HELLENIC REPUBLIC
UNIVERSITY OF CRETE

Distributed Computing

Graduate Course

Section 4: Concurrent Objects – Correctness, Progress and Efficiency

Panagiota Fatourou
Department of Computer Science

Concurrent Objects

- A concurrent object is a data object "shared" by concurrently executing processes.
- Each object has a type, which defines a set of possible values and a set of primitive operations that provide the only means to create and manipulate that object.
- Concurrent objects have been proposed as building blocks for the construction of more complex multi-processing systems.
  - Leads to a system that is simple and well-structured.

Basic Concurrent Objects

Multi-Writer (MW) Register

- All processes are allowed to execute update operations to the register

Single-Writer (SW) Register

- Only one process is allowed to execute update operations to the register.

Register size

- A register is of bounded size if the set of values it may store is bounded. In the opposite case, we say that the register size is unbounded.

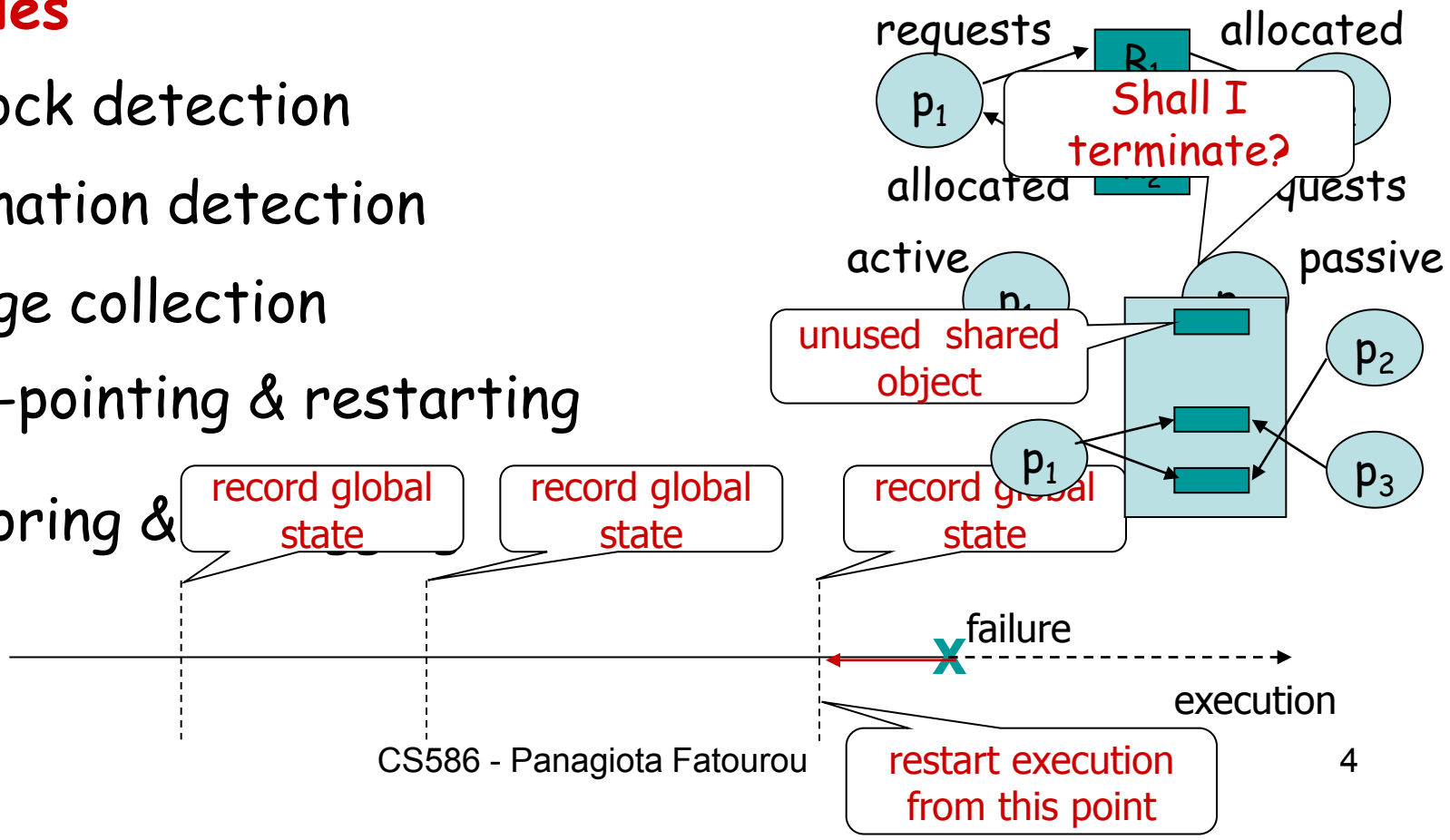
- Different machines support different sets of these operations in hardware.
- The hardware, then, guarantees that these operations are executed atomically.

Global State Predicate Evaluation

In many problems of distributed computing, some action should take place only if some global predicate evaluates to TRUE.

Examples

- deadlock detection
- termination detection
- garbage collection
- check-pointing & restarting
- monitoring &



Debugging Distributed Algorithms

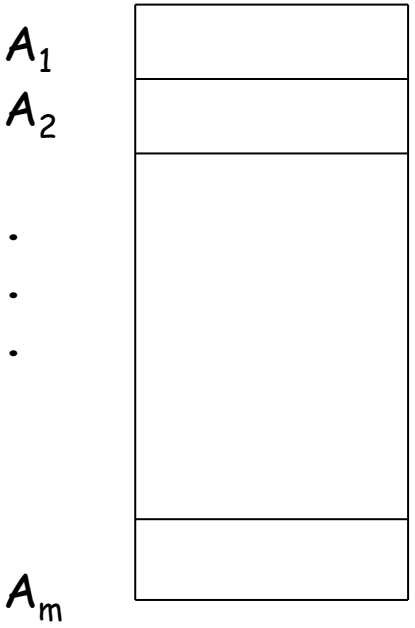
- The biggest difficulty in proving that a distributed algorithm is correct is the necessity to argue based on non-consistent versions of the shared variables.

- Calculating consistent views of the shared variables facilitates the verification of correctness of distributed algorithms.

- Calculating such consistent views is not however an easy problem. ☹

Snapshot Object

- A snapshot object is a concurrent object that is composed by an array of m components, A_1, …, A_m, each capable to store a value from some set.
- The snapshot object supports two operations:
  - UPDATE(i,v): writes v to component A_i.
  - SCAN: returns a vector of consistent values, one for each component.
- Snapshot objects provide "consistent views" of a set of shared variables given that UPDATE operations may concurrently change the values of these variables.



Snapshot Object A

Snapshot Objects

Single-Writer Snapshot

- Only process p_i is allowed to execute UPDATES on component A_i.

Multi-Writer Snapshot

- All processes are allowed to execute UPDATES on every component.

- Snapshots simplify the task of designing distributed algorithms! ☺
- They are too complicated to be provided by the hardware. ☹

Applications of Snapshots

- Verification of global predicates
- Mutual Exclusion [Katseff – ACM STOC'78, Lamport – J. of the ACM'86, Dolev & Gafni & Shavit – ACM STOC'88].
- Concurrent Timestamps [Dolev & Shavit –STOC'88 & SIAM J. on Computing (SICOMP)'97].
- Simplify the design of concurrent data structures [Aspnes & Herlihy – SPAA'90, Herlihy – PODC'91].
- Simplify the design of other concurrent objects [Vitanyi & Awerbuch - FOCS'86, Bloom – PODC'87, Peterson & Burns – FOCS'87].
- Simplify the design of software transactional memory [Shavit & Touitou - PODC'95, Herlihy & Luchangco & Moir & Scherer - PODC'03, Marathe & Scott - PODC'05].

The Problem

- Can we implement snapshot objects in systems that provide only r/w registers?

