Reference counting

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(November 2009)*

In computer science, **reference counting** is a technique of storing the number of references, pointers, or handles to a resource such as an object, block of memory, disk space or other resource. It may also refer, more specifically, to a garbage collection algorithm that uses these reference counts to deallocate objects which are no longer referenced.

Contents [hide]

1 Use in garbage collection
2 Advantages and disadvantages
3 Graph interpretation
4 Dealing with inefficiency of updates
5 Dealing with reference cycles
6 Variants of reference counting
  6.1 Weighted reference counting
  6.2 Indirect reference counting
7 Examples of use
  7.1 COM
  7.2 C++
  7.3 Cocoa
  7.4 Delphi
  7.5 GObject
  7.6 Perl
  7.7 PHP
  7.8 Python
  7.9 Squirrel
  7.10 Tcl
  7.11 File systems
8 References
9 External links

Use in garbage collection [edit]

As a collection algorithm, reference counting tracks, for each object, a count of the number of references to it held by other objects. If an object's reference count reaches zero, the object has become inaccessible, and can be destroyed.

When an object is destroyed, any objects referenced by that object also have their reference counts decreased. Because of this, removing a single reference can potentially lead to a large number of objects being freed. A common modification allows reference counting to be made incremental: instead of destroying an object as soon as its reference count becomes zero, it is added to a list of unreferenced objects, and periodically (or as needed) one or more items from this list are destroyed.

Simple reference counts require frequent updates. Whenever a reference is destroyed or overwritten, the reference count of the object it references is decremented, and whenever one is created or copied, the reference count of the object it references is incremented.

Reference counting is also used in disk operating systems and distributed systems, where full non-incremental tracing garbage collection is too time consuming because of the size of the object graph and slow access speed.

Advantages and disadvantages [edit]

The main advantage of the reference counting over tracing garbage collection is that objects are reclaimed *as soon as* they can no longer be referenced, and in an incremental fashion, without long pauses for collection cycles and with clearly defined lifetime of every object. In real-time applications or systems with limited memory, this is important to maintain responsiveness. Reference counting is also among the simplest forms of memory management to implement. It also allows for effective management of non-memory resources such as operating system objects, which are often much scarcer than memory (tracing GC systems use finalizers for this[*citation needed*], but the delayed reclamation may cause problems). Weighted reference counts are a good solution for garbage collecting a distributed system.

Tracing garbage collection cycles are triggered too often if the set of live objects fills most of the available memory[*citation needed*]; it requires extra space to be efficient[*citation needed*]. Reference counting performance does not deteriorate as the total amount of free space decreases.[1]

Reference counts are also useful information to use as input to other runtime optimizations. For example, systems that depend heavily on immutable objects such as many functional programming languages can suffer an efficiency penalty due to frequent copies.[*citation needed*] However, if the compiler (or runtime system) knows that a particular object has only one reference (as most do in many systems), and that the reference is lost at the same time that a similar new object is created (as in the string append statement `str ← str + "a"`), it can replace the operation with a mutation on the original object.

Reference counting in naive form has two main disadvantages over the tracing garbage collection, both of which require additional mechanisms to ameliorate:

- The frequent updates it involves are a source of inefficiency. While tracing garbage collectors can impact efficiency severely via context switching and cache line faults, they collect relatively infrequently, while accessing objects is done continually. Also, less importantly, reference counting requires every memory-managed object to reserve space for a reference count. In tracing garbage collectors, this information is stored implicitly in the references that refer to that object, saving space, although tracing garbage collectors, particularly incremental ones, can require additional space for other purposes.
- The naive algorithm described above can't handle *reference cycles,* an object which refers directly or indirectly to itself. A mechanism relying purely on reference counts will never consider cyclic chains of objects for deletion, since their reference count is guaranteed to stay nonzero. Methods for dealing with this issue exist but can also increase the overhead and complexity of reference counting — on the other hand, these methods need only be applied to data that might form cycles, often a small subset of all data. One such method is the use of weak references, while another involves using a mark-sweep algorithm that gets called infrequently to clean up.

In addition to these, if the memory is allocated from a free list, reference counting suffers from poor locality. Reference counting alone cannot move objects to improve cache performance, so high performance collectors implement a tracing garbage collector as well. Most implementations (such as the ones in PHP and Objective-C) suffer from poor cache performance since they do not implement copying objects.[2]

Graph interpretation [edit]

When dealing with garbage collection schemes, it is often helpful to think of the **reference graph**, which is a directed graph where the vertices are objects and there is an edge from an object A to an object B if A holds a reference to B. We also have a special vertex or vertices representing the local variables and references held by the runtime system, and no edges ever go to these nodes, although edges can go from them to other nodes.

