Garbage collection

1 Readings

1.1 Set papers

The week's set papers are as follows:

- *A unified theory of garbage collection* (Bacon, Cheng, and Rajan, 2004)
- *Quantifying the performance of garbage collection vs. explicit memory management* (Hertz and Berger, 2005)
- *Fast conservative garbage collection* (Shahriyar, Blackburn, and McKinley, 2014)

You are invited to **compare and contrast two of these papers**.

1.2 Background reading

The book *The Garbage Collection Handbook: The art of automatic memory management* (Jones, Hosking, and Moss, 2011) is a comprehensive and clearly-written guide, and covers significantly more than you will need for this assignment. A second edition (Jones, Hosking, and Moss, 2023) was recently published.

Uniprocessor Garbage Collection Techniques (Wilson, 1992) is a freely available shorter survey, although now rather dated.

2 History

2.1 Beginnings

The first work on automatic memory management coincided with the introduction of LISP: *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I* (McCarthy, 1960) describes a mark-sweep collector for a machine (IBM 704) that is tiny by today's standards, with 32 kilowords of memory.

Reference counting was introduced around the same time (Collins, 1960); despite many subsequent developments, both techniques are still in use today, and can be viewed as algorithmic duals within a single framework, as one of the set papers describes.

2.2 Developments

It is difficult to give precise dates, since several ideas developed gradually over time. In some cases the dates below indicate the point at which a technique first became practical.

1960 **Mark-sweep garbage collection**, described in *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I* (McCarthy, 1960).

1960 **Reference counting**, described in *A method for overlapping and erasure of lists* (Collins, 1960).

1969 A **stop-copy semispace collector**, described in *A LISP Garbage-Collector for Virtual-Memory Computer Systems* (Fenichel and Yochelson, 1969).

1984 Several works introduced **generational collectors** around this time, including *A Real-Time Garbage Collector Based on the Lifetimes of Objects* (Lieberman and Hewitt, 1983) and *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm* (Ungar, 1984).

1993 The first widely-used **conservative collector** was introduced by *Space Efficient Conservative Garbage Collection* (Boehm, 1993).

1999 Arguably the first practical **real-time collector** was described in *On Bounding Time and Space for Multiprocessor Garbage Collection* (Blelloch and Cheng, 1999) ("the first multiprocessor garbage collection algorithm with provable bounds on time and space") and extended in *A Parallel, Real-Time Garbage Collector* (Cheng and Blelloch, 2001).

2.3 Status

Garbage collection is still an active area of research, and likely to remain so, as new architectures and ever larger data sets introduce new challenges. Over the last few decades, CPU speed improvements have significantly outpaced memory speed improvements, and locality and cache usage consequently play a large role in collector design. Similarly, with the widespread availability of parallel architectures, concurrent and parallel collectors have become increasingly important.

