COL380

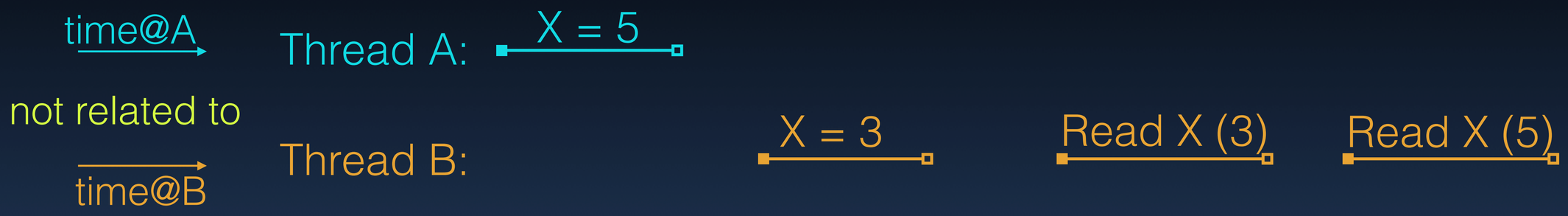
Introduction to
Parallel & Distributed Programming

Sequential Consistency

"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

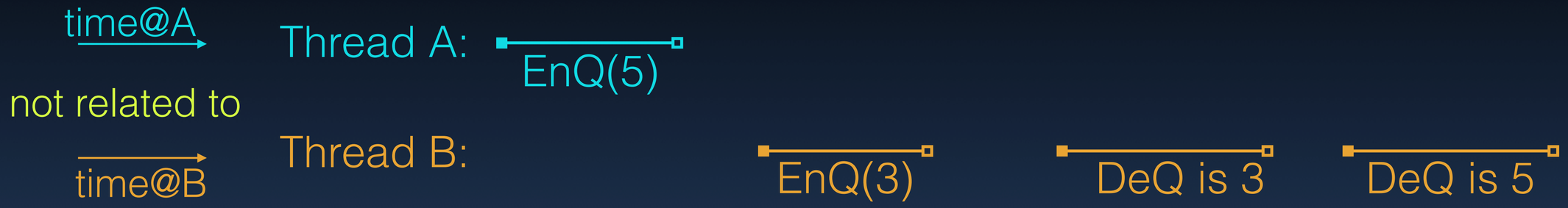
Weaker than Linearizability
Does not depend on global time
(Still not efficient to guarantee.)

Sequentially Consistent



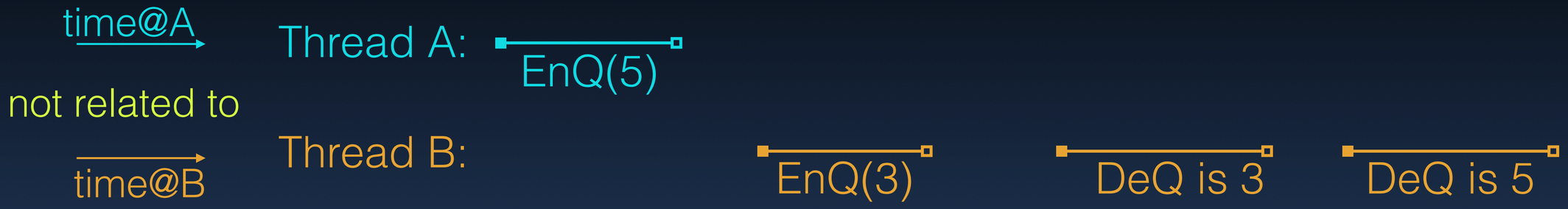
- No global notion of time
  - ➡ Only consistent Order

Sequentially Consistent



- No global notion of time
  - ➡ Only consistent Order

Sequentially Consistent



- No global notion of time

  ➡ Only consistent Order

Thread A: $A_1 \longrightarrow A_2$

Thread B: $B_1 \longrightarrow B_2$

Thread C: $C_1 \longrightarrow C_2 \longrightarrow C_3$

Sequential History

$A_1 \rightarrow A_2 \quad C_1 \rightarrow C_2 \rightarrow C_3 \quad B_1 \rightarrow B_2$

$A_1 \quad C_1 \quad A_2 \quad B_1 \rightarrow B_2 \quad C_2 \rightarrow C_3$

Sequentially Consistent

time@A

not related to

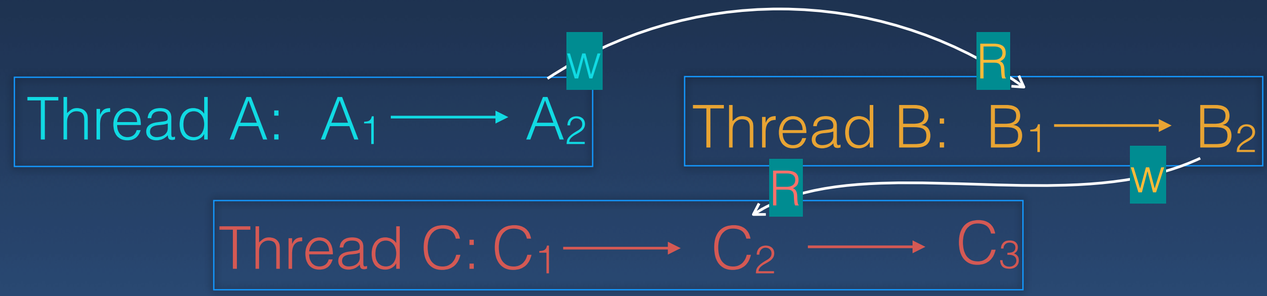
time@B

Thread A: EnQ(5)

