**Why do garbage collection? Won't it be too expensive?**

One of the biggest sources of bookkeeping in systems programs is memory management. We feel it's critical to eliminate that programmer overhead, and advances in garbage collection technology in the last few years give us confidence that we can implement it with low enough overhead and no significant latency.

Another point is that a large part of the difficulty of concurrent and multi-threaded programming is memory management; as objects get passed among threads it becomes cumbersome to guarantee they become freed safely. Automatic garbage collection makes concurrent code far easier to write. Of course, implementing garbage collection in a concurrent environment is itself a challenge, but meeting it once rather than in every program helps everyone.

Finally, concurrency aside, garbage collection makes interfaces simpler because they don't need to specify how memory is managed across them.

The current implementation is a parallel mark-and-sweep collector but a future version might take a different approach.

On the topic of performance, keep in mind that Go gives the programmer considerable control over memory layout and allocation, much more than is typical in garbage-collected languages. A careful programmer can reduce the garbage collection overhead dramatically by using the language well; see the article about [profiling Go programs](https://blog.golang.org/2011/06/profiling-go-programs.html) for a worked example, including a demonstration of Go's profiling tools.

The GOGC variable sets the initial garbage collection target percentage. A collection is triggered when the ratio of freshly allocated data to live data remaining after the previous collection reaches this percentage. The default is GOGC=100. Setting GOGC=off disables the garbage collector entirely. The runtime/debug package's SetGCPercent function allows changing this percentage at run time.

# [Golang memory consumption management](http://stackoverflow.com/questions/14582471/golang-memory-consumption-management)

