Memory Allocation Myths and Half–Truths

Hans–J. Boehm
Silicon Graphics

`boehm@sgi.com`
`http://reality.sgi.com/boehm`

Introduction to Memory Allocation

How do `malloc/free` (`new/delete`, STL allocators) work?

Problem:

- Obtain memory from OS in large chunks.
- Allocator maintains a private data structure to keep track of free vs. in–use blocks.
- Each call to `malloc` reserves and returns a pointer to a smaller amount of memory.
- Each call to `free` allows memory to be reused by another `malloc` call, but not usually by another concurrent process.

Goals:

1) Minimize unused space, a.k.a. *fragmentation*. The ratio

$$\frac{\textit{Max. memory allocated from OS}}{\textit{Max memory } \texttt{malloc}\textit{'d but not } \texttt{free}\textit{'d}}$$

should be as close to one as possible.

2) `Malloc` and `free` should be *fast*.

3) Often they need to be *thread-safe*, i.e. protect against concurrent access to the allocator data structure.

4) Sometimes they need to conform to long obsolete interfaces (e.g. contents of `p` are not immediately altered by `free(p)`) or work around bugs in "important" programs (e.g. `free(p); free(p)`).

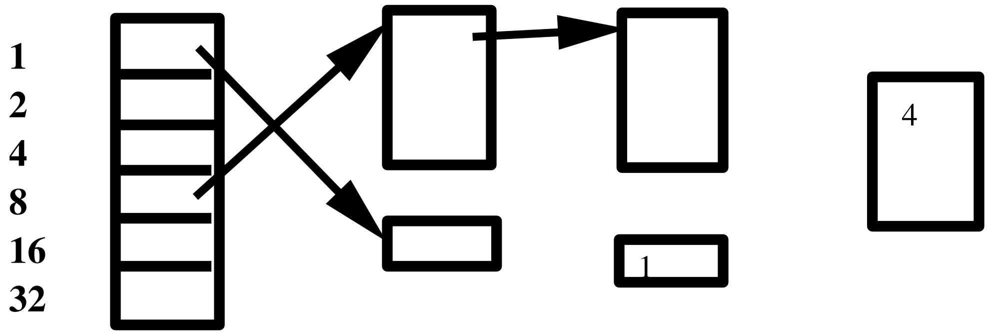
5) Occasionally we need real-time guarantees, but we'll ignore that here ...

An allocation algorithm

(Powers–of–2 segregated storage)

Allocator Data Structure:

We have an array of initially empty singly–linked free lists, one for each possible power of 2 size. We steal one word from each allocated object to remember the size.



Malloc:

1) Round up request size to next power of 2.
2) If desired free list is nonempty, remove and return first element.
3) Find first nonempty free list for larger objects. If there is none, obtain more memory from OS.
4) Split next largest object in 2 and add halves to next smaller free list. Repeat with one of the resulting objects.
5) Remove and return first element from desired free list.

`Free:`

Add object to free list determined by stored size.

Analysis:

- Objects are only split, but never combined.
- The number of objects of each size class (allocated + on free list) is at most one more than was needed at one time.
- Memory use is optimal for size class that used most memory at any point.
- Other size classes each use less memory.
- Memory obtained from OS is at most

  (number of size classes) * (memory used at once)
- This is can be made to be

  O(log(largest/smallest) * (memory used at once))

  This is optimal for the worst case (Robson)
- For a sufficiently large ratio of allocations to memory used, most allocations consist of a check and a free list removal.
- Fragmentation performance is mediocre. Placement algorithm is simplistic. See Wilson et al. (85% expected fragmentation overhead).

Fragmentation

There are many possible *placement algorithms*.
The choice significantly affects space overhead.

For details, I recommend

Wilson, Johnstone, Neely, and Boles, "Dynamic Storage Allocation: A Survey and Critical Review"
`ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps`

The bottom line (my interpretation):

– Good placement (*e.g.* best fit) reduces expected fragmentation (aside from headers, alignment) to **< 15%** on the real programs they measured. One can probably do still better. Contrast with worst case.
– Clever algorithms and data structures can result in reasonably fast allocators with good placement (*e.g.* Doug Lea’s)
– It’s unclear exactly how much good placement costs in execution time.
– It’s unclear whether fragmentation overhead increases significantly for longer running programs. I haven’t seen convincing evidence that it does.

