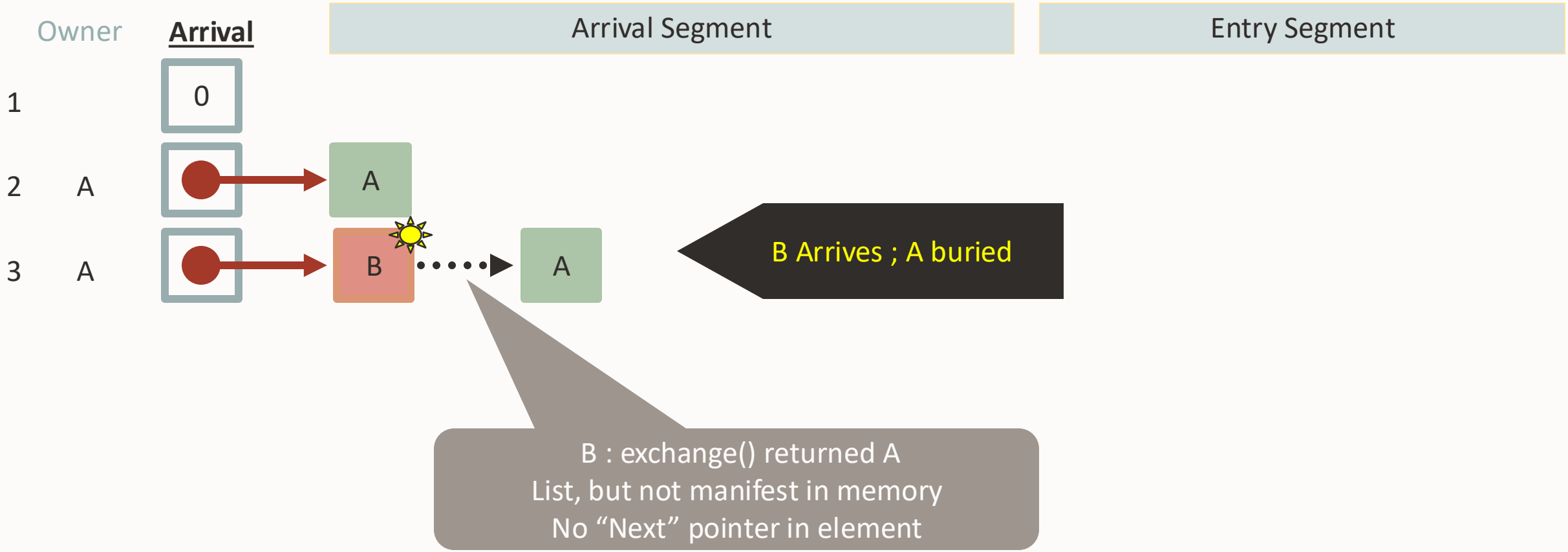
# Informally

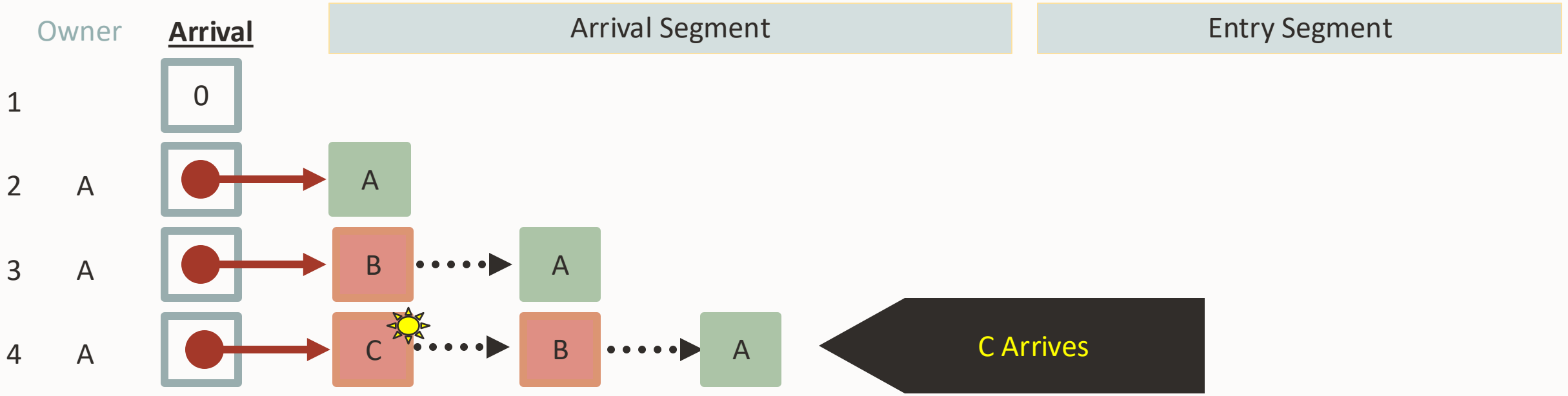
- To Acquire :
  - Push onto Arrival word : 0 ⟶ uncontended
  - Local spinning on element
- To Release : 3 cases
  - Pass ownership to neighbor on Entry Segment, if any
  - Detach Arrivals which become next Entry Segment – exchange(1)
  - Try to switch Arrival word back to unlocked state with CAS
- Invariant : All elements on Entry Segment arrived before any elements on Arrival Segment

