WIKIPEDIA

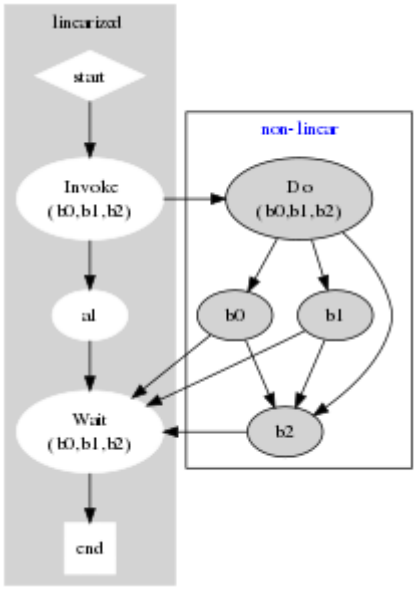
Linearizability

In concurrent programming, an operation (or set of operations) is **linearizable** if it consists of an ordered list of invocation and response events (callbacks), that may be extended by adding response events such that:

1. The extended list can be re-expressed as a sequential history (is serializable), and
2. That sequential history is a subset of the original unextended list.

Informally, this means that the unmodified list of events is linearizable if and only if its invocations were serializable, but some of the responses of the serial schedule have yet to return.[1]

In a concurrent system, processes can access a shared object at the same time. Because multiple processes are accessing a single object, there may arise a situation in which while one process is accessing the object, another process changes its contents. This example demonstrates the need for linearizability. In a linearizable system although operations overlap on a shared object, each operation appears to take place instantaneously. Linearizability is a strong correctness condition, which constrains what outputs are possible when an object is accessed by multiple processes concurrently. It is a safety property which ensures that operations do not complete in an unexpected or unpredictable manner. If a system is linearizable it allows a programmer to reason about the system.[2]



In grey a linear sub-history, processes beginning in b do not have a linearizable history because b0 or b1 may complete in either order before b2 occurs.

Contents

History of linearizability

Linearizability was first introduced as a consistency model by Herlihy and Wing in 1987. It encompassed more restrictive definitions of atomic, such as "an atomic operation is one which cannot be (or is not) interrupted by concurrent operations", which are usually vague about when an operation is considered to begin and end.

An atomic object can be understood immediately and completely from its sequential definition, as a set of operations run in parallel which always appear to occur one after the other; no inconsistencies may emerge. Specifically, linearizability guarantees that the invariants of a system are *observed* and *preserved* by all operations: if all operations individually preserve an invariant, the system as a whole will.

Definition of linearizability

A concurrent system consists of a collection of processes communicating through shared data structures or objects. Linearizability is important in these concurrent systems where objects may be accessed by multiple processes at the same time and a programmer needs to be able to reason about the expected results. An execution of a concurrent system results in a *history*, an ordered sequence of completed operations.

A *history* is a sequence of *invocations* and *responses* made of an object by a set of threads or processes. An invocation can be thought of as the start of an operation, and the response being the signaled end of that operation. Each invocation of a function will have a subsequent response. This can be used to model any use of an object. Suppose, for example, that two threads, A and B, both attempt to grab a lock, backing off if it's already taken. This would be modeled as both threads invoking the lock operation, then both threads receiving a response, one successful, one not.

A invokes *lock*	B invokes *lock*	A gets "failed" response	B gets "successful" response

A *sequential* history is one in which all invocations have immediate responses, that is the invocation and response are considered to take place instantaneously. A sequential history should be trivial to reason about, as it has no real concurrency; the previous example was not sequential, and thus is hard to reason about. This is where linearizability comes in.

A history σ is *linearizable* if there is a linear order of the completed operations such that:

1. For every completed operation in σ, the operation returns the same result in the execution as the operation would return if every operation was completed one by one in order σ.
2. If an operation op_1 completes (gets a response) before op_2 begins (invokes), then op_1 precedes op_2 in σ.[1]

In other words:

- its invocations and responses can be reordered to yield a sequential history;
- that sequential history is correct according to the sequential definition of the object;
- if a response preceded an invocation in the original history, it must still precede it in the sequential reordering.

(Note that the first two bullet points here match serializability: the operations appear to happen in some order. It is the last point which is unique to linearizability, and is thus the major contribution of Herlihy and Wing.)[1]

Let us look at two ways of reordering the locking example above.

A invokes *lock*	A gets "failed" response	B invokes *lock*	B gets "successful" response

Reordering B's invocation below A's response yields a sequential history. This is easy to reason about, as all operations now happen in an obvious order. Unfortunately, it doesn't match the sequential definition of the object (it doesn't match the semantics of the program): A should have successfully obtained the lock, and B should have subsequently aborted.

