The LINEARIZABILITY HIERARCHY

(From sequentiality to concurrency)

Michel Raynal

Univ Rennes (IRISA, CNRS, Inria) France

Table of Contents

From sequential to concurrent specifications

At the very beginning

From structured programming to objects

Once upon a time... sequential computing

- Simula: an algol-based simulation language.
  by O.-J. Dahl and K. Nygaard
  Communications of the ACM, 9(9):671-678 (1966)

- Go To statement considered harmful.
  by E.W. Dijkstra
  Communications of the ACM, 11(3):147-148 (1968)

  Structured programming.
  by O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare
  Academic Press, 220 pages (1972)

- Proof of correctness of data representation.
  by C.A.R. Hoare
  Acta Informatica, 1:271-281 (1972)

- Nondeterminacy and formal derivation of programs.
  E.W. Dijkstra
  Communications of the ACM, 18(8)):453-457 (1975)

- Programming: sorcery or science?
  by C.A.R. Hoare
  IEEE Software, 1(2):5-16 (1984)

- Pre/post conditions (Hoare's logic)
  Pre-condition { statement } Post-condition
- Weakest pre-condition, Predicate transformer (EWD)

From sequential to concurrent computing

Once upon a time... the advent of concurrency

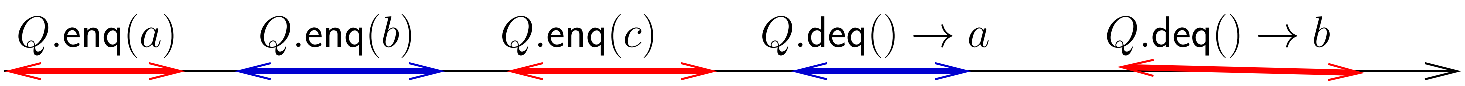
- Solution of a problem in concurrent programming control.
  E.W. Dijkstra
  Communications of the ACM, 8(9):569 (1965)
- Cooperating sequential processes.
  E.W. Dijkstra
  Programming Languages (Genuys Ed.), Academic Press, pp. 43-112 (1968)
- Monitors: an operating system structuring concept.
  C.A.R. Hoare
  Comm. of the ACM, 17(10):549-557 (1974)

Basically reduces concurrency to sequentiality (mutex)

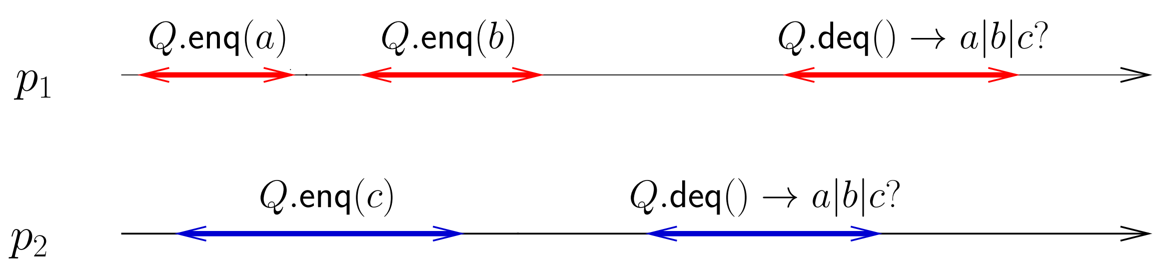
Mastering concurrent computing through sequential thinking.
S. Rajsbaum & M. Raynal
Communications of the ACM, 83(1):78-87 (2020)
(explores the deep continuity from mutex to consensus)

Where is the problem?

- A sequential execution of a queue object



- A concurrent execution of a queue object



On the definition of time: citations

Time is
what makes that all does not arrive at the same time

Time is what is measured by clocks

What is a specification?

- Asynchronous processes, crash failures
- Sequential object:
  all the traces of object operations capturing all the correct behaviors
- Concurrent objects:
  Description of all the ~~traces~~ ??? of object operations capturing all the correct behaviors

  Partial orders ??, How to break atomicity (= at most one operation at a point of the time line? why to break it? etc.)

(BTW, A question is only the formatting of its answer!)

Concurrency: What is a consistency condition (1)?

- Define the (limits on the) way

  concurrency is allowed to impact an execution

- (Always respect process order)

Concurrency: What is a consistency condition (2)?

- Let us consider a concurrent run$R$ involving an object O defined by a specification (e.g. a seq. spec.)

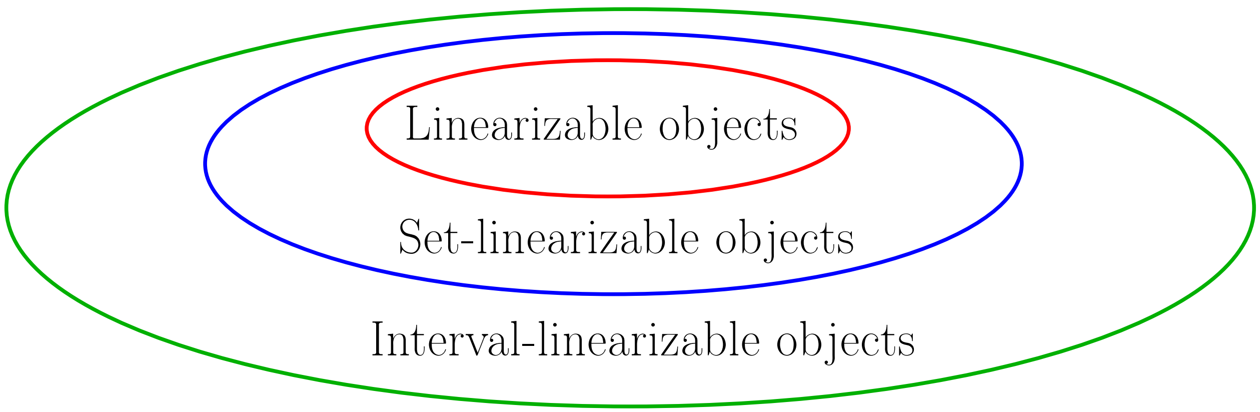
- a consistency condition is a mapping from the operations on the object produced by the run R to the specification of the object

  - ⋆ If (for example) the specification is sequential the consistency condition must produce a trace belonging to the specification

  - ⋆ If no such mapping can be produced, the run does not satisfy the consistency condition

- Linearizability, sequential consistency, serializability, ..., are consistency conditions

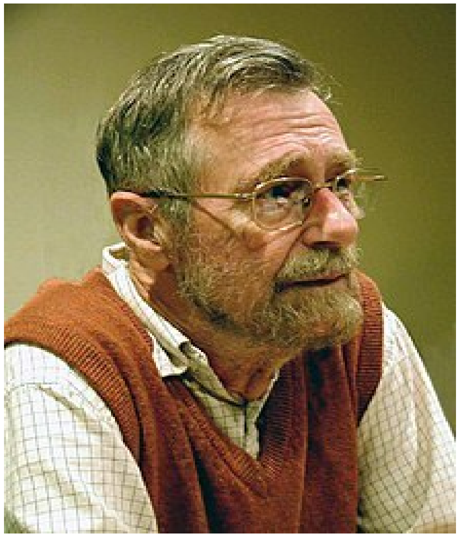
A guided visit to the linearizability hierarchy

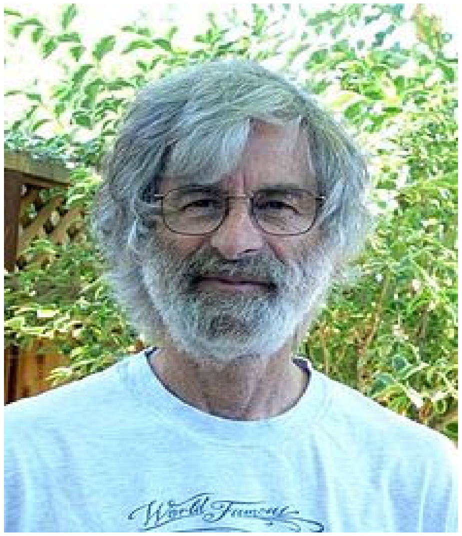


Linearizability

Atomicity, Linearizability, etc.