Thread B: EnQ(3) DeQ is 3 DeQ is 5

- No global notion of time

  ➡ Only consistent Order



Sequential History

$A_1 \rightarrow A_2$ $C_1 \rightarrow C_2 \rightarrow C_3$ $B_1 \rightarrow B_2$

A_1 C_1 $A_2 \rightarrow B_1 \rightarrow B_2 \rightarrow C_2 \rightarrow C_3$

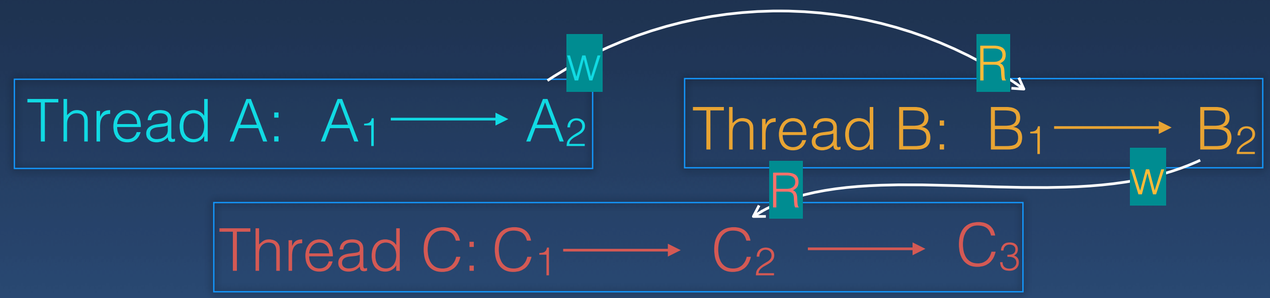
Sequentially Consistent

time@A → not related to → time@B

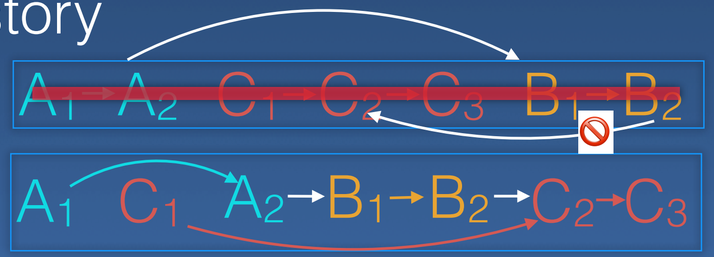
Thread A: EnQ(5)

Thread B: EnQ(3) DeQ is 3 DeQ is 5

- No global notion of time
  - ➡ Only consistent Order



Sequential History



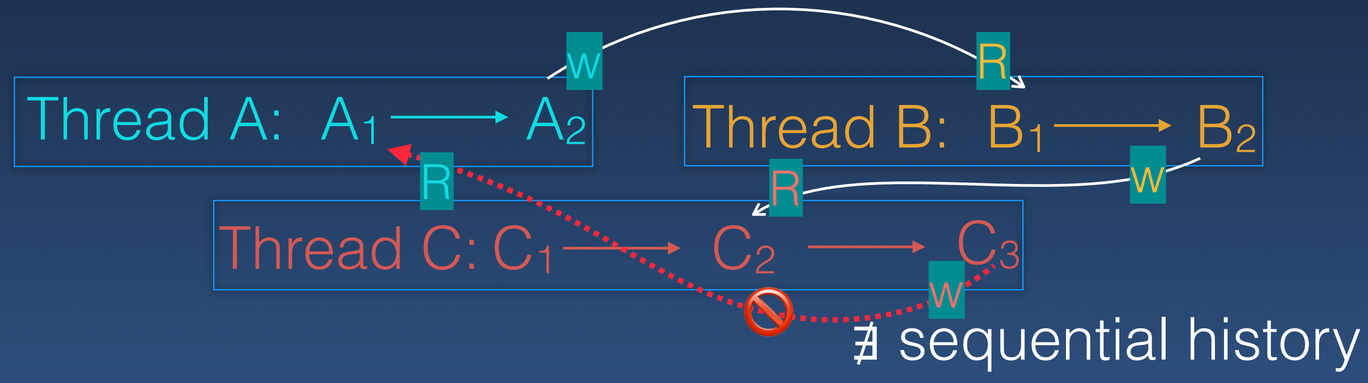
Sequentially Consistent

time@A → not related to → time@B

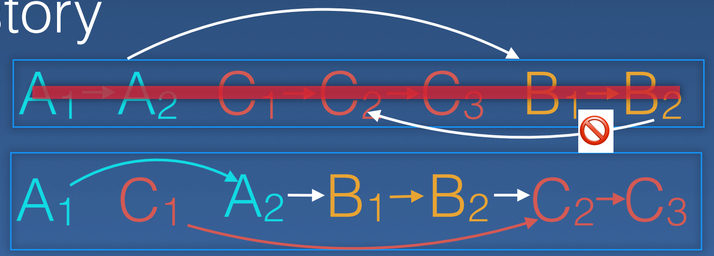
Thread A: EnQ(5)

