# Malloc(3) revisited

Poul-Henning Kamp [1]
*The FreeBSD Project*

## ABSTRACT

*malloc* (3) is one of the oldest parts of the C language environment and not surprisingly the world has changed a bit since it was first conceived. The fact that most UNIX kernels have changed from swap/segment to virtual memory/page based memory management has not been sufficiently reflected in the implementations of the malloc/free API.

A new implementation was designed, written, tested and bench-marked with an eye on the workings and performance characteristics of modern Virtual Memory systems. It works OK.

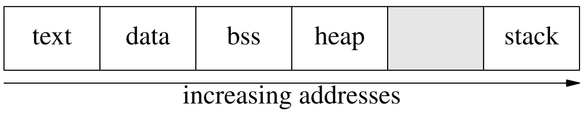
### Introduction

All but the most trivial programs need to allocate storage dynamically in addition to whatever static storage the compiler reserved at compile-time. Programming languages generally come in three flavours on this point: those which handle&hide it for the programmer, those which don't allow for it and the *C* programming language. As with so many other things the *C* environment hands the programmer all the raw bits to play with, and does very little to prevent the programmer from making mistakes.

A modern UNIX kernel provides three means for dynamic memory allocation: the execution stack and the heap, and *mmap* (2).

### The Stack

The stack is usually put at the far upper end of the address-space, from where it grows down [2] as far as needed.




There is no real kernel interface to the stack as such. The kernel will allocate some amount of memory for the stack, usually not even telling the process the exact amount. The process will simply try to access whatever it needs, expecting the kernel to detect the access outside the allocated memory and treat this as a request for extension. If the kernel fails to extend the stack, either because of lack of resources, lack of permissions or because it may just be plain impossible to do in the first place, the process will usually be shot down by the kernel with a terminal signal.

In the C language, there exists a little used interface to the stack, *alloca* (3), which will explicitly allocate space on the stack. This is not a interface to the kernel, but an adjustment done to the stack pointer such that space will be available and unharmed by any subroutine calls yet to be made while the context of the current subroutine is intact. As a consequence of this design, there is no need for an actual "free" operator. The space is returned auto-magically when the current function returns and the stack frame is dismantled. This asymmetry is the cause of much grief, and probably the single most important reason that alloca(3) is not, and should not be, widely used.

### *mmap* (2).

When hardware architectures which provided paging became available, a new API was added which gives the programmer detailed control over the individual pages in the process [3]. The API has two primary functions *mmap* (2) and *munmap* (2) as well as some auxiliary functions. Unfortunately, most programs do not allocate memory in page-sized chunks, so this interface is usually only used in specialised and system applications. One typical and probably the most widespread use in terms of number of calls to this API is shared libraries.

### The heap

The heap is an extension of the data segment of the process, it starts at the end of the *bss* section and extends upwards. The storage in the heap area is explicitly allocated with the system call *brk* (2). which takes one argument: a pointer to where the process wants the heap to end. The *libc* library also provides a function layered on top of *brk* (2) called *sbrk* (2) which takes as argument a (signed) increment to the current end of the heap.

---

[1] This work was not sponsored by anybody. Poul-Henning Kamp was supported by his own daytime job. He would have loved to do this for some sponsors money instead.

[2] A few mostly obsolete CPU designs can be considered antipodic in this respect.

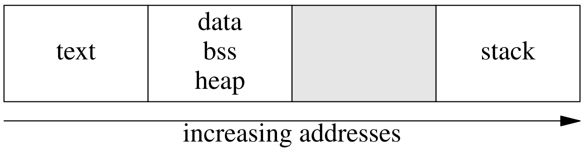
[3] To the extent the kernel implements this API that is. Not all kernels implement more than the bare minimum.

## The kernel and memory

*Brk* (2) is a very inconvenient interface, for most day-to-day uses it is completely impossible to use it. It is easy to allocate memory with it, but you can only free it again in a LIFO order. As so many other things in UNIX, it was probably defined based on what the kernel had to offer rather than a theoretical study of what programmers needed.