|  |  |  |
| --- | --- | --- |
|  | | I am new to Go and trying to figure out how it manages memory consumption.  I have troubles with memory in one of my test projects and it seems that I don't understand why Go uses more and more memory during long execution and doesn't frees it.  I am running test case provided bellow. After first allocation program uses near 350Mb of memory (According to ActivityMonitor). Than I try to free it and ActivityMonitor shows that memory consumption doubles. Why?  I am running this code on OSx, go1.0.3.  What is wrong with this code? And what is the right way to manage large variables in go programs.  P.S.: I ran into misunderstanding of memory management while implementing algorithm which requires a lot of time and memory, and after running it for "some time" it throws "out of memory" exception.  package main  import ("fmt"  "time"  )  func main() {  fmt.Println("getting memory")  tmp := make([]uint32, 100000000)  for kk, \_ := range tmp {  tmp[kk] = 0  }  time.Sleep(5 \* time.Second)  fmt.Println("returning memory")  tmp = make([]uint32, 1)  tmp = nil  time.Sleep(5 \* time.Second)  fmt.Println("getting memory")  tmp = make([]uint32, 100000000)  for kk, \_ := range tmp {  tmp[kk] = 0  }  time.Sleep(5 \* time.Second)  fmt.Println("returning memory")  tmp = make([]uint32, 1)  tmp = nil  time.Sleep(5 \* time.Second)  return  } |
|  | | |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | | 1 |  | | Here's a link to some good article about memory management in Go: [lwn.net/Articles/428100](http://lwn.net/Articles/428100/) –  [OlliP](http://stackoverflow.com/users/1537246/ollip) [Jan 30 '13 at 14:28](http://stackoverflow.com/questions/14582471/golang-memory-consumption-management#comment20393782_14582471) | | |  |  | | --- | --- | | 1 |  | | "I am running this code on OSx, go1.0.3." If you need to do something memory intensive using Go tip (what will become 1.1) is highly recommended. I was leery at first, but after a couple of the Go developers recommended it, it as been more stable than 1.0.3 for me, esp. in regards to memory use. –  [voidlogic](http://stackoverflow.com/users/1013460/voidlogic) [Feb 2 '13 at 6:21](http://stackoverflow.com/questions/14582471/golang-memory-consumption-management#comment20486674_14582471) | |
|  | Currently, go uses a [mark-and-sweep garbage collector](http://en.wikipedia.org/wiki/Mark-and-sweep#Na.C3.AFve_mark-and-sweep), which in general does not define when the object is thrown away.  However, if you look closely, there is a go routine called [runtime·MHeap\_Scavenger](http://code.google.com/p/go/source/browse/src/pkg/runtime/mheap.c#357) which essentially runs as long as your program does and calls the GC periodically:  // If we go two minutes without a garbage collection, force one to run.  forcegc = 2\*60\*1e9;  // If a span goes unused for 5 minutes after a garbage collection,  // we hand it back to the operating system.  limit = 5\*60\*1e9;  forcegc determines the period after which the gc is called by force. limit determines when spans are returned to the operating system. [Spans are a number of memory pages](http://code.google.com/p/go/source/browse/src/pkg/runtime/malloc.h#345) which can hold several objects. They're kept for limit time and are freed if no object is on them and limit is exceeded.  Further down in the code you can see that there is a trace option. You can use this to see, whenever the scavenger thinks he needs to clean up:  $ GOGCTRACE=1 go run gc.go  gc1(1): 0+0+0 ms 0 -> 0 MB 423 -> 350 (424-74) objects 0 handoff  gc2(1): 0+0+0 ms 1 -> 0 MB 2664 -> 1437 (2880-1443) objects 0 handoff  gc3(1): 0+0+0 ms 1 -> 0 MB 4117 -> 2213 (5712-3499) objects 0 handoff  gc4(1): 0+0+0 ms 2 -> 1 MB 3128 -> 2257 (6761-4504) objects 0 handoff  gc5(1): 0+0+0 ms 2 -> 0 MB 8892 -> 2531 (13734-11203) objects 0 handoff  gc6(1): 0+0+0 ms 1 -> 1 MB 8715 -> 2689 (20173-17484) objects 0 handoff  gc7(1): 0+0+0 ms 2 -> 1 MB 5231 -> 2406 (22878-20472) objects 0 handoff  gc1(1): 0+0+0 ms 0 -> 0 MB 172 -> 137 (173-36) objects 0 handoff  getting memory  gc2(1): 0+0+0 ms 381 -> 381 MB 203 -> 202 (248-46) objects 0 handoff  returning memory  getting memory  returning memory  As you can see, no gc invoke is done between getting and returning. However, if you change the delay from 5 seconds to 3 minutes (more than the 2 minutes from forcegc), the objects are removed by the gc:  returning memory  scvg0: inuse: 1, idle: 1, sys: 3, released: 0, consumed: 3 (MB)  scvg0: inuse: 381, idle: 0, sys: 382, released: 0, consumed: 382 (MB)  scvg1: inuse: 1, idle: 1, sys: 3, released: 0, consumed: 3 (MB)  scvg1: inuse: 381, idle: 0, sys: 382, released: 0, consumed: 382 (MB)  gc9(1): 1+0+0 ms 1 -> 1 MB 4485 -> 2562 (26531-23969) objects 0 handoff  gc10(1): 1+0+0 ms 1 -> 1 MB 2563 -> 2561 (26532-23971) objects 0 handoff  scvg2: GC forced // forcegc (2 minutes) exceeded  scvg2: inuse: 1, idle: 1, sys: 3, released: 0, consumed: 3 (MB)  gc3(1): 0+0+0 ms 381 -> 381 MB 206 -> 206 (252-46) objects 0 handoff  scvg2: GC forced  scvg2: inuse: 381, idle: 0, sys: 382, released: 0, consumed: 382 (MB)  getting memory  The memory is still not freed, but the GC marked the memory region as unused. Freeing will begin when the used span is unused and older than limit. From scavenger code:  if(s->unusedsince != 0 && (now - s->unusedsince) > limit) {  // ...  runtime·SysUnused((void\*)(s->start << PageShift), s->npages << PageShift);  }  This behavior may of course change over time, but I hope you now get a bit of a feel when objects are thrown away by force and when not.  As pointed out by zupa, releasing objects may not return the memory to the operating system, so on certain systems you may not see a change in memory usage. This seems to be the case for Plan 9 and Windows according to [this thread on golang-nuts](https://groups.google.com/forum/#%21topic/golang-nuts/vfmd6zaRQVs). | |
|  | |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  |  | | thank you for explanation, as I understood, GC is not very complete now –  [duganets](http://stackoverflow.com/users/111023/duganets) [Jan 30 '13 at 10:44](http://stackoverflow.com/questions/14582471/golang-memory-consumption-management#comment20386564_14586361) | | |  |  | | --- | --- | |  |  | | I have written program which uses a lot of memory, while running with go1.0.3 on OSX program did panic at the middle of the computations after next attempt to alloc more memory (near 1.5Gb). I have optimised heavily to reuse as much objects and structures as I can, but still had memory exception. Then I cloned last release of go and had build it from source. By running with latest version of go my code consumes as much memory as it needs (2.2Gb) and finishes computations successfully. –  [duganets](http://stackoverflow.com/users/111023/duganets) [Jan 30 '13 at 10:50](http://stackoverflow.com/questions/14582471/golang-memory-consumption-management#comment20386796_14586361) | | |  |  | | --- | --- | |  |  | | I think I ran into this kind of issue [code.google.com/p/go/issues/…](http://code.google.com/p/go/issues/detail?id=909&q=memory&colspec=ID%20Status%20Stars%20Priority%20Owner%20Reporter%20Summary) –  [duganets](http://stackoverflow.com/users/111023/duganets) [Jan 30 '13 at 11:19](http://stackoverflow.com/questions/14582471/golang-memory-consumption-management#comment20387635_14586361) | | |  |  | | --- | --- | |  |  | | Glad it worked with tip. The GC seems to be quite incomplete, yes. If you're interested, you can look at the [scanblock function](http://code.google.com/p/go/source/browse/src/pkg/runtime/mgc0.c#399) of the GC to see, how it finds references. –  [nemo](http://stackoverflow.com/users/1643939/nemo) [Jan 30 '13 at 12:41](http://stackoverflow.com/questions/14582471/golang-memory-consumption-management#comment20390178_14586361) | | |  |  | | --- | --- | | 3 |  | | On Windows as of Go 1.1, garbage collected memory is NOT returned to the OS. Man that cost me a day to hunt down. [groups.google.com/forum/#!topic/golang-nuts/vfmd6zaRQVs](https://groups.google.com/forum/#%21topic/golang-nuts/vfmd6zaRQVs) –  [zupa](http://stackoverflow.com/users/926988/zupa) [Oct 1 '13 at 22:45](http://stackoverflow.com/questions/14582471/golang-memory-consumption-management#comment28286039_14586361) | | |

|  |  |
| --- | --- |
|  | To eventually (force) collect unused memory you must call [runtime.GC()](http://golang.org/pkg/runtime/#GC).  variable = nil may make things unreachable and thus eligible for collection, but it per se doesn't free anything |