## Data Intensive Applications (Page - 329)

## 

## ---------------------------------------------------------------------------------------------------------------------

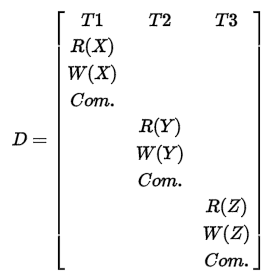
## [Linearizability versus Serializability](http://www.bailis.org/blog/linearizability-versus-serializability/)

Schedule

In the fields of [databases](https://en.wikipedia.org/wiki/Database) and [transaction processing](https://en.wikipedia.org/wiki/Transaction_processing) (transaction management), a **schedule** (or **history**) of a system is an abstract model to describe execution of transactions running in the system. Often it is a *list* of operations (actions) ordered by time, performed by a set of [transactions](https://en.wikipedia.org/wiki/Database_transaction) that are executed together in the system. If order in time between certain operations is not determined by the system, then a [*partial order*](https://en.wikipedia.org/wiki/Partial_order) is used. Examples of such operations are requesting a read operation, reading, writing, aborting, committing, requesting lock, locking, etc. Not all transaction operation types should be included in a schedule, and typically only selected operation types (e.g., data access operations) are included, as needed to reason about and describe certain phenomena. Schedules and schedule properties are fundamental concepts in database [concurrency control](https://en.wikipedia.org/wiki/Concurrency_control) theory.

Formal description

The following is an example of a schedule:

{\displaystyle D={\begin{bmatrix}T1&T2&T3\\R(X)&&\\W(X)&&\\Com.&&\\&R(Y)&\\&W(Y)&\\&Com.&\\&&R(Z)\\&&W(Z)\\&&Com.\end{bmatrix}}}

In this example, the horizontal axis represents the different transactions in the schedule D. The vertical axis represents time order of operations. Schedule D consists of three transactions T1, T2, T3. The schedule describes the actions of the transactions as seen by the [DBMS](https://en.wikipedia.org/wiki/DBMS). First T1 Reads and Writes to object X, and then Commits. Then T2 Reads and Writes to object Y and Commits, and finally T3 Reads and Writes to object Z and Commits. This is an example of a *serial* schedule, i.e., sequential with no overlap in time, because the actions of in all three transactions are sequential, and the transactions are not interleaved in time.

Representing the schedule D above by a table (rather than a list) is just for the convenience of identifying each transaction's operations in a glance. This notation is used throughout the article below. A more common way in the technical literature for representing such schedule is by a list:

D = R1(X) W1(X) Com1 R2(Y) W2(Y) Com2 R3(Z) W3(Z) Com3

Usually, for the purpose of reasoning about concurrency control in databases, an operation is modeled as [atomic](https://en.wikipedia.org/wiki/Atomic_operation), occurring at a point in time, without duration. When this is not satisfactory start and end time-points and possibly other point events are specified (rarely). Real executed operations always have some duration and specified respective times of occurrence of events within them (e.g., "exact" times of beginning and completion), but for concurrency control reasoning usually only the precedence in time of the whole operations (without considering the quite complex details of each operation) matters, i.e., which operation is before, or after another operation. Furthermore, in many cases the before/after relationships between two specific operations do not matter and should not be specified, while being specified for other pairs of operations.

In general operations of transactions in a schedule can interleave (i.e., transactions can be executed concurrently), while time orders between operations in each transaction remain unchanged as implied by the transaction's program. Since not always time orders between all operations of all transactions matter and need to be specified, a schedule is, in general, a [partial order](https://en.wikipedia.org/wiki/Partial_order) between operations rather than a [total order](https://en.wikipedia.org/wiki/Total_order) (where order for each pair is determined, as in a list of operations). Also in the general case each transaction may consist of several processes, and itself be properly represented by a partial order of operations, rather than a total order. Thus in general a schedule is a partial order of operations, containing ([embedding](https://en.wikipedia.org/wiki/Embedding)) the partial orders of all its transactions.