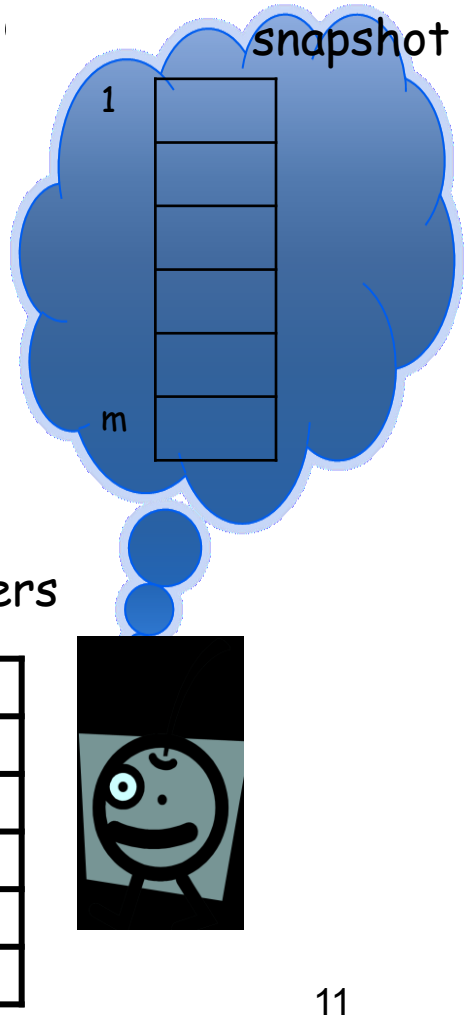
- If yes, how efficient can their implementation be in terms of:
  - Time complexity?
  - Space omplexity?

Model

- **Concurrent Execution**
  - n processes are executed concurrently
- **Asynchrony**
- **Communication via Shared Memory**
- Failures
  - Processes may crash at any point of their execution. A crashed process stops executing and never recovers.

Implementations of Snapshots from r/w registers

- Use the registers to store the values of the snapshot components.
- Provide algorithms to implement SCAN an UPDATE.



Efficiency

- Step Complexity (SCAN or UPDATE)
  - Maximum number of steps executed by any process in any execution in order to perform an operation.
- Space Complexity
  - Number (and size) of registers needed.

Correctness and Termination

Wait-Freedom

- Each process finishes the execution of its operation within a bounded number of its own steps.

➡ Wait-free algorithms are highly fault-tolerant!

Linearizability Intuitively

- In each execution α, each SCAN and UPDATE should have the same response as if it has executed serially (or atomically) at some point in its execution interval. This point is called linearization point of the operation.

and slightly more formally:

- For each (parallel) execution α produced by the implementation, there is a serial execution σ of the operations (SCAN and UPDATE) executed in α, s.t.:
  - each operation has the same response in σ as in α, and
  - σ respects the partial order determined by the execution intervals of the operations.

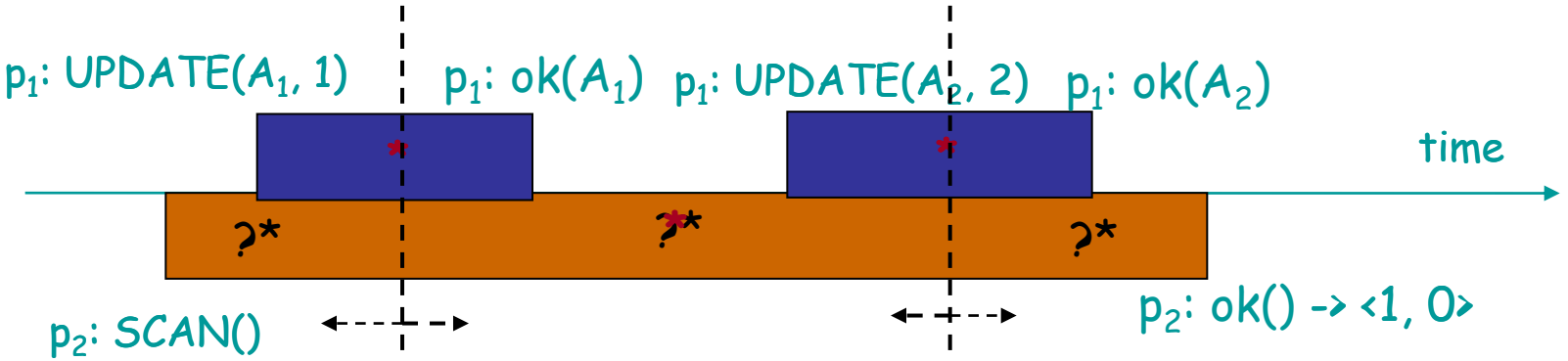
Linearizability – Even more Formally

We say that an execution α is linearizable if it is possible to do all of the following:

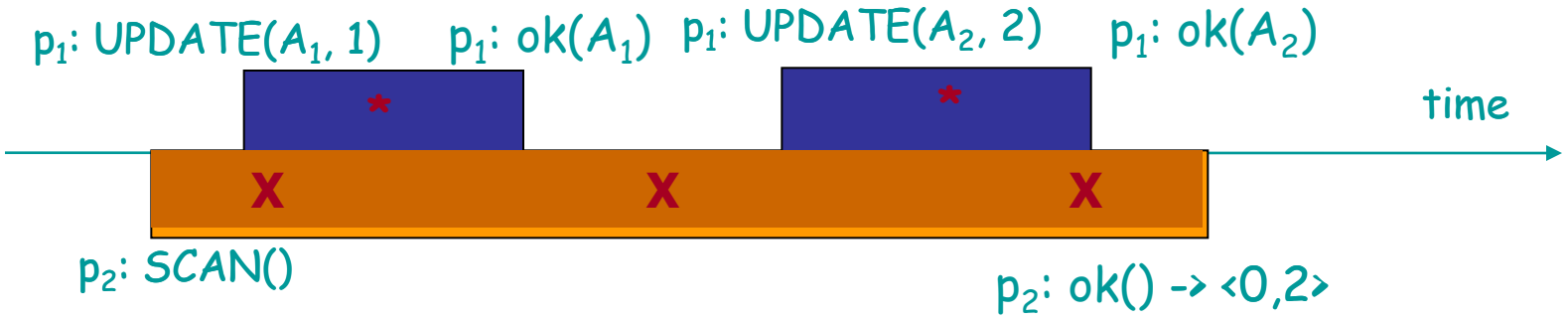
1. For each completed operation π, to insert a linearization point *π somewhere between π's invocation and response in α.
2. To select a subset Φ of the incomplete operations, and for each operation ρ in Φ:
   - to select a response, and
   - to insert a linearization point *ρ somewhere after ρ's invocation in α.
3. These operations and responses should be selected and these linearization points should be inserted, so that, in the sequential execution constructed by serially executing each operation at the point that its linearization point has been inserted, the response of each operation is the same as that in α.