In this context, the simple reference count of an object is the in-degree of its vertex. Deleting a vertex is like collecting an object. It can only be done when the vertex has no incoming edges, so it does not affect the out-degree of any other vertices, but it can affect the in-degree of other vertices, causing their corresponding objects to be collected as well if their in-degree also becomes 0 as a result.

The connected component containing the special vertex contains the objects that can't be collected, while other connected components of the graph only contain garbage. If a reference-counting garbage collection algorithm is implemented, then each of these garbage components must contain at least one cycle; otherwise, they would have been collected as soon as their reference count (i.e., the number of incoming edges) dropped to zero.

Dealing with inefficiency of updates [edit]

Incrementing and decrementing reference counts every time a reference is created or destroyed can significantly impede performance. Not only do the operations take time, but they damage cache performance and can lead to pipeline bubbles. Even read-only operations like calculating the length of a list require a large number of reads and writes for reference updates with naive reference counting.

One simple technique is for the compiler to combine a number of nearby reference updates into one. This is especially effective for references which are created and quickly destroyed. Care must be taken, however, to put the combined update at the right position so that a premature free be avoided.

The Deutsch-Bobrow method of reference counting capitalizes on the fact that most reference count updates are in fact generated by references stored in local variables. It ignores these references, only counting references in data structures, but before an object with reference count zero can be deleted, the system must verify with a scan of the stack and registers that no other reference to it still exists.

Another technique devised by Henry Baker involves **deferred increments**,[3] in which references which are stored in local variables do not immediately increment the corresponding reference count, but instead defer this until it is necessary. If such a reference is destroyed quickly, then there is no need to update the counter. This eliminates a large number of updates associated with short-lived references (such as the above list-length-counting example). However, if such a reference is copied into a data structure, then the deferred increment must be performed at that time. It is also critical to perform the deferred increment before the object's count drops to zero, resulting in a premature free.

A dramatic decrease in the overhead on counter updates was obtained by Levanoni and Petrank.[4][5] They introduce the **update coalescing method** which coalesces many of the redundant reference count updates. Consider a pointer that in a given interval of the execution is updated several times. It first points to an object `O1`, then to an object `O2`, and so forth until at the end of the interval it points to some object `On`. A reference counting algorithm would typically execute `rc(O1)--`, `rc(O2)++`, `rc(O2)--`, `rc(O3)++`, `rc(O3)--`, ..., `rc(On)++`. But most of these updates are redundant. In order to have the reference count properly evaluated at the end of the interval it is enough to perform `rc(O1)--` and `rc(On)++`. The rest of the updates are redundant.

Levanoni and Petrank showed in 2001 how to use such update coalescing in a reference counting collector. When using update coalescing with an appropriate treatment of new objects, more than 99% of the counter updates are eliminated for typical Java benchmarks. In addition, the need for atomic operations during pointer updates on parallel processors is eliminated. Finally, they presented an enhanced algorithm that may run concurrently with multithreaded applications employing only fine synchronization.[6]

Blackburn and McKinley's *ulterior reference counting* method in 2003[7] combines deferred reference counting with a copying nursery, observing that the majority of pointer mutations occur in young objects. This algorithm achieves throughput comparable with the fastest generational copying collectors with the low bounded pause times of reference counting.

More work on improving performance of reference counting collectors[*clarification needed*] can be found in Paz's Ph.D thesis in 2006.[8] In particular, he advocates the use of age oriented collectors[9] and prefetching.[10]

Dealing with reference cycles [edit]

Perhaps the most obvious way to handle reference cycles is to design the system to avoid creating them. A system may explicitly forbid reference cycles; file systems with hard links often do this. Judicious use of "weak" (non-counted) references may also help avoid retain cycles; the Cocoa framework, for instance, recommends using "strong" references for parent-to-child relationships and "weak" references for child-to-parent relationships.[11]

Systems may also be designed to tolerate or correct the cycles they create in some way. Developers may design code to explicitly "tear down" the references in a data structure when it is no longer needed, though this has the cost of requiring them to manually track that data structure's lifetime. This technique can be automated by creating an "owner" object that does the tearing-down when it is destroyed; for instance, a Graph object's destructor could delete the edges of its GraphNodes, breaking the reference cycles in the graph. Cycles may even be ignored in systems with short lives and a small amount of cyclic garbage, particularly when the system was developed using a methodology of avoiding cyclic data structures wherever possible, typically at the expense of efficiency.