B invokes *lock*	B gets "successful" response	A invokes *lock*	A gets "failed" response

This is another correct sequential history. It is also a linearization! Note that the definition of linearizability only precludes responses that precede invocations from being reordered; since the original history had no responses before invocations, we can reorder it as we wish. Hence the original history is indeed linearizable.

An object (as opposed to a history) is linearizable if all valid histories of its use can be linearized. Note that this is a much harder assertion to prove.

Linearizability versus serializability

Consider the following history, again of two objects interacting with a lock:

A invokes lock	A successfully locks	B invokes unlock	B successfully unlocks	A invokes unlock	A successfully unlocks

This history is not valid because there is a point at which both A and B hold the lock; moreover, it cannot be reordered to a valid sequential history without violating the ordering rule. Therefore, it is not linearizable. However, under serializability, B's unlock operation may be moved to *before* A's original lock, which is a valid history (assuming the object begins the history in a locked state):

B invokes unlock	B successfully unlocks	A invokes lock	A successfully locks	A invokes unlock	A successfully unlocks

This reordering is sensible provided there is no alternative means of communicating between A and B. Linearizability is better when considering individual objects separately, as the reordering restrictions ensure that multiple linearizable objects are, considered as a whole, still linearizable.

Linearization points

This definition of linearizability is equivalent to the following:

- All function calls have a *linearization point* at some instant between their invocation and their response.
- All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition.

This alternative is usually much easier to prove. It is also much easier to reason about as a user, largely due to its intuitiveness. This property of occurring instantaneously, or indivisibly, leads to the use of the term *atomic* as an alternative to the longer "linearizable".[1]

In the examples above, the linearization point of the counter built on compare-and-swap is the linearization point of the first (and only) successful compare-and-swap update. The counter built using locking can be considered to linearize at any moment while the locks are held, since any potentially conflicting operations are excluded from running during that period.

Primitive atomic instructions

Processors have instructions that can be used to implement locking and lock-free and wait-free algorithms. The ability to temporarily inhibit interrupts, ensuring that the currently running process cannot be context switched, also suffices on a uniprocessor. These instructions are used directly by compiler and operating system writers but are also abstracted and exposed as bytecodes and library functions in higher-level languages:

- atomic read-write;
- atomic swap (the RDLK instruction in some Burroughs mainframes, and the XCHG x86 instruction);
- test-and-set;
- fetch-and-add;
- compare-and-swap;

- load-link/store-conditional.

Most processors include store operations that are not atomic with respect to memory. These include multiple-word stores and string operations. Should a high priority interrupt occur when a portion of the store is complete, the operation must be completed when the interrupt level is returned. The routine that processes the interrupt must not access the memory being changed. It is important to take this into account when writing interrupt routines.

When there are multiple instructions which must be completed without interruption, a CPU instruction which temporarily disables interrupts is used. This must be kept to only a few instructions and the interrupts must be re-enabled to avoid unacceptable response time to interrupts or even losing interrupts. This mechanism is not sufficient in a multi-processor environment since each CPU can interfere with the process regardless of whether interrupts occur or not. Further, in the presence of an instruction pipeline, uninterruptible operations present a security risk, as they can potentially be chained in an infinite loop to create a denial of service attack, as in the Cyrix coma bug.

The C standard and SUSv3 provide `sig_atomic_t` for simple atomic reads and writes; incrementing or decrementing is not guaranteed to be atomic.[3] More complex atomic operations are available in C11, which provides `stdatomic.h`.

The ARM instruction set provides `LDREX` and `STREX` instructions which can be used to implement atomic memory access by using exclusive monitors implemented in the processor to track memory accesses for a specific address.[4] However, if a context switch occurs between calls to `LDREX` and `STREX`, the documentation notes that `STREX` will fail, indicating the operation should be retried.

High-level atomic operations

The easiest way to achieve linearizability is running groups of primitive operations in a critical section. Strictly, independent operations can then be carefully permitted to overlap their critical sections, provided this does not violate linearizability. Such an approach must balance the cost of large numbers of locks against the benefits of increased parallelism.

Another approach, favoured by researchers (but not yet widely used in the software industry), is to design a linearizable object using the native atomic primitives provided by the hardware. This has the potential to maximise available parallelism and minimise synchronisation costs, but requires mathematical proofs which show that the objects behave correctly.