Linearizability - Examples

Example of linearizable execution

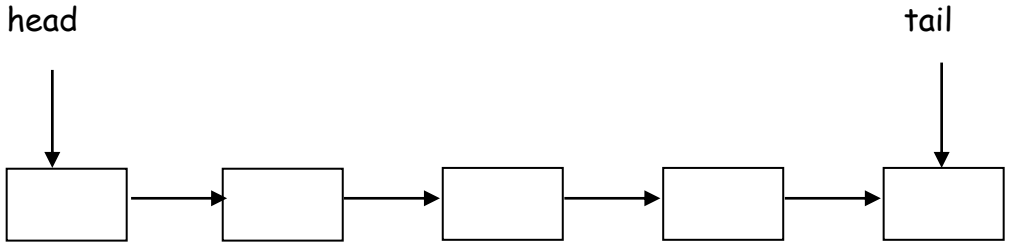


Example of non-linearizable execution



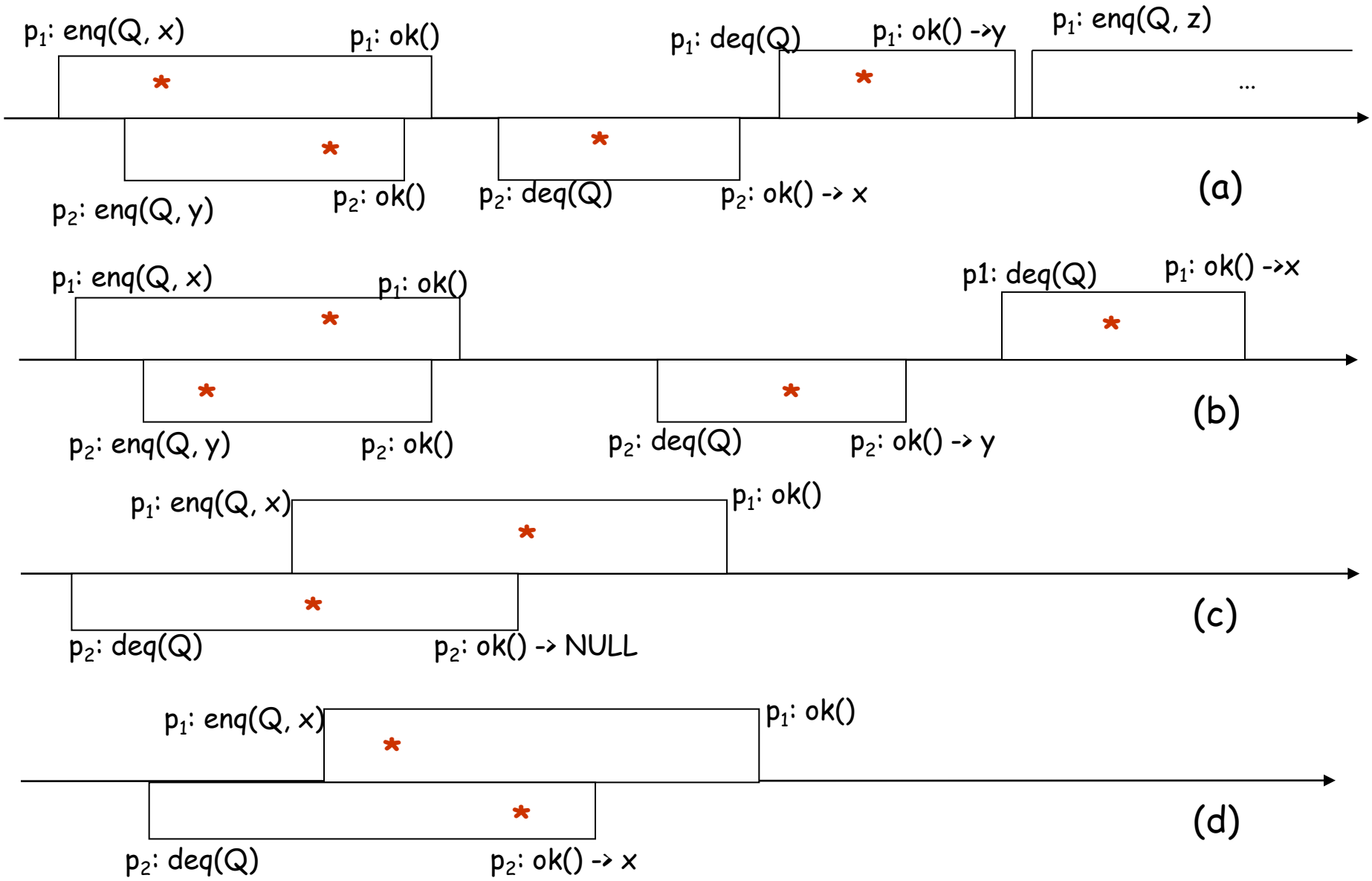
Linearizability - Examples

- A FIFO queue Q supports the following two operations:
  - enq(Q,v): add an element with value v as the last element of Q
  - deq(Q): deletes and returns the first element of Q
- In a concurrent FIFO queue, several processes try to apply enq() and deq() operations concurrently.

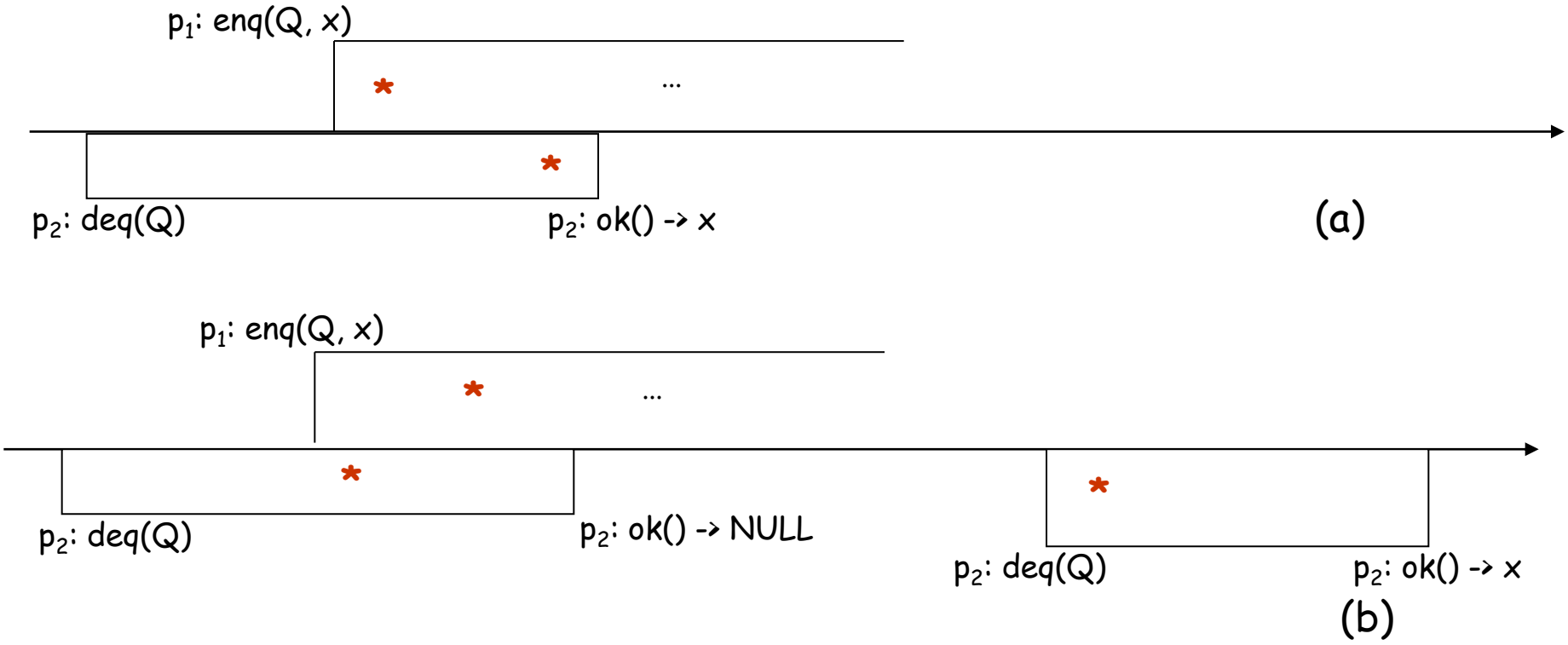


- Concurrent FIFO queues can be implemented by simpler concurrent objects, such as Test-And-Set and LL/SC registers.
- The same is true for other concurrent objects, such as stacks, lists, skip lists, graphs, etc.