Computer scientists have also discovered ways to detect and collect reference cycles automatically, without requiring changes in the data structure design. One simple solution is to periodically use a tracing garbage collector to reclaim cycles; since cycles typically constitute a relatively small amount of reclaimed space, the collector can be run much less often than with an ordinary tracing garbage collector.

Bacon describes a cycle-collection algorithm for reference counting systems with some similarities to tracing systems, including the same theoretical time bounds, but that takes advantage of reference count information to run much more quickly and with less cache damage. It is based on the observation that an object cannot appear in a cycle until its reference count is decremented to a nonzero value. All objects which this occurs to

are put on a *roots* list, and then periodically the program searches through the objects reachable from the roots for cycles. It knows it has found a cycle that can be collected when decrementing all the reference counts on a cycle of references brings them all down to zero. An enhanced version of this algorithm by Paz et al.[12] is able to run concurrently with other operations and improve its efficiency by using the update coalescing method of Levanoni and Petrank.[13]

Variants of reference counting [edit]

Although it is possible to augment simple reference counts in a variety of ways, often a better solution can be found by performing reference counting in a fundamentally different way. Here we describe some of the variants on reference counting and their benefits and drawbacks.

Weighted reference counting [edit]

In weighted reference counting, we assign each reference a *weight*, and each object tracks not the number of references referring to it, but the total weight of the references referring to it. The initial reference to a newly created object has a large weight, such as 2^{16}. Whenever this reference is copied, half of the weight goes to the new reference, and half of the weight stays with the old reference. Because the total weight does not change, the object's reference count does not need to be updated.

Destroying a reference decrements the total weight by the weight of that reference. When the total weight becomes zero, all references have been destroyed. If an attempt is made to copy a reference with a weight of 1, we have to "get more weight" by adding to the total weight and then adding this new weight to our reference, and then splitting it. An alternative in this situation is to create an *indirection* reference object, the initial reference to which is created with a large weight which can then be split.

The property of not needing to access a reference count when a reference is copied is particularly helpful when the object's reference count is expensive to access, for example because it is in another process, on disk, or even across a network. It can also help increase concurrency by avoiding many threads locking a reference count to increase it. Thus, weighted reference counting is most useful in parallel, multiprocess, database, or distributed applications.

The primary problem with simple weighted reference counting is that destroying a reference still requires accessing the reference count, and if many references are destroyed this can cause the same bottlenecks we seek to avoid. Some adaptations of weighted reference counting seek to avoid this by attempting to give weight back from a dying reference to one which is still active.

Weighted reference counting was independently devised by Bevan, in the paper *Distributed garbage collection using reference counting*,[14] and Watson & Watson, in the paper *An efficient garbage collection scheme for parallel computer architectures*,[15] both in 1987.

Indirect reference counting [edit]

In indirect reference counting, it is necessary to keep track of whom the reference was obtained from. This means that two references are kept to the object: a direct one which is used for invocations; and an indirect one which forms part of a diffusion tree, such as in the Dijkstra-Scholten algorithm, which allows a garbage collector to identify dead objects. This approach prevents an object from being discarded prematurely.

Examples of use [edit]

COM [edit]

Microsoft's Component Object Model (COM) makes pervasive use of reference counting. In fact, two of the three methods that all COM objects must provide (in the IUnknown interface) increment or decrement the reference count. Much of the Windows Shell and many Windows applications (including MS Internet Explorer, MS Office, and countless third-party products) are built on COM, demonstrating the viability of reference counting in large-scale systems.

One primary motivation for reference counting in COM is to enable interoperability across different programming languages and runtime systems. A client need only know how to invoke object methods in order to manage object life cycle; thus, the client is completely abstracted from whatever memory allocator the implementation of the COM object uses. As a typical example, a Visual Basic program using a COM object is agnostic towards whether that object was allocated (and must later be deallocated) by a C++ allocator or another Visual Basic component.

However, this support for heterogeneity has a major cost: it requires correct reference count management by all

parties involved. While high-level languages like Visual Basic manage reference counts automatically, C/C++ programmers are entrusted to increment and decrement reference counts at the appropriate time. C++ programs can and should avoid the task of managing reference counts manually by using smart pointers. Bugs caused by incorrect reference counting in COM systems are notoriously hard to resolve, especially because the error may occur in an opaque, third-party component.

Microsoft abandoned reference counting in favor of tracing garbage collection for the .NET Framework. However, it has been reintroduced in the COM-based WinRT and the new C++/CX (Component Extensions) language.

C++ [edit]

C++11 provides reference counted smart pointers, via the `std::shared_ptr` class. Programmers can use weak pointers (via `std::weak_ptr`) to break cycles. C++ does not *require* all objects to be reference counted; in fact, programmers can choose to apply reference counting to only those objects that are truly shared; objects not intended to be shared can be referenced using a `std::unique_ptr`, and objects that are shared but not owned can be accessed via an iterator.