The masters of time (concurrency)





To synchronize or not to synchronize, that is the question
and what to synchronize?

Basic articles

- Solution of a problem in concurrent programming control
  E.W. Dijkstra
  Communications of the ACM, 8(9):569 (1965)

  First article on concurrency

- On interprocess communication, Part I: basic formalism, Part II: algorithms
  L. Lamport
  Distributed Computing, 1(2):77-101 (1986)

  This article analyzes the nature of what is atomic, and what is not

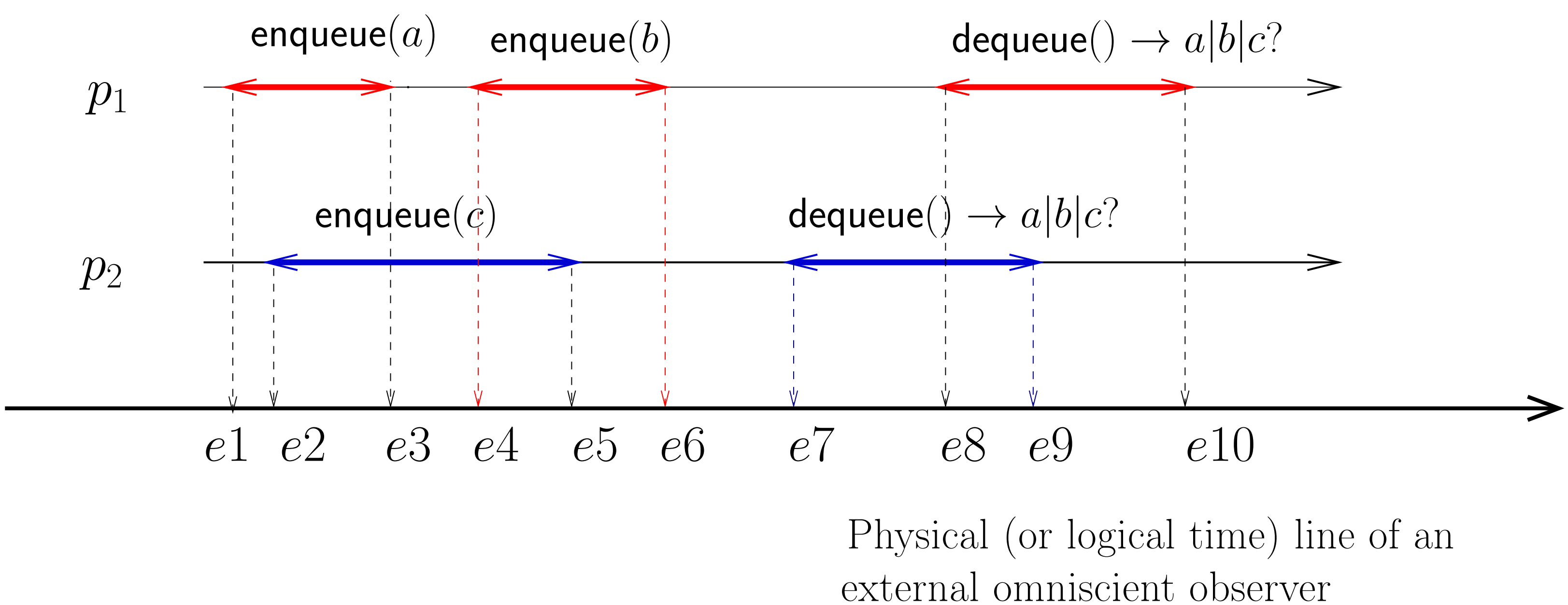
- Linearizability: a correctness condition for concurrent objects
  M.P. Herlihy and J.M. and Wing J.M.
  ACM Transactions on Progr. Languages and Systems, 12(3):463-492 (1990)

  This article introduced linearizability and its properties

For a pedagogical presentation see also chapter 4 (Atomicity: Formal Definition and Properties) in *Concurrent Programming: Algorithms, Principles, and Foundations*, Springer, 528 pages (2013) M. Raynal

Object operations vs events

Asynchronous processes, crash failures



Linearizability: definition

From sequential specifications to concurrent executions

- Linearizability considers objects defined by a sequential specification on total operations
- An execution of an object is linearizable if it is possible to totally order all the operations on the object in such a way that this order respects real-time order

  (if an operation on the object op1 terminated before an operation op2 started, op1 appears before op2 in the total order)

Remarks:
- total operation: always returns a result
- always respects process order

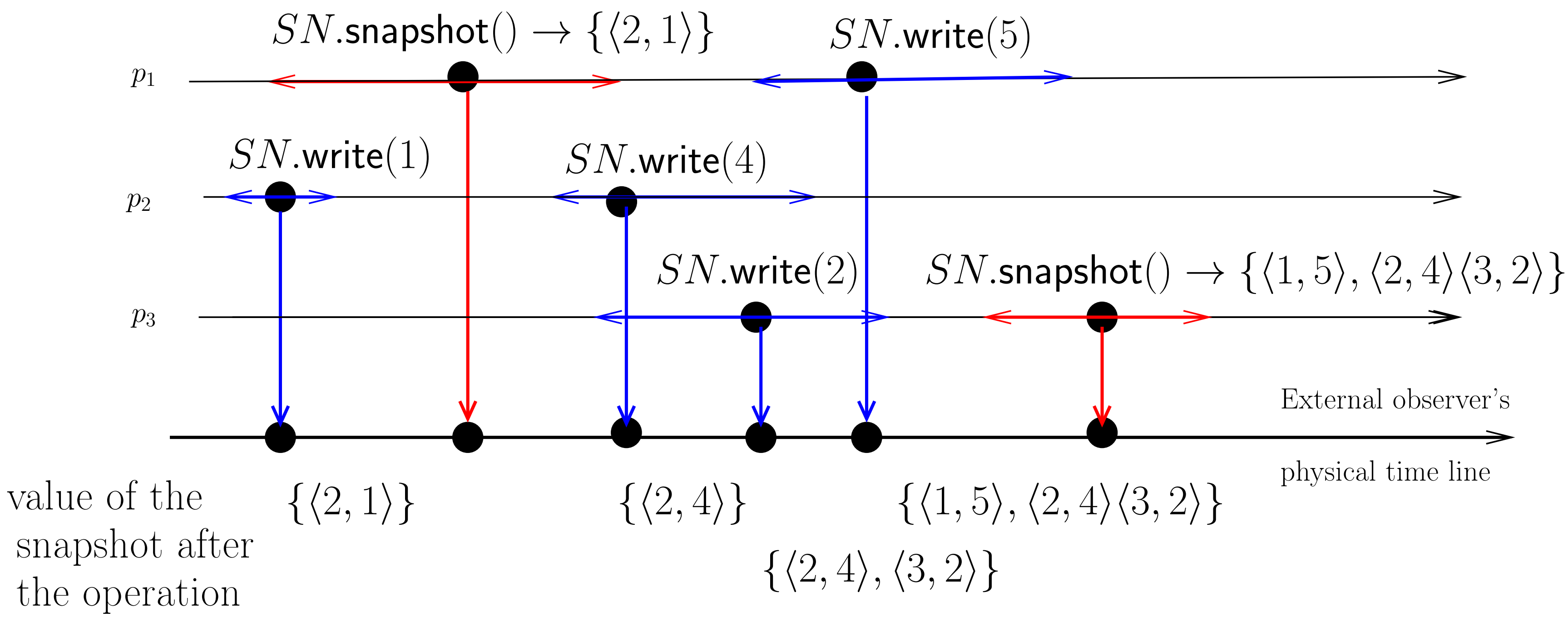
Linearizability example: snapshot object (1)

