Reprinted from: *Proceedings of the San Francisco USENIX Conference*, pp. 295-303, June 1988.

Design of a General Purpose Memory Allocator for the 4.3BSD UNIX† Kernel

Marshall Kirk McKusick

Michael J. Karels

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

The 4.3BSD UNIX kernel uses many memory allocation mechanisms, each designed for the particular needs of the utilizing subsystem. This paper describes a general purpose dynamic memory allocator that can be used by all of the kernel subsystems. The design of this allocator takes advantage of known memory usage patterns in the UNIX kernel and a hybrid strategy that is time-efficient for small allocations and space-efficient for large allocations. This allocator replaces the multiple memory allocation interfaces with a single easy-to-program interface, results in more efficient use of global memory by eliminating partitioned and specialized memory pools, and is quick enough that no performance loss is observed relative to the current implementations. The paper concludes with a discussion of our experience in using the new memory allocator, and directions for future work.

1. Kernel Memory Allocation in 4.3BSD

The 4.3BSD kernel has at least ten different memory allocators. Some of them handle large blocks, some of them handle small chained data structures, and others include information to describe I/O operations. Often the allocations are for small pieces of memory that are only needed for the duration of a single system call. In a user process such short-term memory would be allocated on the run-time stack. Because the kernel has a limited run-time stack, it is not feasible to allocate even moderate blocks of memory on it. Consequently, such memory must be allocated through a more dynamic mechanism. For example, when the system must translate a pathname, it must allocate a one kilobye buffer to hold the name. Other blocks of memory must be more persistent than a single system call and really have to be allocated from dynamic memory. Examples include protocol control blocks that remain throughout the duration of the network connection.

Demands for dynamic memory allocation in the kernel have increased as more services have been added. Each time a new type of memory allocation has been required, a specialized memory allocation scheme has been written to handle it. Often the new memory allocation scheme has been built on top of an older allocator. For example, the block device subsystem provides a crude form of memory allocation through the allocation of empty buffers [Thompson78]. The allocation is slow because of the implied semantics of finding the oldest buffer, pushing its contents to disk if they are dirty, and moving physical memory into or out of the buffer to create the requested size. To reduce the overhead, a ''new'' memory allocator was built in 4.3BSD for name translation that allocates a pool of empty buffers. It keeps them on a free list so they can be quickly allocated and freed [McKusick85].

†UNIX is a registered trademark of AT&T in the US and other countries.

This memory allocation method has several drawbacks. First, the new allocator can only handle a limited range of sizes. Second, it depletes the buffer pool, as it steals memory intended to buffer disk blocks to other purposes. Finally, it creates yet another interface of which the programmer must be aware.

A generalized memory allocator is needed to reduce the complexity of writing code inside the kernel. Rather than providing many semi-specialized ways of allocating memory, the kernel should provide a single general purpose allocator. With only a single interface, programmers do not need to figure out the most appropriate way to allocate memory. If a good general purpose allocator is available, it helps avoid the syndrome of creating yet another special purpose allocator.

To ease the task of understanding how to use it, the memory allocator should have an interface similar to the interface of the well-known memory allocator provided for applications programmers through the C library routines *malloc*() and *free*(). Like the C library interface, the allocation routine should take a parameter specifying the size of memory that is needed. The range of sizes for memory requests should not be constrained. The free routine should take a pointer to the storage being freed, and should not require additional information such as the size of the piece of memory being freed.

2. Criteria for a Kernel Memory Allocator

The design specification for a kernel memory allocator is similar to, but not identical to, the design criteria for a user level memory allocator. The first criterion for a memory allocator is that it make good use of the physical memory. Good use of memory is measured by the amount of memory needed to hold a set of allocations at any point in time. Percentage utilization is expressed as:

$$utilization = \frac{requested}{required}$$

Here, "requested" is the sum of the memory that has been requested and not yet freed. "Required" is the amount of memory that has been allocated for the pool from which the requests are filled. An allocator requires more memory than requested because of fragmentation and a need to have a ready supply of free memory for future requests. A perfect memory allocator would have a utilization of 100%. In practice, having a 50% utilization is considered good [Korn85].