Thread B: EnQ(3) DeQ is 3 DeQ is 5

- No global notion of time
  - ➡ Only consistent Order



Sequential History



Sequentially Consistent

time@A

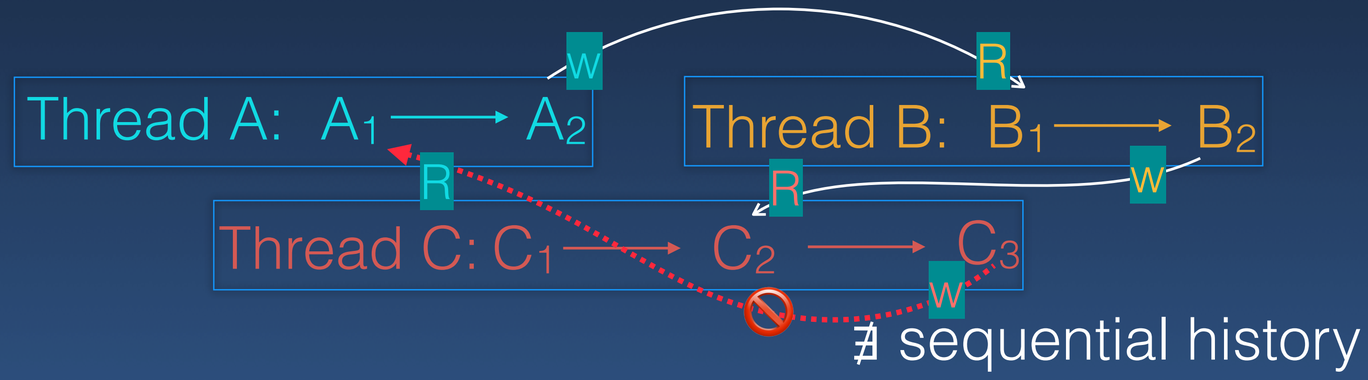
not related to

time@B

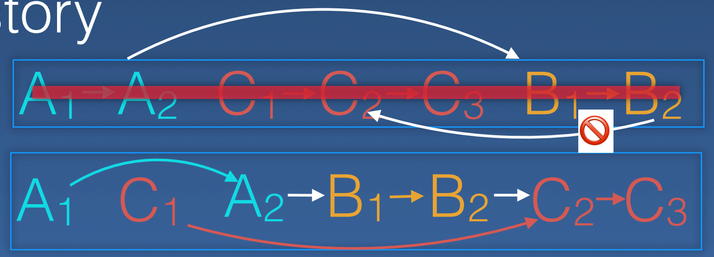
Thread A: EnQ(5)