Allocation Myth 1:
Long running programs will fragment the heap more and more, consuming unbounded memory.

The Truth:
This is provably false, for even our simple allocator.

***Worst–case bound*: Factor of 20 or so overhead.**

***Experience with real programs*: Overhead of more than 50% is very unlikely with a good allocator.**

Allocation Myth 2:
Malloc is inherently slow.

The truth:
Many modern malloc implementations basically allocate small objects by removing the first element from a free list, and deallocate by adding to a free list. That's often the best one could possibly do.

They may or may not be slower than a custom allocator:

- **Locking needed for thread–safety.**
- **Lack of inlining.**
- **Better placement at the expense of execution time.**
- **Backward compatibility constraints.**
- **Poor implementation.**

Allocation Myth 3:
It's usually better to implement your own per–class memory allocator in C++.

The truth:

- **This often doesn't beat a single centralized memory allocator for speed. It's easy to be slower. The SGI STL shared allocator is usually no slower than the HP STL per class allocators. (The generated code is basically identical, so long as the allocator is inlined.)**
- **It risks serious fragmentation in large systems. Every class is likely to retain as much memory as it ever needed. This is potentially much worse than even the 85% for a fast, dumb malloc.**
- **It is very hard to make the result thread–safe. Many of the private free lists must be protected by a lock. This is likely to be the only reason many classes will be thread–package––dependent.**
- **Interacts poorly with garbage collection. Free lists may point to other objects, which will appear to be reachable.**

Allocation Myth 4:
Non–garbage–collected programs should always deallocate all memory they allocate.

The Truth:
Omitted deallocations in frequently executed code cause growing leaks. They are rarely acceptable.

but

Programs that retain most allocated memory until program exit often perform better without any intervening deallocation. `Malloc` is much easier to implement if there is no `free`.

In most cases, deallocating memory just before program exit is pointless. The OS will reclaim it anyway. `Free` will touch and page in the dead objects; the OS won't.

Consequence:

Be careful with "leak detectors" that count allocations. *Some "leaks" are good!*

Allocation Myth 5: (less commonly heard)
There is no need to deallocate anything; the OS will just page it out.

The truth:
1) This may or may not cause serious paging locality problems. Live memory may be distributed over many pages.

2) Many users don’t have that much swap space.

3) For a program that does this every page of new allocation will eventually require some other page to be written to disk. On a modern workstation, with a good allocator, memory allocation rates are much higher than disk transfer rates, never mind seek times ...

Some Simple Measurements

- Nondestructively reverse a list of 1000 40–byte elements. Deallocate old list. Check every other time. Repeat 2000 times.
- 80 MB allocation, 2 million objects, only 80KB live at once.
- Run on a 64MB 133MHz R4600 Indy SC (average, slightly obsolete SGI workstation, 20% variations observed)

Seconds	user	system	elapsed
vendor libc#	9	0.05	9.5
Doug Lea+	4.7	0.04	6.0
custom	3.2	0.04	3.4
vendor fast*	3.3	0.04	3.5
incr. ptr/no free	2.7	4	22
incr. ptr/free on exit	3.1	9	86

#`malloc` in precompiled dynamic library, thread–safe

*SGI 7.2, –IPA:use_intrinsic=on (thread–unsafe, inlinable, less emphasis on space)

+with IPA, i.e. whole program optimization

Garbage collectors

Automatically reclaim memory: no `free`

Quick Taxonomy:

- **Reference counting**
- **Tracing collectors – traverse live objects**
  - **Mark–and–Sweep collectors**
    - **Nonmoving Mark and Sweep**
    - **Compacting Mark and Sweep**
      - **(Important, but not considered here.)**
  - **Copying collectors**
  - **Hybrids**
    - **(Probably often the best solution, but not considered here)**

Other collector characteristics:

Conservative tracing

Tracing with incomplete pointer information.
If it looks like a pointer, treat it as one.

Generational tracing:

Try to reclaim younger objects more frequently.

(Important, but not considered here.)

Real time or incremental

(Important, but not considered here.)

Does it trace from "dead" variables?