An object SN containing pairs with two operations

- SN.write(v):
  adds the pair $\langle i, v \rangle$ to SN
  and suppress the previous pair $\langle i, - \rangle \in SN$ if any
- SN.snapshot(): returns the "current" set of pairs

- Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890 (1993)

Linearizability example: snapshot object (2)

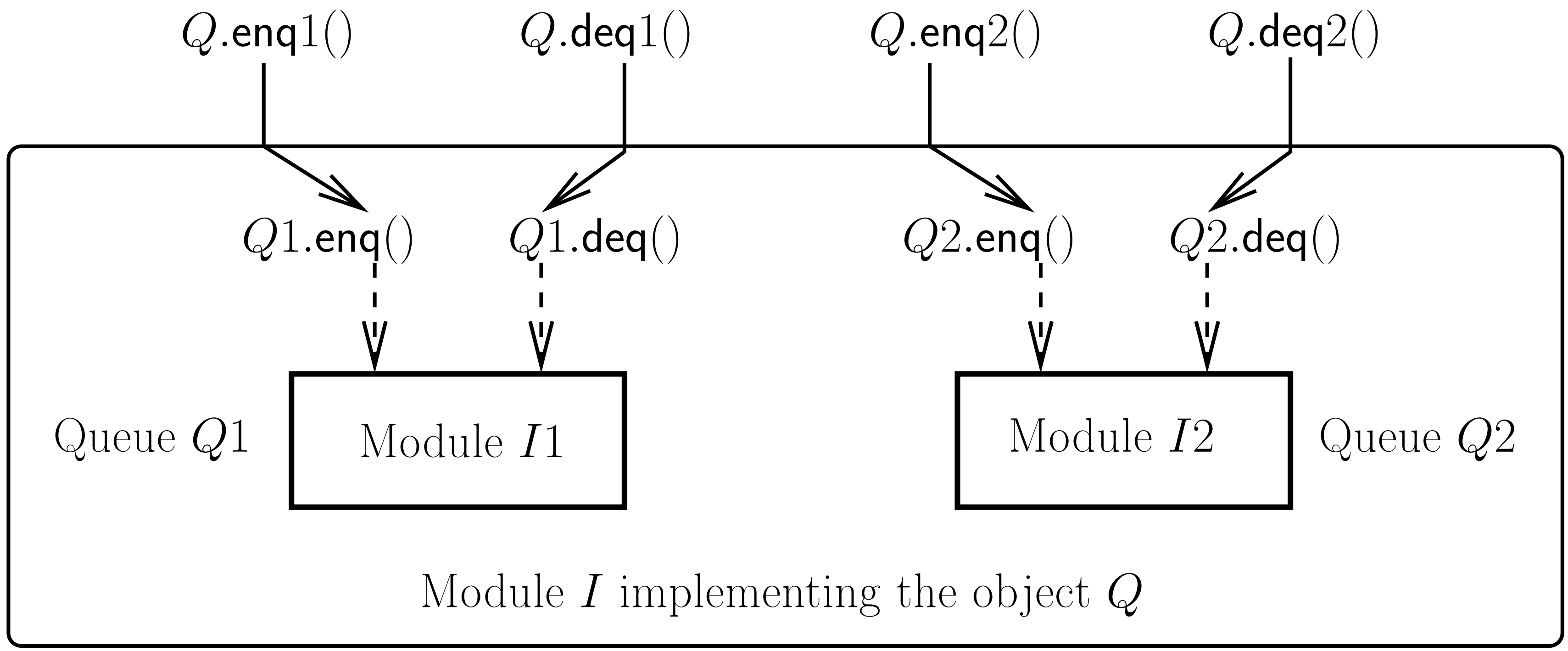


- Internally (implementation): concurrency
- Externally (spec. for users): sequentiality

Fundamental properties of linearizability

- Non-blocking:
  To complete an object operation does not need to wait for another to terminate
- Composability:
  Lineararizable objects compose for free

Composability: example



Sequential consistency

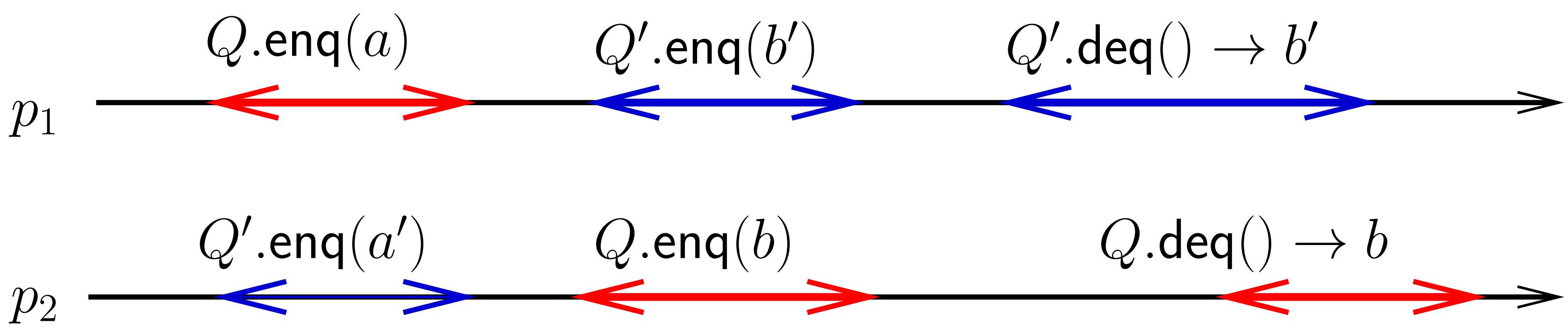
- *How to make a multiprocessor computer that correctly executes multiprocess programs*. L. Lamport
  IEEE Transactions on Computers, C28(9):690-691 (1979)

- Definition:
  an execution of an object is sequentially consistent if if it is possible to totally order all the operations on the object while respecting each process order

- The “witness” total order is “physical” in linearizability and “logical” only in sequential consistency

- Seq. consistent objects do not compose for free!