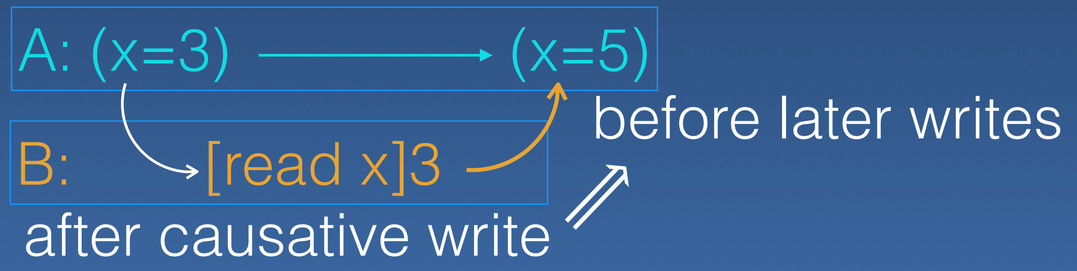
Thread B: EnQ(3) DeQ is 3 DeQ is 5

- No global notion of time
  - ➡ Only consistent Order



Sequential History





Subodh Kumar

Applying Sequential Consistency

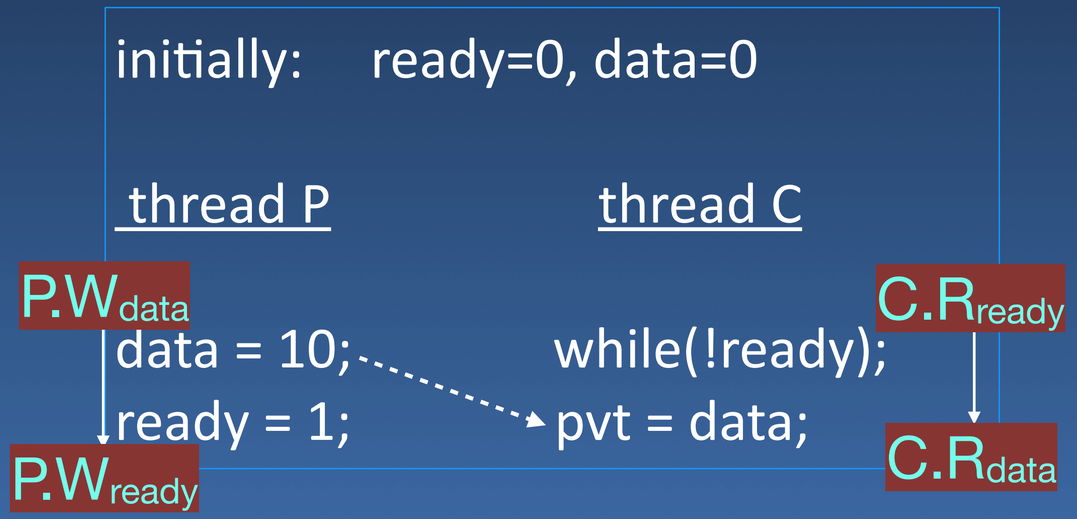
- Threads always see values written by some thread
  - ➡ No garbage (update is atomic)
- The value seen is constrained by thread-order
  - ➡ for every thread

initially: ready=0, data=0

thread P	thread C
data = 10;	while(!ready);
ready = 1;	pvt = data;

Applying Sequential Consistency

- Threads always see values written by some thread
  - ➡ No garbage (update is atomic)
- The value seen is constrained by thread-order
  - ➡ for every thread



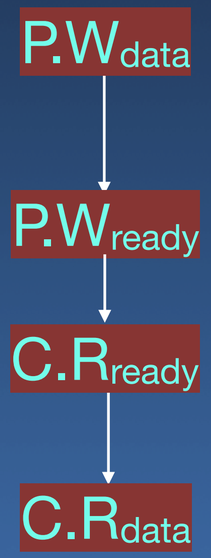
Show: If C sees the updated ready (=1), C must also see the updated data (=10)

Applying Sequential Consistency

- Threads always see values written by some thread
  - ➡ No garbage (update is atomic)
- The value seen is constrained by thread-order
  - ➡ for every thread

initially: ready=0, data=0

thread P	thread C
data = 10;	while(!ready);
ready = 1;	pvt = data;

