Non-blocking algorithms use low-level atomic machine instructions such as compare-and-swap instead of locks to ensure data integrity under concurrent access.

They are more complicated to implement than lock-based alternatives, but have many advantages:

- coordinate at a finer level of granularity and can greatly reduce scheduling overhead because they don't block multiple threads
- immune to deadlock and other liveness problems —- impervious to individual thread failures

Disadvantages of locking

- Modern JVMs can optimize uncontended lock acquisition and release fairly effectively, but if multiple threads request the lock, the JVM may
 use profiling data to decide to use spin waiting
 enlist the help of the operating system: suspend some thread and resume it later, after that the thread may have to wait until other threads finish their scheduling quanta
 * a lot of overhead: context switches, scheduling delays
 * also, for fine-grained operations the ratio of overhead compared to useful work may be quite high
- Lighter-weight synchronization, volatile, they cannot be used to construct atomic compound actions
 can't be used when one variable depends on another
 can't be used when new value depends on the old one
- When a thread holding a lock is delayed (page fault, scheduling delay, etc.), then no thread that needs that lock can make progress
 this makes higher-priority threads effecrively become lower-priority threads —- priority inversion
 if the thread is locked permanently, any threads waiting for that lock can never make progress

Hardware support of concurrency

- Exclusive locking is a pessimistic technique —- it assumes the worst and doesn't proceed until you can guarantee (by acquiring the locks) that other threads will not interfere
- Optimistic approach is more efficient for fine-grained operations, you proceed with an update hopreful that you can proceed withoud interference. In case of interference the operation fails and may be retried

Most processors have **read-modify-write** instruction: **compare-and-swap**, **load-linked/store-conditional**.

Most CPUs implement **CAS**:

- 3 operands, memory location V, the expected value A, the new value B

- atomically updates V but only if its current value is A, otherwise does nothing
- returns current value in V

When using **CAS**

- read A from V
- derive the new value B from A
- use CAS to atomically update B
- one threads wins, successfully updates the value
- other threads fail, may decide to retry or do nothing

Performance comparison: **CAS** significantly outperforms locking:

- the fast path for uncontended lock acquisition typically requires at least one CAS plus other lock-related housekeeping => more work in the case of locking
  * lock: at least one **CAS**, traversing a relatively complicated code path in the JVM, may entail OS-level locking, thread suspension, context switches
  * **CAS**: no JVM code, system calls or scheduling activity
- **CAS** succeeds most of the time, the CPU will correctly predict the branch implicit in the retry loop.

Costs:

- single core CPU: a few instructions
- multiple CPU, uncontended: ~150 cycles
- price of uncontended lock acquisition and release is about 2 times the price of **CAS**

Disadvantage of **CAS**:

- makes client code to deal with contention (retrying, backing off, giving up)
- the difficulty of constructing the surrounding algorithms correctly

Low-level support exists to support **CAS** operations, JVM compiles into the most efficient code for the hardware:

- if platform supports **CAS**, the JVM inlines into the appropriate machine instructions
- if **CAS**-like instructions are not available, the JVM uses a spin lock

Atomic variable classes

Difference
THe atomic variable classes provide a generalization of volatile variables to support atomic conditional read-modify-write operations.

- AtomicInteger* has **compareAndSet** method, atomic add, increment and decrement methods
- compareAndSet if successful has a semantics of both reading and writing a volatile variable

Four groups of atomic variables classes

- scalars
  AtomicInteger, **AtomicLong**, **AtomicBoolean**, **AtomicReference**
  use **floatToIntBits** or **doubleToLongBits** for float and double types
  do not redefine **equals** or **hash** methods, not good candidates for keys in hash-based collections
- field updaters
  AtomicLongFieldUpdater, **AtomicLongFieldUpdater**, **AtomicReferenceFieldUpdater**
- arrays —- elements can be update atomically
  AtomicDoubleArray, **AtomicIntegerArray**, **AtomicLongArray**
- compound variables
  AtomicStampedReference, **AtomicMarkableReference**

Performance

- At high contention levels locking tends to slightly outperform atomic variables
  Lock reacts to contention by suspending threads, reducing CPU usage and synchronization traffic on the shared memory bus
- At more realistic contention levels atomic vairables outperform locks
  contention management is pushed back to the calling class, CAS-based algorithms react to contention by retrying immediately
- CAS-based algorithms always outperform lock-based algorithms on single-CPU systems
  CPU always succeeds on a single-CPU system except in the unlikely case that a thread is preempted in the middle of read-modify-write operation

Non-blocking algorithms

An algorithm is **non-blocking** if failure or suspension of any thread cannot cause failure or suspension of another thread.
An algorithm us **lock-free** if at each step some thread can make progress.

Algorithms tha use CAS excusively for coordination between threads can, if constructed properly, be both **lock-free** and **non-blocking**

- Making progress: unconteded CAS always succeeds; if multiple threads content, one always wins
- Immune to deadlock or priority inversion, though can exhibit starvation or livelock

CAS-based algorithms derive their thread-safety from the fact that, like locking, **compareAndSet** provides both atomicity and visibility guarantees.

- changing the state with **compareAndSet** which has the memory semantics of a volatile write
- when a thread examines the stack, it does so by calling **get** which has the memory effects of a volatile read

Key to creating non-blocking algorithms is figuring out how to limit the scope of atomic changes to a single variable while maintaing data consistency.

- In linked collections you can sometimes get away with expressing state transformations as changes to individual links and using an **AtomicReference** to represent each link that must be updated atomically
   Stack: construct a new node that refers to the top; V=reference to the top node; try to atomically update the link to the top, if it’s still V
- If two variales need to be updated
   LinkedList: the tail and the next reference in the tail node; update the tail node’s link:
  * successful —- complete the operation, may fail because another thread helped to complete
  * failed —- help to complete the operation (may fail —- the thread finished itself or another thread helped) and retry (Michal-Scott algorithm)
   AtomicUpdaters
  * Single updater instance for all nodes
  * Inner state of node must be protected —- **final** fields for immutable references, **volatile** fields for mutating references
  * Performance issue —- for frequently allocated, short-lived objects eliminating the creation of **AtomicReference** for each node may be significant enough to reduce the cost of insertion operations

The ABA problem

The **ABA problem**: is it sufficient to check that the value is A, or that it stayed A, not became B and then A again?

Solution: instead of updating the value of a reference, update a pair of values, a reference and a version number

- use **AtomicStampedReference** —- reference with a version
- use **AtomicMarkedReference** —- reference with a flag (e.g., “removed”)