Malloc(3) in modern Virtual Memory environments.

Revised Fri Apr 5 12:50:07 1996

Poul-Henning Kamp

<phk@FreeBSD.org>
Den Andensidste Viking
Valbygaardsvej 8
DK-4200 Slagelse
Denmark

ABSTRACT

Malloc/free is one of the oldest parts of the C language environment and obviously the world has changed a bit since it was first made. 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.

1. Introduction

Most programs need to allocate storage dynamically in addition to whatever static storage the compiler reserved at compile-time. To C programmers this fact is rather obvious, but for many years this was not an accepted and recognized fact, and many languages still used today don't support this notion adequately.

The classic UNIX kernel provides two very simple and powerful mechanisms for obtaining dynamic storage, the execution stack and the heap. The stack is usually put at the far upper end of the address-space, from where it grows down as far as needed, though this may depend on the CPU design. The heap starts at the end of the **bss** segment and grows upwards as needed.

There isn't really a kernel-interface to the stack as such. The kernel will allocate some amount of memory for it, not even telling the process the exact size. If the process needs more space than that, it will simply try to access it, hoping that the kernel will detect that access have been attempted outside the allocated memory, and try to extend it. If the kernel fails to extend the stack, this could be because of lack of resources or permissions or because it may just be impossible to do in the first place, the process will usually be shot down by the kernel.

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 merely 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.

Due to the nature of normal use of the stack, there is no corresponding "free" operator, but instead the space is returned when the current function returns to its caller and the stack frame is dismantled. This is the cause of much grief, and probably the single most important reason that alloca(3) is not, and should not be, used widely.

The heap on the other hand has an explicit kernel-interface in the system call **brk(2)**. The argument to brk(2) is a pointer to where the process wants the heap to end. There is also a interface called **sbrk(2)** taking an increment to the current end of the heap, but this is merely a **libc** front for brk(2).

In addition to these two memory resources, modern virtual memory kernels provide the mmap(2)/mmunmap(2) interface which allows almost complete control over any bit of virtual memory in the process address space.

Because of the generality of the mmap(2) interface and the way the data structures representing the regions are laid out, sbrk(2) is actually faster in use than the equivalent mmap(2) call, simply because mmap(2) has to search for information that is implicit in the sbrk(2) call.

2. The kernel and memory

Brk(2) isn't a particularly convenient interface, it was probably made more to fit the memory model of the hardware being used, than to fill the needs of the programmers.

Before paged and/or virtual memory systems became common, the most popular memory management facility used for UNIX was segments. This was also very often the only 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. On some systems the text shared a segment with the data and bss, and was consequently just as writable as them.

In this setup all the brk(2) system call has 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.

As such the difference is very small, the real difference is that in the old world of swapping, either the entire process was in primary storage or it wouldn't be selected to be run. In a modern VM kernel, a process might only have a subset of its pages in primary memory, the rest will be paged in, if and when the process tries to access them.

Only very few programs deal with the brk(2) interface directly. The few that do usually have their own memory management facilities. LISP or FORTH interpreters 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.

3. Malloc and free

The job of malloc(3) is to turn the rather simple brk(2) facility into a service programs can actually use without getting hurt.

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).

For reasons of efficiency, the original implementation of malloc(3) put the small 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 [Kernighan & Ritchie]

Various optimisations can be applied to the above basic algorithm:

If in 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 the first 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 will present themselves.

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, depending on their size, to speed allocation.

...

4. The problems

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

Writing to memory outside the allocated chunk. The most likely result being that the data structure used to hold the links and flags about this chunk or the next one gets thrashed.

Freeing a pointer to memory not allocated by malloc. This is often a pointer that points to an object on the stack or in the data-section, in newer implementations of C it may even be in the text- section where it is likely to be readonly. Some malloc implementations detect this, some don’t.

Freeing a modified pointer. This is a very common mistake, freeing not the pointer malloc(3) returned, but rather some offset from it. Some mallocs will handle this correctly if the offset is positive.

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." message before the core-dump. Even worse though, very often the program will just continue, possibly giving wrong results.

An entirely different form of problem is that the memory returned by malloc(3) can contain any value. Unfortunately most kernels, correctly, zero out the storage they provide with brk(2), and thus the storage malloc returns will be zeroed in many cases as well, so programmers are not particular apt to notice that their code depends on malloc’ed storage being zeroed.

With problems this big and error handling this weak, it is not surprising that problems are hard and time consuming to find and fix.

5. Alternative implementations

These problems were actually 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, and possibly filling these guard zones with some pattern, so accesses outside the allocated chunk could be detected 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 in use and what isn’t, but also detects if the first reference is a read (which would return undefined values) and other such violations.

Later actual complete implementations of malloc arrived, but many of these still based their workings on the basic schema mentioned previously, disregarding that in the meantime virtual memory and paging have become the standard environment.