Example of sequential consistency



A lot of works relaxing linearizability

Many++ works investigated weakening of linearizability

Relaxing linearizability: a few examples

- A scalable lock-free stack algorithm

  D. Hendler, N. Shavit and L. Yerushalmi
  Proc. 32nd ACM SPAA, pp. 206-215 (2004)

- Quasi-linearizability: relaxed consistency for improved concurrency
  Afek Y., Korland G., and Yanovsky E.

  Proc. 14th OPODIS, Springer LNCS 6490, pp. 395-410 (2010)

  Idea: Each run is at a bounded distance of a linearizable run

- Data structures in the multicore age

  Shavit N.,
  Commmunications of the ACM, 54(3):76-84 (2011)

- Local linearizability for concurrent container-type data structures

  Haas A., Henzinger T.A., Holzer A., Kirsch Ch.M, Lippautz M., Payer H., Sezgin A., Sokolova A., and Veith H.
  Proc. 27th CONCUR, LIPIcs Vol. 59, pages 6:1–6:15 (2016)

  Introduced the notion of container object (RW is not a container)

Relaxing linearizability: a few examples: Cont'd

- The computability of relaxed data structures: queues and stacks as examples

  Shavit N. and Taubenfeld G.,
  Distributed Computing, 29(5):395-407 (2016)

- Distributionally linearizable data structures

  Alistarh D., Brown T., Kopinsky J., Li J. and Nadiradze G.,
  Proc. 30th ACM SPAA, ACM Press, pp. 133-142 (2018)

- Intermediate value linearizability: a quantitative correctness condition

  Rinberg A. and Keidar I.,
  Proc. 34th DISC, LIPICs 179, 17 pages (2020)

- Relaxed queues and stacks from read/write operations

  A. Castañeda,S. Rajsbaum, M. Raynal.
  Proc. 24th OPODIS, LIPICs 184, 19 pages (2020)

- Upper and lower bounds for deterministic approximate objects

  Hendler D., Khattabi A., Milani A.,and Travers C.,
  Proc. 41st IEEE ICDCS, LIPICs, pp. 438-448 (2021)

Set-linearizability

- Introduced by Gil Neiger:
  Set linearizability.
  Proc. 13th ACM symposium on Principles of distributed computing (PODC'94),
  Brief announcement, ACM Press, page 396 (1994)

- Later investigated in:

  - ⋆ Hemed N., Rinetzky N., and Vafeiadis V.,
    Modular verification of concurrency-aware linearizability.
    Proc. 29th DISC, Springer LNCS 9363, pp. 371-387 (2015)

  - ⋆ Castañeda A., Rajsbaum S., and Raynal M.,
    Unifying concurrent objects and distributed tasks: interval-linearizability.
    Journal of the ACM, 65(6), Article 45, 42 pages (2018)

Why set-linearizability?

- Motivation example: k-set agreement object
  - ⋆ Each process proposes a value and decides a value
  - ⋆ a decided value is a proposed value
  - ⋆ at most k different values are decided
- Linearizability:
  - ⋆ cannot capture the full generality of k-set agreement (and many other objects)
  - ⋆
  - ⋆ Due to its very definition: restricted to seq. spec.
- need to free from the "burden of the (seq.) past"

What does set-linearizability add

	Linearizability	Set-linearizability
	Atomicity	Atomicity + simultaneity
User level: specification	Sequential	Concurrent
Implementation level	FT + Concurrent	FT +Concurrent

- Due to its very definition: linearizability $\leftrightarrow$ seq. spec.
- Set-linearizability
  - $\star$ allows to capture simultaneity of operations
  - $\star$ captures the notion of *point contention*
- Suited to a class of concurrent object specification

Set-lin = linearizability + simultaneity

Set-lin example: Immediate snapshot object

- Immediate atomic snapshots and fast renaming.
  Borowsky E. and Gafni E., *Proc. 12th ACM PODC'93*, pp. 41–51 (1993)

- A snapshot object with concurrent specification

- A single operation denoted im_snapshot(v)

- When a process p_i invokes im_snapshot(v_i)

  ⋆ it deposits the pair $\langle i, v_i \rangle$ in the object

  ⋆ and returns a set of pairs denoted $view_i$

Set-LIN: Immediate snapshot object

- Termination. If a process invokes im_snapshot() and does not crash, its invocation terminates
- Self-inclusion.
  im_snapshot(v_i) returns $view_i$ to p_i $\Rightarrow$ $(\langle i, v_i\rangle \in view_i)$
- Global inclusion (Containment).
  invocation of im_snapshot(v_i) by p_i returns $view_i$ and invocation of im_snapshot(v_j) by p_j returns $view_j$ $\Rightarrow$ $view_i \subseteq view_j$ or $view_j \subseteq view_i$
- Immediacy.
  $(\langle i, v_i\rangle \in view_j) \wedge (\langle j, v_j\rangle \in view_i) \Rightarrow (view_i = view_j)$

Immediacy $\Rightarrow$ simultaneity

Set-lin: immediate snapshot algorithm

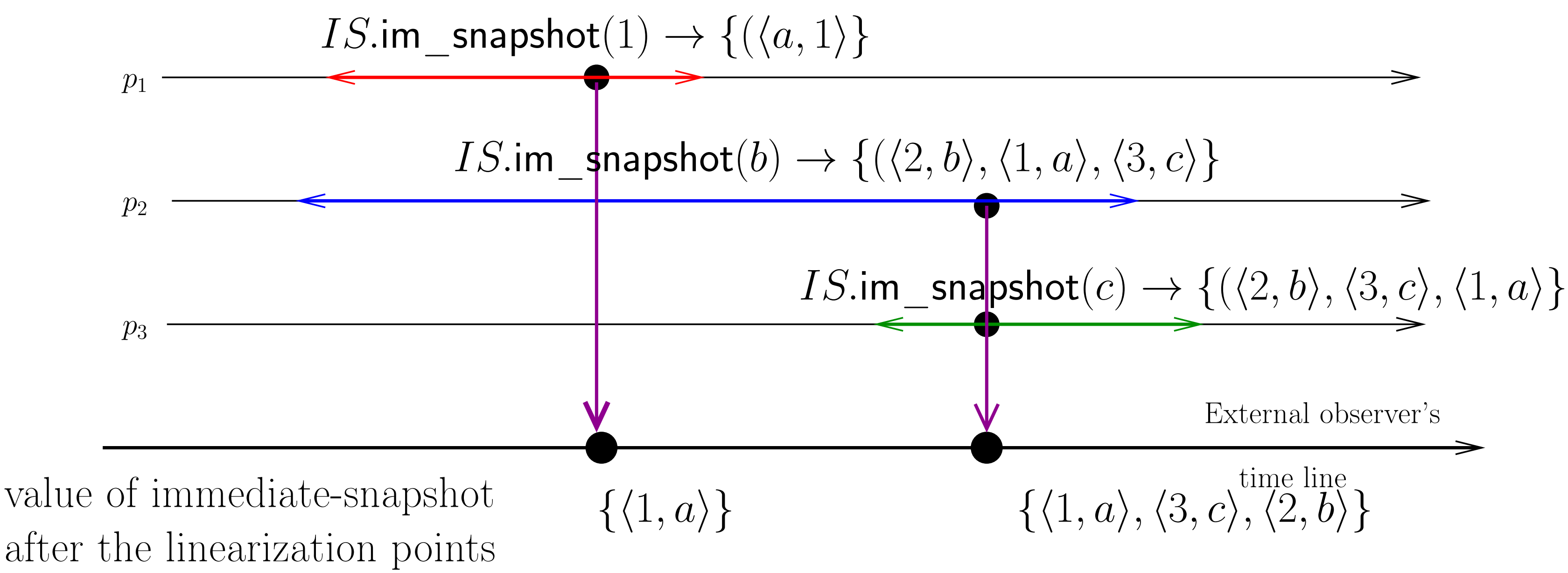
Shared registers:

$MEM[1..n]$ init to $[\perp, \cdots, \perp]$
$LEVEL[1..n]$ init to $[(n+1), \cdots, (n+1)]$

operation im_snapshot(v)**is** % code for process p_i
$MEM[i] \leftarrow v$;
repeat $LEVEL[i] \leftarrow LEVEL[i] - 1$;
(L3) **for each** $j \in \{1, \ldots, n\}$ **do** $level_i[j] \leftarrow LEVEL[j]$ **end for**;
$set_i \leftarrow \{x \mid level_i[x] \leq level_i[i]\}$
until $(|set_i| \geq level_i[i])$ **end repeat**;
(L6) **let** $view_i = \{ \langle x, MEM[x] \rangle \mid x \in set_i \}$;
return($view_i$)
end operation.

Immediate snapshot example of an execution

A possible run of the previous algorithm



Interval-linearizability

What does interval linearizability add

Consist. cond.	Specification	Implementation
Linearizability	Sequentiality	Concurrent
Set Lin.	Lin + simultaneity	Concurrent
Interval Lin.	Set Lin + time ubiquity	Concurrent

- Castañeda A., Rajsbaum S., and Raynal M.,
Unifying concurrent objects and distributed tasks: interval-linearizability.
Journal of the ACM, 65(6), Article 45, 42 pages (2018)

Int-lin: write-snapshot object (definition)

- Its is a snapshot object in which the two operations write() and snapshot() are pieced together into a single operation denoted write_snapshot()

- Properties:

  - ⋆ Self-inclusion: $(\langle i, v_i \rangle \in view_i)$
  - ⋆ Containment: $view_i \subseteq view_j$ or $view_j \subseteq view_i$

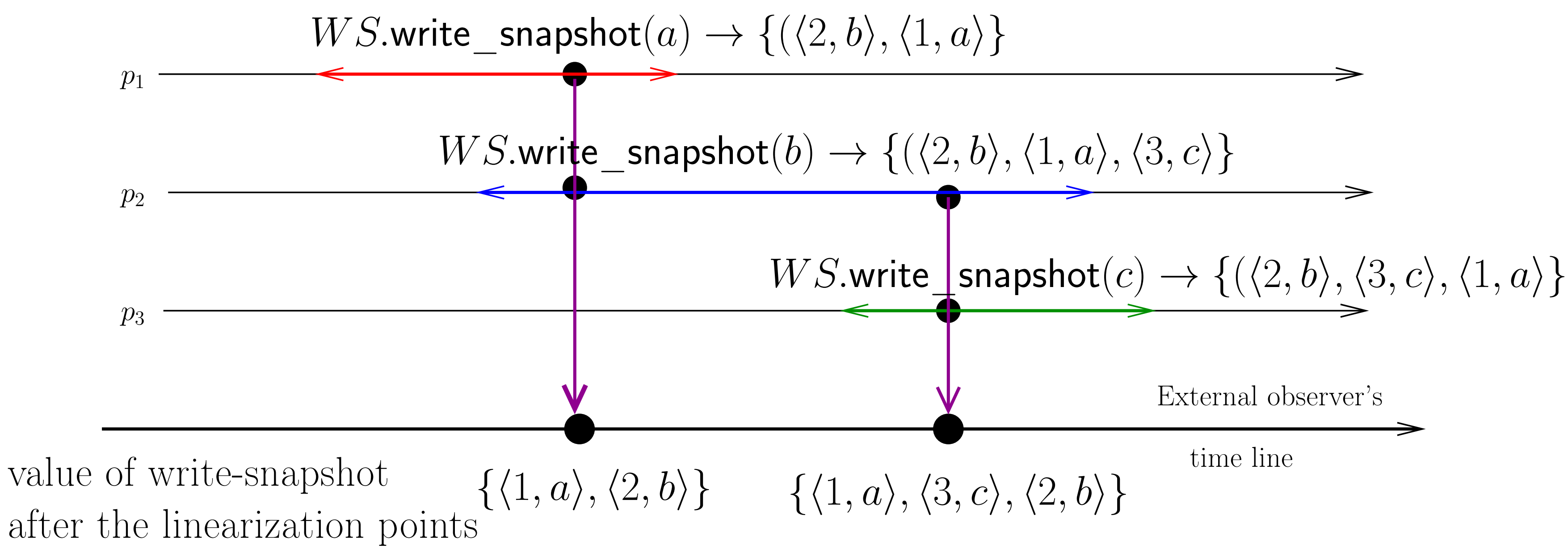
Reminder: Self-inclusion is not a property required by the base snapshot object (operations write() and snapshot())

One-shot write-snapshot object: algorithm

operation write_snapshot(v) **is** % code for process p_i
  $MEM[i] \leftarrow \langle i, v\rangle$;
  $new_i \leftarrow \cup_{1\leq j\leq n}\{\langle j, MEM[j]\rangle$ such that $MEM[j] \neq \perp\}$;
  repeat $old_i \leftarrow new_i$;
    $new_i \leftarrow \cup_{1\leq j\leq n}\{MEM[j]$ such that $MEM[j] \neq \perp\}$
  until $(old_i = new_i)$ **end repeat**;
  return(new_i)
end operation