In addition, C++11's move semantics further reduce the extent to which reference counts need to be modified.

Cocoa [edit]

Apple's Cocoa and Cocoa Touch frameworks (and related frameworks, such as Core Foundation) use manual reference counting, much like COM. Traditionally this was accomplished by the programmer manually sending `retain` and `release` messages to objects, but Automatic Reference Counting, a Clang compiler feature that automatically inserts these messages as needed, was added in iOS 5[16] and Mac OS X 10.7.[17] Mac OS X 10.5 introduced a tracing garbage collector as an alternative to reference counting, but it was deprecated in OS X 10.8 and is expected to be removed in a future version. iOS has never supported a tracing garbage collector.

Delphi [edit]

One language that uses reference counting for garbage collection is Delphi. Delphi is not a completely garbage collected language, in that user-defined types must still be manually allocated and deallocated. It does provide automatic collection, however, for a few built-in types, such as strings, dynamic arrays, and interfaces, for ease of use and to simplify the generic database functionality. It is up to the programmer to decide whether to use the built-in types or not; Delphi programmers have complete access to low-level memory management like in C/C++. So all potential cost of Delphi's reference counting can, if desired, be easily circumvented.

Some of the reasons reference counting may have been preferred to other forms of garbage collection in Delphi include:

- The general benefits of reference counting, such as prompt collection.
- Cycles either cannot occur or do not occur in practice because all of the small set of garbage-collected built-in types are not arbitrarily nestable.
- The overhead in code size required for reference counting is very small (on native x86, typically a single LOCK INC, LOCK DEC or LOCK XADD instruction, which ensures atomicity in any environment), and no separate thread of control is needed for collection as would be needed for a tracing garbage collector.
- Many instances of the most commonly used garbage-collected type, the string, have a short lifetime, since they are typically intermediate values in string manipulation.
- The reference count of a string is checked before mutating a string. This allows reference count 1 strings to be mutated directly whilst higher reference count strings are copied before mutation. This allows the general behaviour of old style pascal strings to be preserved whilst eliminating the cost of copying the string on every assignment.
- Because garbage-collection is only done on built-in types, reference counting can be efficiently integrated into the library routines used to manipulate each datatype, keeping the overhead needed for updating of reference counts low. Moreover a lot of the runtime library is in hand-optimized assembler.

GObject [edit]

The GObject object-oriented programming framework implements reference counting on its base types, including weak references. Reference incrementing and decrementing uses atomic operations for thread safety. A significant amount of the work in writing bindings to GObject from high-level languages lies in adapting GObject reference counting to work with the language's own memory management system.

The Vala programming language uses GObject reference counting as its primary garbage collection system,

along with copy-heavy string handling.[18]

Perl [edit]

Perl also uses reference counting, without any special handling of circular references, although (as in Cocoa and C++ above), Perl does support weak references, which allows programmers to avoid creating a cycle.

PHP [edit]

PHP uses a reference counting mechanism for its internal variable management. Since PHP 5.3, it implements the algorithm from Bacon's above mentioned paper. PHP allows you to turn on and off the cycle collection with user-level functions. It also allows you to manually force the purging mechanism to be run.

Python [edit]

Python also uses reference counting and offers cycle detection as well.[19]

Squirrel [edit]

Squirrel also uses reference counting and offers cycle detection as well. This tiny language is relatively unknown outside the video game industry; however, it is a concrete example of how reference counting can be practical and efficient (especially in realtime environments).[*citation needed*]

Tcl [edit]

Tcl 8 uses reference counting for memory management of values (Tcl Obj structs[*disambiguation needed*]). Since Tcl's values are immutable, reference cycles are impossible to form and no cycle detection scheme is needed. Operations that would replace a value with a modified copy are generally optimized to instead modify the original when its reference count indicates it to be unshared. The references are counted at a data structure level, so the problems with very frequent updates discussed above do not arise.

File systems [edit]

Many file systems maintain a count of the number of references to any particular block or file, for example the inode *link count* on Unix-style file systems. When the count falls to zero, the file can be safely deallocated. In addition, while references can still be made from directories, some Unixes allow that the referencing can be solely made by live processes, and there can be files that do not exist in the file system hierarchy.