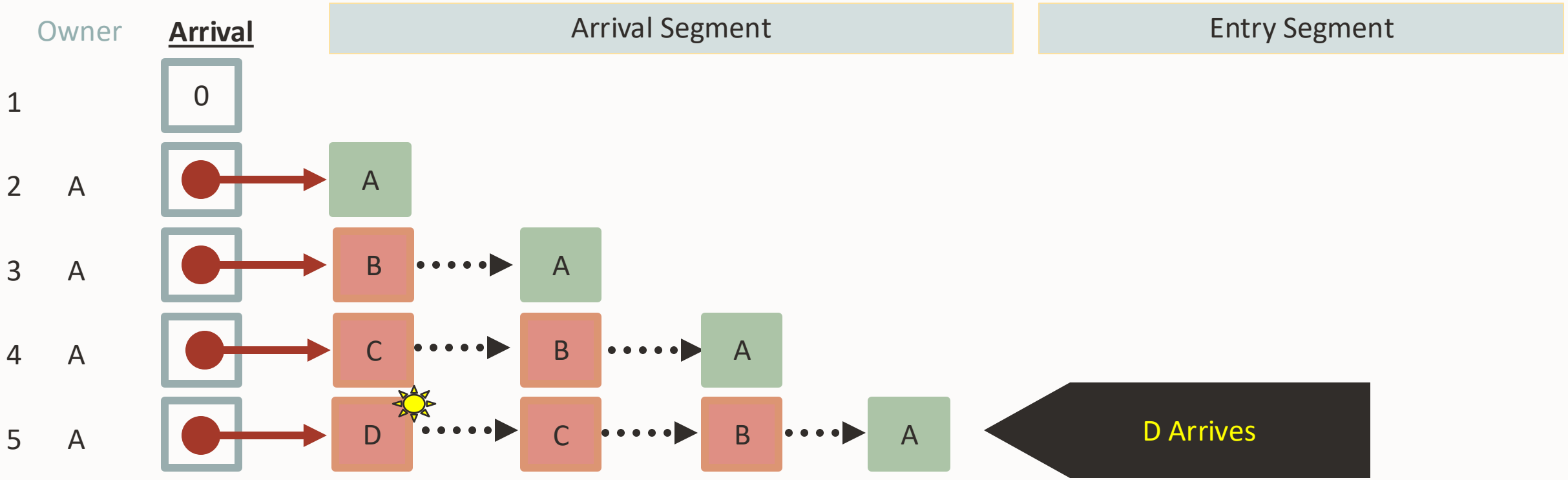


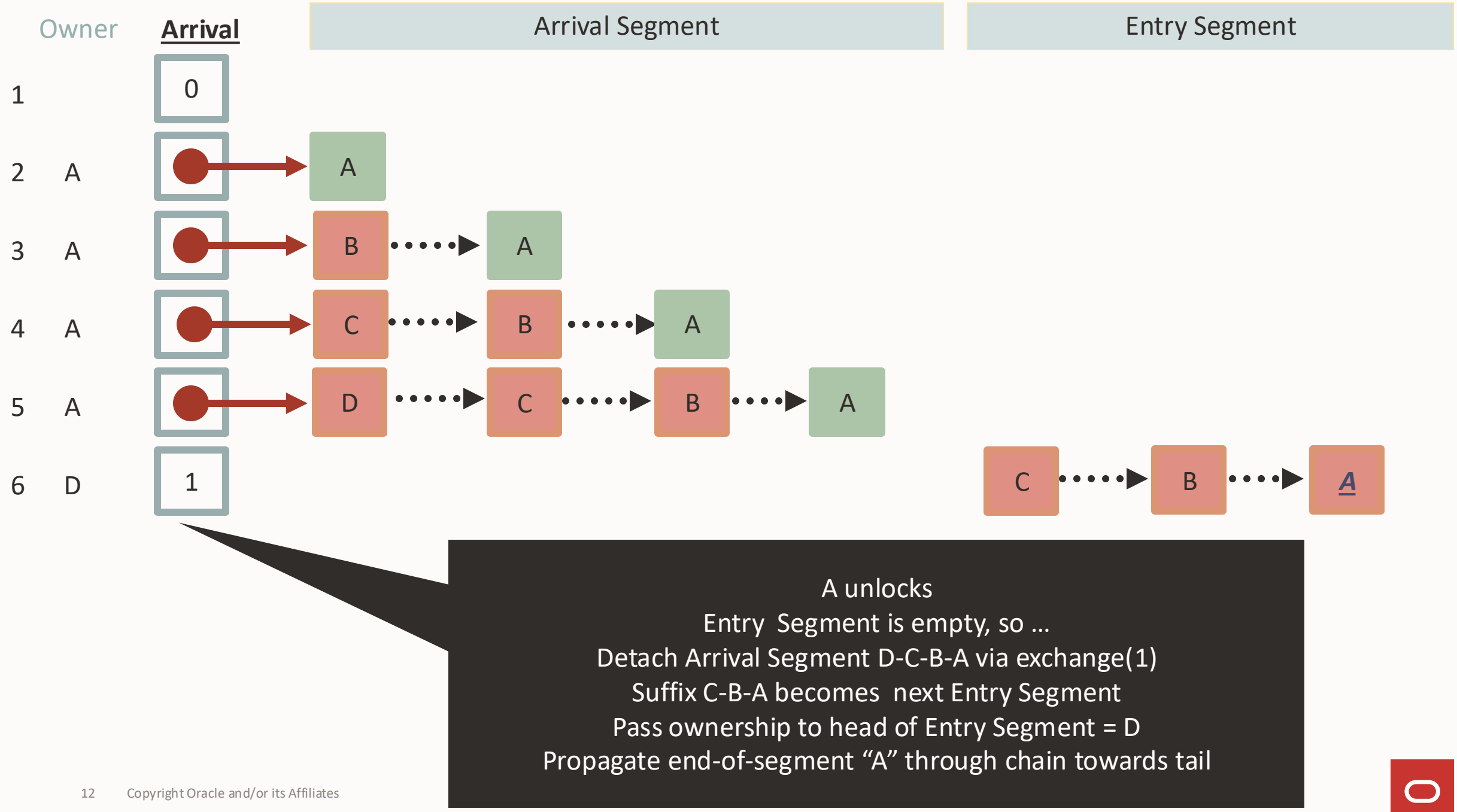

# Reciprocating Locks in Action