Are there guarantees about what will be reclaimed?

(Important, but not considered here.)

Quality of implementation

(Important: treat measurements with caution.)

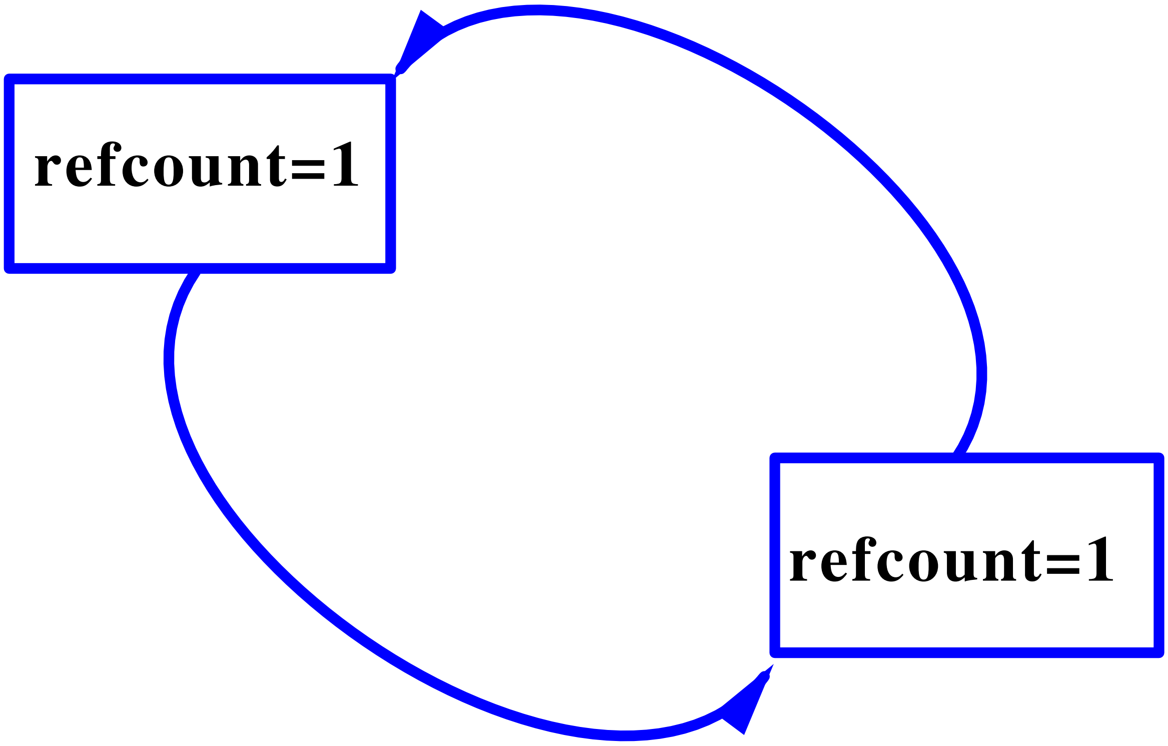
Reference Counting:

- **Conventional nonmoving allocator.**
- **Each object maintains a count of the number of pointers that refer to it.**
- **Object can be deallocated when count reaches 0.**
- **Can (more or less) be implemented with C++ "smart pointers"**
- **p = q;   becomes**

```
if (q != NULL)
  atomically{++(q -> refcount);}
if (p != NULL) {
  size_t new_count;
  atomically{
     new_count =
          --(p -> refcount);
  }
  if (new_count == 0)
     dealloc_decr_counts(p);
}
p = q;
```

Reference count evaluation:

- Small space overhead.
- Pointer assignments and pointer parameters are expensive.
- Multithreaded version requires atomic increment and decrement. (Generic locks are too slow.)
- Cycles of garbage:



Nonmoving Mark–Sweep GC

– Can use a mostly conventional allocator.
  (In practice it's tuned to the collector.)

– Occasionally, when more memory is needed:

1. Mark all objects referenced by pointer variables.
2. Mark all objects referenced by newly marked objects.
3. Repeat step 2 until no more objects are marked. (Typically a stack of newly marked objects is maintained.)
4. Deallocate unmarked objects.

