Garbage collection (computer science)

From Wikipedia, the free encyclopedia

In computer science, **garbage collection** (**GC**) is a form of automatic memory management. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to abstract away manual memory management in Lisp.[1][2]

Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of approaches, including other techniques such as stack allocation and region inference. Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and, as a result, can have significant influence on performance. With good implementations, enough memory, and depending on application, garbage collection can be faster than manual memory management, while the opposite can also be true and has often been the case in the past with sub-optimal GC algorithms.

Resources other than memory, such as network sockets, database handles, user interaction windows, and file and device descriptors, are not typically handled by garbage collection. Methods used to manage such resources, particularly destructors, may suffice to manage memory as well, leaving no need for GC. Some GC systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed; this is called *finalization*. Finalization may introduce complications limiting its usability, such as intolerable latency between disuse and reclaim of especially limited resources, or a lack of control over which thread performs the work of reclaiming.

Contents

Principles

The basic principles of garbage collection are to find data objects in a program that cannot be accessed in the future, and to reclaim the resources used by those objects.

Many programming languages require garbage collection, either as part of the language specification (for example, Java, C#, D,[3] Go and most scripting languages) or effectively for practical implementation (for example, formal languages like lambda calculus); these are said to be *garbage collected languages*. Other languages were designed for use with manual memory management, but have garbage-collected implementations available (for example, C and C++). Some languages, like Ada, Modula-3, and C++/CLI, allow both garbage collection and manual memory management to co-exist in the same application by using separate heaps for collected and manually managed objects; others, like D, are garbage-collected but allow the user to manually delete objects and also entirely disable garbage collection when speed is required.

While integrating garbage collection into the language's compiler and runtime system enables a much wider choice of methods, *post-hoc* GC systems exist, such as ARC, including some that do not require recompilation. (*Post-hoc* GC is sometimes distinguished as *litter collection*.) The garbage collector will almost always be closely integrated with the memory allocator.

Advantages

Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of bugs are eliminated or substantially reduced:

- *Dangling pointer bugs*, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is dereferenced. By then the memory may have been reassigned to another use, with unpredictable results.
- *Double free bugs*, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again.
- Certain kinds of *memory leaks*, in which a program fails to free memory occupied by objects that have become unreachable, which can lead to memory exhaustion. (Garbage collection typically does not deal with the unbounded accumulation of data that is reachable, but that will actually not be used by the program.)
- Efficient implementations of persistent data structures

Some of the bugs addressed by garbage collection can have security implications.

Disadvantages

Typically, garbage collection has certain disadvantages, including consuming additional resources, performance impacts, possible stalls in program execution, and incompatibility with manual resource management.

Garbage collection consumes computing resources in deciding which memory to free, even though the programmer may have already known this information. The penalty for the convenience of not annotating object lifetime manually in the source code is overhead, which can lead to decreased or uneven performance.[4] A peer-reviewed paper came to the conclusion that GC needs five times the memory to compensate for this overhead and to perform as fast as explicit memory management.[5] Interaction with memory hierarchy effects can make this overhead intolerable in circumstances that are hard to predict or to detect in routine testing. The impact on performance was also given by Apple as a reason for not adopting garbage collection in iOS despite being the most desired feature.[6]

The moment when the garbage is actually collected can be unpredictable, resulting in stalls (pauses to shift/free memory) scattered throughout a session. Unpredictable stalls can be unacceptable in real-time environments, in transaction processing, or in interactive programs. Incremental, concurrent, and real-time garbage collectors address these problems, with varying trade-offs.

The modern GC implementations try to avoid blocking "stop-the-world" stalls by doing as much work as possible on the background (i.e. on a separate thread), for example marking unreachable garbage instances while the application process continues to run. In spite of these advancements, for example in .NET CLR paradigm it is still very difficult to maintain large heaps (millions of objects) with resident objects that get promoted to older generations without incurring noticeable delays (sometimes a few seconds).

Non-deterministic GC is incompatible with RAII based management of non-GCed resources. As a result, the need for explicit manual resource management (release/close) for non-GCed resources becomes transitive to composition. That is: in a non-deterministic GC system, if a resource or a resource-like object requires manual resource management (release/close), and this object is used as "part of" another object, then the composed object will also become a resource-like object that itself requires manual resource management (release/close).

Strategies

Tracing

Tracing garbage collection is the most common type of garbage collection, so much so that "garbage collection" often refers to tracing garbage collection, rather than other methods such as reference counting. The overall strategy consists of determining which objects should be garbage collected by tracing which objects are *reachable* by a chain of references from certain root objects, and considering the rest as garbage and collecting them. However, there are a large number of algorithms used in implementation, with widely varying complexity and performance characteristics.

