Brief Announcement: Linearizability: A Typo

Gal Sela
Technion
Israel
galy@cs.technion.ac.il

Maurice Herlihy
Brown University
USA
mph@cs.brown.edu

Erez Petrank
Technion
Israel
erez@cs.technion.ac.il

ABSTRACT

Linearizability [4] is the de facto consistency condition for concurrent objects, widely used in theory and practice. Loosely speaking, linearizability classifies concurrent executions as correct if operations on shared objects appear to take effect instantaneously during the operation execution time. This paper calls attention to a somewhat-neglected aspect of linearizability: restrictions on how pending invocations are handled, an issue that has become increasingly important for software running on systems with non-volatile main memory. Interestingly, the original published definition of linearizability includes a typo (a symbol is missing a prime) that concerns exactly this issue. In this paper we point out the typo and provide an amendment to make the definition complete. We believe that pointing this typo out rigorously and proposing a fix is important and timely.

CCS CONCEPTS

• **Software and its engineering** → **Correctness**; • **Theory of computation** → *Concurrent algorithms*; • **Computing methodologies** → Concurrent algorithms.

KEYWORDS

Linearizability; Correctness; Verification; Concurrent Algorithms; Concurrent Data Structures

ACM Reference Format:
Gal Sela, Maurice Herlihy, and Erez Petrank. 2021. Brief Announcement: Linearizability: A Typo. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing (PODC '21), July 26–30, 2021, Virtual Event, Italy*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3465084.3467944

The full version of this paper is available at [6].

1 INTRODUCTION

Linearizability is the prevalent correctness condition for concurrent executions on shared objects. It determines whether a concurrent execution is correct by relating it to a sequential execution that satisfies the sequential specification of the object. To relate a valid sequential execution to a concurrent one, linearizability specifies an order of the concurrent operations, denoted *linearization order*. On the one hand, linearizability requires that if we execute the operations sequentially one by one according to their linearization order (with the same parameters as in the concurrent execution), we obtain a sequential execution (with the same operation results as in the concurrent execution) that satisfies the sequential specification of the object. On the other hand, linearizability dictates that the linearization order preserve the order of non-overlapping operations in the original concurrent execution. Namely, if an operation op_1 completes before another operation op_2 begins, then op_1 must precede op_2 in the linearization order. A concurrent execution is called linearizable if it can be related as above to a legal sequential execution (i.e., one that satisfies the sequential specification of the object). The formal definition is provided in Section 2.2.

So far, we ignored pending invocations in the execution. These are invocations of operations that start during the execution, but do not complete. The issue that we point out in this paper concerns the treatment of pending invocations. The original linearizability definition provides a treatment of pending invocations, stating which executions with pending invocations are linearizable. However, in this paper we argue that, due to a typo, this treatment of pending invocations is lacking. It contradicts our intuition about linearizable executions, and furthermore, causes the locality and nonblocking properties to not hold. We then propose a simple fix for the typo that fits intuition and obtains these desirable properties also for executions with pending invocations.

Linearizability allows eliding some of the pending invocations, namely, excluding their operations from the related sequential execution. This can be interpreted as operations that do not yet take effect before the execution ends. The rest of the pending invocations appear in the sequential execution with a *response*, i.e., completion and returned results. These can be interpreted as operations in the concurrent execution that have taken effect although their responses have not yet been returned to the caller. The responses appended in the sequential execution are determined in a way that fits the overall execution. In particular, responses are set so that the related sequential execution satisfies the sequential specification of the object.

The problem that arises, due to the typo in the original definition, is that the definition does not restrict the order of operations that have pending invocations, even when these operations take effect, i.e., are included in the related sequential execution. In particular, a pending invocation of an operation *op* may be placed in a linearization order before other operations that completed earlier in the execution, even operations that completed before *op* started. Imagine an execution that starts with an operation op_1 that reads a shared variable x. While x is initially 0, the operation weirdly reads the value 1, and then completes and returns 1. Later a new operation op_2 is invoked on a different process. Operation op_2 writes 1 into x and does not complete before the execution ends. Intuitively, this

This work was supported by the United States - Israel BSF grant No. 2018655.