Time-order between two operations can be represented by an [ordered pair](https://en.wikipedia.org/wiki/Ordered_pair) of these operations (e.g., the existence of a pair (OP1,OP2) means that OP1 is always before OP2), and a schedule in the general case is a [set](https://en.wikipedia.org/wiki/Set_(mathematics)) of such ordered pairs. Such a set, a schedule, is a [partial order](https://en.wikipedia.org/wiki/Partial_order) which can be represented by an [acyclic directed graph](https://en.wikipedia.org/wiki/Acyclic_directed_graph) (or *directed acyclic graph*, DAG) with operations as nodes and time-order as a [directed edge](https://en.wikipedia.org/wiki/Directed_edge) (no cycles are allowed since a cycle means that a first (any) operation on a cycle can be both before and after (any) another second operation on the cycle, which contradicts our perception of [Time](https://en.wikipedia.org/wiki/Time)). In many cases a graphical representation of such graph is used to demonstrate a schedule.

**Comment:** Since a list of operations (and the table notation used in this article) always represents a total order between operations, schedules that are not a total order cannot be represented by a list (but always can be represented by a DAG).

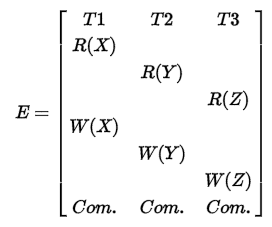
Types of schedule

**Serial**

The transactions are executed non-interleaved (see example above) i.e., a serial schedule is one in which no transaction starts until a running transaction has ended.

**Serializable**

A schedule that is equivalent (in its outcome) to a serial schedule has the [serializability](https://en.wikipedia.org/wiki/Serializability) property. In schedule E, the order in which the actions of the transactions are executed is not the same as in D, but in the end, E gives the same result as D.

{\displaystyle E={\begin{bmatrix}T1&T2&T3\\R(X)&&\\&R(Y)&\\&&R(Z)\\W(X)&&\\&W(Y)&\\&&W(Z)\\Com.&Com.&Com.\end{bmatrix}}}

**Conflicting actions**

Two actions are said to be in conflict (conflicting pair) if:

1. The actions belong to different transactions.
2. At least one of the actions is a write operation.
3. The actions access the same object (read or write).

The following set of actions is conflicting:

* R1(X), W2(X), W3(X) (3 conflicting pairs)

While the following sets of actions are not:

* R1(X), R2(X), R3(X)
* R1(X), W2(Y), R3(X)

**Conflict equivalence**

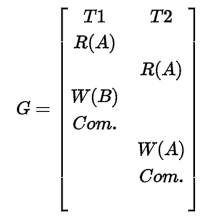
The schedules S1 and S2 are said to be conflict-equivalent if the following two conditions are satisfied:

1. Both schedules S1 and S2 involve the same set of transactions (including ordering of actions within each transaction).
2. Both schedules have same set of conflicting operations.

**Conflict-serializable**

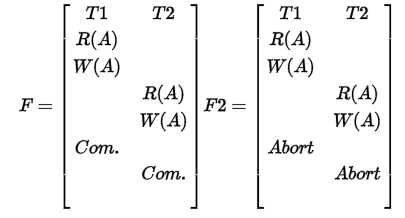
A schedule is said to be conflict-serializable when the schedule is conflict-equivalent to one or more serial schedules.

Another definition for conflict-serializability is that a schedule is conflict-serializable if and only if its [precedence graph](https://en.wikipedia.org/wiki/Precedence_graph)/serializability graph, when only committed transactions are considered, is acyclic (if the graph is defined to include also uncommitted transactions, then cycles involving uncommitted transactions may occur without conflict serializability violation).



**Recoverable**

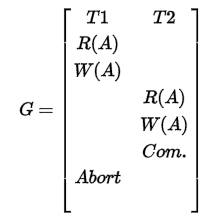
Transactions commit only after all transactions whose changes they read, commit.

{\displaystyle F={\begin{bmatrix}T1&T2\\R(A)&\\W(A)&\\&R(A)\\&W(A)\\Com.&\\&Com.\\&\end{bmatrix}}F2={\begin{bmatrix}T1&T2\\R(A)&\\W(A)&\\&R(A)\\&W(A)\\Abort&\\&Abort\\&\end{bmatrix}}}

These schedules are recoverable. F is recoverable because T1 commits before T2, that makes the value read by T2 correct. Then T2 can commit itself. In F2, if T1 aborted, T2 must abort because the value of A it read is incorrect. In both cases, the database is left in a consistent state.