Before paged and/or virtual memory systems became common, the memory management facility used for UNIX was segments. This was also very often the only available vehicle for imposing protection on various parts of memory. Depending on the hardware, segments can be anything, and consequently how the kernels exploited them varied a lot from UNIX to UNIX and from machine to machine.

Typically a process would have one segment for the text section, one for the data and bss section combined and one for the stack. [4]




In this setup all the *brk* (2) system call needs to do is to find the right amount of free storage, possibly moving things around in physical memory, maybe even swapping out a segment or two to make space, and change the upper limit on the data segment according to the address given.

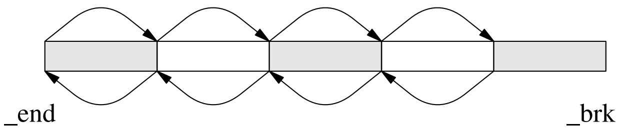
In a more modern page based virtual memory implementation this is still pretty much the situation, except that the granularity is now pages. The kernel finds the right number of free pages, possibly paging some pages out to free them up, and then plugs them into the page-table of the process.

Only very few programs deal with the *brk* (2) interface directly. The few that do usually have their own memory management facilities. LISP, MODULA-3 or FORTH interpreters and runtimes are good examples. Most other programs use the *malloc* (3) interface instead, and leave it to the malloc implementation to use *brk* (2) to get storage allocated from the kernel.

[4] On some systems the text shared a segment with the data and bss, and was consequently just as writable as them. Some people will remember the undocumented way of compiling *awk* (1) programs: given the right option *awk* (1) would load and parse the program and then write the address space from the start of the text to the top of the heap into a file. Another option read this file back in. This reduced the startup time because the program was already parsed into internal form. The initial version of this hack didn't work on machines where the text segment could not be written to. TeX and GNU-emacs are other programs which have used similar methods for similar reasons.

## Malloc(3), realloc(3) and free(3)

The archetypical *malloc* (3) implementation keeps track of the memory between the end of the bss section, as defined by the **_end** symbol, and the current *brk* (2) point using a linked list of chunks of memory. Each item on the list has a status as either free or used, a pointer to the next entry and in most cases to the previous as well, to speed up inserts and deletes in the list.




When a *malloc* (3) request comes in, the list is traversed from the front and if a free chunk big enough to hold the request is found, it is returned. If the free chunk is bigger than the size requested, a new free chunk is made from the excess and put back on the list. When a chunk is *free* (3)'ed, the chunk is found in the list, its status is changed to free and if one or both of the surrounding chunks are free, they are collapsed to one.

A third kind of request, *realloc* (3), will resize a chunk, trying to avoid copying the contents if possible. It is seldom used, and has only had a significant impact on performance in a few special situations. The typical pattern of use is to *malloc* (3) a chunk of the maximum size needed, read in the data and adjust the size of the chunk to match the size of the data read using *realloc* (3), or alternatively, to allocate with *malloc* (3) a chunk which can handle a large fraction of the requests, and if this proves insufficient, reallocating with *realloc* (3), possibly several times, until the necessary amount of memory has been obtained.

For reasons of efficiency, the original implementation of *malloc* (3) put the small data structure used to contain the next and previous pointers plus the state of the chunk right before the chunk itself. As a matter of fact, the canonical *malloc* (3) implementation can be studied in the ''Old testament'', chapter 8 verse 7 [5]

Various optimisations can be applied to the above basic algorithm:

- If when freeing a chunk, we end up with the last chunk on the list being free, we can return that to the kernel by calling *brk* (2)with address of that chunk and then make the previous chunk the last on the chain by terminating its ''next'' pointer.
- A best-fit algorithm can be used instead of first-fit at an expense of memory, because statistically fewer chances to *brk* (2)backwards
- Splitting the list in two, one for used and one for free chunks, to speed the searching.
- Putting free chunks on one of several free lists,

[5] Kernighan & Ritchie: The C programming language

depending on their size, to speed allocation.