Conservative variant:

– Start with stack, registers, and static data.
– Treat every bit pattern that happens to be inside an allocated heap object as a pointer.

––> various partially conservative strategies are possible and often desirable.

Nonmoving GC evaluation

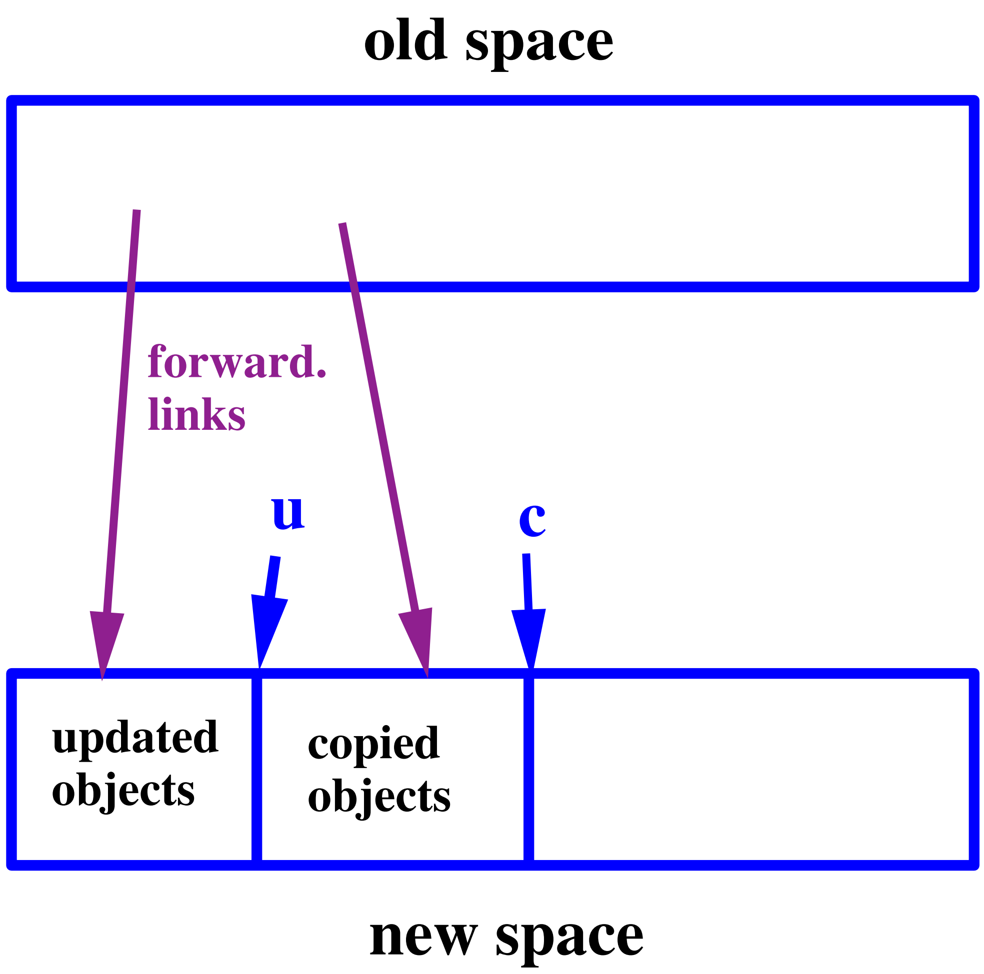
- no leaks, premature frees, ...
- less synchronization (no `free`)
- Subject to fragmentation
- Don’t want to collect after every allocation
  ==> Garbage is not immediately deallocated
  ==> Some space overhead
- Usually some space overhead for GC data structures (may matter for small heaps)
- Stack of recently marked objects may consume space and/or complicate algorithm
- Garbage collection touches much of reachable memory

Conservative GC:

- Usually (99.9%) works with unmodified C/C++ compilers.
- May occasionally require some layout information to reclaim sufficient memory.
- May (at least theoretically) fail with some compiler optimizations.

Copying collection (Cheney)

- Compacts: free space is contiguous
- Can use bump–the–pointer allocator
- Memory is divided into 2 spaces.
  Only one is active at a time.
- Garbage collector copies live memory from one (old space) to another (new space) which then becomes active.



GC advances u pointer.