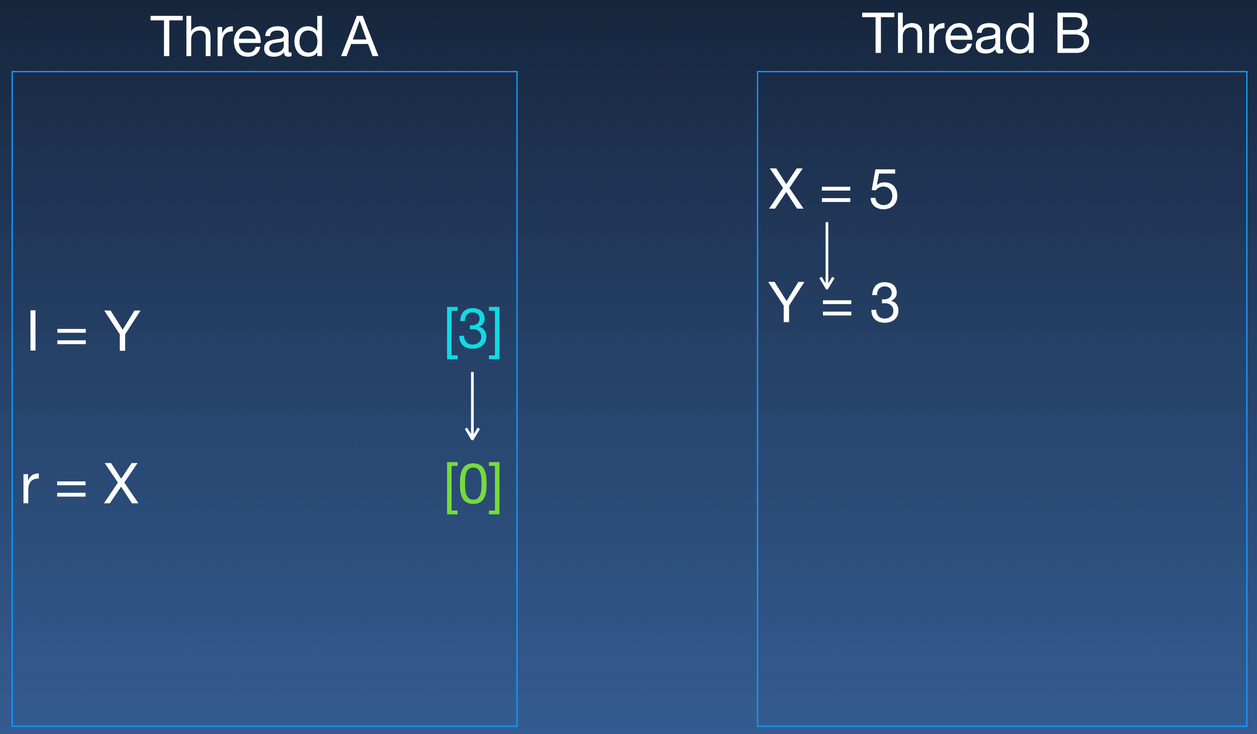


If C sees ready =	then C sees data =
0	0 or 10
1	10

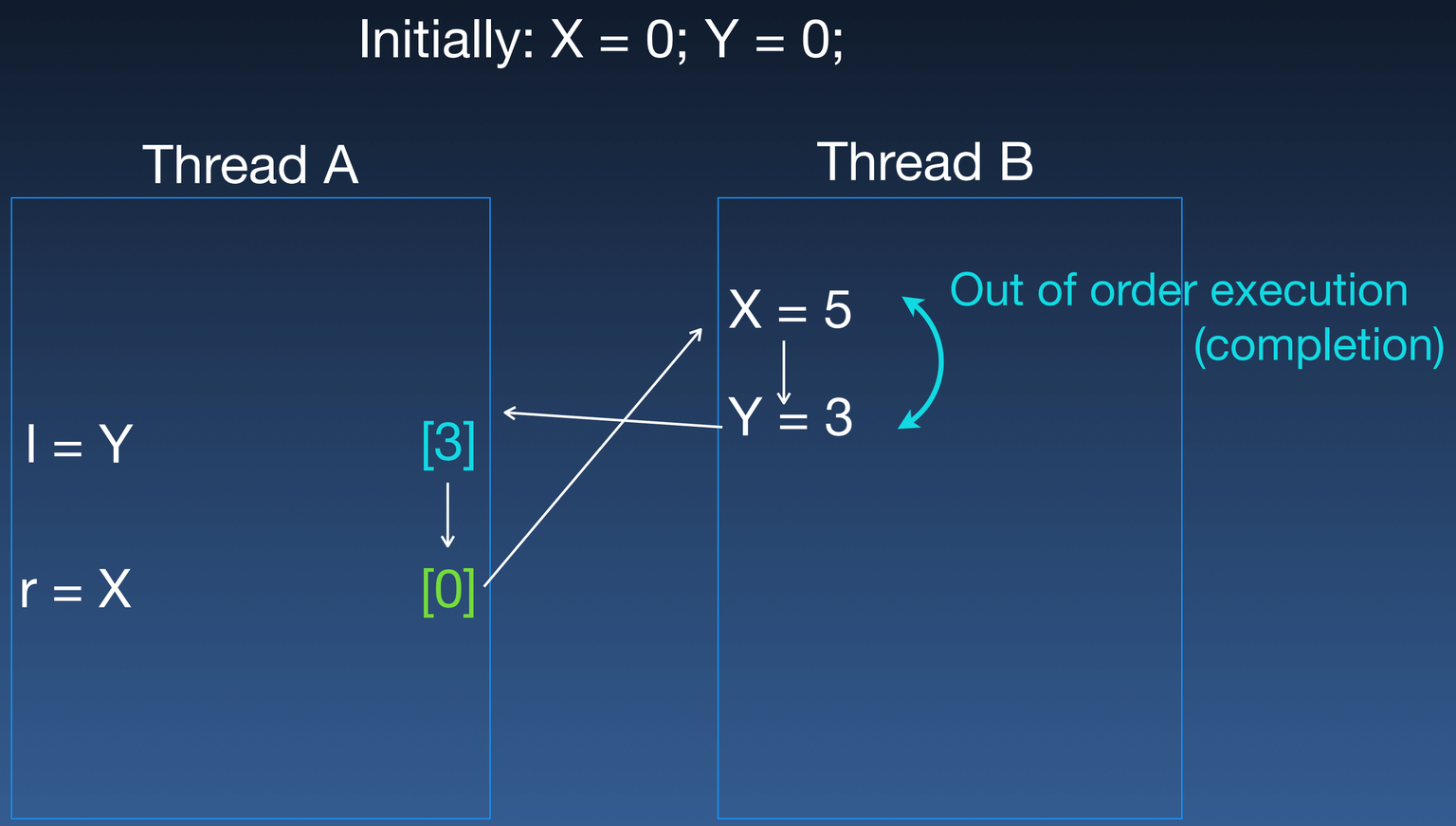
Show: If C sees the updated ready (=1), C must also see the updated data (=10)

Coherence *vs* Consistency

Initially: X = 0; Y = 0;