**Unrecoverable**

If a transaction T1 aborts, and a transaction T2 commits, but T2 relied on T1, we have an unrecoverable schedule.

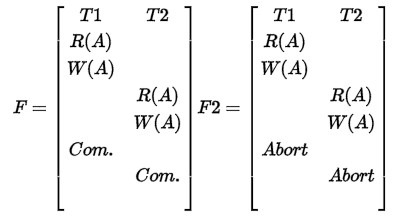
{\displaystyle G={\begin{bmatrix}T1&T2\\R(A)&\\W(A)&\\&R(A)\\&W(A)\\&Com.\\Abort&\\&\end{bmatrix}}}

In this example, G is unrecoverable, because T2 read the value of A written by T1, and committed. T1 later aborted, therefore the value read by T2 is wrong, but since T2 committed, this schedule is unrecoverable.

#### **{\displaystyle G={\begin{bmatrix}T1&T2\\R(A)&\\&R(A)\\W(B)&\\Com.&\\&W(A)\\&Com.\\&\end{bmatrix}}}Avoids cascading aborts / rollbacks (ACA)**

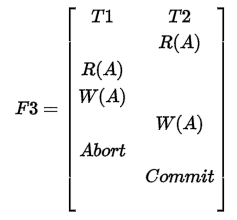
Also named cascadeless. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

The following examples are the same as the ones in the discussion on recoverable:



In this example, although F2 is recoverable, it does not avoid cascading aborts. It can be seen that if T1 aborts, T2 will have to be aborted too in order to maintain the correctness of the schedule as T2 has already read the uncommitted value written by T1.

The following is a recoverable schedule which avoids cascading abort. Note, however, that the update of A by T1 is always lost (since T1 is aborted).



Note that this Schedule would not be serializable if T1 would be committed. Cascading aborts avoidance is sufficient but not necessary for a schedule to be recoverable.

#### **Strict**

A schedule is strict - has the strictness property - if for any two transactions T1, T2, if a write operation of T1 precedes a *conflicting* operation of T2 (either read or write), then the commit or abort event of T1 also precedes that conflicting operation of T2.

Any strict schedule is cascadeless, but not the converse. Strictness allows efficient recovery of databases from failure.

<https://en.wikipedia.org/wiki/Schedule_(computer_science)#Recoverable>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Serializability

In [concurrency control](https://en.wikipedia.org/wiki/Concurrency_control) of [databases](https://en.wikipedia.org/wiki/Database),[[1]](https://en.wikipedia.org/wiki/Serializability#cite_note-Bernstein87-1)[[2]](https://en.wikipedia.org/wiki/Serializability#cite_note-Weikum01-2) [transaction processing](https://en.wikipedia.org/wiki/Transaction_processing) (transaction management), and various [transactional](https://en.wikipedia.org/wiki/Database_transaction) applications (e.g., [transactional memory](https://en.wikipedia.org/wiki/Transactional_memory)[[3]](https://en.wikipedia.org/wiki/Serializability#cite_note-Herlihy1993-3) and [software transactional memory](https://en.wikipedia.org/wiki/Software_transactional_memory)), both centralized and [distributed](https://en.wikipedia.org/wiki/Distributed_computing), a transaction [schedule](https://en.wikipedia.org/wiki/Schedule_(computer_science)) is **serializable** if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, i.e. without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of [isolation](https://en.wikipedia.org/wiki/Isolation_(computer_science)) between [transactions](https://en.wikipedia.org/wiki/Database_transaction), and plays an essential role in [concurrency control](https://en.wikipedia.org/wiki/Concurrency_control). As such it is supported in all general purpose database systems. [*Strong strict two-phase locking*](https://en.wikipedia.org/wiki/Two-phase_locking) (SS2PL) is a popular serializability mechanism utilized in most of the database systems (in various variants) since their early days in the 1970s.

**Serializability theory** provides the formal framework to reason about and analyze serializability and its techniques. Though it is [mathematical](https://en.wikipedia.org/wiki/Mathematics) in nature, its fundamentals are informally (without mathematics notation) introduced below.