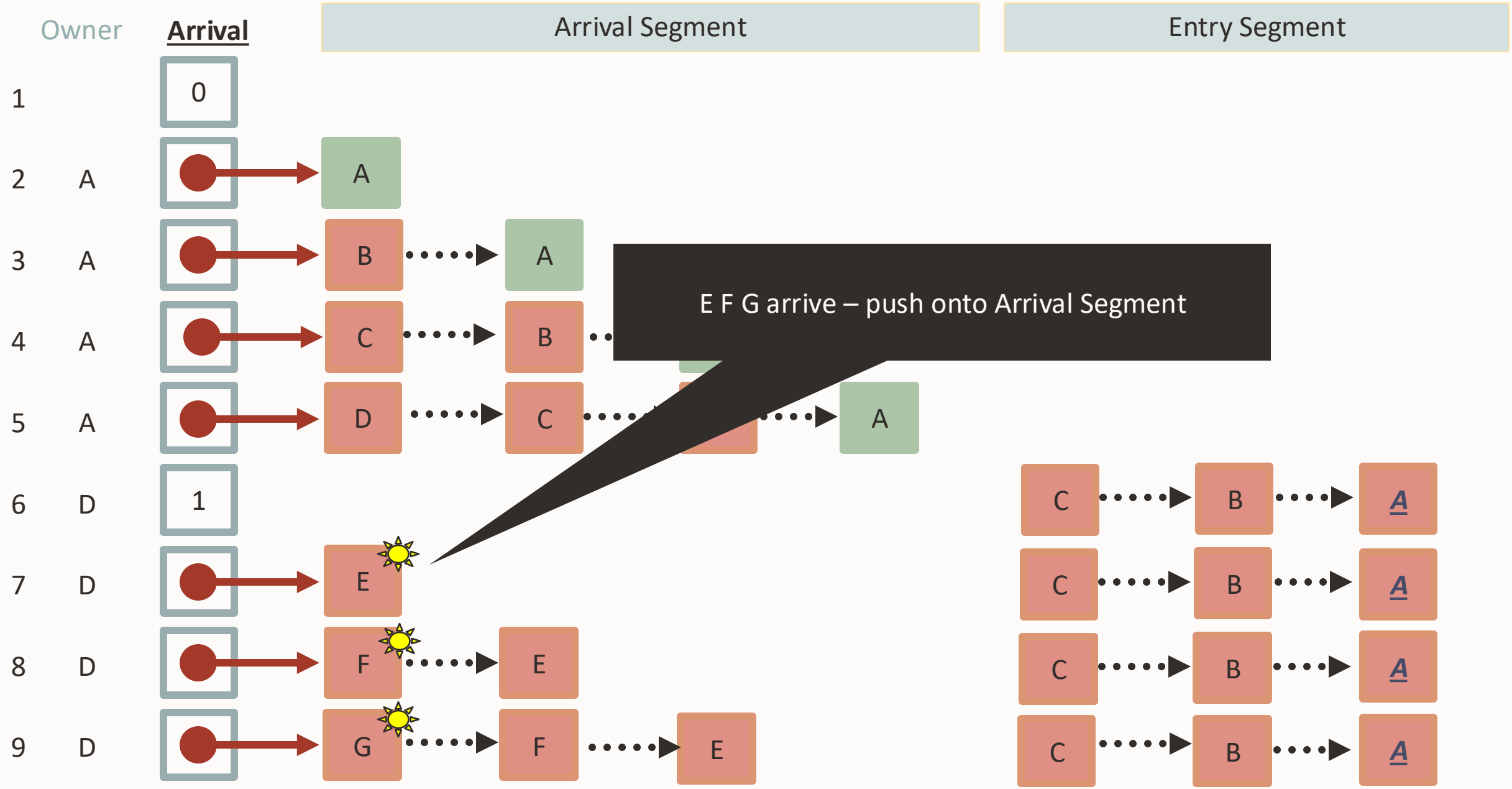


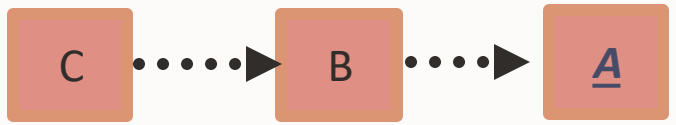


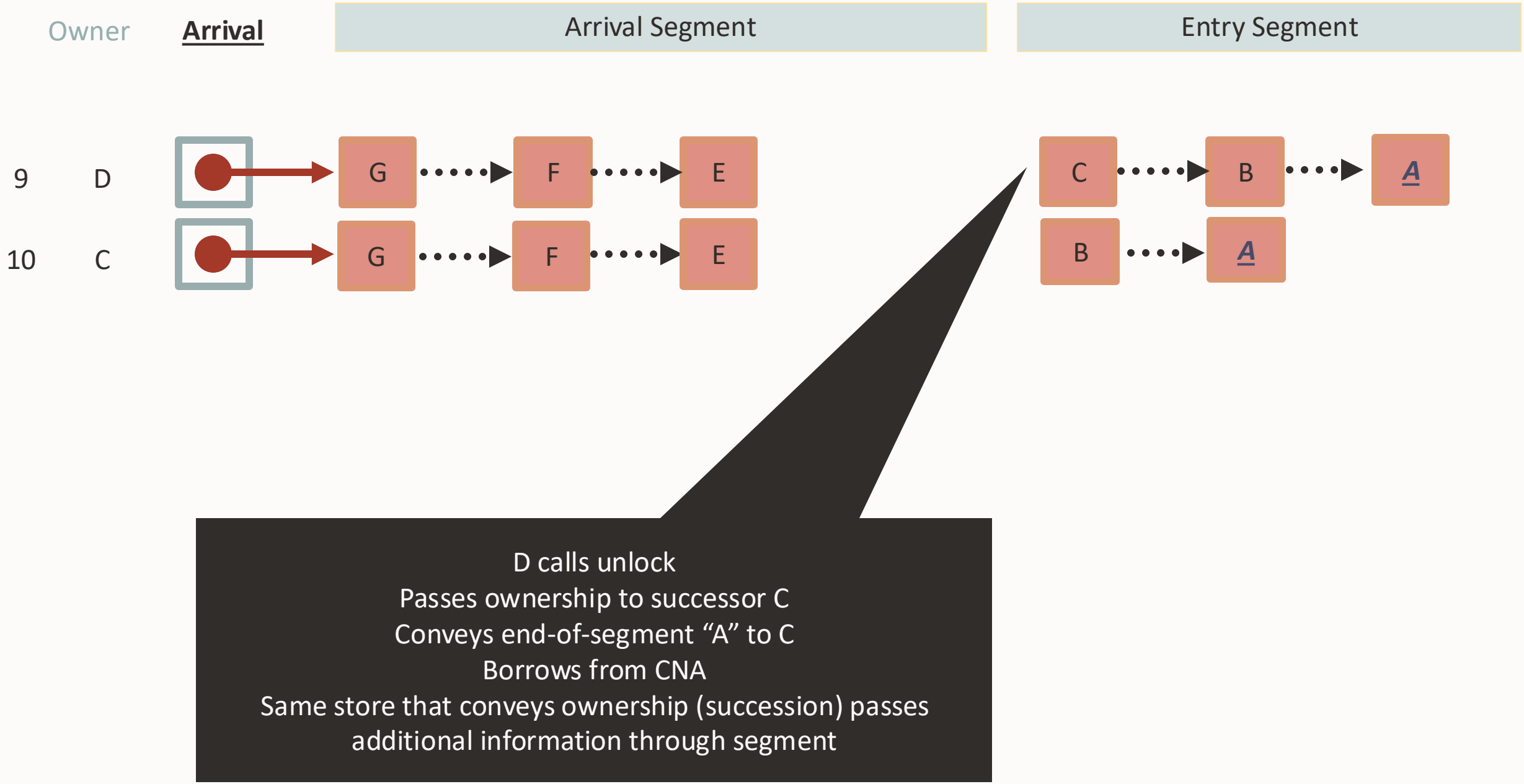