- &c &c &c

### The problems

Even though *malloc* (3) is a lot simpler to use than the raw *brk* (2) interface, or maybe exactly because of that, a lot of problems arise from its use.

- Writing to memory outside the allocated chunk.
- Freeing a pointer to memory not allocated by malloc.
- Freeing a modified pointer.
- Freeing the same pointer more than once.
- Accessing memory in a chunk after it has been *free* (3)'ed.

The handling of these problems have traditionally been weak. A core-dump was the most common form for ‘‘handling’’, but in rare cases one could experience the famous ‘‘malloc: corrupt arena.’’, or similarly informative messages right before the core dump. Much worse though, very often the program will just continue, quite possibly giving wrong results or weird behaviour.

An entirely different kind of problem is normal sloppy thinking: The manual pages clearly state the memory returned by *malloc* (3) can contain any value, and that one should explicitly initialise the memory before use. Unfortunately most kernels, correctly so, zero out the storage they provide with *brk* (2) for security reasons, and thus the storage *malloc* (3) return happen to be zeroed in many cases as well, so programmers are not particular apt to notice that their code depends on malloc'ed storage being zero.

*Malloc* (3) has somewhat deserved the reputation it has gotten for being the first of ‘‘the usual suspects’’ to round up when programs act weird.

### Alternative implementations

Detecting some or all of these problems was the inspiration for the first alternative malloc implementations. Since their main aim was debugging, they would often use techniques like allocating a guard zone before and after the chunk, usually filling these guard zones with some known predictable pattern[6], so that write accesses outside the allocated chunk could be detected as changes to these patterns with some decent probability. Another widely used technique is to use tables to keep track of which chunks are actually in which state and so on.

This class of debugging has been taken to its practical extreme by the product ‘‘Purify’’ which does the entire memory-colouring exercise and not only keeps track of what is and what isn't in use, but also detects if the first reference is a read (which would return undefined values) and other such violations. Purify is a commercial product of high quality and priced to reflect this.

Later actual complete alternative implementations of malloc arrived, but many of these as well as the code which sat comfortably in the libc library of FreeBSD, still based their workings on the basic schema mentioned previously, oblivious to the fact that in the meantime virtual memory and paging have become the standard environment rather than segments.

The most widely used ‘‘alternative’’ malloc is undoubtedly ‘‘gnumalloc’’ which has received wide acclaim and certainly runs faster than most stock mallocs. It does, however, just like most other malloc implementations, have a tendency to fare badly in cases where paging is the norm rather than the exception.

The particular malloc that prompted this work basically didn't bother reusing storage until the kernel forced it to do so by refusing further allocations with *sbrk* (2). That may make sense if you work alone on your own personal mainframe, but as a general policy it is much less than optimal.

In order to select a candidate amongst the various available free implementations of malloc, I tried to benchmark them from end to other. This was done on a tiny laptop with only 8MB of RAM [7], and it soon transpired that as soon as RAM was over-committed things went downhill very fast. This prompted me to study what ‘‘performance’’ meant for a malloc implementation.

### Performance

Performance for a *malloc* (3) has two sides:

A) How much time does it use for searching and manipulating data structures. We will refer to this as ‘‘overhead’’.

B) How well does it manage the storage. This rather vague metric we call ‘‘quality of allocation’’.

The overhead is easy to measure: Just do a lot of malloc/free calls of various kinds and combination, and compare the results. This is unfortunately the most common basis for systematic comparison of malloc implementations. I say ‘‘unfortunately’’ because it should be obvious to anybody that if you can save just one disk access, you can do almost anything you like to your internal data structures for several milliseconds and still come out being faster in the end. To compound this oversight, most people who have compared malloc implementations have done so on systems where RAM was not over-committed, and consequently the implementations abilities in this area have

[6] Amongst the many creative patterns are 0xDEADBEEF, 0xCOFEBABE, 0xDEADDEAD and so on.