Database transaction

A **database transaction** is a specific intended run (with specific parameters, e.g., with transaction identification, at least) of a computer program (or programs) that accesses a database (or databases). Such a program is written with the assumption that it is running in *isolation* from other executing programs, i.e., when running, its accessed data (after the access) are not changed by other running programs. Without this assumption the transaction's results are unpredictable and can be wrong. The same transaction can be executed in different situations, e.g., in different times and locations, in parallel with different programs. A *live* transaction (i.e., exists in a computing environment with already allocated computing resources; to distinguish from a *transaction request*, waiting to get execution resources) can be in one of three states, or phases:

1. *Running* - Its program(s) is (are) executing.
2. *Ready* - Its program's execution has ended, and it is waiting to be *Ended (Completed)*.
3. *Ended* (or *Completed*) - It is either *Committed* or *Aborted (Rolled-back)*, depending whether the execution is considered a success or not, respectively . When committed, all its *recoverable* (i.e., with states that can be controlled for this purpose), *durable* resources (typically *database data*) are put in their *final* states, states after running. When aborted, all its recoverable resources are put back in their *initial* states, as before running.

A failure in transaction's computing environment before ending typically results in its abort. However, a transaction may be aborted also for other reasons as well (e.g., see below).

Upon being ended (completed), transaction's allocated computing resources are released and the transaction disappears from the computing environment. However, the effects of a committed transaction remain in the database, while the effects of an aborted (rolled-back) transaction disappear from the database. The concept of *atomic transaction* ("all or nothing" semantics) was designed to exactly achieve this behavior, in order to control correctness in complex faulty systems

**Serializability** is used to keep the data in the data item in a consistent state. Serializability is a property of a transaction [schedule](https://en.wikipedia.org/wiki/Schedule_(computer_science)) (history). It relates to the [isolation](https://en.wikipedia.org/wiki/Isolation_(database_systems)) property of a [database transaction](https://en.wikipedia.org/wiki/Database_transaction).

**Serializability** of a schedule means equivalence (in the outcome, the database state, data values) to a *serial schedule* (i.e., sequential with no transaction overlap in time) with the same transactions. It is the major criterion for the correctness of concurrent transactions' schedule, and thus supported in all general purpose database systems.

**The rationale behind serializability** is the following:

If each transaction is correct by itself, i.e., meets certain integrity conditions, then a schedule that comprises any *serial* execution of these transactions is correct (its transactions still meet their conditions): "Serial" means that transactions do not overlap in time and cannot interfere with each other, i.e. complete *isolation* between each other exists. Any order of the transactions is legitimate, if no dependencies among them exist, which is assumed (see comment below). Thus, a schedule that comprises any execution (not necessarily serial) that is equivalent (in its outcome) to any serial execution of these transactions, is correct.

Schedules that are not serializable are likely to generate erroneous outcomes. Well known examples are with transactions that debit and credit accounts with money: If the related schedules are not serializable, then the total sum of money may not be preserved. Money could disappear, or be generated from nowhere. This and violations of possibly needed other [invariant](https://en.wikipedia.org/wiki/Invariant_(computer_science)) preservations are caused by one transaction writing, and "stepping on" and erasing what has been written by another transaction before it has become permanent in the database. It does not happen if serializability is maintained.

If any specific order between some transactions is requested by an application, then it is enforced independently of the underlying serializability mechanisms. These mechanisms are typically indifferent to any specific order, and generate some unpredictable [partial order](https://en.wikipedia.org/wiki/Partial_order) that is typically compatible with multiple serial orders of these transactions. This partial order results from the scheduling orders of concurrent transactions' data access operations, which depend on many factors.

A major characteristic of a database transaction is [atomicity](https://en.wikipedia.org/wiki/Atomicity_(database_systems)), which means that it either *commits*, i.e., all its operations' results take effect in the database, or *aborts* (rolled-back), all its operations' results do not have any effect on the database ("all or nothing" semantics of a transaction). In all real systems, transactions can abort for many reasons, and serializability by itself is not sufficient for correctness. Schedules also need to possess the [recoverability](https://en.wikipedia.org/wiki/Schedule_(computer_science)#Recoverable) (from abort) property. **Recoverability** means that committed transactions have not read data written by aborted transactions (whose effects do not exist in the resulting database states). While serializability is currently compromised on purpose in many applications for better performance (only in cases when application's correctness is not harmed), compromising recoverability would quickly violate the database's integrity, as well as that of transactions' results external to the database. A schedule with the recoverability property (a *recoverable* schedule) "recovers" from aborts by itself, i.e., aborts do not harm the integrity of its committed transactions and resulting database. This is false without recoverability, where the likely integrity violations (resulting incorrect database data) need special, typically manual, corrective actions in the database.