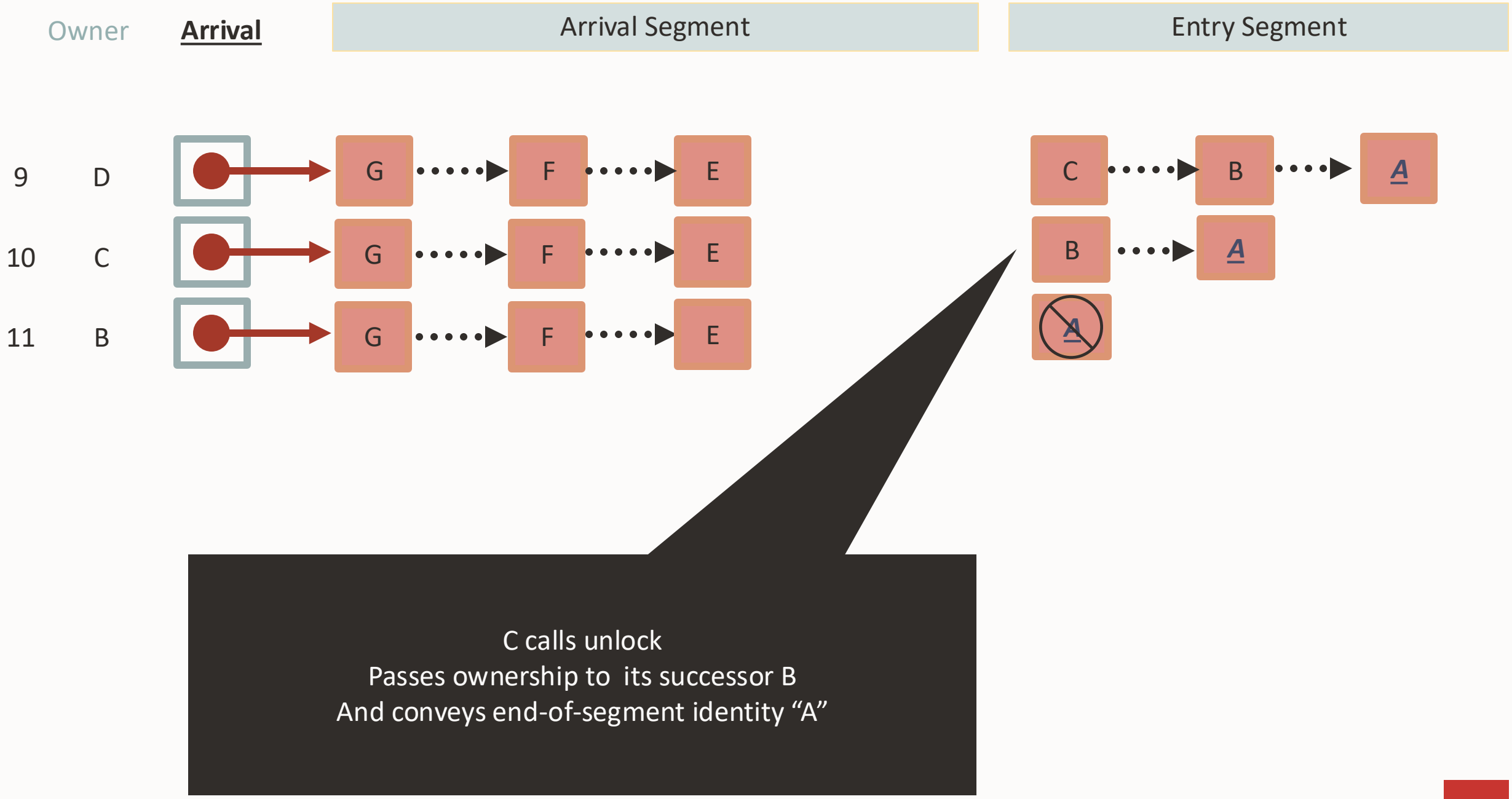


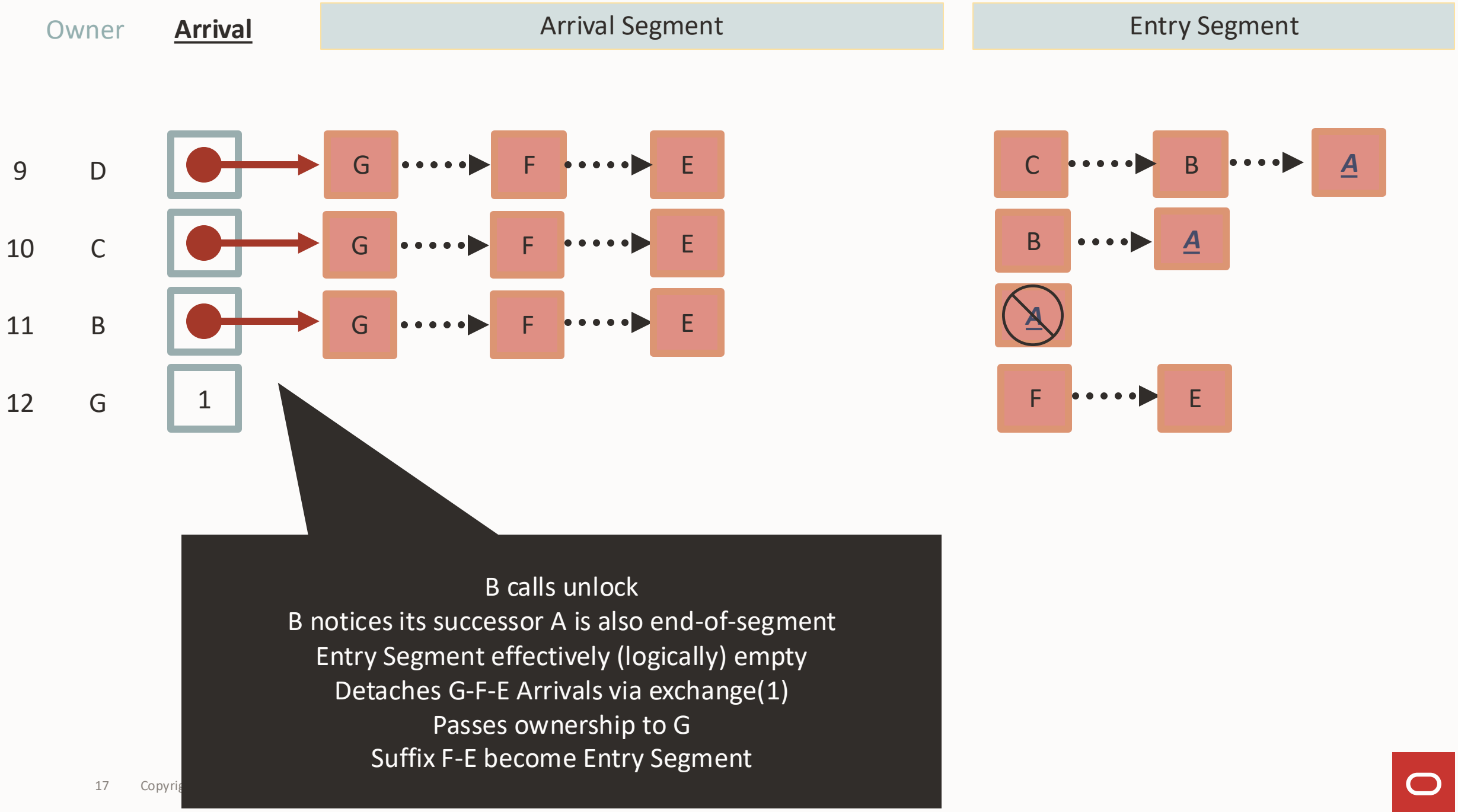


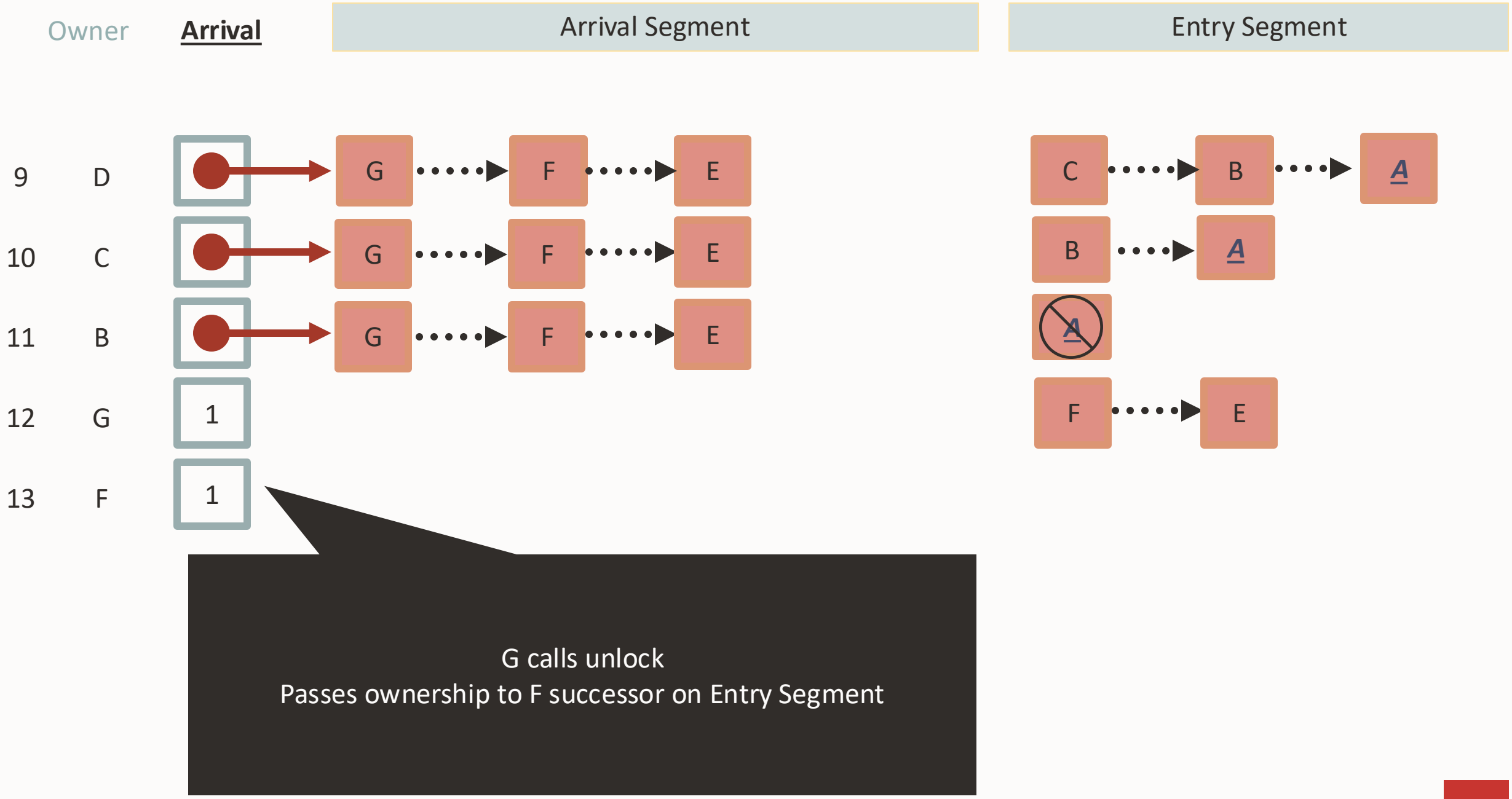