Mark

```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```

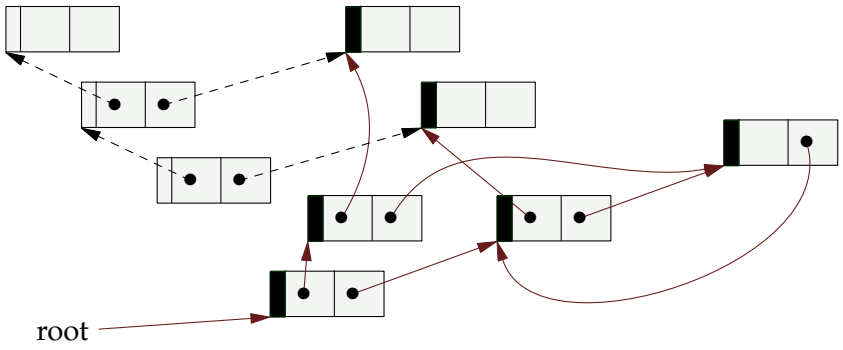

Figure 1: Mark & sweep

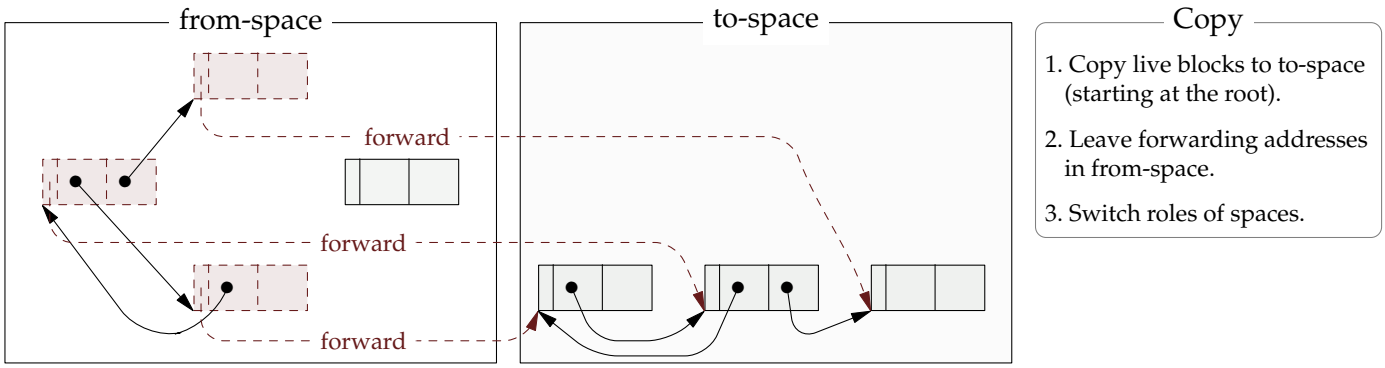

Figure 2: Copying collection

3 Basics

Much of the vocabulary around garbage collection has become standard:

- The **heap** consists of one or more blocks of contiguous words
- An **object** is a heap-allocated (typically) contiguous region addressed by zero or more pointers in an application
- A **mutator** is an application execution thread, opaque to the collector except for operations affecting the heap (allocate, discard, read, write)
- A **root** is a location accessible to the mutator (typically in static global storage, stack space, or registers) that points into the heap
- An object is **live** if a mutator will access it in the future; liveness is approximated by **reachability** via a chain of pointers from some root

There are several simple standard algorithms:

1. **mark & sweep** involves a traversal, starting from the roots, that sets a bit in every reachable object, followed by a second pass over the heap to reclaim unmarked objects.

   Figure 1 shows the state of a heap after the marking traversal.

2. **stop & copy** divides the heap into **from-space** and **to-space** regions, copies the reachable object graph from from-space to to-space before switching the roles of the two regions.

   Figure 2 shows the state of a heap after copying but before the role switch. The **forwarding pointers** allow the objects in to-space to be located from the corresponding objects in from-space during traversal.

3. **reference counting** involves tracking the number of pointers to each object, typically via a counter stored in the object itself.

4. **conservative collection** is used when limited run-time support (e.g. in a language compiled to C) leaves the collector with imperfect information about object layout; instead a safe(?) approximation is used based on examining bit patterns to guess whether they represent pointers.

5. **generational collection** involves dividing the heap into several regions: minor regions that hold **young** (recently allocated) objects are collected frequently and reachable objects are **promoted** to infrequently-collected major regions, under the (empirically justified, e.g. Jones, Hosking, and Moss (2011, p113)) assumption that such objects are likely to have long lifetimes

4 Practical considerations

4.1 Metrics

It is relatively straightforward to write a correct collector, but writing a collector with good performance is challenging, in part due to need to balance competing performance needs:

1. **throughput** refers to the performance of mutators. Time spent in collection clearly negatively affects throughput, but the relationship is more subtle, since the design of a collector may impose additional costs on heap operations performed by a mutator.
2. **latency** refers to the pauses in mutator execution introduced by collection.
3. **space overhead** may vary: most collector designs involve expanding the size of allocated objects to track additional information (e.g. mark bits or layout information).
4. Other metrics relevant to performance may be consequences of the combination of program behaviour and collector design; they include **maximum heap size**, **allocation rate**, collection frequency, **mean object size**, and the proportion of the heap occupied by large objects.

In practice, many of these metrics are more subtle, or are only useful in combination. For example, pause times alone provide little information; a good distribution of pause times is needed to ensure that there is opportunity for mutators to make progress.

4.2 Hybrid systems

The standard algorithms have different performance characteristics that each one suited to particular circumstances. For example, **compaction** (which takes place naturally with stop-and-copy collection) can make collection slower, but may increase mutator throughput by improving locality.

In practice, many mature systems use several of the standard algorithms in some combination. For example, the implementation of the Cedar langauge (Rovner, 1985) combined reference counting with a periodic mark-and-sweep to reclaim cycles; furthermore, to reduce reference count update frequency it used conservative collection for activation frames, avoiding the need to modify the count when heap objects were referenced from the stack.

Other innovations to improve collector performance involve partitioning the heap according to the characteristics of the objects stored in each partition (mobility, size, space, kind, yield, thread, availabiltiy, mutability, etc.).

4.3 Extras: finalisers, weak references, ephemerons, etc.

The semantics of many garbage-collected programming languages make no mention of garbage collection, since there is no semantic difference between a correct collector and a collector that never runs. However, to support resource management, practical systems often expose the operation of the collector to the application via mechanisms such as finalisers (which run application code on object collection), ephemerons (Hayes, 1997) or weak references (i.e. pointers that do not prevent collection).

5 Questions and challenges

In addition to performance improvements and adaptations for new architectures, work is ongoing on a several other aspects of garbage collection design and implementation.