Implementing recoverability in its general form may result in *cascading aborts*: Aborting one transaction may result in a need to abort a second transaction, and then a third, and so on. This results in a waste of already partially executed transactions, and may result also in a performance penalty. [Avoiding cascading aborts](https://en.wikipedia.org/wiki/Schedule_(computer_science)#Avoids_cascading_aborts_.28rollbacks.29) (ACA, or Cascadelessness) is a special case of recoverability that exactly prevents such phenomenon. Often in practice a special case of ACA is utilized: [Strictness](https://en.wikipedia.org/wiki/Schedule_(computer_science)#Strict). Strictness allows an efficient database recovery from failure.

Note that the *recoverability* property is needed even if no database failure occurs and no database *recovery* from failure is needed. It is rather needed to correctly automatically handle aborts, which may be unrelated to database failure and recovery from failure.

Distributed serializability

**Distributed serializability** is the serializability of a schedule of a transactional [distributed system](https://en.wikipedia.org/wiki/Distributed_system) (e.g., a [distributed database](https://en.wikipedia.org/wiki/Distributed_database) system). Such system is characterized by [distributed transactions](https://en.wikipedia.org/wiki/Distributed_transaction) (also called *global transactions*), i.e., transactions that span computer processes (a process abstraction in a general sense, depending on computing environment; e.g., [operating system](https://en.wikipedia.org/wiki/Operating_system)'s [thread](https://en.wikipedia.org/wiki/Thread_(computer_science))) and possibly network nodes. A distributed transaction comprises more than one *local sub-transactions* that each has states as described above for a [database transaction](https://en.wikipedia.org/wiki/Serializability#Database_transaction). A local sub-transaction comprises a single process, or more processes that typically fail together (e.g., in a single [processor core](https://en.wikipedia.org/wiki/Processor_core)). Distributed transactions imply a need in [Atomic commit](https://en.wikipedia.org/wiki/Atomic_commit) protocol to reach consensus among its local sub-transactions on whether to commit or abort. Such protocols can vary from a simple (one-phase) hand-shake among processes that fail together, to more sophisticated protocols, like [Two-phase commit](https://en.wikipedia.org/wiki/Two-phase_commit), to handle more complicated cases of failure (e.g., process, node, communication, etc. failure). Distributed serializability is a major goal of [distributed concurrency control](https://en.wikipedia.org/wiki/Distributed_concurrency_control) for correctness. With the proliferation of the [Internet](https://en.wikipedia.org/wiki/Internet), [Cloud computing](https://en.wikipedia.org/wiki/Cloud_computing), [Grid computing](https://en.wikipedia.org/wiki/Grid_computing), and small, portable, powerful computing devices (e.g., [smartphones](https://en.wikipedia.org/wiki/Smartphone)) the need for effective distributed serializability techniques to ensure correctness in and among distributed applications seems to increase.

Distributed serializability is achieved by implementing distributed versions of the known centralized techniques.[[1]](https://en.wikipedia.org/wiki/Serializability#cite_note-Bernstein87-1)[[2]](https://en.wikipedia.org/wiki/Serializability#cite_note-Weikum01-2) Typically all such distributed versions require utilizing conflict information (either of materialized or non-materialized conflicts, or equivalently, transaction precedence or blocking information; conflict serializability is usually utilized) that is not generated locally, but rather in different processes, and remote locations. Thus, information distribution is needed (e.g., precedence relations, lock information, timestamps, or tickets). When the distributed system is of a relatively small scale, and message delays across the system are small, the centralized concurrency control methods can be used unchanged, while certain processes or nodes in the system manage the related algorithms. However, in a large-scale system (e.g., *Grid* and *Cloud*), due to the distribution of such information, substantial performance penalty is typically incurred, even when distributed versions of the methods (Vs. centralized) are used, primarily due to computer and communication [latency](https://en.wikipedia.org/wiki/Latency_(engineering)). Also, when such information is distributed, related techniques typically do not scale well. A well-known example with scalability problems is a [distributed lock manager](https://en.wikipedia.org/wiki/Distributed_lock_manager), which distributes lock (non-materialized conflict) information across the distributed system to implement locking techniques