Reference counting

Reference counting is a form of garbage collection whereby each object has a count of the number of references to it. Garbage is identified by having a reference count of zero. An object's reference count is incremented when a reference to it is created, and decremented when a reference is destroyed. When the count reaches zero, the object's memory is reclaimed.

As with manual memory management, and unlike tracing garbage collection, reference counting guarantees that objects are destroyed as soon as their last reference is destroyed, and usually only accesses memory which is either in CPU caches, in objects to be freed, or directly pointed by those, and thus tends to not have significant negative side effects on CPU cache and virtual memory operation.

There are a number of disadvantages to reference counting; this can generally be solved or mitigated by more sophisticated algorithms:

Cycles
: If two or more objects refer to each other, they can create a cycle whereby neither will be collected as their mutual references never let their reference counts become zero. Some garbage collection systems using reference counting (like the one in CPython) use specific cycle-detecting algorithms to deal with this issue.[7] Another strategy is to use weak references for the "backpointers" which create cycles. Under reference counting, a weak reference is similar to a weak reference under a tracing garbage collector. It is a special reference object whose existence does not increment the reference count of the referent object. Furthermore, a weak reference is safe in that when the referent object becomes garbage, any weak reference to it *lapses*, rather than being permitted to remain dangling, meaning that it turns into a predictable value, such as a null reference.

Space overhead (reference count)
: Reference counting requires space to be allocated for each object to store its reference count. The count may be stored adjacent to the object's memory or in a side table somewhere else, but in either case, every single reference-counted object requires additional storage for its reference count. Memory space with the size of an unsigned pointer is commonly used for this task, meaning that 32 or 64 bits of reference count storage must be allocated for each object. On some systems, it may be possible to mitigate this overhead by using a tagged pointer to store the reference count in unused areas of the object's memory. Often, an architecture does not actually allow programs to access the full range of memory addresses that could be stored in its native pointer size; certain number of high bits in the address is either ignored or required to be zero. If an object reliably has a pointer at a certain location, the reference count can be stored in the unused bits of the pointer. For example, each object in Objective-C has a pointer to its class at the beginning of its memory; on the ARM64 architecture using iOS 7, 19 unused bits of this class pointer are used to store the object's reference count.[8][9]

Speed overhead (increment/decrement)
: In naive implementations, each assignment of a reference and each reference falling out of scope often require modifications of one or more reference counters. However, in the common case, when a reference is copied from an outer scope variable into an inner scope variable, such that the lifetime of the inner variable is bounded by the lifetime of the outer one, the reference incrementing can be eliminated. The outer variable "owns" the reference. In the programming language C++, this technique is readily implemented and demonstrated with the use of `const` references. Reference counting in C++ is usually implemented using "smart pointers"[10] whose constructors, destructors and assignment operators manage the references. A smart pointer can be passed by reference to a function, which avoids the need to copy-construct a new smart pointer (which would increase the reference count on entry into the function and decrease it on exit). Instead the function receives a reference to the smart pointer which is produced inexpensively.

Requires atomicity
: When used in a multithreaded environment, these modifications (increment and decrement) may need to be atomic operations such as compare-and-swap, at least for any objects which are shared, or potentially shared among multiple threads. Atomic operations are expensive on a multiprocessor, and even more expensive if they have to be emulated with software algorithms. It

is possible to avoid this issue by adding per-thread or per-CPU reference counts and only accessing the global reference count when the local reference counts become or are no longer zero (or, alternatively, using a binary tree of reference counts, or even giving up deterministic destruction in exchange for not having a global reference count at all), but this adds significant memory overhead and thus tends to be only useful in special cases (it is used, for example, in the reference counting of Linux kernel modules).

Not real-time

Naive implementations of reference counting do not in general provide real-time behavior, because any pointer assignment can potentially cause a number of objects bounded only by total allocated memory size to be recursively freed while the thread is unable to perform other work. It is possible to avoid this issue by delegating the freeing of objects whose reference count dropped to zero to other threads, at the cost of extra overhead.

Escape analysis

Escape analysis can be used to convert heap allocations to stack allocations, thus reducing the amount of work needed to be done by the garbage collector. This is done using a compile-time analysis to determine whether an object allocated within a function is not accessible outside of it (i.e. escape) to other functions or threads. In such a case the object may be allocated directly on the thread stack and released when the function returns, reducing its potential garbage collection overhead.

Availability

Generally speaking, higher-level programming languages are more likely to have garbage collection as a standard feature. In some languages that do not have built in garbage collection, it can be added through a library, as with the Boehm garbage collector for C and C++.