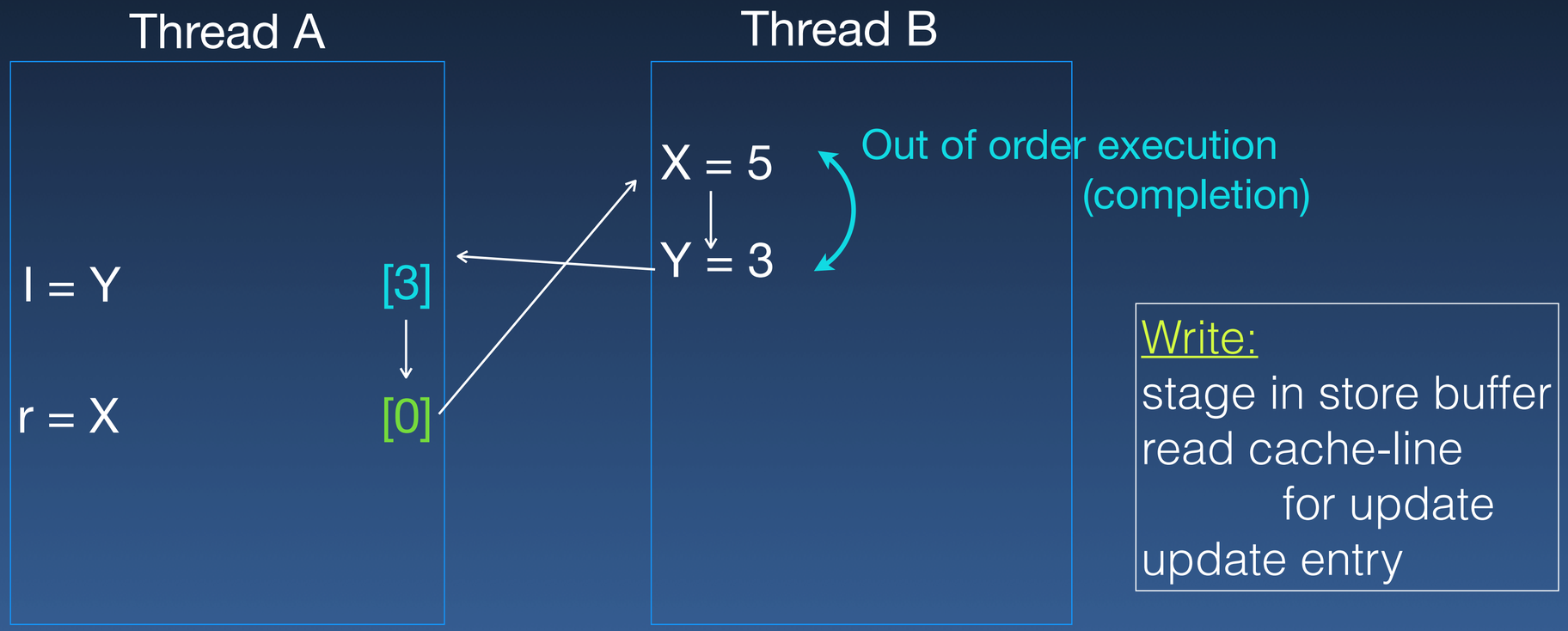


Coherence *vs* Consistency



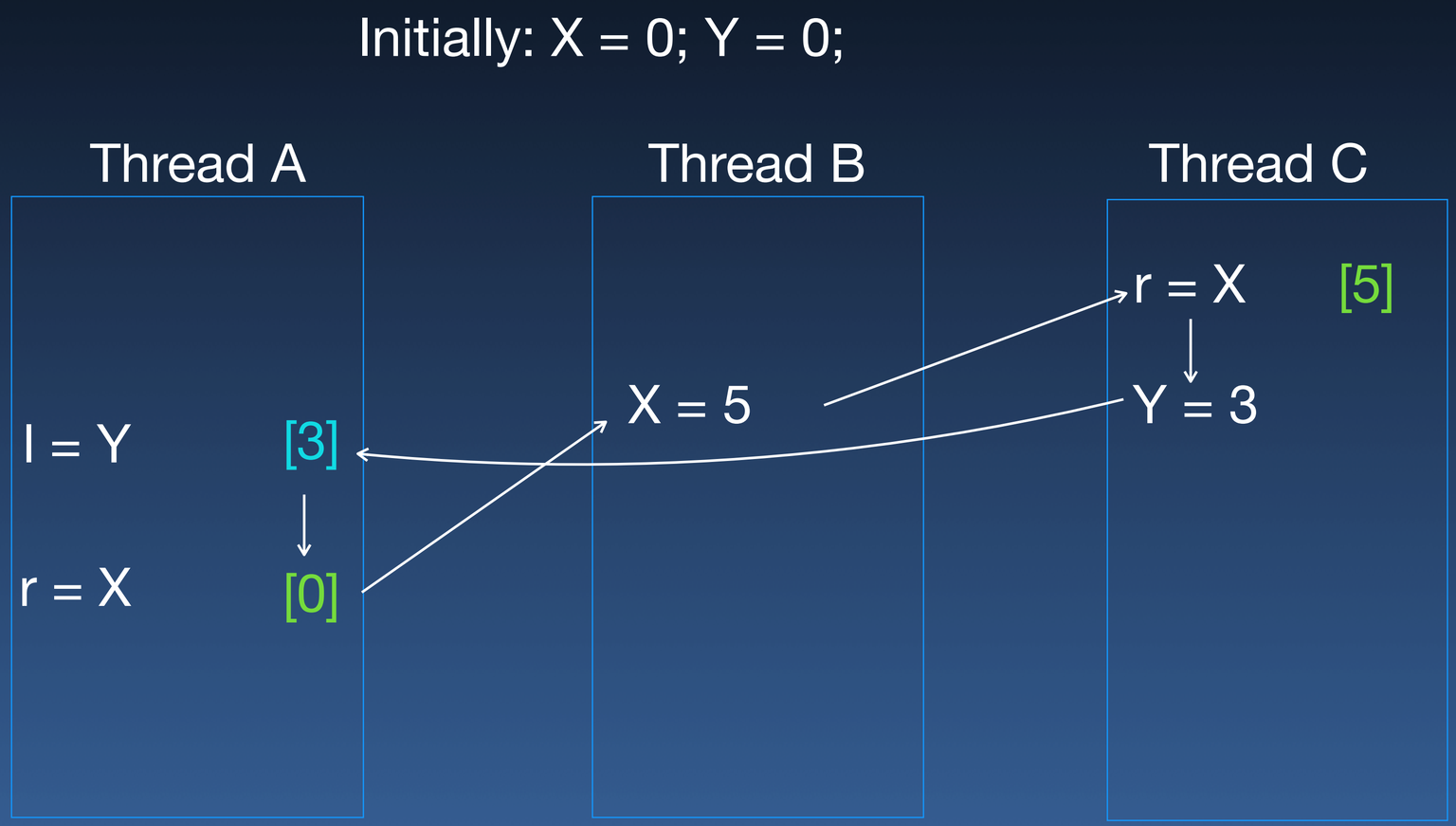
Coherence *vs* Consistency

Initially: X = 0; Y = 0;



Write:
stage in store buffer
read cache-line
for update
update entry

Coherence *vs* Consistency



Memory inconsistency

- Consistency is about global state (not per-variable)
  - ➡ Program must not assume higher consistency than available
- Only local memory dependencies visible to compiler/architecture
  - ➡ Can allocate a register or stack entry for some shared variable
  - ➡ Batching of memory transactions
  - ➡ Network can also reorder two memory messages

Memory inconsistency

- Consistency is about global state (not per-variable)