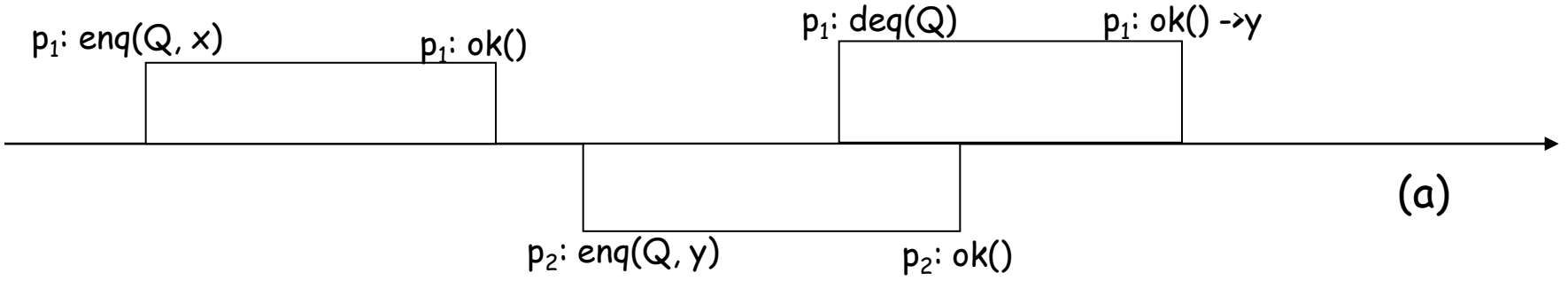
Linearizable Executions



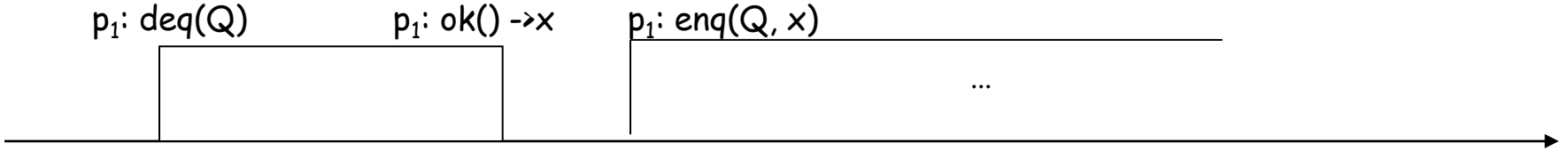
Linearizable Executions



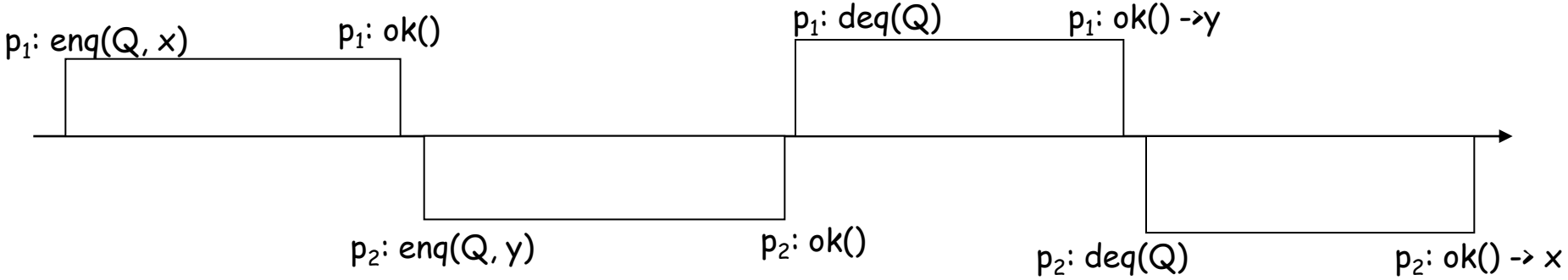
Non-Linearizable Executions



(a)

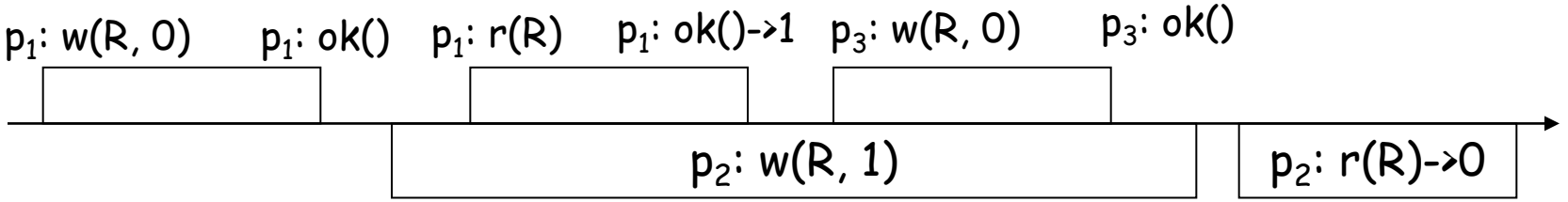


(b)

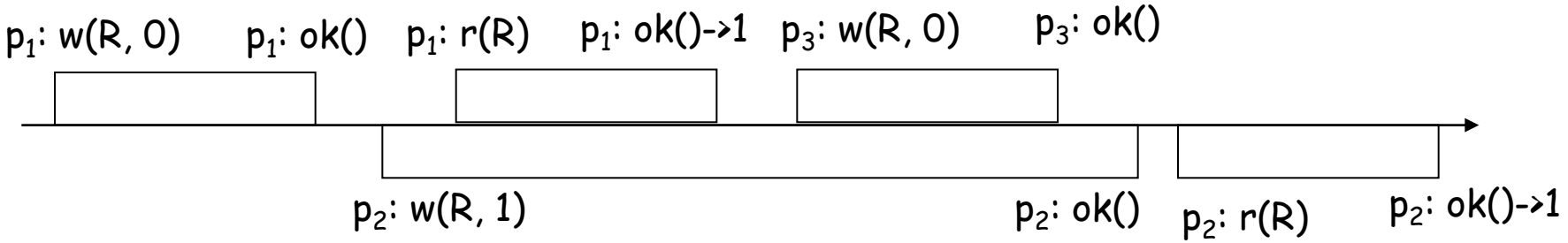


(c)

More Examples - Register Executions



(a) linearizable



(b) non-linearizable

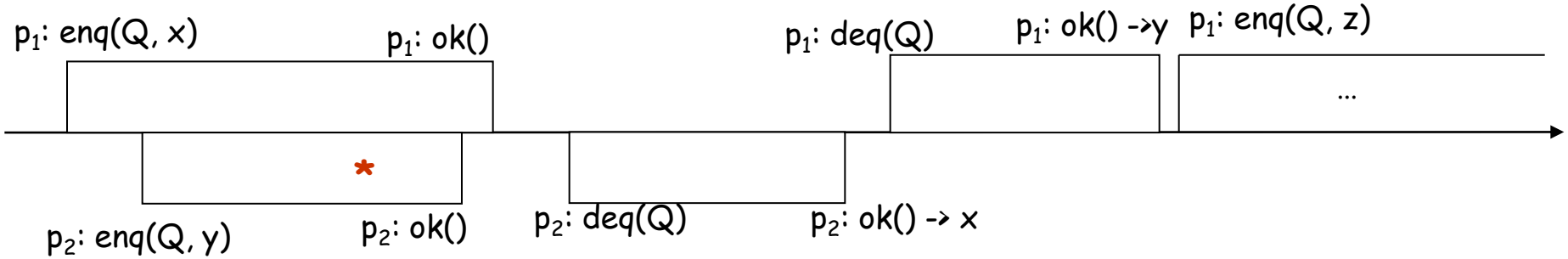
Histories

- A history H is a finite sequence of operation invocation and response events.
- A subhistory of H is a subsequence of the events of H.
- The invocation of an operation is denoted by <p_i: op(O, args*)>, where O is an object's name, op is an operation name, args* denotes a sequence of argument values and p_i is a process name.
- The response of an operation is denoted by <p_i: term(O)->res*>, where term is a termination condition, and res* is a list of results.
- A response matches an invocation if their object names agree and their process names agree.
- complete(H): maximal subsequence of H consisting only of invocations and matching responses.