Write-snapshot: example of an execution

A possible run of the previous algorithm



Lattice agreement

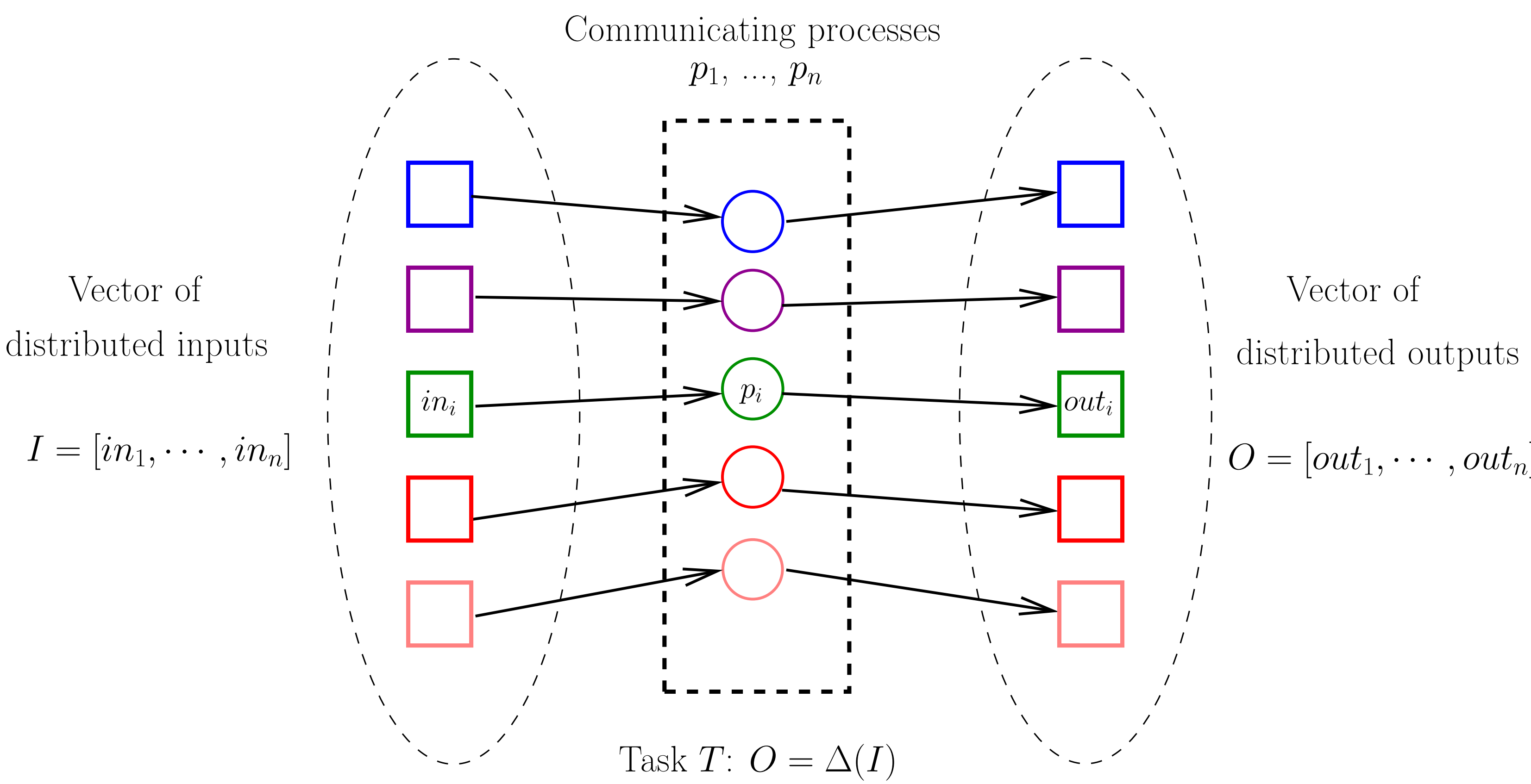
- A set L partially ordered by a binary relation $\sqsubseteq$ s. t. any pair $x, y \in L$ has a least upper bound called *join*
- A one-shot operation propose(v) with input $v \in L$, returns a value $v' \in L$, such that:
  - ⋆ Validity: v' is a join of some proposed values including v and all values returned by previous propose() operations
  - ⋆ Consistency: returned values are ordered by $\sqsubseteq$

Used in distributed state reconciliation:

Accountability and Reconfiguration: Self-Healing Lattice Agreement, OPODIS 2021: 25:1-25:23 (2021), Freitas de Souza L., Kuznetsov P., Rieutord Th., Tucci Piergiovanni S.

Many objects are defined by distributed tasks

Distributed tasks: no notion of "order" on operation execution



Two important theorems

- Concurrent specifications: beyond linearizability
  Goubault E., Ledent J., and Mimram S.,
  22nd OPODIS, LIPIcs 125, 16 pages (2018)

  Theorem:
  Every concurrent specification is interval-linearizable

- Unifying concurrent objects and distributed tasks: interval-linearizability
  Castañeda A., A., Rajsbaum S., and Raynal M.,
  Journal of the ACM, 65(6), 42 pages (2018)

  Theorem:
  interval-linearizable objects and (refined) tasks have the same expressive power (both are complete in the sense they are able to specify any prefix-closed set of well-formed executions)

On progress conditions

On the progress in the presence of failures

(Net effect of asynchrony and failures: mutex is irrelevant)

- 1991: Wait-freedom: If a process does crash (while executing an object operation) it terminates
- 1990: Non-blocking $\sim$ no deadlock
- 2005: Obstruction-freedom: if a process executes alone during a long enough period (and does not crash) it terminates its operation

(All these properties are due to M. Herlihy and co-authors)

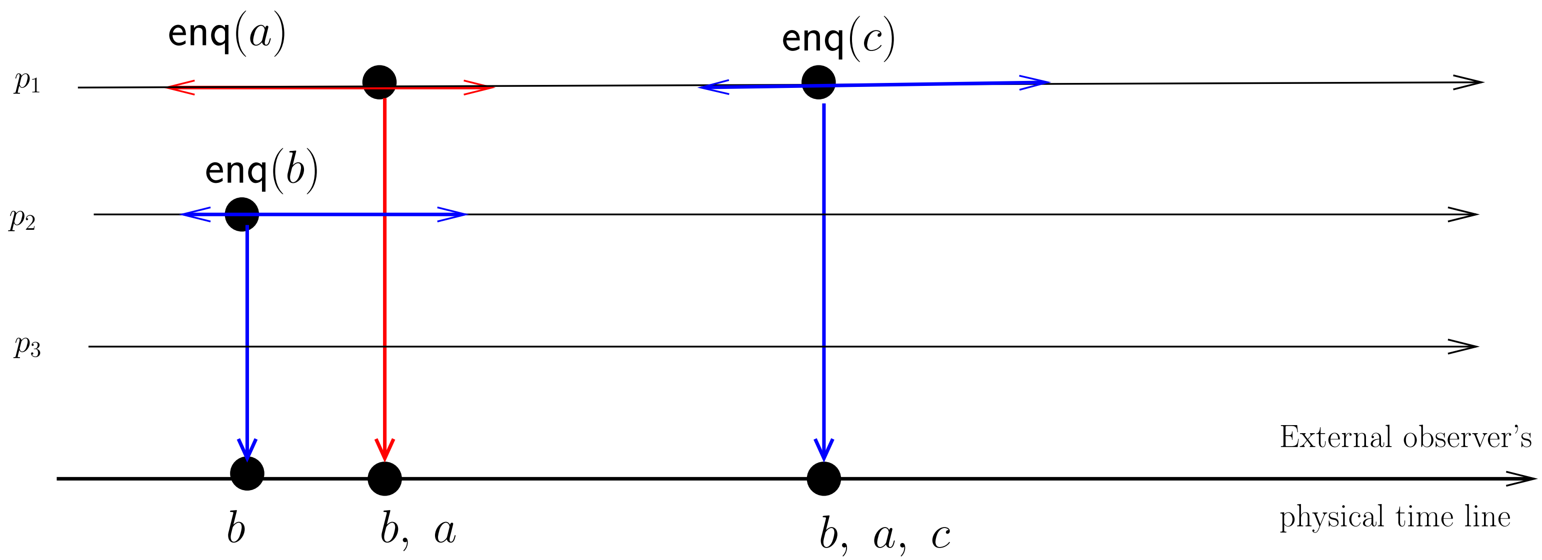
On the interplay betwen safety and liveness

Queue in the consensus number (CN) 1 and 2 worlds

i .e., with the help of the
weakest computability/synchronization power
in the presence of asynchrony and crashes