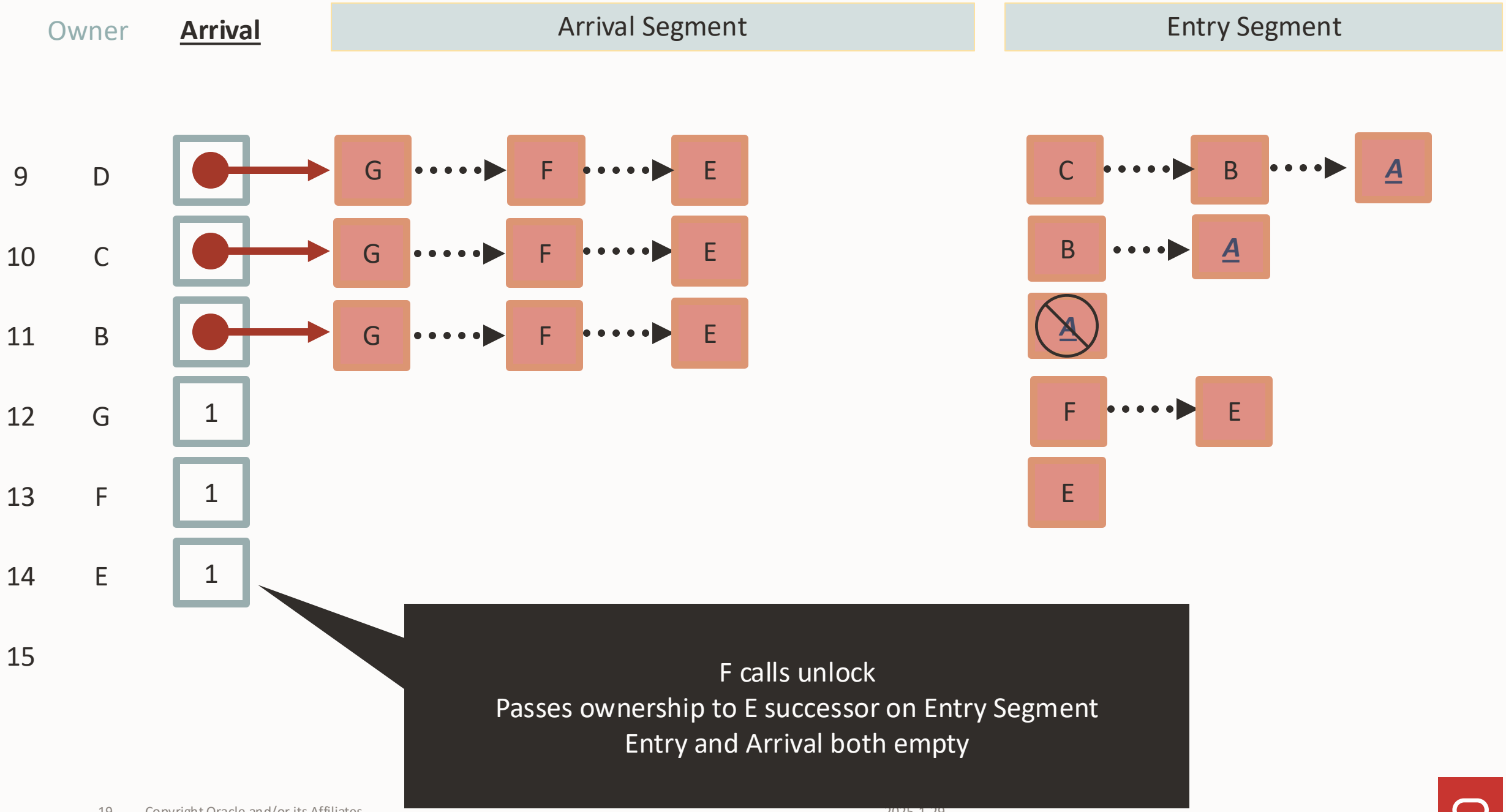
| Owner | **Arrival** | Arrival Segment | Entry Segment |
| --- | --- | --- | --- |
| 9 | D |  |  |