References [edit]

1. ^ Wilson, Paul R. "Uniprocessor Garbage Collection Techniques". *Proceedings of the International Workshop on Memory Management*. London, UK: Springer-Verlag. pp. 1–42. ISBN 3-540-55940-X. Retrieved 5 December 2009. Section 2.1.
2. ^ Rifat Shahriyar, Stephen M. Blackburn, Xi Yang and Kathryn S. McKinley (2013). "Taking Off the Gloves with Reference Counting Immix". *24th ACM SIGPLAN conference on Object Oriented Programming Systems, Languages and Applications*. OOPSLA 2013. doi:10.1145/2509136.2509527.
3. ^ Henry Baker (September 1994). "Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures". *ACM SIGPLAN Notices* **29** (9): 38–43. doi:10.1145/185009.185016.
4. ^ Yossi Levanoni, Erez Petrank (2001). "An on-the-fly reference-counting garbage collector for java". *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA 2001. pp. 367–380. doi:10.1145/504282.504309.
5. ^ Yossi Levanoni, Erez Petrank (2006). "An on-the-fly reference-counting garbage collector for java". *ACM Trans. Program. Lang. Syst.* **28**: 31–69. doi:10.1145/1111596.1111597.
6. ^ http://www.cs.technion.ac.il/%7Eerez/Papers/refcount.pdf
7. ^ Stephen Blackburn, Kathryn McKinley (2003). "Ulterior Reference Counting: Fast Garbage Collection without a Long Wait". *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*. OOPSLA 2003. pp. 344–358. doi:10.1145/949305.949336. ISBN 1-58113-712-5.
8. ^ http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2006/PHD/PHD-2006-10.pdf
9. ^ http://www.cs.technion.ac.il/~erez/Papers/ao-cc.pdf
10. ^ http://www.cs.technion.ac.il/~erez/Papers/rc-prefetch-cc07.pdf
11. ^ http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmObjectOwnership.html#//apple_ref/doc/uid/20000043-1044135-BCICCFAE
12. ^ Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, V. T. Rajan (2007). "An efficient on-the-fly cycle collection". *ACM Transactions on Programming Languages and Systems (TOPLAS)*. *TOPLAS*. doi:10.1145/1255450.1255453

doi:10.1145/1233450.1233453.
13. ^ http://www.research.ibm.com/people/d/dfb/papers/Bacon01Concurrent.pdf
14. ^ Bevan, D. I. (1987). "Distributed garbage collection using reference counting". *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*. Eindhoven, The Netherlands: Springer-Verlag. pp. 176–187. ISBN 0-387-17945-3.
15. ^ Watson, Paul; Watson, Ian (1987). "An efficient garbage collection scheme for parallel computer architectures". *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*. Eindhoven, The Netherlands: Springer-Verlag. pp. 432–443. ISBN 0-387-17945-3.
16. ^ http://developer.apple.com/technologies/ios5/
17. ^ http://developer.apple.com/library/mac/#releasenotes/MacOSX/WhatsNewInOSX/Articles/MacOSX10_7.html#//apple_ref/doc/uid/TP40010355-SW62
18. ^ https://live.gnome.org/Vala/ReferenceHandling
19. ^ https://docs.python.org/extending/extending.html#reference-counts

External links [edit]

- The Memory Manager Reference: Beginner's Guide: Recycling: Reference Counts

- *Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures*, Henry G. Baker
- *Concurrent Cycle Collection in Reference Counted Systems*, David F. Bacon
- *An On-the-Fly Reference-Counting Garbage Collector for Java*, Yossi Levanoni and Erez Petrank
- *Atomic Reference Counting Pointers: A lock-free, async-free, thread-safe, multiprocessor-safe reference counting pointer*, Kirk Reinholtz
- *Extending and Embedding the Python Interpreter: Extending Python with C or C++: Reference Counts*, Guido van Rossum
- *Down for the count? Getting reference counting back in the ring*, Rifat Shahriyar, Stephen M. Blackburn and Daniel Frampton

v · t · e	**Memory management**	[hide]
Memory management as a function of an operating system		
Manual memory management	Static memory allocation · C dynamic memory allocation · new (C++) · delete (C++)	
Virtual memory	Demand paging · Page table · Paging · Virtual memory compression	
Hardware	Memory management unit · Translation lookaside buffer	
Garbage collection	Boehm garbage collector · Finalizer · Garbage · Mark-compact algorithm · **Reference counting** · Strong reference · Weak reference	
Memory segmentation	Protected mode · Real mode · Virtual 8086 mode · x86 memory segmentation	
Memory safety	Buffer overflow · Buffer over-read · Dangling pointer · Stack overflow	
Issues	Fragmentation · Memory leak · Unreachable memory	
Other	Automatic variable · International Symposium on Memory Management · Region-based memory management	

Categories: Automatic memory management | Memory management