A promising hybrid of these two is to provide a transactional memory abstraction. As with critical sections, the user marks sequential code that must be run in isolation from other threads. The implementation then ensures the code executes atomically. This style of abstraction is common when interacting with databases; for instance, when using the Spring Framework, annotating a method with @Transactional will ensure all enclosed database interactions occur in a single database transaction. Transactional memory goes a step further, ensuring that all memory interactions occur atomically. As with database transactions, issues arise regarding composition of transactions, especially database and in-memory transactions.

A common theme when designing linearizable objects is to provide an all-or-nothing interface: either an operation succeeds completely, or it fails and does nothing. (ACID databases refer to this principle as atomicity.) If the operation fails (usually due to concurrent operations), the user must retry, usually performing a different operation. For example:

- Compare-and-swap writes a new value into a location only if the latter's contents matches a supplied old value. This is commonly used in a read-modify-CAS sequence: the user reads the location, computes a new value to write, and writes it with a CAS (compare-and-swap); if the value changes concurrently, the CAS will fail and the user tries again.
- Load-link/store-conditional encodes this pattern more directly: the user reads the location with load-link, computes a new value to write, and writes it with store-conditional; if the value has changed concurrently, the SC (store-conditional) will fail and the user tries again.

- In a database transaction, if the transaction cannot be completed due to a concurrent operation (e.g. in a deadlock), the transaction will be aborted and the user must try again.

Examples of linearizability

Counters

To demonstrate the power and necessity of linearizability we will consider a simple counter which different processes can increment.

We would like to implement a counter object which multiple processes can access. Many common systems make use of counters to keep track of the number of times an event has occurred.

The counter object can be accessed by multiple processes and has two available operations.

1. Increment - adds 1 to the value stored in the counter, return acknowledgement
2. Read - returns the current value stored in the counter without changing it.

We will attempt to implement this counter object using shared registers

Our first attempt which we will see is non-linearizable has the following implementation using one shared register among the processes.

Non-atomic

The naive, non-atomic implementation:

Increment:

1. Read the value in the register R
2. Add one to the value
3. Writes the new value back into register R

Read:

Read register R

This simple implementation is not linearizable, as is demonstrated by the following example.

Imagine two processes are running accessing the single counter object initialized to have value 0:

1. The first process reads the value in the register as 0.
2. The first process adds one to the value, the counter's value should be 1, but before it has finished writing the new value back to the register it may become suspended, meanwhile the second process is running:
3. The second process reads the value in the register, which is still equal to 0;
4. The second process adds one to the value;
5. the second process writes the new value into the register, the register now has value 1.

The second process is finished running and the first process continues running from where it left off:

1. The first process writes 1 into the register, unaware that the other process has already updated the value in the register to 1.

In the above example, two processes invoked an increment command, however the value of the object only increased from 0 to 1, instead of 2 as it should have. One of the increment operations was lost as a result of the system not being linearizable.

The above example shows the need for carefully thinking through implementations of data structures and how linearizability can have an effect on the correctness of the system.

Atomic

To implement a linearizable or atomic counter object we will modify our previous implementation so **each process P_i will use its own register R_i**

Each process increments and reads according to the following algorithm:

Increment:

1. Read value in register R_i.
2. Add one to the value.
3. Write new value back into R_i

Read:

1. Read registers R_1, R_2, ... R_n.
2. Return the sum of all registers.

This implementation solves the problem with our original implementation. In this system the increment operations are linearized at the write step. The linearization point of an increment operation is when that operation writes the new value in its register R_i. The read operations are linearized to a point in the system when the value returned by the read is equal to the sum of all the values stored in each register R_i.

This is a trivial example. In a real system, the operations can be more complex and the errors introduced extremely subtle. For example, reading a 64-bit value from memory may actually be implemented as two sequential reads of two 32-bit memory locations. If a process has only read the first 32 bits, and before it reads the second 32 bits the value in memory gets changed, it will have neither the original value nor the new value but a mixed-up value.

Furthermore, the specific order in which the processes run can change the results, making such an error difficult to detect, reproduce and debug.

Compare-and-swap

Most systems provide an atomic compare-and-swap instruction that reads from a memory location, compares the value with an "expected" one provided by the user, and writes out a "new" value if the two match, returning whether the update succeeded. We can use this to fix the non-atomic counter algorithm as follows:

1. Read the value in the memory location;
2. add one to the value;
3. use compare-and-swap to write the incremented value back;
4. retry if the value read in by the compare-and-swap did not match the value we originally read.

Since the compare-and-swap occurs (or appears to occur) instantaneously, if another process updates the location while we are in-progress, the compare-and-swap is guaranteed to fail.