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, tend 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 less than optimal.

6. Performance

Performance for a malloc(3) implementation comes as two variables:

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

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

The overhead time is easy to measure, just to a lot of malloc/free calls of various kinds and combination, and compare the results.

The quality of allocation is not quite as simple as that. One measure of quality is the size of the process, that should obviously be minimized. Another measure is the execution time of the process. This is not an obvious indicator of quality, but people will generally agree that it should be minimized as well, and if malloc(3) can do anything to do so, it should. Explanation why it is still a good metric follows:

In a traditional segment/swap kernel, the desirable behaviour of a process is to keep the brk(2) as low as possible, thus minimizing 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 some subset of the total amount of virtual memory occupied by the process. This number 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’’. It has been defined many different ways over the years, mostly to satisfy and support claims in marketing or benchmark contexts.

For now we can simply say that it is the number of pages the process needs in order to run at a sufficiently low paging rate in a congested primary storage. (If primary storage isn’t congested, this is not really important of course, but most systems would be better off using the pages for disk-cache or similar functions, so from that perspective it will always be congested.) 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.

From the view of any single process, this number of pages is "all of my pages", but from the point of view of the OS it should be tuned to maximise the total throughput of all the processes on the machine at the time. This is usually done using various kinds of least-recently-used replacement algorithms to select page candidates for replacement.

With this knowledge, can we decide what the performance goal is for a modern malloc(3) ? Well, it’s almost as simple as it used to be: **Minimize 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 is in a VM system) work better.

It’s interesting to notice that the classical malloc fails on this one because the information about free chunks is kept with the free chunks themselves. In some of the benchmarks this came out as all the pages

being paged in every time a malloc call was made, because malloc had to traverse the free list to find a suitable chunk for the allocation. If memory is not in use, then you shouldn't access it.

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 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.

With this analysis in the luggage, we can start coding.

7. Implementation

A new malloc(3) implementation was written to meet the goals, and to the extent possible to address the shortcomings listed previously.

The source is 1218 lines of C code, and can be found in FreeBSD 2.2 (and probably later versions as well) 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 a number of pages is done by changing their state in the page directory to MALLOC_FREE, and then traversing the free-pages list to find the right place for this run of pages, possibly collapsing with the two neighbouring runs into one run and, if possible, releasing some memory 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 page table for that page contains a pointer to the pginfo structure, where the free bit is set for the chunk, the number of free chunks increased by one, and if equal to one, the pginfo structure is linked into the proper place on the list for this size of chunks. If the count increases to match the number of chunks on the page, the pginfo structure is unlinked from the list and free(3)'ed and the actual page itself is 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. Interestingly enough, in practice this comes out to almost the same thing performance wise.

It's not that surprising after all, it's the difference between following the crowd or actively directing where it can go, in both ways you can end up in the middle of it all.

The side effect of this compromise is that it also uses less storage, and the list never has to be reordered, all the ordering happens when pages are added or deleted.

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.

8. 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 by the kernel to zero the pages before handing them over, so 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 handle this correctly. It's amazing how few programs actually handle this condition correctly, and consequently the havoc they can create is the more creative or destructive.

Dump Writes malloc statistics to a file called ''malloc.out'' prior to process termination.

Hint Pass a hint to the kernel about pages we no longer need through the madvise(2) system call. This can help performance on machines that page heavily by eliminating unnecessary page-ins and page-outs of unused data.

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-class.

Junk will explicitly fill the allocated area with a particular value to try to detect if programs rely on it being zero.

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 catch out of the chunk references.

9. The road not yet taken.

A couple of avenues were explored that could be interesting in some set of circumstances.

Using mmap(2) instead of brk(2) was actually slower, since brk(2) knows a lot of the things that mmap has to find out first.

In general there is little room for further improvement of the time-overhead of the malloc, further improvements will have to be in the area of improving paging behaviour.

It is still under consideration to add a feature such that if realloc is called with two zero arguments, the internal allocations will be reallocated to perform a garbage collect. This could be used in certain types of programs to collapse the memory use, but so far it doesn't seem to be worth the effort.

Malloc/Free can be a significant point of contention in multi-threaded programs. Low-grain locking of the data-structures inside the implementation should be implemented to avoid excessive spin-waiting.

10. 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, in which case gnumalloc wastes time paging in pages it's not going to use. In such cases as much as a factor of five in wall-clock time has been seen in difference. Apart from that gnumalloc and this implementation are pretty much head-on performance wise.

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.

11. 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.

Peter Wemm <peter@FreeBSD.org> came up with the idea to store the page-directory in mmap(2)'ed memory instead of in the heap. This has proven to be a good move.

Lars Fredriksen <fredriks@mcs.com> found and identified a fence-post bug in the code.