[7] A ‘‘GateWay 2000 Handbook’’, too bad they don't make them anymore.

not been measured.

The "quality of allocation" metric tries to measure this aspect. It is actually horribly complex to measure. In fact, the only manageable way to measure it is to run some complex and deterministic test cases on a system where RAM is over-committed, measure the time it took and use that as the metric.

To design an algorithm on the other hand, an analytical attack is needed. Here is the one I used in the design of my malloc implementation:

One indicator of this quality is the size of the process, that should obviously be minimised. Another indicator is the execution time of the process. This is not an obvious indicator of quality for malloc, but people will generally agree that it should be minimised as well, and if *malloc* (3), as we will see shortly, can do anything to do so, it should.

In a traditional segment/swap kernel, because the entire process will either be swapped out to disk or be resident in RAM, the desirable behaviour of a process is to keep the *brk* (2) point as low as possible, thus minimising the size of the data/bss/heap segment, which in turn translates to a smaller process and a smaller probability of the process being swapped out. QED: faster execution time as an average.

In a paging environment this is not a bad choice for a default, but a couple of details needs to be looked at much more carefully. First of all, the size of a process becomes a more vague concept since only the pages that are actually used need to be in primary storage for execution to progress, and they only need to be there when used. That implies that many more processes can fit in the same amount of primary storage, since most processes have a high degree of locality of reference and thus only need some fraction of their pages to actually do their job. From this it follows that the interesting size of the process is a subset of the total amount of virtual memory occupied by the process. This subset isn't a constant. It varies depending on the whereabouts of the process, and it may indeed fluctuate wildly over the lifetime of the process.

One of the names for this vague concept is ‘‘current working set’’. This is a most horribly ill-defined number, but for now we can simply say that it is the number of pages the process needs in order to run at a acceptable low paging rate in a congested primary storage. If the number of pages is too small, the process will wait for its pages to be read from secondary storage much of the time. If it's too big, the space could be used better for something else. If primary storage isn't congested, this may not seem important. But many kernels today can use any available pages for disk-cache or similar functions, so from that perspective main storage is always congested.

From the view of any single process, this number of pages is of course ‘‘all of my pages’’, since this guarantees that no pages will need to be paged in or out. From the point of view of the OS it should be tuned to maximise the total throughput of all the processes running on the machine at that time. This is usually done using various kinds of least-recently-used replacement algorithms to select page candidates for replacement.

With this miniature analysis, we can define the performance goals for a modern *malloc* (3) implementation as: **Minimise the number of pages accessed.**

This really is the core of it all. If the number of accessed pages is smaller, then locality of reference is higher, and all kinds of caches (which is essentially what the primary storage in a VM system is) work better.

It's interesting to notice that the classical malloc, and most of the alternatives available, fail decisively according to this criteria. The information about free chunks is kept in the free chunks themselves. In other words, even though the application as such do not need these chunks of memory, the malloc implementation still does, and consequently those pages if paged out, will not stay there longer than till the next call to *malloc* (3) or *free* (3) needs to traverse the free-list.

In some of the benchmarks this came out as all the pages being paged in every time a malloc call was made. This made as much difference as a factor of five in wall-clock time for certain scenarios.

The secondary goal is more evident: **Try to work in pages.** That makes it easier for the kernel, and wastes less virtual memory. Most modern implementations do this when they interact with the kernel, but only a few try to avoid objects spanning pages.

If an object's size is less than or equal to a page, there is no reason for it to span two pages. Having objects span pages means that two pages must be paged in, if that object is accessed.

## Implementation

The implementation is 1136 lines of C code, and can be found in FreeBSD 2.2 and later versions of FreeBSD as src/lib/libc/stdlib/malloc.c.