Histories

- A process subhistory, H_P, of a history H is the subsequence of all events in H whose process names are P.
- An object subhistory, H_O, is similarly defined for an object O.
- A history H is sequential if:
  - The first event of H is an invocation, and
  - Each invocation, except possibly the last one, is immediately followed by a matching response. Each response is immediately followed by an invocation from the same process.
- A history that is not sequential is concurrent.
- A history H is well-formed if, for every process p, H_p is sequential.
- Two histories, H and H', are equivalent if for every process p, $H_p = H'_p$

Histories - Example



H:

p_1: enq(Q, x)
p_2: enq(Q, y)
p_2: ok(Q)
p_1: ok(Q)
p_2: deq(Q)
p_2: ok(Q)->x
p_1: deq(Q)
p_1: ok(Q)->y
p_1:enq(Q,z)

complete(H):

p_1: enq(Q, x)
p_2: enq(Q, y)
p_2: ok(Q)
p_1: ok(Q)
p_2: deq(Q)
p_2: ok(Q)->x
p_1: deq(Q)
p_1: ok(Q)->y

H_{p1}:

p_1: enq(Q, x)
p_1: ok(Q)
p_1: deq(Q)
p_1: ok(Q)->y
p_1:enq(Q,z)

H_{p2}:

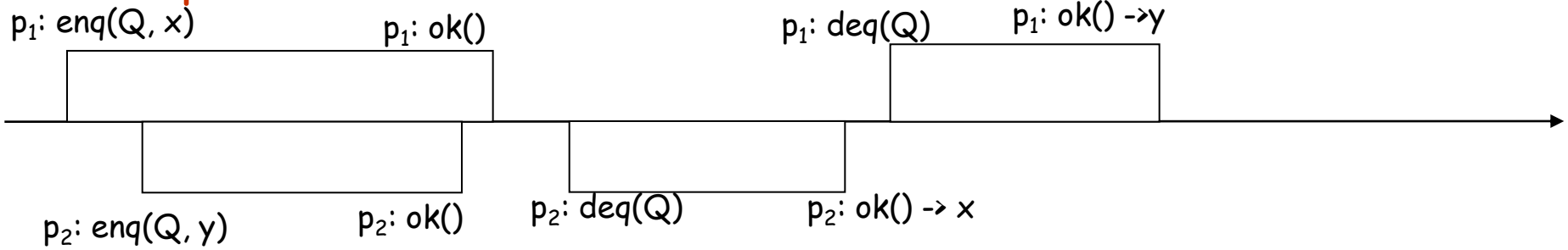
p_2: enq(Q, y)
p_2: ok(Q)
p_2: deq(Q)
p_2: ok(Q)->x

Partial Order induced by histories

- A history H induces an irreflexive partial order $<_H$ on operations:
  - $e_0 <_H e_1$ if $res(e_0)$ precedes $inv(e_1)$ in H
- Informally, $<_H$ captures the real time precedence ordering of operations in H.
- Operations unrelated by $<_H$ are said to be concurrent.
- If H is sequential, $<_H$ is a total order.

(a)

Example



$<_H$ = {<[p_1: enq(Q, x)/p_1: ok(Q)], [p2: deq(Q)/p_2: ok(Q)->x]>,
<[p_1: enq(Q, x)/p_1: ok(Q)], [p1: deq(Q)/p_1: ok(Q)->y]>,
<[p_2: enq(Q, y)/p_2: ok(Q)],[p_2: deq(Q)/p_2: ok(Q)->x]>
<[p_2: enq(Q, y)/p_2: ok(Q)], [p_1: deq(Q)/p_1: ok(Q)->y]>
<[p_2: deq(Q)/p_2: ok(Q)->x], [p_1:deq(Q)/p_1: ok(Q)->y]> }

Linearizability defined in terms of Histories

- A history H is linearizable if it can be extended (by appending zero or more response events) to some history H' such that:
  - L_1: complete(H') is equivalent to some legal sequential history S, and
  - L_2: $<_H \subseteq <_S$
- S is called a linearization of H.
- Nondeterminism is inherent in the notion of linearizability:
  1. There may be more than one extension H' satisfying the two conditions, L_1 and L_2.
  2. For each extension H', there may be more than one linearization S

Properties of Linearizability - Locality

- A property P of a concurrent system is said to be local if the system as a whole satisfies P whenever each individual object satisfies P.

Theorem

H is linearizable if and only if, for each object O, H_O is linearizable.

This theorem has two parts:

- If part: If for each object O, H_0 is linearizable, then H is linearizable.
- Only-if part: If H is linearizable, then for each object O, H_0 is linearizable.

- The only-if part is obvious.
- The if part requires a proof (see original paper by Herlihy and Wing Proof of Theorem 1).

Locality

- Locality is important because it allows concurrent systems to be designed and constructed in a modular fashion.
- Linearizable objects can be implemented, verified and executed independently.
- Locality should not taken for granted; alternative correctness properties that have been proposed in the literature are not local.

Linearizable implementations

- An implementation is linearizable if all the executions (histories) it produces are linearizable.
- In this course, an implementation of a concurrent object is called correct, if it is linearizable.

Comparison to other correctness conditions

Sequential Consistency (Intuitively)

- In each execution α, each operation should have the same response as if it has executed serially (or atomically).

and slightly more formally:

For each (parallel) execution α produced by the implementation, there is a serial execution σ of the operations executed in α, s.t.,

- each operation has the same response in σ as in α.

- for each process p_i, if op by p_i precedes op′ by p_i then op appears in σ before op′.

Sequential Consistency

even more formally:

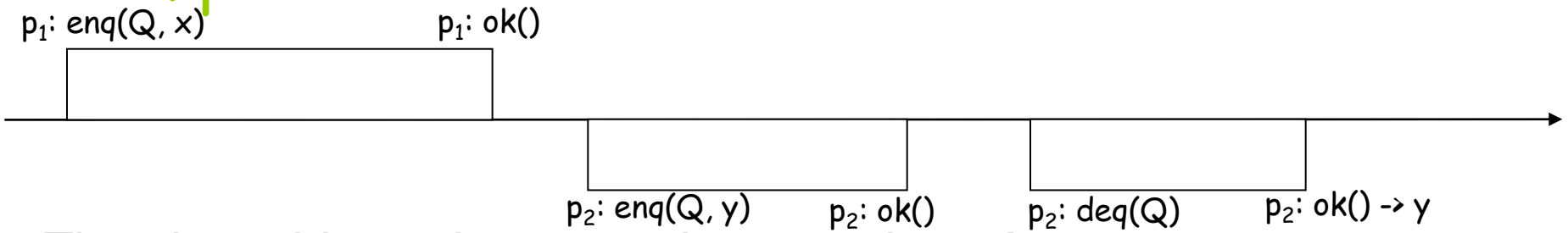
We say that an execution a is sequntially consistent if it is possible to do all of the following:

1. For each completed operation π, to insert a serialization point $*\pi$.
2. To select a subset Φ of the incomplete operations, and for each operation π in Φ:
   - to select a response for it, and
   - to insert a linearization point $*\pi$ for it.
3. These operations and responses should be selected and these linearization points should be inserted, so that, in the sequential execution constructed by serially executing each operation at the point that its linearization point has been inserted, the response of each operation is the same as that in a.

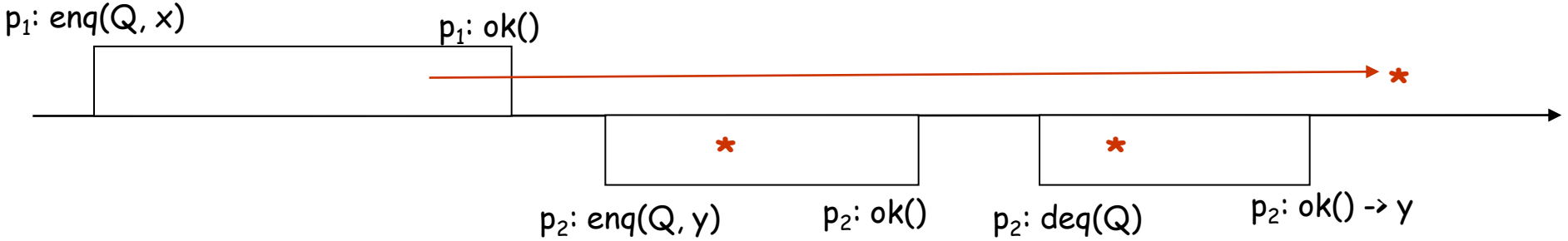
and finally:

- A history H is sequentially consistent if it can be extended (by appending zero or more response events) to some history H' such that complete(H') is equivalent to some legal sequential history S.