Copying GC evaluation

- no leaks, premature frees
- less synchronization
- very cheap allocation
- no fragmentation
- often simple to implement
- doubles address space usage
- both old and new space must exist simultaneously
- space overhead to reduce GC frequency
- garbage collections touches 2 copies of reachable memory, including pointerfree objects
- harder to use with C/C++

Disclaimer:

There are more sophisticated variants of copying collection that do not share all of the space overhead. Combinations with mark–sweep collection seem particularly attractive, but are not discussed here.

GC Myth 1:

Copying Collection is faster than mark–sweep collection since it touches only reachable memory; there is no sweep phase.

The truth:
Heap size is generally a constant (e.g. 1.5) times the amount of live memory.
The sweep phase is not the bottleneck.
The fact that copying collectors touch pointerfree memory (e.g. 60% of memory) is typically more significant in practice.

In fact, it's hard to make this claim precise:
It relies on an arbitrary distinction between allocation and collection.

Mark–sweep collectors don't really need a sweep phase anyway:

- **Sweeping can be interleaved with allocation, or**
- **The mark bits can be used directly during allocation, or**
- **Sweeping can be done in constant time (Baker treadmill).**

Copying does allow faster allocation.

GC Myth 2:
Copying collection has better paging behavior because it compacts memory.

The truth:
Copying collectors tend to page much sooner, since they tend to touch roughly twice as much memory. This factor of 2 is much larger than the expected fragmentation overhead for a nonmoving collector.

Copying collectors do rearrange data, but there seems to be no clear consensus over whether this yields better or worse locality than the ordering you get from a well–designed mark–sweep collector. (Cheney copying produces breadth–first ordering, which may be much worse than a nonmoving collector. But one can do better.)

GC Myth 3:
Garbage collectors are always slower than explicit memory deallocation.
GC Myth 4:
Garbage collectors are always faster than explicit memory deallocation.

The truth:
Explicit allocation/deallocation cost is usually independent of the object cost. Garbage collection frequency increases in proportion to object size.

Very small objects & large sparse heap ==>
GC is usually cheaper, especially with threads

Large objects & small dense heap ==>
Explicit deallocation is cheaper

Reverse test, 8 byte objects, 2MB heap, 10000 reversals, allocator in separate library, no threads:

Seconds	user	system	elapsed
Libc `malloc`	27.9	.07	29
Doug Lea	19.5	.05	20.2
GC (no free loop)	17.0	.21	18.1

Copying GC would do better still under these conditions.

GC Myth 5:
Reference counting is a cheap and low latency memory management technique.

The truth:
Simple reference count implementations can introduce substantial delays when large data structures are dropped and deallocated. This can be avoided by delayed deallocation, which adds space overhead.

Simple pointer assignments change from a register–to–register move to sequences that involve at least 4 potential memory references.

C++ "smart pointer" are typically more expensive than compiler/runtime supported techniques.

It does often save space *if it doesn't leak cycles*.

Reference count costs:

- noninvasive implementation
  (Barton & Nackman, uses extra allocation).
- original benchmark, 40 byte objects, default parameters for our GC, GC has no pointer info.
- most real benchmarks will probably involve less allocation --> reference counting is likely to do worse.

Seconds	user	system	elapsed
libc, explicit	9	0.05	9.5
fast, explicit	3.3	0.04	3.5
libc, rc	34	0.1	36
fast, rc	17.8	0.08	23
tracing GC	13	0.3	14
tracing GC, 2MB hp	6.4	0.2	7

GC Myth 6:

GC is much faster in languages like Java and Scheme that were designed for it than in C/C++.

The truth:

Conservative garbage collectors are more constrained, and cannot use some techniques that speed up a garbage collector, and good Scheme/ML implementations with a nonconservative collector are faster for some benchmarks with lots of very short–lived data, but:

- **It is hard to take advantage of that freedom in a consistently useful way.**

- **Scheme/ML nonconservative collectors are not consistently faster. (Some performance competitive implementations use a conservative GC.)**

- **I haven't yet found a Java collector that is performance competitive with our conservative C/C++ collector. (I don't expect that to last, and I haven't tried all implementations ...)**

Why is this area so prone to dubious folk−wisdoms?