PODC '21, July 26–30, 2021, Virtual Event, Italy

ACM ISBN 978-1-4503-8548-0/21/07.
https://doi.org/10.1145/3465084.3467944

does not seem like an acceptable linearizable execution. However, under the existing definition with the typo, it is linearizable, because the pending invocation of operation op_2 (that writes 1) can be ordered before the completed operation op_1 that reads 1.

Interestingly, beyond contradicting intuition, the typo in the original definition does not allow it to yield neither locality nor the nonblocking property. In Section 3 we describe the intuitive problem with the typo in the definition and show that it is not local. In Section 3.3 in [6] we show that it is also not nonblocking.

We propose a (syntactically minor) modification to the definition that restricts the linearization order of operations with pending invocations that take effect. Similarly to completed operations, operations with pending invocations are ordered later than any operation that completes before they start. This modification makes the odd execution described above not linearizable. In Section 2 we recall the formal original definition of linearizability (with the typo). In Section 4 we formally specify the amended definition, and in Section 5 we show that the amendment makes odd executions like the one described above non-linearizable as expected. We show that the amendment also makes linearizability local and nonblocking in Sections 5.2 and 5.3 in [6]. We also cover an equivalent definition of linearizability based on linearization points in Section 7 in [6].

It is clear that the flaw in the definition is a typo, not a conceptual error, and the authors' intended meaning is clear in context. We believe no prior paper was rendered incorrect by relying on the original definition. However, linearizability is extremely important for concurrent executions. It has been used in thousands of papers and the definition with the typo has been replicated in numerous subsequent publications. We believe it is important to point out this typo and provide a rigorous discussion and a fix. The issue of pending invocations is becoming increasingly important as architectures with non-volatile main memory become commonplace. Non-volatile memory models encompass various definitions [e.g. 1–3, 5] where a major focus is dealing with invocations pending at the time of a crash. In this realm, pending invocations become critically important, making the fix of this typo timely.

2 SYSTEM MODEL AND LINEARIZABILITY DEFINITION

We follow the terminology of the original linearizability paper [4], which contains additional motivation and detailed explanations.

2.1 Histories Terminology

An execution of a concurrent system is modeled by a history. A *history* is a finite sequence of operation *invocation* and *response* events. Each invocation or response event is associated with some object and some process. An invocation includes also an operation name and argument values, and a response includes a termination condition and results. A response *matches* an invocation if it is associated with the same object and process. An invocation is *pending* in a history if no matching response follows the invocation. An *extension* of H is a history constructed by appending to the end of H responses to zero or more pending invocations of H. A *subhistory* of a history H is a subsequence of the events of H. *complete(H)* is the maximal subhistory of H consisting only of invocations and matching responses, without any pending invocations. For a process P, the *process subhistory* $H|P$ is the subsequence of all events in H associated with the process P. For an object x, the *object subhistory* $H|x$ is the subsequence of all events in H associated with the object x. Two histories H and H' are *equivalent* if for every process P, $H|P = H'|P$.

A history H is *sequential* if it comprises a sequence of pairs of an invocation and a matching response, except possibly the last invocation, which might be the last event in the history, not accompanied by a matching response. A history that is not sequential is *concurrent*. A history is *well-formed* if each of its process subhistories is sequential. A *single-object* history is one in which all events are associated with the same object. A *sequential specification* for an object is a prefix-closed set of single-object sequential histories for that object. A sequential history H is *legal* if each object subhistory $H|x$ belongs to the sequential specification for x.

We recall the exact definition of operations from [4], which is inherent to the definition of linearizability:

Definition 2.1. (Operation) An *operation* in a history is a pair consisting of an invocation and the next matching response.

An operation e_0 *precedes* (synonymously *happens before*) an operation e_1 in a history H if e_0 ends before e_1 begins, namely, e_1's invocation event occurs after e_0's response event in H. Precedence in H induces a partial order on operations of H, denoted $<_H$. Informally, $<_H$ captures the "real-time" precedence order of operations in H. We stress that only invocations that have matching responses are considered *operations* and the order $<_H$ applies only to them.