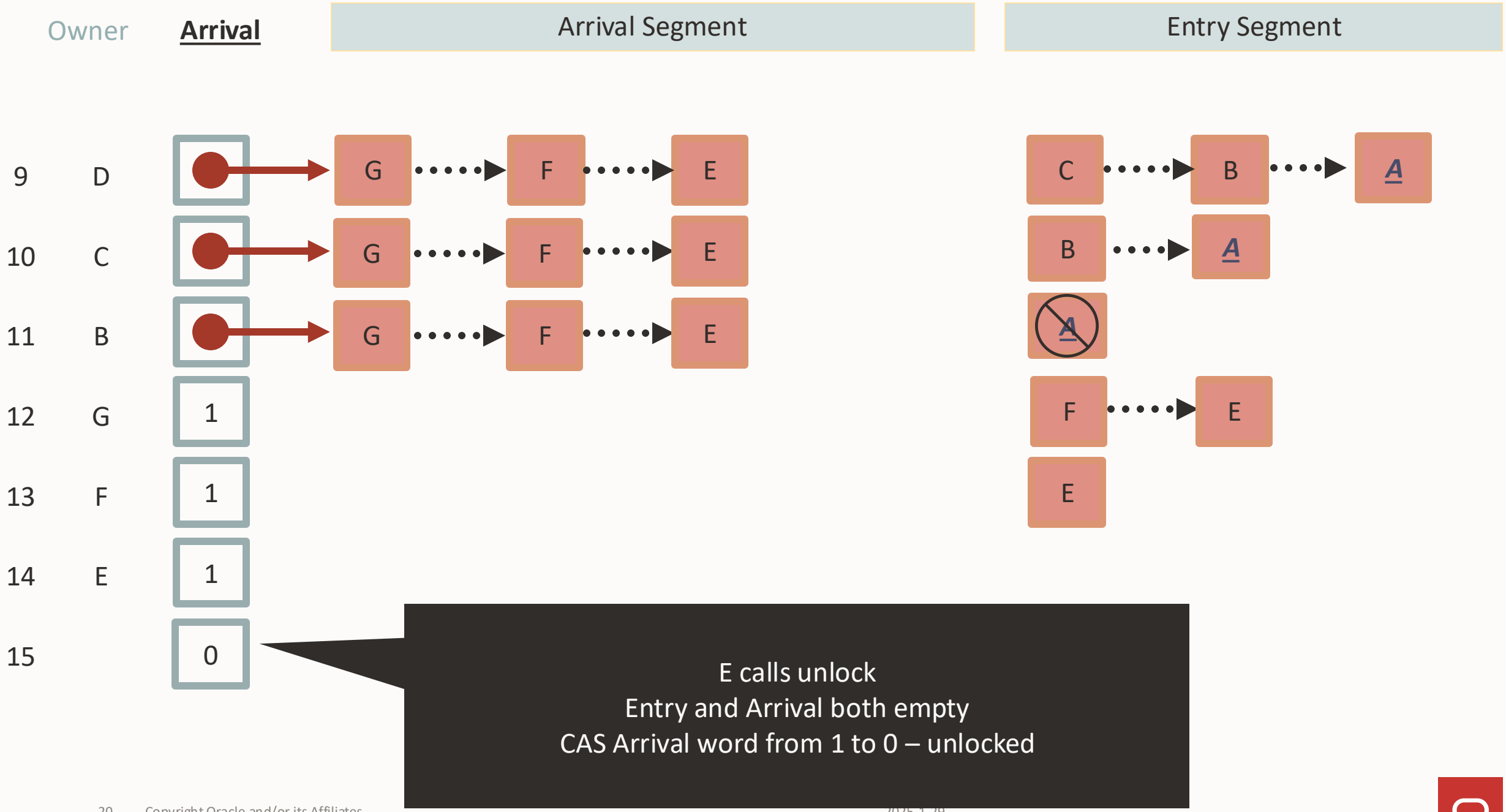





Owner
Arrival
Arrival Segment
Entry Segment
9
D
10
C
11
B
12
G
1
G
F
E
C
B
A
B
A
A
F
E
B calls unlock
B notices its successor A is also end-of-segment
Entry Segment effectively (logically) empty
Detaches G-F-E Arrivals via exchange(1)
Passes ownership to G
Suffix F-E become Entry Segment







# Advantages

- **Constant-time** arrival “doorway” and unlock paths
- Elements “queue nodes” don’t circulate  -- like MCS, unlike CLH
- Local spinning  on local memory
  - Helpful for home-based coherence (Intel UPI fabric)
  - Passively NUMA-friendly but not NUMA-aware
- Amenable to modern waiting :  WFE | MONITOR-MWAIT | Futex
  - Unlike HemLock
- No explicit lists in segments -- like CLH
- **Bounded Bypass** : avoid starvation
  - Entry segment is LIFO but arrival segments FIFO
  - If T is waiting then S > T can bypass T at most once per episode

# Advantages

- Easy to integrate or retrofit under existing APIs – practical
- Thread needs at most one element
  - Regardless of the number of locks they hold
  - Singleton in TLS or on-stack  (arXiv)
  - Avoid lifecycle management issues
  - Tight space bounds
- Throughput  and scalability





# Advantages

- Improved Throughput  -- why ?
- Reduced **coherence traffic** relative to MCS or CLH
- Conserves interconnect bandwidth and reduces latency
- Fewer coherence misses per acquire-release episode
- Fewer misses to remote memory
  - Helps if home-based coherence  : Intel UPI
  - Coherence miss to a line homed locally to requester may be faster





# Experiment – Coherence Traffic

- Ad-hoc contention benchmark with single lock
- Non-critical section : empty
- Critical section : advance local PRNG 100 steps – no shared accesses
- Runs fully in CPU-local L2 cache when solo
- Multiple threads : Only coherence misses are from the lock operation itself
- All misses are coherence misses induced by lock algorithm
- Configure for sustained contention
- Linux “perf stat” to read hardware performance counters
- Tally coherence traffic per acquire-release episode
- Ensure performance counter ratios and static analysis agree
- Compare CLH and Reciprocating Locks …
- Abridged : Show only paths and shared accesses used under contention