<https://en.wikipedia.org/wiki/Serializability>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Linearizability

In [concurrent programming](https://en.wikipedia.org/wiki/Concurrent_programming), an operation (or set of operations is **atomic**, **linearizable**, **indivisible** or **uninterruptible** if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of [isolation](https://en.wikipedia.org/wiki/Isolation_(computer_science)) from [interrupts](https://en.wikipedia.org/wiki/Interrupt), [signals](https://en.wikipedia.org/wiki/Signal_(IPC)), [concurrent](https://en.wikipedia.org/wiki/Concurrency_(computer_science)) [processes](https://en.wikipedia.org/wiki/Process_(computing)) and [threads](https://en.wikipedia.org/wiki/Thread_(computing)). It is relevant for [thread safety](https://en.wikipedia.org/wiki/Thread_safety) and [reentrancy](https://en.wikipedia.org/wiki/Reentrancy_(computing)). Additionally, atomic operations commonly have a [succeed-or-fail](https://en.wikipedia.org/wiki/Atomicity_(database_systems)) definition—they either successfully change the state of the system, or have no apparent effect.

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.

Atomicity is often enforced by [mutual exclusion](https://en.wikipedia.org/wiki/Mutual_exclusion), whether at the hardware level building on a [cache coherency](https://en.wikipedia.org/wiki/Cache_coherency) protocol, or the software level using [semaphores](https://en.wikipedia.org/wiki/Semaphore_(programming)) or [locks](https://en.wikipedia.org/wiki/Lock_(computer_science)). Thus, an atomic operation does not necessarily *actually* occur instantaneously. The benefit comes from the *appearance*: the system behaves *as if* each operation occurred instantly, separated by pauses. This makes the system consistent. Because of this, implementation details may be ignored by the user, except insofar as they affect performance. If an operation is not atomic, the user will also need to understand and cope with sporadic extraneous behaviour caused by interactions between concurrent operations, which by their nature are likely to be hard to reproduce and debug.

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](https://en.wikipedia.org/wiki/Thread_(computer_science)) 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 op1 completes (gets a response) before op2 begins (invokes), then op1 precedes op2 in σ.[[2]](https://en.wikipedia.org/wiki/Linearizability#cite_note-:0-2)

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](https://en.wikipedia.org/wiki/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.)[[](https://en.wikipedia.org/wiki/Linearizability#cite_note-:0-2)

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 |

While weird, 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".[[2]](https://en.wikipedia.org/wiki/Linearizability#cite_note-:0-2)

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

High-level atomic operations

The easiest way to achieve linearizability is running groups of primitive operations in a [critical section](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Lock_(computer_science)) 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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Spring_Framework), annotating a method with @Transactional will ensure all enclosed database interactions occur in a single [database transaction](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/ACID) databases refer to this principle as [atomicity](https://en.wikipedia.org/wiki/Atomicity_(database_systems)).) If the operation fails (usually due to concurrent operations), the user must retry, usually performing a different operation. For example:

* [Compare-and-swap](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Database_transaction), if the transaction cannot be completed due to a concurrent operation (e.g. in a [deadlock](https://en.wikipedia.org/wiki/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 register](https://en.wikipedia.org/wiki/Shared_register)s

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:

1. the second process reads the value in the register, which is still equal to 0;
2. the second process adds one to the value;
3. 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 because of the system not being linearizable.

The above example shows the need for carefully thinking through implementations of data structures and how Linearizability can influence the correctness of the system.

#### **Atomic**

To implement a Linearizable or Atomic Counter object we will modify our previous implementation so each Process Pi will use its own Register Ri

Each process Increments and Reads per the following algorithm:

**Increment:**

1. read value in Register Ri
2. add one to the value
3. write new value back into Ri

**Read:**

1. read Registers R1, R2, ... Rn
2. return 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 Ri. 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 Ri.

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](https://en.wikipedia.org/wiki/64-bit) value from memory may actually be implemented as two [sequential](https://en.wikipedia.org/wiki/Sequence) reads of two [32-bit](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/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.

### Locking

Another approach is to turn the naive algorithm into a [critical section](https://en.wikipedia.org/wiki/Critical_section), preventing other threads from disrupting it, using a [lock](https://en.wikipedia.org/wiki/Lock_(computer_science)). 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](https://en.wikipedia.org/wiki/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.