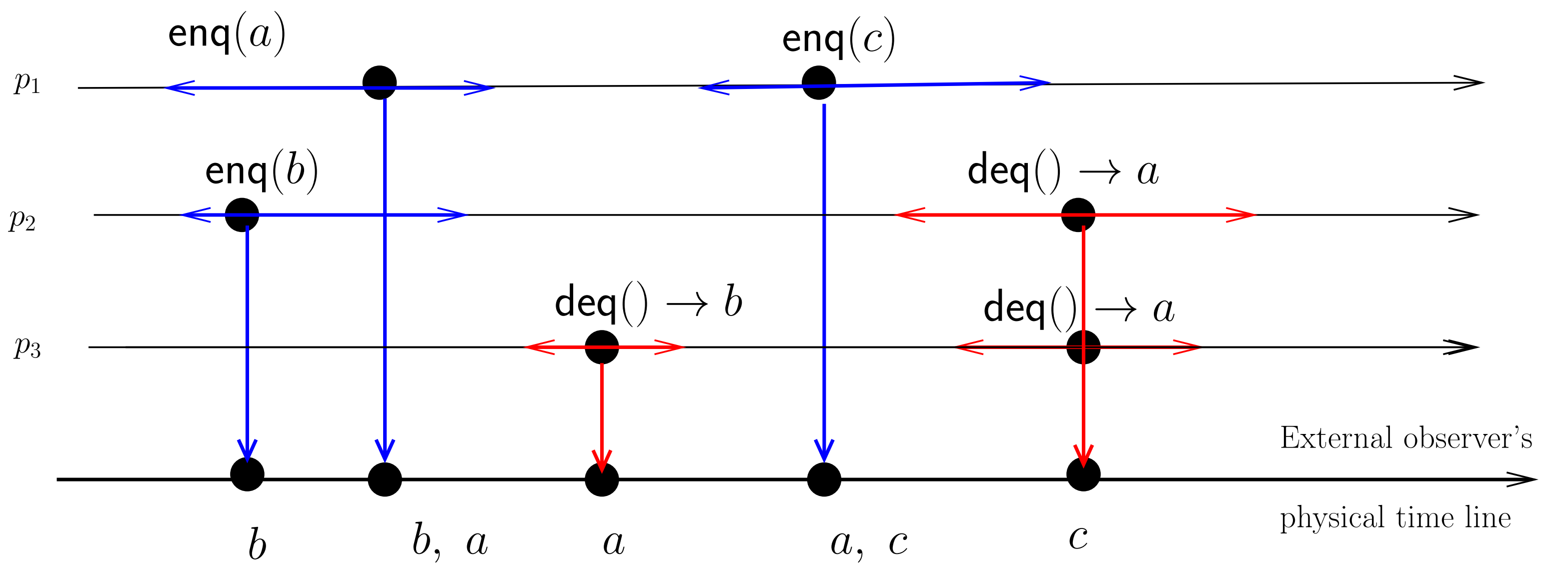
- Castañeda A., Rajsbaum S. and Raynal M., Relaxed queues and stacks from read/write operations. *Proc.24th Conference on Principles of Distributed Systems (OPODIS 2020)*, LIPICS Vol. 184, Article 13, 19 pages (2020)

Additional results



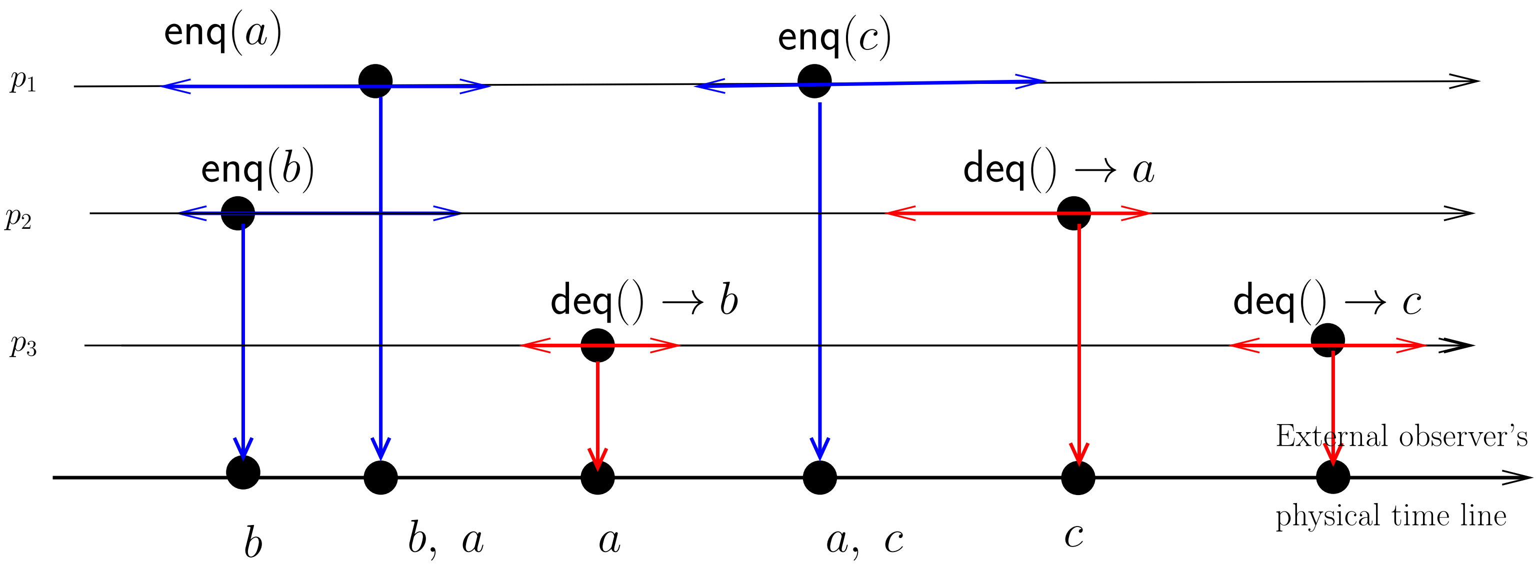
This execution fragment is linearizable

Additional results



This execution fragment is set-linearizable
(See work-stealing for idempotent jobs)

Additional results



Additional results

Base object	Liveness	Safety
CN = 1	enqueue(): wait-freedom dequeue(): non-blocking	enqueue(): linearizability dequeue(): set-linearizability
CN = 1	enqueue(): wait-freedom dequeue(): wait-freedom	enqueue(): linearizability dequeue(): interval-linearizability
CN = 2	enqueue(): wait-freedom dequeue(): non-blocking	enqueue(): linearizability dequeue(): linearizability
CN = 2	enqueue(): wait-freedom dequeue(): wait-freedom	enqueue(): linearizability dequeue(): interval-linearizability