The main data structure is the *page-directory* which contains a **void*** for each page we have control over. The value can be one of:

- **MALLOC_NOT_MINE** Another part of the code may call *brk* (2) to get a piece of the cake. Consequently, we cannot rely on the memory we get from the kernel being one consecutive piece of memory, and therefore we need a way to mark such pages as "untouchable".
- **MALLOC_FREE** This is a free page.
- **MALLOC_FIRST** This is the first page in a (multi-)page allocation.
- **MALLOC_FOLLOW** This is a subsequent page in a multi-page allocation.
- **struct pginfo*** A pointer to a structure describing a

partitioned page.

In addition, there exists a linked list of small data structures that describe the free space as runs of free pages.

Notice that these structures are not part of the free pages themselves, but rather allocated with malloc so that the free pages themselves are never referenced while they are free.

When a request for storage comes in, it will be treated as a ‘‘page’’ allocation if it is bigger than half a page. The free list will be searched and the first run of free pages that can satisfy the request is used. The first page gets set to MALLOC_FIRST status. If more than that one page is needed, the rest of them get MALLOC_FOLLOW status in the page-directory.

If there were no pages on the free list, *brk* (2) will be called, and the pages will get added to the page-directory with status MALLOC_FREE and the search restarts.

Freeing an allocation of pages is done by changing their state in the page directory to MALLOC_FREE, traversing the free-pages list to find the right place for this run of pages, collapsing with either or both of the two neighbouring entries if possible, and if above the threshold: releasing some pages back to the kernel by calling *brk* (2).

If the request is less than or equal to half of a page, its size will be rounded up to the nearest power of two before being processed and if the request is less than some minimum size, it is rounded up to that size.

These sub-page allocations are served from pages which are split up into some number of equal size chunks. For each of these pages a **struct pginfo** describes the size of the chunks on this page, how many there are, how many are free and so on. The description consist of a bitmap of used chunks, and various counters and numbers used to keep track of the stuff in the page.

For each size of sub-page allocation, the pginfo structures for the pages that have free chunks in them form a list. The heads of these lists are stored in predetermined slots at the beginning of the page directory to make access fast.

To allocate a chunk of some size, the head of the list for the corresponding size is examined, and a free chunk found. The number of free chunks on that page is decreased by one and, if zero, the pginfo structure is unlinked from the list.

To free a chunk, the page is derived from the pointer, the pginfo info structure found from the page directory and the bit corresponding to this chunk is set in the bitmap, and the counter for free chunks is increased by one. If this page has exactly one free chunk now, it is linked back into the list for chunks of this size, if all chunks are free both the page and the pginfo structure are *free* (3)’ed too.

To be 100% correct performance-wise these lists should be ordered according to the recent number of accesses to that page. This information is not available and it would essentially mean a reordering of the list on every memory reference to keep it up-to-date. Instead they are ordered according to the address of the pages. Other criteria has been tried and it looks like any kind of stable and repeatable sorting of these result in the same performance. Sorting by address statistically keeps. *brk* (2) as lower.

It is an interesting twist to the implementation that the **struct pginfo** Is allocated with malloc. That is, "as with malloc" to be painfully correct. The code knows the special case where the first (couple) of allocations on the page is actually the pginfo structure and deals with it accordingly. This avoids some silly "chicken and egg" issues.

## Bells and whistles.

*brk* (2) is actually not a very fast system call when you ask for storage. This is mainly because of the need for the kernel to zero the pages before handing them over. Therefore this implementation does not release heap pages until there is a large chunk to release back to the kernel. Chances are pretty good that we will need it again pretty soon anyway. Since these pages are not accessed at all, they will soon be paged out and don’t affect anything but swap-space usage.

The page directory is actually kept in a *mmap* (2)’ed piece of anonymous memory. This avoids some rather silly cases that would otherwise have to be handled when the page directory has to be extended.