# Coherence traffic under sustained Contention

| | CLH | CT |
|---|---|---|
| Lock | auto E = TLS | |
| | E➞Gate = 0 | ✗ |
| | auto pred = L➞Tail.exchange(E) | ✗ |
| | TLS = pred | |
| | while pred➞Gate == 0 : Pause | ✗ ✗ |
| | <CriticalSection> | |
| unlock | E➞Gate = 1 | ✗ |

| | Reciprocating | CT |
|---|---|---|
| Lock | TLS.Gate = 0 | X |
| | auto succ = L➞Arrivals.exchange(&TLS) | ✗ |
| | while TLS.Gate == 0 : Pause | X |
| | <CriticalSection> | |
| unlock | succ➞Gate = NonZero | ✗ |

Local

Remote

Fewer CPU stalls
Less Indirection
Less pointer chasing

ARMv8 l2c_cache_inval or hnf_snp_sent 5:4 ratio

# Disadvantages

- Branch-y paths : 3 cases in unlock
  - Branch predictor misses
  - Code complexity higher than MCS or CLH
- More accesses to central arrival word to reprovision Entry Segment than MCS | CLH
- Not FIFO
- Long-term admission unfairness
  - “Palindromic” schedule
  - Long-standing repeating admission schedules
  - Some threads can be admitted **twice** as often – 2X worst-case bound





# Future

- Address long-term fairness concerns
  - Impose statistical long-term admission fairness
  - Randomize order within Entry Segment  - maintains bounded bypass property
  - Perturb palindromic schedule – will converge to new unfair mode
  - Slight periodic perturbations suffice
- Improved encoding of Arrival word simplifies unlock path
  - 2 cases instead of 3 – reduced path complexity
  - Atomic fetch_add()  using low order 2 bits of Arrival word as a tag to encode state
  - Described in arXiv long-form
- Apply ”CTR” Coherence Traffic Reduction optimizations from HemLock
  - Local spinning with atomic exchange or fetch_add() instead of load()
  - Remove 1 additional coherence transaction from contended acquire-release episode