Most functional programming languages, such as ML, Haskell, and APL, have garbage collection built in. Lisp is especially notable as both the first functional programming language and the first language to introduce garbage collection.[11]

Other dynamic languages, such as Ruby and Julia (but not Perl 5 or PHP before version 5.3,[12] which both use reference counting), also tend to use GC. Object-oriented programming languages such as Smalltalk, Java, JavaScript and ECMAScript usually provide integrated garbage collection. Notable exceptions are C++ and Delphi which have destructors.

BASIC

Historically, languages intended for beginners, such as BASIC and Logo, have often used garbage collection for heap-allocated variable-length data types, such as strings and lists, so as not to burden programmers with manual memory management. On early microcomputers, with their limited memory and slow processors, BASIC garbage collection could often cause apparently random, inexplicable pauses in the midst of program operation.

Some BASIC interpreters, such as Applesoft BASIC on the Apple II family, repeatedly scanned the string descriptors for the string having the highest address in order to compact it toward high memory, resulting in $O(n^2)$ performance, which could introduce minutes-long pauses in the execution of string-intensive

programs. A replacement garbage collector for Applesoft BASIC published in Call-A.P.P.L.E. (January 1981, pages 40–45, Randy Wigginton) identified a group of strings in every pass over the heap, which cut collection time dramatically. BASIC.System, released with ProDOS in 1983, provided a windowing garbage collector for BASIC that reduced most collections to a fraction of a second.

Objective-C

While the Objective-C traditionally had no garbage collection, with the release of OS X 10.5 in 2007 Apple introduced garbage collection for Objective-C 2.0, using an in-house developed runtime collector.[13] However, with the 2012 release of OS X 10.8, garbage collection was deprecated in favor of LLVM's automatic reference counter (ARC) that was introduced with OS X 10.7.[14] Furthermore, since May 2015 Apple even forbids the usage of garbage collection for new OS X applications in the App Store.[15][16] For iOS, garbage collection has never been introduced due to problems in application responsivity and performance;[6][17] instead, iOS uses ARC.[18][19]

Limited environments

Garbage collection is rarely used on embedded or real-time systems because of the perceived need for very tight control over the use of limited resources. However, garbage collectors compatible with such limited environments have been developed.[20] The Microsoft .NET Micro Framework and Java Platform, Micro Edition are embedded software platforms that, like their larger cousins, include garbage collection.

Compile-time use

Compile-time garbage collection is a form of static analysis allowing memory to be reused and reclaimed based on invariants known during compilation. This form of garbage collection has been studied in the Mercury programming language,[21] and it saw greater usage with the introduction of LLVM's automatic reference counter (ARC) into Apple's ecosystem (iOS and OS X) in 2011.[18][19][15]

Real-time systems

Incremental, concurrent, and real-time garbage collectors have been developed, such as Baker's algorithm or Lieberman's algorithm.[22][23][24]

In Baker's algorithm, the allocation is done in either half of a single region of memory. When it becomes half full, a garbage collection is performed which moves the live objects into the other half and the remaining objects are implicitly deallocated. The running program (the 'mutator') has to check that any object it references is in the correct half, and if not move it across, while a background task is finding all of the objects.[25]

Generational garbage collection schemes are based on the empirical observation that most objects die young. In generational garbage collection two or more allocation regions (generations) are kept, which are kept separate based on object's age. New objects are created in the "young" generation that is regularly collected, and when a generation is full, the objects that are still referenced from older regions are copied into the next oldest generation. Occasionally a full scan is performed.

Some high-level language computer architectures include hardware support for real-time garbage collection.

Most implementations of real-time garbage collectors use tracing. Such real-time garbage collectors meet hard real-time constraints when used with a real-time operating system.[26]

See also

- Destructor (computer programming)
- International Symposium on Memory Management
- Memory management