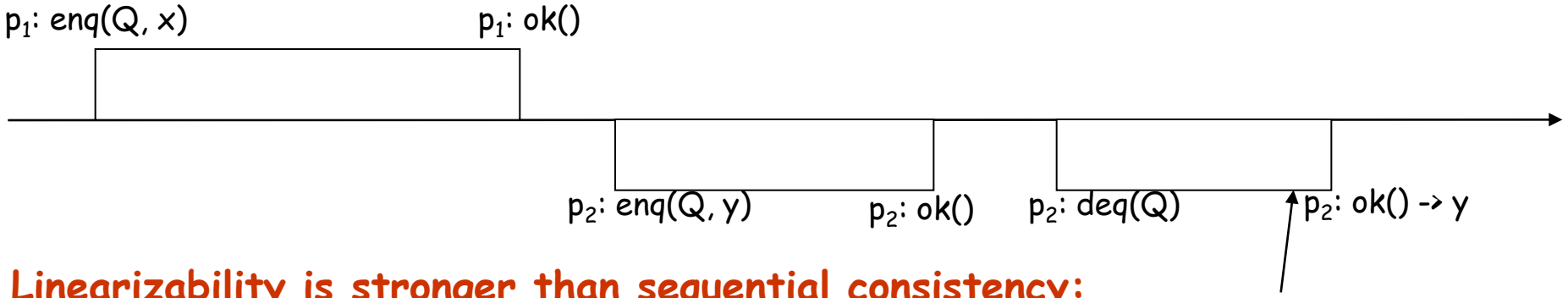
Comparison to other correctness conditions



- The above history is sequentially consistent!



- However, it is not linearizable (since p_2 should return x instead of y).

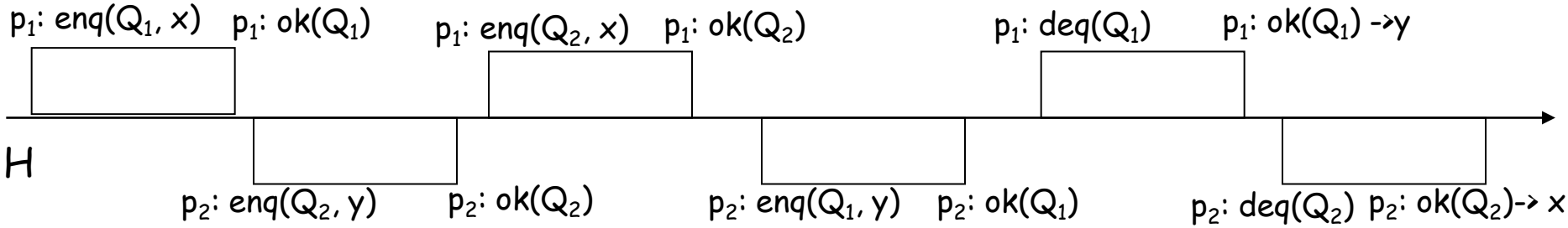


Linearizability is stronger than sequential consistency:

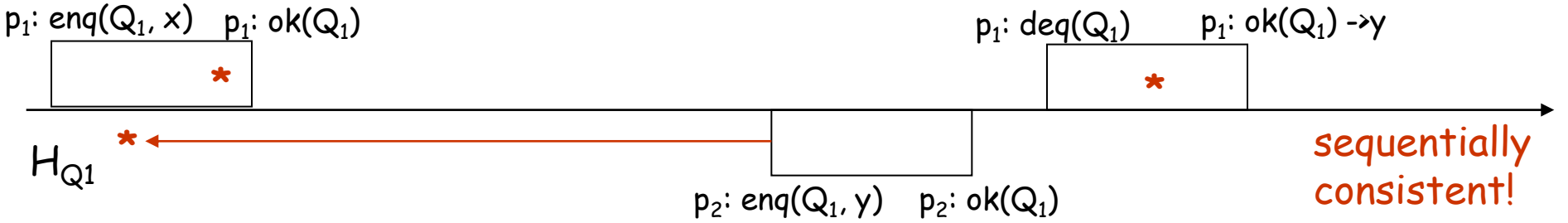
- Any linearizable execution is sequentially consistent!
- The opposite is not TRUE!

Comparison to other correctness conditions

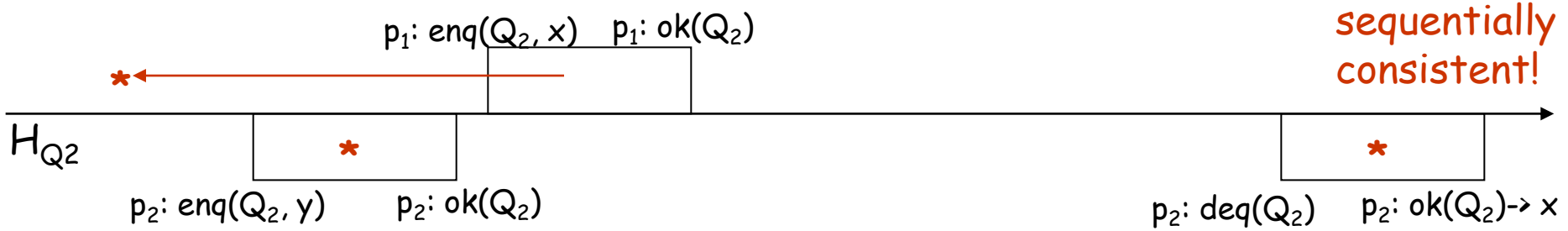
- Sequential consistency **is not** a local property!



not sequentially consistent!



sequentially consistent!



sequentially consistent!

The Problem

- Can we implement snapshot objects in systems that provide only r/w registers?

The trivial solution does not work!

- Assign a register R_i to each component A_i.
- **UPDATE(i,v):** write(R_i,v);
- **SCAN():** Read all registers and return a vector consisting of the values you read.

This algorithm is not linearizable!

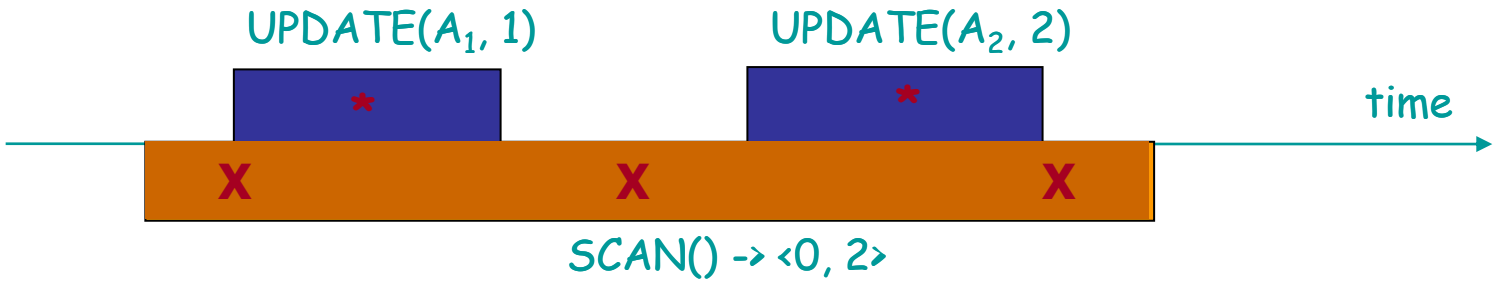
The trivial solution does not work!

- Assign a register R_i to each component A_i.
- **UPDATE(i,v):** write(R_i,v);
- **SCAN():** Read all registers and return a vector consisting of the values you read.

Processes		Register Values	
P	Q	R_1	R_2
read(R_1)		0	0
	write(R_1,1)		
		1	0
	write(R_2,2)		
		1	2
read(R_2)			

The trivial solution does not work!

Processes		Register Values	
P	Q	R_1	R_2
read(R_1)		0	0
	write(R_1,1)	1	0
	write(R_2,2)	1	2
read(R_2)			



The trivial solution does not work!

Are there wait-free, linearizable implementations of atomic snapshot objects?

Snapshot Implementations from Registers

- Anderson – PODC'90, PODC'93, DISC'91
- Afek, Attiya, Dolev, Gafni, Merritt and Shavit – PODC'90 + JACM'93
- Attiya, Herlihy & Rachman - DISC'92
- Attiya & Rachman – PODC'93 + SICOMP'98
- Israeli & Shaham – PODC'92
- Israeli, Shaham & Shirazi – DISC'93
- Inoue, Chen, Masuzawa & Tokura – DISC'94
- Afek, Stupp & Touitou – FOCS'99
- Afek, Attiya, Fouren, Stupp & Touitou – PODC'99
- Attiya & Fouren – SICOMP'01
- Jayanti – PODC'02, STOC'05
- Fatourou, Fich & Ruppert - STOC'03
- Fatourou & Kallimanis – PODC'06, PODC'07

A simple implementation using unbounded-size registers

- Each component, A_j, $1 \leq j \leq n$, can be updated only by process p_j (single-writer snapshot).
- The implementation uses n registers R_1, .., R_n, one for each component. A register R_j has been assigned to each component A_j. UPDATE operations on A_j write only into R_j. Each register R_j is written only by p_i but it can be read by all processes (single-writer registers).
- Each register R_j is big enough to store the following information:
  - val_j: the current value of A_j
  - tag_j: a timestamp used by p_j to differentiate the UPDATE operations it initiates
  - $view_j$: a vector of n values, one for each component.
- This assumption is not realistic (since the registers are too big to be provided by the hardware).
- However, for the time being, we are only interested to design a simple such algorithm.