<https://en.wikipedia.org/wiki/Linearizability#Linearizability_versus_serializability>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## [Linearizability versus Serializability](http://www.bailis.org/blog/linearizability-versus-serializability/)

Linearizability and serializability are both important properties about interleavings of operations in databases and distributed systems, and it’s easy to get them confused. This post gives a short, simple, and hopefully practical overview of the differences between the two.

#### Linearizability: single-operation, single-object, real-time order

[*Linearizability*](http://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf) is a guarantee about single operations on single objects. It provides a real-time (i.e., wall-clock) guarantee on the behavior of a set of single operations (often reads and writes) on a single object (e.g., distributed register or data item).

In plain English, under linearizability, writes should appear to be instantaneous. Imprecisely, once a write completes, all later reads (where “later” is defined by wall-clock start time) should return the value of that write or the value of a later write. Once a read returns a value, all later reads should return that value or the value of a later write.

Linearizability for read and write operations is synonymous with the term “atomic consistency” and is the “C,” or “consistency,” in Gilbert and Lynch’s [proof of the CAP Theorem](http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf). We say linearizability is composable (or “local”) because, if operations on each object in a system are linearizable, then all operations in the system are linearizable.

#### Serializability: multi-operation, multi-object, arbitrary total order

Serializability is a guarantee about transactions, or groups of one or more operations over one or more objects. It guarantees that the execution of a set of transactions (usually containing read and write operations) over multiple items is equivalent to some serial execution (total ordering) of the transactions.

Serializability is the traditional “I,” or isolation, in [ACID](http://sites.fas.harvard.edu/~cs265/papers/haerder-1983.pdf). If users’ transactions each preserve application correctness (“C,” or consistency, in ACID), a serializable execution also preserves correctness. Therefore, serializability is a mechanism for guaranteeing database correctness.[1](http://www.bailis.org/blog/linearizability-versus-serializability/#fn:mechanism)

Unlike linearizability, serializability does not—by itself—impose any real-time constraints on the ordering of transactions. Serializability is also not composable. Serializability does not imply any kind of deterministic order—it simply requires that some equivalent serial execution exists.

#### Strict Serializability: Why don’t we have both?

Combining serializability and linearizability yields strict serializability: transaction behavior is equivalent to some serial execution, and the serial order corresponds to real time. For example, say I begin and commit transaction T1, which writes to item x, and you later begin and commit transaction T2, which reads from x. A database providing strict serializability for these transactions will place T1 before T2 in the serial ordering, and T2 will read T1’s write. A database providing serializability (but not strict serializability) could order T2 before T1.[2](http://www.bailis.org/blog/linearizability-versus-serializability/#fn:implementation)

As [Herlihy and Wing](http://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf) note, “linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object.”

#### Coordination costs and real-world deployments

Neither linearizability nor serializability is achievable without coordination. That is we can’t provide either guarantee with availability (i.e., CAP “AP”) under an asynchronous network.[3](http://www.bailis.org/blog/linearizability-versus-serializability/#fn:hardness)

In practice, your database is [unlikely to provide serializability](http://www.bailis.org/blog/when-is-acid-acid-rarely/), and your multi-core processor is [unlikely to provide linearizability](http://preshing.com/20120930/weak-vs-strong-memory-models/)—at least by default. As the above theory hints, achieving these properties requires a lot of expensive coordination. So, instead, real systems often use cheaper-to-implement and often [harder-to-understand](http://www.bailis.org/blog/understanding-weak-isolation-is-a-serious-problem/) models. This trade-off between efficiency and programmability represents a fascinating and challenging design space

#### A note on terminology, and more reading

One of the reasons these definitions are so confusing is that linearizability hails from the distributed systems and concurrent programming communities, and serializability comes from the database community. Today, almost everyone uses both distributed systems and databases, which often leads to overloaded terminology (e.g., “consistency,” “atomicity”).

There are many more precise treatments of these concepts. I like [this book](http://link.springer.com/book/10.1007%2F978-3-642-15260-3), but there is plenty of free, concise, and (often) accurate material on the internet, such as [these notes](https://www.cs.rochester.edu/~scott/458/notes/04-concurrent_data_structures).

<http://www.bailis.org/blog/linearizability-versus-serializability/>