References

1. "Recursive functions of symbolic expressions and their computation by machine, Part I". Portal.acm.org. Retrieved 29 March 2009.
2. "Recursive functions of symbolic expressions and their computation by machine, Part I". Retrieved 29 May 2009.
3. "Overview — D Programming Language". *dlang.org*. Digital Mars. Retrieved 2014-07-29.
4. Zorn, Benjamin (1993-01-22). "The Measured Cost of Conservative Garbage Collection". Department of Computer Science, University of Colorado Boulder. CiteSeerX 10.1.1.14.1816.
5. Matthew Hertz; Emery D. Berger (2005). "Quantifying the Performance of Garbage Collection vs. Explicit Memory Management" (PDF). OOPSLA 2005. Retrieved 2015-03-15.
6. "Developer Tools Kickoff — session 300" (PDF). *WWDC 2011*. Apple, inc. 2011-06-24. Retrieved 2015-03-27.
7. "Reference Counts". *Extending and Embedding the Python Interpreter*. 21 February 2008. Retrieved 22 May 2014.
8. Mike Ash. "Friday Q&A 2013-09-27: ARM64 and You". mikeash.com. Retrieved 2014-04-27.
9. "Hamster Emporium: [objc explain]: Non-pointer isa". Sealiesoftware.com. 2013-09-24. Retrieved 2014-04-27.
10. RAII, Dynamic Objects, and Factories in C++, Roland Pibinger, 3 May 2005 (http://www.codeproject.com/Articles/10141/RAII-Dynamic-Objects-and-Factories-in-C#1)
11. Chisnall, David (2011-01-12). *Influential Programming Languages, Part 4: Lisp*.
12. "PHP: Performance Considerations". *php.net*. Retrieved 14 January 2015.
13. Objective-C 2.0 Overview (https://web.archive.org/web/20100724195423/http://developer.apple.com/leopard/overview/objectivec2.html)
14. Mac OS X 10.7 Lion: the Ars Technica review (http://arstechnica.com/apple/2011/07/mac-os-x-10-7/11/) John Siracusa (20 Juli 2011)
15. Apple says Mac app makers must transition to ARC memory management by May (http://appleinsider.com/articles/15/02/20/apple-says-mac-app-makers-must-transition-to-arc-memory-management-by-may) by AppleInsider (February 20, 2015)
16. Cichon, Waldemar (2015-02-21). "App Store: Apple entfernt Programme mit Garbage Collection". Heise.de. Retrieved 2015-03-30.
17. Silva, Precious (2014-11-18). "iOS 8 vs Android 5.0 Lollipop: Apple Kills Google with Memory Efficiency". International Business Times. Retrieved 2015-04-07.
18. Rob Napier, Mugunth Kumar (2012-11-20). "iOS 6 Programming Pushing the Limit". John Wiley & Sons. Retrieved 2015-03-30.
19. Cruz, José R.C. (2012-05-22). "Automatic Reference Counting on iOS". Dr. Dobbs. Retrieved 2015-03-30.
20. "Wei Fu and Carl Hauser, "A Real-Time Garbage Collection Framework for Embedded Systems". ACM SCOPES '05, 2005". Portal.acm.org. Retrieved 9 July 2010.

21. Mazur, Nancy (May 2004). *Compile-time garbage collection for the declarative language Mercury* (PDF) (Thesis). Katholieke Universiteit Leuven.
22. Lorenz Huelsbergen, Phil Winterbottom. "Very Concurrent Mark-&-Sweep Garbage Collection without Fine-Grain Synchronization" (http://doc.cat-v.org/inferno/concurrent_gc/concurrent_gc.pdf). 1998.
23. "GC FAQ" (http://www.iecc.com/gclist/GC-faq.html).
24. Henry Lieberman. A Real-Time Garbage Collector Based on the Lifetimes of Objects (http://web.media.mit.edu/~lieber/Lieberary/GC/Realtime/Realtime.html)
25. Baker, H.G. List processing in real time on a serial computer. Commun. ACM 21, 4 (April 1978) 280- 294. see also description (http://web.media.mit.edu/~lieber/Lieberary/GC/Realtime/Realtime.html)
26. McCloskey, Bacon, Cheng, Grove."Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors" (http://researcher.watson.ibm.com/researcher/files/us-groved/rc24504.pdf). 2008.

Further reading

- Jones, Richard; Hosking, Antony; Moss, Eliot (19 August 2011). *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures Series. Chapman and Hall/CRC. ISBN 1-4200-8279-5.
- Jones, Richard; Lins, Rafael D. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley. ISBN 0-471-94148-4.
- Wilson, Paul R.; Johnstone, M. S.; Neely, M.; Boles, D. (1995). "Dynamic Storage Allocation: A Survey and Critical Review". *International Workshop on Memory Management*. CiteSeerX 10.1.1.47.275.
- Wilson, Paul R. (1992). "Uniprocessor Garbage Collection Techniques". *IWMM '92 Proceedings of the International Workshop on Memory Management*. Springer-Verlag. CiteSeerX 10.1.1.47.2438.

External links

The Wikibook *Memory Management* has a page on the topic of: ***Garbage Collection***

- The Memory Management Reference (http://www.memorymanagement.org/)
- The Very Basics of Garbage Collection (http://basen.oru.se/kurser/koi/2008-2009-p1/texter/gc/index.html)
- Java SE 6 HotSpot™ Virtual Machine Garbage Collection Tuning (http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html)
- TinyGC - an independent implementation of the BoehmGC API (http://tinygc.sourceforge.net/)
- Conservative Garbage Collection Implementation for C Language (http://www.codeproject.com/KB/cpp/conservative_gc.aspx)
- MeixnerGC - an incremental mark and sweep garbage collector for C++ using smart pointers (http://sourceforge.net/projects/meixnergc/)

Categories: Memory management | Automatic memory management | Solid-state computer storage