The UnboundedSnapshot Algorithm

UPDATE(v), code for process p_i:

view := SCAN();
tag := increment p_i's tag by 1;
write(R_i,<v, tag,view>);

SCAN, code for process p_i:

repeatedly read R_1, …, R_n until:

1. Two **sets of reads** return the same R_j.tag for every j; then, return the vector of values R_j.val, $1 \leq j \leq n$, returned by the second set of reads, or
2. For some j, three distinct values of R_j.tag have been seen; return the vector of values in R_j.view associated with the last of the three values read in R_j.tag.

Step Complexity = $O(n^2)$ for SCAN and UPDATE.
The implementation uses n SW registers of unbounded size.

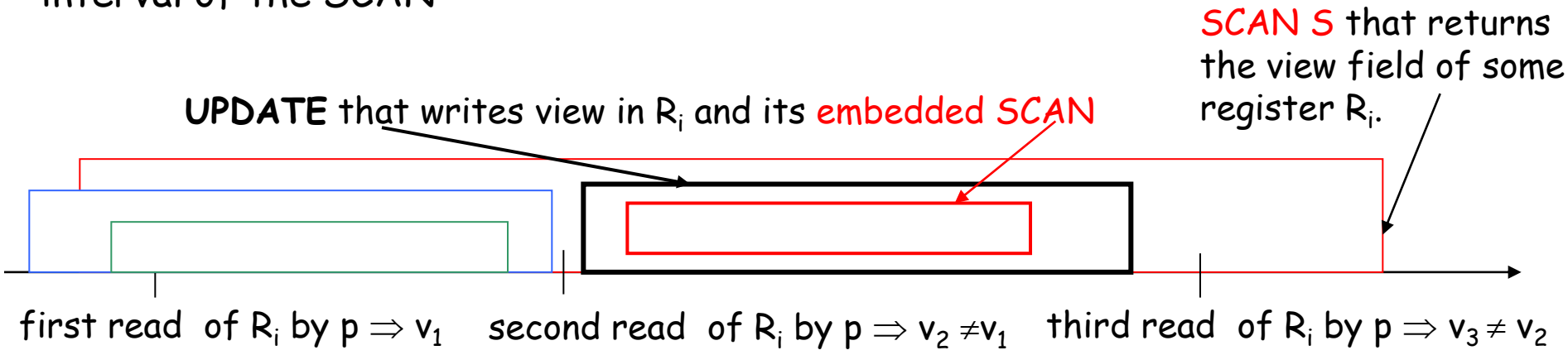
Linearizability

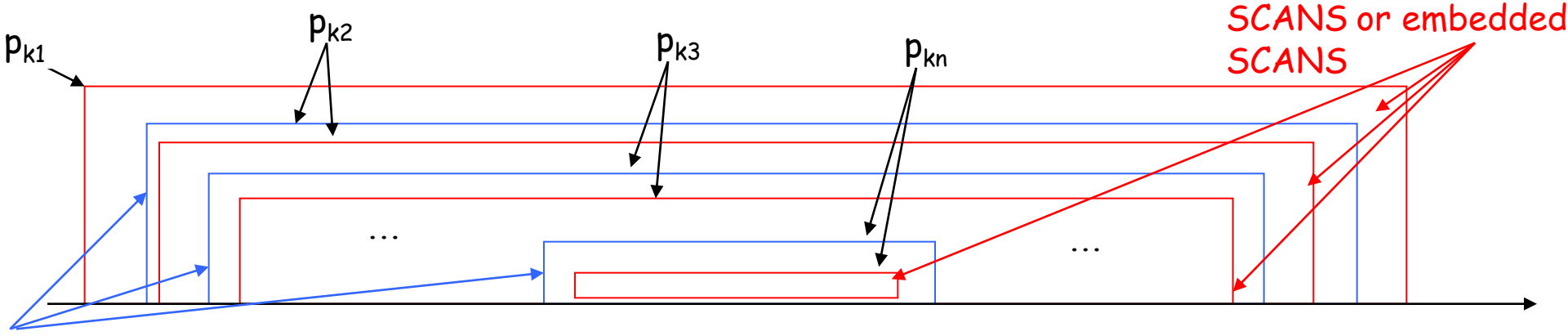
Definition: A SCAN operation S returns a *consistent vector* if, for each component A_i, S returns for A_i the value written by the last UPDATE on A_i for which the linearization point precedes the linearization point of S (or the initial value if such an UPDATE does not exist).

- **To prove that the implementation is linearizable, we have to do the following:**
  - assign linearization points to the operations of any execution
  - prove that the linearization point of each operation is within its execution interval, and
  - prove that each SCAN returns a consistent vector

The UnboundedSnapshot Algorithm

- If process j, while performing repeated sets of reads on behalf of a SCAN, ever sees the same register R_i with three different tags, then it knows that some $UPDATE_i$ is completely contained within the execution interval of the SCAN





All p_{k1}, …, p_{kn} are simultaneously active. Since we have n processes in the system, the embedded SCAN executed by p_{kn} terminates by evaluating condition (1) of the SCAN code to TRUE.

The UnboundedSnapshot Algorithm

Assignment of Linearization Points

UPDATE Operations

- We insert the linearization point of an UPDATE at the point where its write occurs.

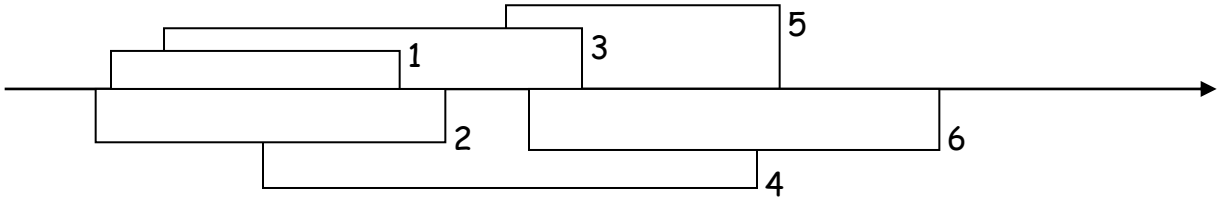
SCAN operations

- We assign linearization points not just to SCANs but also to embedded SCANs.
- We partition SCANs in two categories:
  - Direct SCANs: those that terminate by evaluating condition (1) to TRUE
  - Indirect SCAN: those that terminate by evaluating condition (2) to TRUE
- The linearization point of a direct SCAN can be placed anywhere within the end of the first of its last two sets of reads and the beginning of its second.

The UnboundedSnapshot Algorithm

Linearization points of indirect SCANs

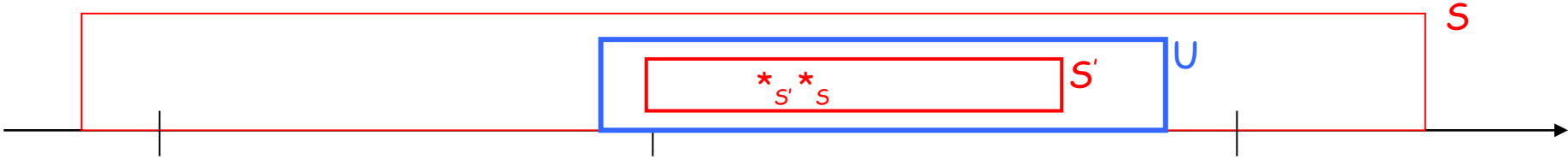
By induction on their response events (i.e., we linearize these SCANs one by one, in the order of their response events).



The UnboundedSnapshot Algorithm

Induction Base: Linearization point of indirect SCAN S which responds first.