2.2 Linearizability Definition

The original definition of linearizability according to [4] follows:

Definition 2.2. (Linearizability) A well-formed history H is *linearizable* if it has an extension H' such that:

L1: There exists a legal sequential history S, to which $complete(H')$ is equivalent.

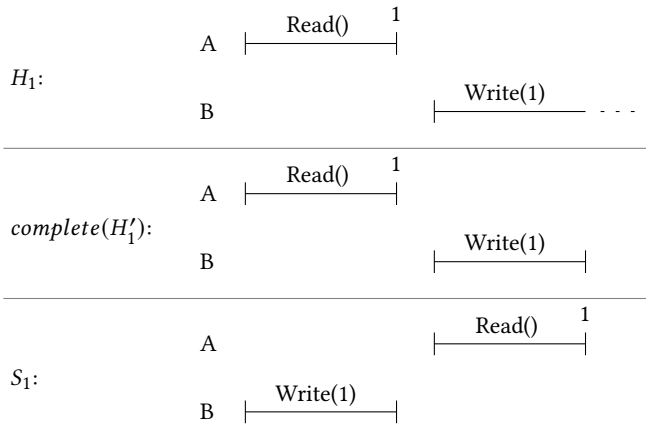
L2: $<_H \subseteq <_S$.

S is denoted the *linearization* of H, and operations that appear in $complete(H')$ are denoted *linearized* operations. Definition 2.2 requires the existence of a linearization S that satisfies two conditions. Condition L1 refers to each process individually, guaranteeing that all its linearized operations are in the same order and with the same results as in the legal sequential history S. Due to this equivalence to a legal sequential history, operations in H act as if they were interleaved at the granularity of complete operations, and adhere to the sequential specification. Condition L2 guarantees that S preserves the order of non-concurrent operations in H, so that it respects possible dependencies between operations in H.

3 ISSUES WITH THE ORIGINAL DEFINITION

Linearizability enforces real-time precedence order on operations. Definition 2.2 enforces it only on operations that include both an invocation and a response in the given history. Fixing the typo extends the enforcement to linearized operations related to pending invocations as well, so that overall, the order is enforced on all linearized operations. We establish the necessity of the typo fix in this section, with a complementary argument regarding the nonblocking property in Section 3.3 in [6].

Figure 1: H_1: an execution on a register, $complete(H_1') = H_1'$: an extension of H_1, and S_1: a linearization of H_1



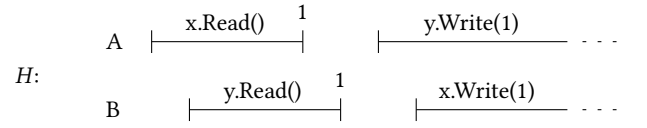
3.1 Executions Counter-Intuitively Classified As Linearizable

We bring executions that intuitively seem non-linearizable, but are classified as linearizable by Definition 2.2. This stems from allowing operations related to pending invocations to appear to take effect before operations by other processes that precede them, since L2 enforces order among H's operations only, and excludes all pending invocations of H. Here we bring an example of an execution on a register, and an example execution on a FIFO queue appears in [6].

Consider the execution H_1 that appears in Figure 1, involving two processes: A and B, operating on a register object initialized to 0. H_1 is intuitively unacceptable, as A cannot "predict the future" and read the value that B has not yet even asked to write, and it cannot distinguish between the given execution and an execution in which B does not invoke any write. Therefore, A should return 0 and not 1. Nevertheless, Definition 2.2 classifies the execution as linearizable, as there is an extension H_1' and a legal sequential execution S_1 (see Figure 1) that adhere to the conditions in Definition 2.2: L1 holds since the events per process in $complete(H_1')$ and in S_1 are identical. L2 vacuously holds since it enforces order only on operations of H_1, and H_1 has a single operation (since a pending invocation does not count as an operation, see Definition 2.1). In particular, L2 does not force B's write to occur in S_1 after the read operation by A.