One particularly nice feature is that all pointers passed to *free* (3) and *realloc* (3) can be checked conclusively for validity. First the pointer is masked to find the page. The page directory is then examined, it must contain either MALLOC_FIRST, in which case the pointer must point exactly at the page, or it can contain a struct pginfo*, in which case the pointer must point to one of the chunks described by that structure. Warnings will be printed on **stderr** and nothing will be done with the pointer if it is found to be invalid.

An environment variable **MALLOC_OPTIONS** allows the user some control over the behaviour of malloc. Some of the more interesting options are:

**Abort** If malloc fails to allocate storage, core-dump the process with a message rather than expect it to handle this correctly.

**Hint** Pass a hint to the kernel about pages we no longer need using the *madvise* (2) system call. This allows the kernel to discard the contents of the page and reuse it as free. If this process accesses that page later on, the kernel can just map a new page into the address space. This can improve performance a fair bit in certain applications since it has the potential to save a page-out and a page-in operation.

**Realloc** Always do a free and malloc when *realloc* (3) is called. For programs doing garbage collection using *realloc* (3), this make the heap collapse faster since malloc will reallocate from the lowest available address. The default is to leave things alone if the size of the allocation is still in the same size-bracket.

**Junk** will explicitly fill the allocated area with a particular value to try to detect if programs rely on it being zero. The value used, 0xd0, is selected to maximize the probability of a coredump.

**Zero** will explicitly zero out the allocated chunk of memory, while any space after the allocation in the chunk will be filled with the junk value to try to detect out of the chunk references.

**sys-V** quite to my surprise there were one bit of the API which were not well agreed upon. What should *realloc* (3) return when given a pointer and a new size of zero ? Well, some people expect it to return a NULL pointer, which makes sense, and some people expect it to return a valid pointer, which also makes sense. This option lets the programmer choose.

All these and a few other options can also be set in a system-wide fashion, or at compile time. They have proved very popular with developers, and users alike, and in particular the 'H' option can have a decisive performance impact.

### Future improvements

It is not obvious that having the free-page list is an actual benefit, it may be equally fast to just search for free pages in the page directory.

Truly transient programs like *echo* (1), *date* (1) and similar shouldn't bother with malloc/free, they should simply use *sbrk* (2) for their needs. Maybe a grace period should be implemented in *malloc* (3) so serious memory management would only start after a certain number of chunks or bytes have actually been freed back.

Universally huge improvements in performance in the future seems unlikely unless the *malloc* (3) API is changed significantly. But doing so is by no means a guarantee of better performance. The main stumbling block is that it is not possible for the *malloc* (3) implementation to relocate in-use memory to improve locality of reference.

This is not the same as to say that a few programs out there could not use a better and more intelligent memory allocation policy.

### Conclusion and experience.

In general the performance differences between gnumalloc and this malloc are not that big. The major difference comes when primary storage is seriously over-committed, and gnumalloc wastes time paging in pages it's not going to really use, in such cases as much as a factor of five in time has been observed for various programs.

Several legacy programs in the BSD 4.4 Lite distribution had code that depended on the memory returned from malloc being zeroed. In a couple of cases, free(3) was called more than once for the same allocation, and a few cases even called free(3) with pointers to objects in the data section or on the stack.

A couple of users have reported that using this malloc on other platforms yielded "pretty impressive results", but no hard benchmarks have been made.

### Acknowledgements & references.

The first implementation of this algorithm was actually a file system, done in assembler using 5-hole ‘‘Baudot’’ paper tape for a drum storage device attached to a 20 bit germanium transistor computer with 2000 words of memory, but that was many years ago.

A lot of people have provided ideas, bug-fixes and portability changes to the code. Special thanks and mention goes to: Peter Wemm, Lars Fredriksen, Keith Bostic, Dmitrij Tejblum, John-Mark Gurney, Joel Maslak, John Birrell, Warner Losh, Kaleb Keithly, Mike Pritchard, John D. Polstra and Archie Cobbs.