WIKIPEDIA
The Free Encyclopedia

Mark-compact algorithm

From Wikipedia, the free encyclopedia

In computer science, a **mark-compact algorithm** is a type of garbage collection algorithm used to reclaim unreachable memory. Mark-compact algorithms can be regarded as a combination of the mark-sweep algorithm and Cheney's copying algorithm. First, reachable objects are marked, then a compacting step relocates the reachable (marked) objects towards the beginning of the heap area. Compacting garbage collection is used by Microsoft's Common Language Runtime and by the Glasgow Haskell Compiler.

Contents [hide]

Algorithms [edit]

After marking the live objects in the heap in the same fashion as the mark-sweep algorithm, the heap will often be fragmented. The goal of mark-compact algorithms is to shift the live objects in memory together so the fragmentation is eliminated. The challenge is to correctly update all pointers to the moved objects, most of which will have new memory addresses after the compaction. The issue of handling pointer updates is handled in different ways.

Table-based compaction [edit]

A table-based algorithm was first described by Haddon and Waite in 1967.[1] It preserves the relative placement of the live objects in the heap, and requires only a constant amount of overhead.

Compaction proceeds from the bottom of the heap (low addresses) to the top (high addresses). As live (that is, marked) objects are encountered, they are moved to the first available low address, and a record is appended to a **break table** of relocation information. For each live object, a record in the break table consists of the object's original address before the compaction and the difference between the original address and the new address after compaction. The break table is stored in the heap that is being compacted, but in an area that are marked as unused. To ensure that compaction will always succeed, the minimum object size in the heap must be larger than or the same size as a break table record.

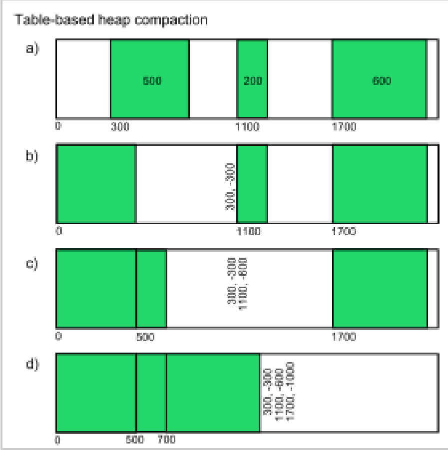

Illustration of the table-heap compaction algorithm. Objects that the marking phase has determined to be reachable (live) are colored, free space is blank.

As compaction progresses, relocated objects are copied towards the bottom of the heap. Eventually an object will need to be copied to the space occupied by the break table, which now must be relocated elsewhere. These movements of the break table, (called *rolling the table* by the authors) cause the relocation records to become disordered, requiring the break table to be sorted after the compaction is complete. The cost of sorting the break table is $O(n \log n)$, where n is the number of live objects that were found in the mark stage of the algorithm.

Finally, the break table relocation records are used to adjust pointer fields inside the relocated objects. The live objects are examined for pointers, which can be looked up in the sorted break table of size n in $O(\log n)$ time if the break table is sorted, for a total running time of $O(n \log n)$. Pointers are then adjusted by the amount specified in the relocation table.

LISP2 Algorithm [edit]

In order to avoid $O(n \log n)$ complexity, the LISP2 algorithm uses 3 different passes over the heap. In addition, heap objects must have a separate forwarding pointer slot that is not used outside of garbage collection.

After standard marking, the algorithm proceeds in the following 3 passes:

1. Compute the forwarding location for live objects.
   - Keep track of a *free* and *live* pointer and initialize both to the start of heap.
   - If the *live* pointer points to a live object, update that object's forwarding pointer to the current *free* pointer and increment the *free* pointer according to the object's size.
   - Move the *live* pointer to the next object
   - End when the *live* pointer reaches the end of heap.
2. Update all pointers
   - For each live object, update its pointers according to the forwarding pointers of the objects they point to.
3. Move objects
   - For each live object, move its data to its forwarding location.

This algorithm is $O(n)$ on the size of the heap; it has a better complexity than the table-based approach, but the table-based approach's n is the size of the used space only, not the entire heap space as in the LISP2 algorithm. However, the LISP2 algorithm is simpler to implement.

References [edit]

1. ^ B. K. Haddon and W. M. Waite (August 1967). "A compaction procedure for variable length storage elements". *Computer Journal* **10**: 162–165.

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 algorithms

This page was last modified on 26 January 2014, at 07:22.

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

Privacy policy About Wikipedia Disclaimers Contact Wikipedia Developers Mobile view