3.2 Linearizability With The Typo Is Not Local

A property of a concurrent system is *local* (synonymously *composable*) if the system as a whole satisfies it whenever each object in the system satisfies it individually. Locality enables implementing and verifying objects independently, thus maintaining modularity. To demonstrate that linearizability as defined with the typo is not local, we bring the following execution H, involving two processes: A and B, operating on two register objects initialized to 0: x and y.



For each of x and y, the object subhistory of the presented execution H is similar to H_1 (see Figure 1). As shown in Section 3.1, these subhistories are linearizable by Definition 2.2. However, H as a whole is not linearizable by Definition 2.2: An appropriate extension H' must include responses to both writes for the writes to be included in $complete(H')$, otherwise there will be no legal sequential execution S equivalent to $complete(H')$, because the read operations could not legally return 1. Together with the order enforced by Condition L1 on operations by each process, we get the following order requirements, which form a cycle: x.Read() must occur before y.Write(1) for S to preserve the order of A's events (due to Condition L1 that requires $S|A = complete(H')|A$); y.Write(1) must occur before y.Read() for S to be a legal register history (which dictates in particular that 1 be a legal return value of y.Read()); y.Read() must occur before x.Write(1) for S to preserve the order of B's events (due to Condition L1 that requires $S|B = complete(H')|B$); and finally x.Write(1) must occur before x.Read() for S to be legal.

4 AMENDED LINEARIZABILITY

We bring the amended definition of linearizability, which fixes a typo in Condition L2 with a fix that enforces real-time precedence order on linearized operations related to pending invocations:

Definition 4.1. (Amended Linearizability) A well-formed history H is *linearizable* if it has an extension H' such that:

L1: There exists a legal sequential history S, to which $complete(H')$ is equivalent.

L2: $<_{complete(H')} \subseteq <_S$.

L2 is equivalent to $<_{H'} \subseteq <_S$ because $complete(H')$ and H' differ only in pending invocations and according to the definitions in [4], which we use too, pending invocations are not considered operations and a happens-before order does not apply to them. While writing L2 as above makes the definition easier to understand, note that writing L2 as $<_{H'} \subseteq <_S$ provides a fix that is within a single prime sign from the original definition. This missing prime is the typo in the original definition.

Some papers have used an alternative definition of operations, in which pending invocations are also considered as operations, to which a happens-before relation applies. We consider this alternative definition (with the original linearizability definition) in Section 6 in [6] and show that it does not yield an adequate definition for linearizability.

5 ISSUES REVISITED

Here, we explain how the typo fix makes H_1 (see Figure 1) non-linearizable as expected. In Sections 5.2 and 5.3 in [6] we show how the amendment also makes linearizability local and nonblocking.

H_1 is non-linearizable by Definition 4.1: To satisfy Condition L1, an appropriate extension H_1' must include a response for the write operation, otherwise there will be no legal sequential execution S_1 equivalent to $complete(H_1')$, because the read operation could not legally return 1. In addition, an appropriate linearization S_1 must satisfy the amended L2 Condition, which dictates that the read precede the write in S_1 because it precedes it in $complete(H_1')$. This means the read must return 0 in S_1 for S_1 to be legal, which contradicts Condition L1 that requires S_1 to have the same return values as $complete(H_1')$.

REFERENCES

[1] Marcos K Aguilera and Svend Frølund. 2003. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241* (2003). https://hpl.hp.com/techreports/2003/HPL-2003-241.html

[2] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. 2015. Robust shared objects for non-volatile main memory. In *OPODIS*. https://doi.org/10.4230/LIPIcs.OPODIS.2015.20

[3] Rachid Guerraoui and Ron R Levy. 2004. Robust emulations of shared memory in a crash-recovery model. In *ICDCS*. https://doi.org/10.1109/ICDCS.2004.1281605

[4] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *TOPLAS* 12, 3 (1990). https://doi.org/10.1145/78969.78972

[5] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*. https://doi.org/10.1007/978-3-662-53426-7_23

[6] Gal Sela, Maurice Herlihy, and Erez Petrank. 2021. Linearizability: A typo. *arXiv preprint* (2021). arXiv:2105.06737