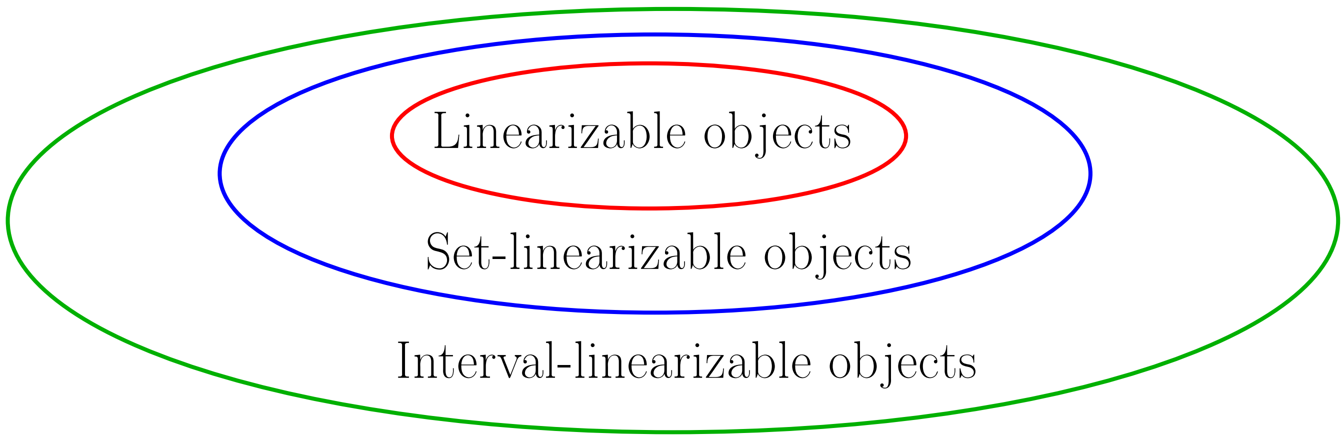
Additional results cont'd

Stack in the consensus number (CN) 1 and 2 worlds

Base object	Liveness	Safety
CN = 1	push(): wait-freedom pop(): wait-freedom	push(): linearizability pop(): set-linearizability
CN = 2	push(): wait-freedom pop(): wait-freedom	push(): linearizability pop(): linearizability

THE GLOBAL PICTURE

Consistency condition	User layer specification	Implementation layer
Linearizability	Sequential	FT + Concurrent
Set-linearizability	concurrent: simultaneity	FT + Concurrent
Int-linearizability	concurrent: time-ubiquity	FT + Concurrent



As Lin, Set lin and Int lin are composable for free!

A look at the underlying theory

Two articles

- Introduced in:

  Unifying concurrent objects and distributed tasks: interval-linearizability
  Castañeda A., A., Rajsbaum S., and Raynal M.,
  Journal of the ACM, 65(6), Article 45, 42 pages (2018)

- Analyzed in:
  Concurrent specifications: beyond linearizability
  Goubault E., Ledent J., and Mimram S.,
  22nd OPODIS, LIPIcs 125, 16 pages (2018)

Notations (1)

- n processes p_1, ..., p_n
- $\mathcal{V}$: values (integers) exchanged by the processes
  - ⋆ inv_i^x: invocations of the object by p_i with input x
  - ⋆ resp_j^y: responses of the object to p_j with output y
- $\mathcal{A}$: set of all the actions (events) on the object
- execution trace: finite seq of actions (events)
- $\mathcal{T} = \mathring{A}^*$: set of possible traces
- ϵ: empty trace
- $T \cdot T'$: trace concatenation

Notations cont'd

- $\pi_i(T)$ trace obtained by removing all the actions of the processes $p_j \neq p_i$
- Alternating trace: $\pi_i(T)$ is empty or alternates between invocations and responses
- If $\pi_i(T)$ terminates with an invocation: pending inv.
- Complete trace: no pending invocation

Specification of concurrent objects (1/2)

Definition

A concurrent specification Σ is
a subset of $\mathcal{T}$ satisfying the following eight properties

- Alternating: every $T \in \Sigma$ is alternating
- Prefix-closed: if $T \cdot T' \in \Sigma$ then $T \in \Sigma$
- non-empty: $\epsilon \in \Sigma$
- receptive: if $T \in \Sigma$ and p_i has no pending invocation, then $T \cdot \mathsf{inv}_i^x \in \Sigma$ for any x

Specification of concurrent objects (2/2)

- Total:

  if $T \in \Sigma$ and p_i has a pending invocation, then it exists $x \in \mathcal{V}$ such that $T \cdot \mathsf{resp}_i^x \in \Sigma$

- Commuting invocations:
  if $T \cdot \mathsf{inv}_i^x \cdot \mathsf{inv}_j^y \cdot T' \in \sigma$ then $T \cdot \mathsf{inv}_j^y \cdot \mathsf{inv}_i^x \cdot T' \in \sigma$

- Commuting responses:
  if $T \cdot \mathsf{resp}_i^x \cdot \mathsf{resp}_j^y \cdot T' \in \sigma$ then $T \cdot \mathsf{resp}_j^y \cdot \mathsf{resp}_i^x \cdot T' \in \sigma$

- Closure under expansions:
  if $T \cdot \mathsf{resp}_j^y \cdot \mathsf{inv}_i^x \cdot T' \in \sigma$ then $T \cdot \mathsf{inv}_i^x \cdot \mathsf{resp}_j^y \cdot T' \in \sigma$

Meaning of "an algo implements an conc. object"

- Consider an automaton-based representation of a prog. language
- Decision function $\delta()$: defines which object the program will call
- Transition function $\tau()$: defines the next state of the object
- An algorithm A (concurrent program) is defined by a set of automata A_i, each one associated with a process p_i
- A implements a concurrent specification Σ if all the traces it generates belong to Σ

Two important theorems

- Concurrent specifications: beyond linearizability
  Goubault E., Ledent J., and Mimram S.,
  22nd OPODIS, LIPIcs 125, 16 pages (2018)

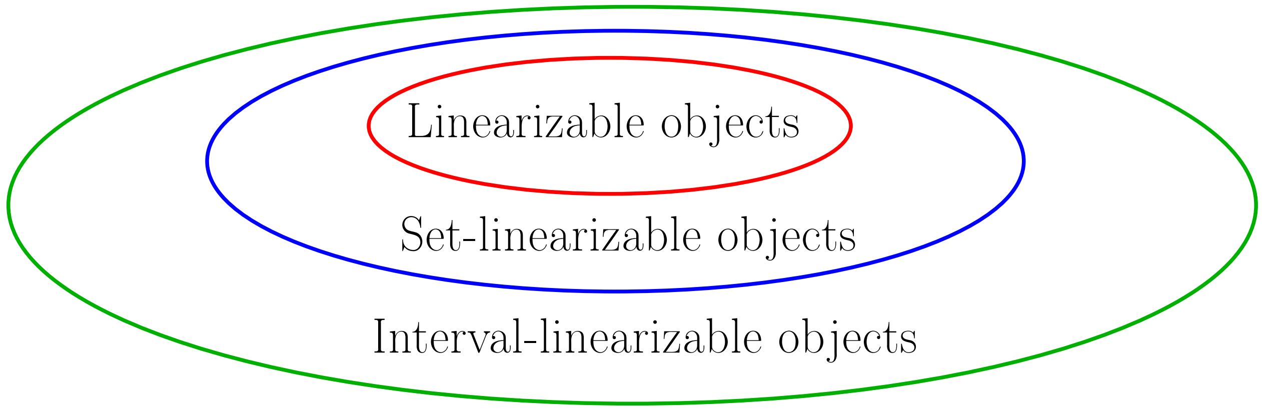
  Theorem:
  Every concurrent specification is interval-linearizable

- Unifying concurrent objects and distributed tasks: interval-linearizability
  Castañeda A., A., Rajsbaum S., and Raynal M.,
  Journal of the ACM, 65(6), 42 pages (2018)

  Theorem:
  interval-linearizable objects and (refined) tasks have the same expressive power and both are complete in the sense that they are able to specify any prefix-closed set of well-formed executions

Conclusion

A visit to



- Concurrent objects
- Specification of concurrent objects
- Linearizability hierarchy

Important:
Int-LIN $\Rightarrow$ Composability (for free) of concurrent objects

Colorin colorado,

est cuento NO se ha acabado...