- S returns a vector of values written by an UPDATE U (i.e., this vector has been calculated by the embedded SCAN S' of U).
- The execution interval of U and therefore also of S' (which is executed by U) is included in the execution interval of S.
- S' is a direct SCAN (since otherwise, the indirect SCAN that has the first response event would be S' and not S).
- Thus, a linearization point for S' has already been assigned.
- We linearize S at the same place as S'.



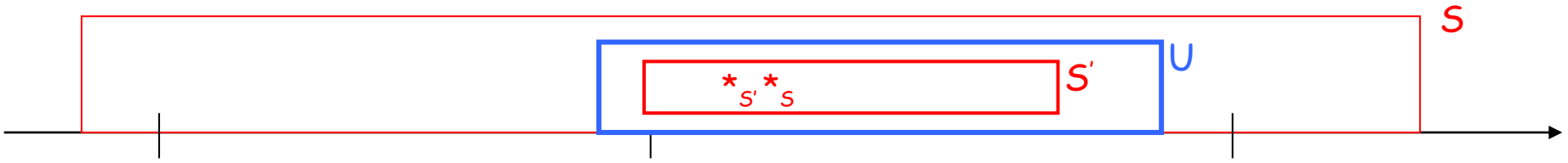
The UnboundedSnapshot Algorithm

Induction Hypothesis

- Let S be the indirect SCAN which has the k-th respond event. Assume that we have assigned linearization points to all SCAN operations for which it holds that their response events precede that of S.

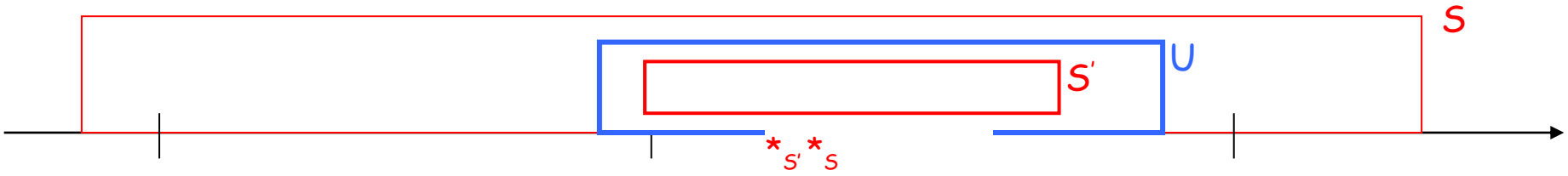
Induction Step: We assign a linearization point to S.

- S returns a vector of values written by an UPDATE U (i.e., this vector has been calculated by the embedded SCAN S′ of U).
- The execution interval of U and therefore also of S′ (which is executed by U) is included in the execution interval of S.
- If S′ is an indirect SCAN, by induction hypothesis, a linearization point has been assigned to S′. The same is TRUE for all direct SCANs.
- We linearize S at the same place as S′.



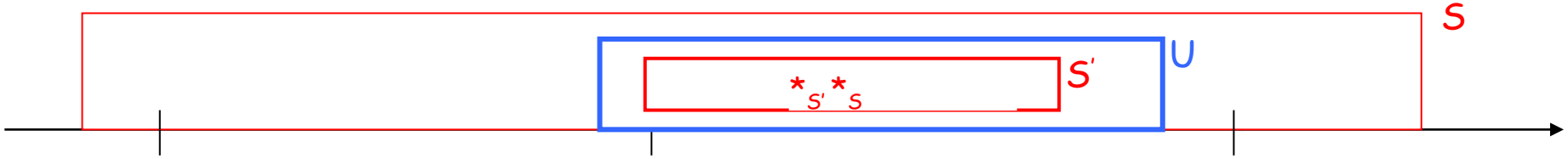
The UnboundedSnapshot Algorithm

- **Lemma:** The linearization point of each SCAN or UPDATE is within its execution interval.
- **Proof:** For UPDATES and direct SCANS this is obvious.
- We prove the claim for indirect SCANS by induction on the order of their response events.
- **Induction Base**
  - Recall that the indirect SCAN S with the first response event is linearized at the same point as the direct SCAN S' from which it borrows its vector. Moreover, the execution interval of S' is included in the execution interval of S.
  - The linearization point of S' is in its execution interval ⇒ Thus, the linearization point of S is in its execution interval.



The UnboundedSnapshot Algorithm

- **Induction Step:** We prove the claim for the indirect SCAN S with the k-th response event.
- S returns a vector of values written by an UPDATE U (i.e., this vector has been calculated by the embedded SCAN S′ of U).
- The execution interval of U and therefore also of S′ (which is executed by U) is included in the execution interval of S.
- If S′ is a direct SCAN, its linearization point is obviously within its execution interval. The same is TRUE, if S′ is an indirect SCAN (by induction hypothesis since the execution interval of S′ is included in the execution interval of S, and therefore the response event of S′ precedes that of S).
- The linearization point of S is placed at the same point as that of S′ ⇒ thus, the linearization point of S is within its execution interval.



The UnboundedSnapshot Algorithm

- **Lemma:** In every execution of UnboundedSnapshot, each SCAN operation returns a consistent vector of values.
- **Proof:** Obviously true for direct SCANs.
- For indirect SCANs, the claim will be proved by induction on the order of response events
- **Induction Base:** The indirect SCAN S which responds first, is linearized at the same point as the direct SCAN S' from which it borrows its vector. Since S' returns a consistent vector, the same holds for S.
- **Induction Step:** We prove the claim for the indirect SCAN S that has the kth response event. S returns the same vector of values as an embedded SCAN S' and is linearized at the same point as S'. Moreover, the execution interval of S' is within the execution interval of S.
- If S' is direct, it obviously returns a consistent vector. If S' is an indirect SCAN, by the induction hypothesis (since the response event of S' precedes that of S), it follows that it returns a consistent vector. Thus, S returns a consistent vector.

A1	A2	A3	Vals	Scans
U(1)			[1,0,0]	
	U(1)		[1,1,0]	
				S1
U(3)			[3,1,0]	
	U(4)		[3,4,0]	
		U(5)	[3,4,5]	
		U(6)	[3,4,6]	
				S2
				S3
				S4

time

$*_{U(1,1)}^{[1,0,0]} *_{U(2,1)}^{[1,1,0]} *_{S1} *_{U(1,3)}^{[3,1,0]} *_{U(2,4)}^{[3,4,0]} *_{U(3,5)}^{[3,4,5]} *_{U(3,6)}^{[3,4,6]} *_{S2} *_{S3} *_{S4}$

Bibliography

These slides are based on material that appears in the following books:

- Herlihy and Wing, "Linearizability: a correctness condition for concurrent objects", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3): 463-492, 1990.
- N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996 (Chapter 13, Section 3).

End of Section

Financing

- The present educational material has been developed as part of the educational work of the instructor.
- The project "Open Academic Courses of the University of Crete" has only financed the reform of the educational material.
- The project is implemented under the operational program "Education and Lifelong Learning" and funded by the European Union (European Social Fund) and National Resources

Notes

Licensing Note

- The current material is available under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0[1] International license or later International Edition. The individual works of third parties are excluded, e.g. photographs, diagrams etc. They are contained therein and covered under their conditions of use in the section «Use of Third Parties Work Note».

[1] http://creativecommons.org/licenses/by-nc-nd/4.0/

- As Non-Commercial is defined the use that:
  - Does not involve direct or indirect financial benefits from the use of the work for the distributor of the work and the license holder
  - Does not include financial transaction as a condition for the use or access to the work
  - Does not confer to the distributor and license holder of the work indirect financial benefit (e.g. advertisements) from the viewing of the work on website

- The copyright holder may give to the license holder a separate license to use the work for commercial use, if requested.

Reference Note

 Panagiota Fatourou. «Distributed Computing. Section 4: Concurrent Objects – Correctness, Progress and Efficiency». Edition: 1.0. Heraklion 2015. Available at: https://opencourses.uoc.gr/courses/course/view.php?id=359.

Preservation Notices

Any reproduction or adaptation of the material should include:

- the Reference Note
- the Licensing Note
- the declaration of Notices Preservation
- the Use of Third Parties Work Note (if is available)

together with the accompanied URLs.