  - ➡ Program must not assume higher consistency than available
- Only local memory dependencies visible to compiler/architecture
  - ➡ Can allocate a register or stack er
  - ➡ Batching of memory transactions
  - ➡ Network can also reorder two mer

X=1; Y=1; X=2;

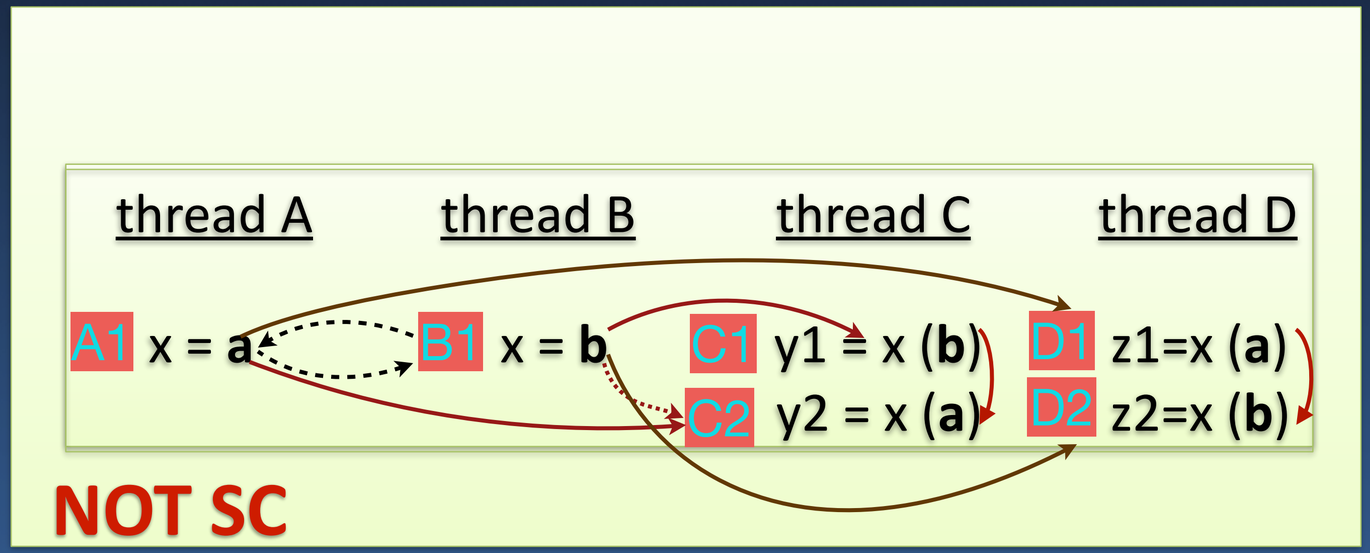
- ➡ Second write to X may happen before Y's
- ➡ 1st write may never happen

Memory inconsistency

- Consistency is about global state (not per-variable)
  - ➡ Program must not assume higher consistency than available
- Only local memory dependencies visible to compiler/architecture
  - ➡ Can allocate a register or stack entry for some shared variable
  - ➡ Batching of memory transactions
  - ➡ Network can also reorder two memory messages