# Finis





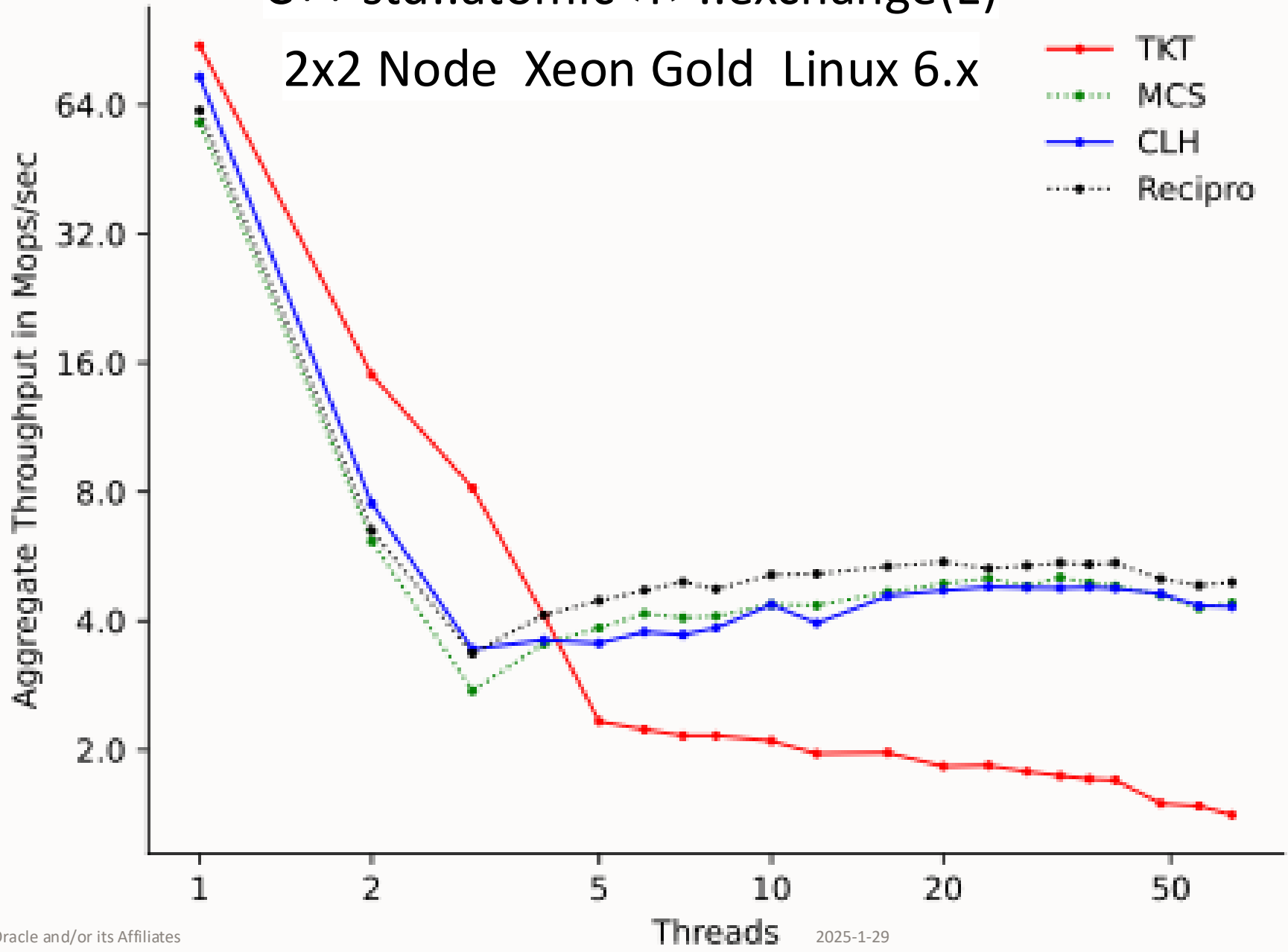
# BACKUP



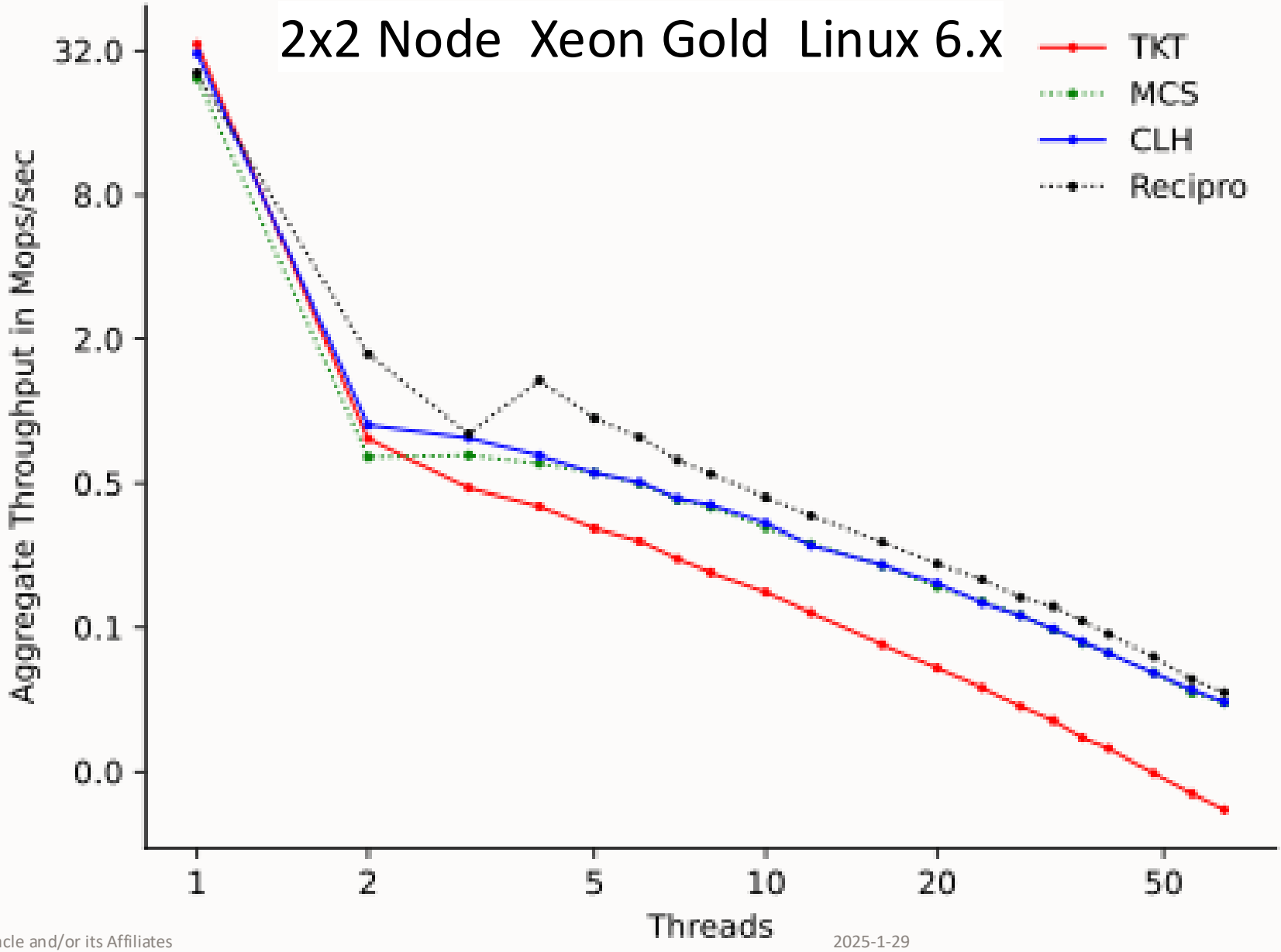


# C++ std::atomic<T>::exchange(E)

## 2x2 Node Xeon Gold Linux 6.x



# C++ std::atomic<T>::compare_and_exchange(E)



# Coherence traffic under sustained Contention

| | CLH | CT |
|---|---|---|
| Lock | auto E = TLS | |
| | E→Gate = 0 | ↑ |
| | auto pred = L→Tail.exchange(E) | ✗ |
| | TLS = pred | |
| | while pred→Gate == 0 : Pause | ✗ ✗ |
| | <CriticalSection> | |
| unlock | E→Gate = 1 | ✗ |

| | Reciprocating | CT |
|---|---|---|
| Lock | TLS.Gate = 0 | ↑ Upgrade |
| | auto succ = L→Arrivals.exchange(&TLS) | ✗ Remote |
| | while TLS.Gate == 0 : Pause | X Local |
| | <CriticalSection> | |
| unlock | succ→Gate = NonZero | ✗ |

Fewer CPU stalls
Less Indirection
Less pointer chasing

ARMv8 l2c_cache_inval or hnf_snp_sent 5:4 ratio

# Historical Origins

- HotSpot JVM objectMonitor.cpp : Contention queue and Entry segment
- 24+ years old
- https://github.com/openjdk/jdk/blob/master/src/hotspot/share/runtime/objectMonitor.cpp
- https://github.com/openjdk/jdk/blob/785e7b47e05a4c6a2b28a16221fbeaa74db4db7d/src/hotspot/share/runtime/objectMonitor.cpp#L183
- Only recently realized we could make it constant-time
- Written for SPARC : no 64-bit atomic exchange – CAS loop





# Drivers of change for synchronization  (why new locks?)

- Software
  - DoS Attacks
  - More flexible locking APIs (C++) allow more algorithms
    Less constraints implies more latitude
  - Maximum dispersal placement policy in Linux 6.x scheduler
    NUMA effects appear at low thread counts
    Favors Reciprocating Locks
  - Greenfield :  Python ?  GIL removal and JIT means synchronization becomes important
- Economic : less concern about power-awareness





# Drivers of change for synchronization

- Hardware
  - More cores on die : broke “16” barrier, now 128 and increasing
  - Waiting mechanisms : WFE ; MONITOR-MWAIT
  - ARM ecosystem evolves away from LL-SC (deeply unfair) with LSE (fairer)
  - ARM and RISC-V : weak memory models; fence efficiency
  - Apple M silicon entirely different world
  - Asymmetric MP : P/E ; Fire/Ice; Big/Small
  - Spectre mitigation impacts synchronization economics
  - Coherence
    - UPI (vs QPI) home-based coherence
    - On-die mesh
    - On-chip NUMA : Cluster-on-Die
    - MESI vs MOESI

# Backup : Modern coherent interconnects

- RMR complexity : useful but we want refined measures that better aligned with modern hardware
- NUMA vs NUCA distinction
- MESI  MESIF (Intel) MOESI (AMD)
- Invalidation diameter of a store
- Distance in hops : local vs remote

# Backup : Modern coherent interconnects

- Simple :
  - Performance governed by location of requester vs location of cache(s) that hold that line
  - Number of caches ; location of caches; state in those caches
  - Don’t care about NUMA home location of a cache line except for cold miss
    Home location not relevant to performance
- Home-based coherence : intel
  - Home ”node” of cache line handles coherence probes – moderates
  - More latency and hops but reduced bandwidth – avoid broadcast snoops
  - Home location vs requester location becomes a concern
  - Performance : requester location; what caches; what states **+** home location
  - CLH element migration particularly undesirable
  - Local misses (home = requester) vs remote misses

# Miscellany

- Borrows from CNA :
  - Same store that conveys ownership also propagates end-of-segment address toward tail of entry segment
- Modern linux scheduler placement policy :
  - Maximum dispersal over NUMA nodes
  - Round-robin equidistribution
  - Advantage for reciprocating locks appears with even small number of threads

# HemLock - disadvantages

- Unlock() path is not constant-time
  - Unlocking() thread waits positive ack from successor
  - Need to know that ”mailbox” is available for re-use
  - Can mitigate by using multiple per thread
- Multi-waiting
  - Rare, but results in unbounded theoretical RMR complexity