Fetch-and-increment

Many systems provide an atomic fetch-and-increment instruction that reads from a memory location, unconditionally writes a new value (the old value plus one), and returns the old value. We can use this to fix the non-atomic counter algorithm as follows:

1. Use fetch-and-increment to read the old value and write the incremented value back.

Using fetch-and increment is always better (requires fewer memory references) for some algorithms—such as the one shown here—than compare-and-swap,[5] even though Herlihy earlier proved that compare-and-swap is better for certain other algorithms that can't be implemented at all using only fetch-and-increment. So CPU designs with both

fetch-and-increment and compare-and-swap (or equivalent instructions) may be a better choice than ones with only one or the other.[5]

Locking

Another approach is to turn the naive algorithm into a critical section, preventing other threads from disrupting it, using a lock. Once again fixing the non-atomic counter algorithm:

1. Acquire a lock, excluding other threads from running the critical section (steps 2-4) at the same time;
2. read the value in the memory location;
3. add one to the value;
4. write the incremented value back to the memory location;
5. release the lock.

This strategy works as expected; the lock prevents other threads from updating the value until it is released. However, when compared with direct use of atomic operations, it can suffer from significant overhead due to lock contention. To improve program performance, it may therefore be a good idea to replace simple critical sections with atomic operations for non-blocking synchronization (as we have just done for the counter with compare-and-swap and fetch-and-increment), instead of the other way around, but unfortunately a significant improvement is not guaranteed and lock-free algorithms can easily become too complicated to be worth the effort.

See also

- Atomic transaction
- Consistency model
- ACID
- Read-copy-update (RCU)
- Read-modify-write
- Time of check to time of use

References

1. Herlihy, Maurice P.; Wing, Jeannette M. (1990). "Linearizability: A Correctness Condition for Concurrent Objects". *ACM Transactions on Programming Languages and Systems*. **12** (3): 463–492. CiteSeerX 10.1.1.142.5315 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.5315). doi:10.1145/78969.78972 (https://doi.org/10.1145%2F78969.78972).
2. Shavit, Nir and Taubenfel,Gadi (2016). "The Computability of Relaxed Data Structures: Queues and Stacks as Examples" (http://www.faculty.idc.ac.il/gadi/MyPapers/2015ST-RelaxedDataStructures.pdf) (PDF). *Distributed Computing*. **29** (5): 396–407. doi:10.1007/s00446-016-0272-0 (https://doi.org/10.1007%2Fs00446-016-0272-0).
3. Kerrisk, Michael (7 September 2018). *The Linux Programming Interface* (https://books.google.de/books?id=2SAQAQAAQBAJ&pg=PA428). No Starch Press. ISBN 9781593272203 – via Google Books.
4. "ARM Synchronization Primitives Development Article" (https://developer.arm.com/products/architecture/a-profile/docs/dht0008/latest/1-arm-synchronization-primitives).
5. Fich, Faith; Hendler, Danny; Shavit, Nir (2004). "On the inherent weakness of conditional synchronization primitives". *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing – PODC '04*. New York, NY: ACM. pp. 80–87. doi:10.1145/1011767.1011780 (https://doi.org/10.1145%2F1011767.1011780). ISBN 978-1-58113-802-3.

Further reading

- Herlihy, Maurice P.; Wing, Jeannette M. (1987). *Axioms for Concurrent Objects* (http://repository.cmu.edu/cgi/viewcontent.cgi?article=2597&context=compsci). *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on*

Principles of Programming Languages, POPL '87. p. 13. doi:10.1145/41625.41627 (https://doi.org/10.1145%2F41625.41627). ISBN 978-0-89791-215-0.

- Herlihy, Maurice P. (1990). *A Methodology for Implementing Highly Concurrent Data Structures*. *ACM SIGPLAN Notices*. **25**. pp. 197–206. CiteSeerX 10.1.1.186.6400 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.186.6400). doi:10.1145/99164.99185 (https://doi.org/10.1145%2F99164.99185). ISBN 978-0-89791-350-8.
- Herlihy, Maurice P.; Wing, Jeannette M. (1990). "Linearizability: A Correctness Condition for Concurrent Objects". *ACM Transactions on Programming Languages and Systems*. **12** (3): 463–492. CiteSeerX 10.1.1.142.5315 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.5315). doi:10.1145/78969.78972 (https://doi.org/10.1145%2F78969.78972).
- Aphyr. "Strong Consistency Models" (https://aphyr.com/posts/313-strong-consistency-models). *aphyr.com*. Aphyr. Retrieved 13 April 2018.