★ Solutions: Handle inconsistency, force synchronization

Example: Not SC



SC Inefficiency

- Hard to implement efficiently
  - ➡ Need to enforce serialization of operations
  - ➡ No re-ordering of instructions allowed
- Some solutions:
  - ➡ Allow out-of-order execution, Detect and recover from SC violation
  - ➡ Only enforce ordering when required
    - ▸ programmer enforced

thread A	thread B	thread C	thread D
x = **a**	x = **b**	y1 = x (**b**) y2 = x (**a**)	z1=x (**a**) z2=x (**b**)

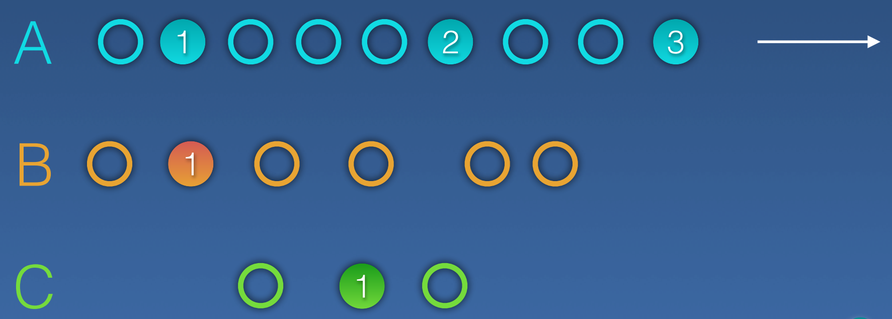
Relaxing SC

initially: ready=0, data=0

thread P	thread C
data1 = 1	Wait for OK
data2 = 1	sav1 = data1
Say OK	sav2 = data2

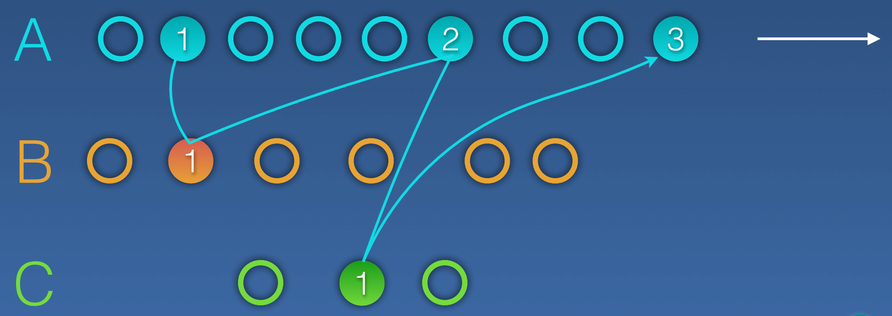
Weak Consistency

- Special synchronization accesses are sequentially consistent
- Regular accesses ordered only with respect to synchronization accesses
  - Before any regular read/write is allowed to be visible to any other thread, all previous synchronization accesses must become visible
  - Before a synchronization access is allowed to complete, all previous ordinary read/write accesses must be completed
- Suitable for many optimizations



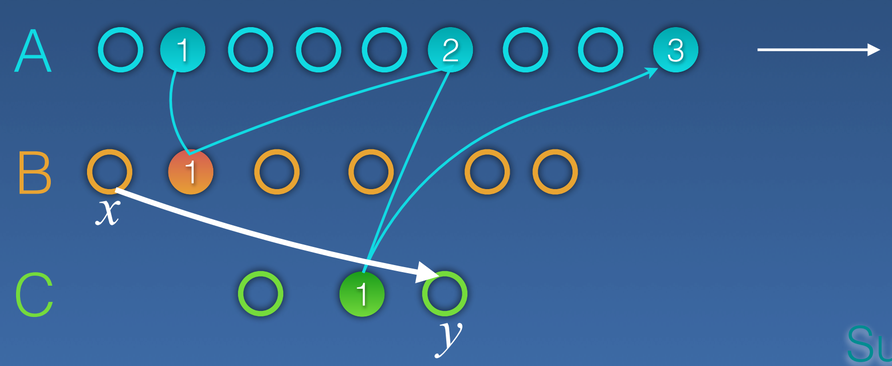
Weak Consistency

- Special synchronization accesses are sequentially consistent
- Regular accesses ordered only with respect to synchronization accesses
  - Before any regular read/write is allowed to be visible to any other thread, all previous synchronization accesses must become visible
  - Before a synchronization access is allowed to complete, all previous ordinary read/write accesses must be completed
- Suitable for many optimizations



Weak Consistency

- Special synchronization accesses are sequentially consistent
- Regular accesses ordered only with respect to synchronization accesses
  - Before any regular read/write is allowed to be visible to any other thread, all previous synchronization accesses must become visible
  - Before a synchronization access is allowed to complete, all previous ordinary read/write accesses must be completed
- Suitable for many optimizations



Weak Consistency

- Special synchronization accesses are sequentially consistent
- Regular accesses ordered only with respect to synchronization accesses
  - Before any regular read/write is allowed to be visible to any other thread, all previous synchronization accesses must become visible
  - Before a synchronization access is allowed to complete, all previous ordinary read/write accesses must be completed
- Suitable for many optimizations