5.1 Cross-language garbage collection

Many applications are implemented in a combination of languages, whose implementations may employ quite different memory management strategies. Foreign function interfaces allow such applications to construct object graphs that span the heaps of the implementations, making collection challenging, since none of the implementations have sufficient information to reclaim such graphs. *Collecting Cyclic Garbage across Foreign Function Interfaces: Who Takes the Last Piece of Cake?* (Yamazaki, Nakamaru, Shioya, et al., 2023) describes a solution that coordinates different runtimes using proxy objects.

5.2 Verified garbage collection

Verified compilers have been developed for a number of languages. Where the language implementations support garbage collection, it is natural to wish to verify the collector, too. *A Verified Generational Garbage Collector for CakeML* (Ericsson, Myreen, and Pohjola, 2019) describes a recent verified implementation of a moderately complex collector.

5.3 Hardware support for garbage collection

Hardware support for garbage collection dates back at least to the Symbolics 3600 Lisp machine (1983), and is still an active area of research. For example, *A Hardware Accelerator for Tracing Garbage Collection* (Maas, Asanović, and Kubiatowicz, 2018) present a specialized accelerator with support for mark-and-sweep collection that improves mark speed by 4.2×.

References

Bacon, D. F., P. Cheng, and V. T. Rajan (2004). "A unified theory of garbage collection". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*. Ed. by J. M. Vlissides and D. C. Schmidt. ACM, pp. 50–68. DOI: 10.1145/1028976.1028982.

Blelloch, G. E. and P. Cheng (1999). "On Bounding Time and Space for Multiprocessor Garbage Collection". In: *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*. Ed. by B. G. Ryder and B. G. Zorn. ACM, pp. 104–117. DOI: 10.1145/301618.301648.

Boehm, H. (1993). "Space Efficient Conservative Garbage Collection". In: *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*. Ed. by R. Cartwright. ACM, pp. 197–206. DOI: 10.1145/155090.155109.

Cheng, P. and G. E. Blelloch (2001). "A Parallel, Real-Time Garbage Collector". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. Ed. by M. Burke and M. L. Soffa. ACM, pp. 125–136. DOI: 10.1145/378795.378823.

Collins, G. E. (1960). "A method for overlapping and erasure of lists". In: *Commun. ACM* 3.12, pp. 655–657. DOI: 10.1145/367487.367501.

Ericsson, A. S., M. O. Myreen, and J. Å. Pohjola (2019). "A Verified Generational Garbage Collector for CakeML". In: *J. Autom. Reason.* 63.2, pp. 463–488. DOI: 10.1007/s10817-018-9487-z.

Fenichel, R. R. and J. C. Yochelson (Nov. 1969). "A LISP Garbage-Collector for Virtual-Memory Computer Systems". In: *Commun. ACM* 12.11, pp. 611–612. ISSN: 0001-0782. DOI: 10.1145/363269.363280.

Hayes, B. (1997). "Ephemerons: A New Finalization Mechanism". In: *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1997, Atlanta, Georgia, October 5-9, 1997*. Ed. by M. E. S. Loomis, T. Bloom, and A. M. Berman. ACM, pp. 176–183. DOI: 10.1145/263698.263733.

Hertz, M. and E. D. Berger (2005). "Quantifying the performance of garbage collection vs. explicit memory management". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. Ed. by R. E. Johnson and R. P. Gabriel. ACM, pp. 313–326. DOI: 10.1145/1094811.1094836.

Jones, R. E., A. L. Hosking, and J. E. B. Moss (2011). *The Garbage Collection Handbook: The art of automatic memory management*. Chapman and Hall / CRC Applied Algorithms and Data Structures Series. CRC Press. ISBN: 978-1-4200-8279-1.

Jones, R. E., A. L. Hosking, and J. E. B. Moss (2023). *The Garbage Collection Handbook: The art of automatic memory management*. Chapman and Hall / CRC Applied Algorithms and Data Structures Series. CRC Press. ISBN: 978-1032218038.

Lieberman, H. and C. E. Hewitt (June 1983). "A Real-Time Garbage Collector Based on the Lifetimes of Objects". In: 26.6. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981, pp. 419–429. DOI: 10.1145/358141.358147.

Maas, M., K. Asanović, and J. Kubiatowicz (June 2018). "A Hardware Accelerator for Tracing Garbage Collection". In: *45th Annual*. Los Angeles, CA. DOI: 10.1109/ISCA.2018.00022.

McCarthy, J. (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Commun. ACM* 3.4, pp. 184–195. DOI: 10.1145/367177.367199.

Rovner, P. (July 1985). *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language*. Technical Report CSL–84–7.

Shahriyar, R., S. M. Blackburn, and K. S. McKinley (2014). "Fast conservative garbage collection". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by A. P. Black and T. D. Millstein. ACM, pp. 121–139. DOI: 10.1145/2660193.2660198.

Ungar, D. M. (1984). "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm". In: *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*. Ed. by W. E. Riddle and P. B. Henderson. ACM, pp. 157–167. DOI: 10.1145/800020.808261.

Wilson, P. R. (1992). "Uniprocessor Garbage Collection Techniques". In: *Proceedings of the International Workshop on Memory Management*. IWMM '92. Berlin, Heidelberg: Springer-Verlag, pp. 1–42. ISBN: 354055940X.

Yamazaki, T., T. Nakamaru, R. Shioya, T. Ugawa, and S. Chiba (June 2023). "Collecting Cyclic Garbage across Foreign Function Interfaces: Who Takes the Last Piece of Cake?" In: *Proc. ACM Program. Lang.* 7.PLDI. DOI: 10.1145/3591244.