Good memory utilization in the kernel is more important than in user processes. Because user processes run in virtual memory, unused parts of their address space can be paged out. Thus pages in the process address space that are part of the "required" pool that are not being "requested" need not tie up physical memory. Because the kernel is not paged, all pages in the "required" pool are held by the kernel and cannot be used for other purposes. To keep the kernel utilization percentage as high as possible, it is desirable to release unused memory in the "required" pool rather than to hold it as is typically done with user processes. Because the kernel can directly manipulate its own page maps, releasing unused memory is fast; a user process must do a system call to release memory.

The most important criterion for a memory allocator is that it be fast. Because memory allocation is done frequently, a slow memory allocator will degrade the system performance. Speed of allocation is more critical when executing in the kernel than in user code, because the kernel must allocate many data structure that user processes can allocate cheaply on their run-time stack. In addition, the kernel represents the platform on which all user processes run, and if it is slow, it will degrade the performance of every process that is running.

Another problem with a slow memory allocator is that programmers of frequently-used kernel interfaces will feel that they cannot afford to use it as their primary memory allocator. Instead they will build their own memory allocator on top of the original by maintaining their own pool of memory blocks. Multiple allocators reduce the efficiency with which memory is used. The kernel ends up with many different free lists of memory instead of a single free list from which all allocation can be drawn. For example, consider the case of two subsystems that need memory. If they have their own free lists, the amount of memory tied up in the two lists will be the sum of the greatest amount of memory that each of the two subsystems has ever used. If they share a free list, the amount of memory tied up in the free list may be as low as the greatest amount of memory that either subsystem used. As the number of subsystems grows, the savings from having a single free list grow.

3. Existing User-level Implementations

There are many different algorithms and implementations of user-level memory allocators. A survey of those available on UNIX systems appeared in [Korn85]. Nearly all of the memory allocators tested made good use of memory, though most of them were too slow for use in the kernel. The fastest memory allocator in the survey by nearly a factor of two was the memory allocator provided on 4.2BSD originally written by Chris Kingsley at California Institute of Technology. Unfortunately, the 4.2BSD memory allocator also wasted twice as much memory as its nearest competitor in the survey.

The 4.2BSD user-level memory allocator works by maintaining a set of lists that are ordered by increasing powers of two. Each list contains a set of memory blocks of its corresponding size. To fulfill a memory request, the size of the request is rounded up to the next power of two. A piece of memory is then removed from the list corresponding to the specified power of two and returned to the requester. Thus, a request for a block of memory of size 53 returns a block from the 64-sized list. A typical memory allocation requires a roundup calculation followed by a linked list removal. Only if the list is empty is a real memory allocation done. The free operation is also fast; the block of memory is put back onto the list from which it came. The correct list is identified by a size indicator stored immediately preceding the memory block.

4. Considerations Unique to a Kernel Allocator

There are several special conditions that arise when writing a memory allocator for the kernel that do not apply to a user process memory allocator. First, the maximum memory allocation can be determined at the time that the machine is booted. This number is never more than the amount of physical memory on the machine, and is typically much less since a machine with all its memory dedicated to the operating system is uninteresting to use. Thus, the kernel can statically allocate a set of data structures to manage its dynamically allocated memory. These data structures never need to be expanded to accommodate memory requests; yet, if properly designed, they need not be large. For a user process, the maximum amount of memory that may be allocated is a function of the maximum size of its virtual memory. Although it could allocate static data structures to manage its entire virtual memory, even if they were efficiently encoded they would potentially be huge. The other alternative is to allocate data structures as they are needed. However, that adds extra complications such as new failure modes if it cannot allocate space for additional structures and additional mechanisms to link them all together.

Another special condition of the kernel memory allocator is that it can control its own address space. Unlike user processes that can only grow and shrink their heap at one end, the kernel can keep an arena of kernel addresses and allocate pieces from that arena which it then populates with physical memory. The effect is much the same as a user process that has parts of its address space paged out when they are not in use, except that the kernel can explicitly control the set of pages allocated to its address space. The result is that the ‘‘working set’’ of pages in use by the kernel exactly corresponds to the set of pages that it is really using.

A final special condition that applies to the kernel is that all of the different uses of dynamic memory are known in advance. Each one of these uses of dynamic memory can be assigned a type. For each type of dynamic memory that is allocated, the kernel can provide allocation limits. One reason given for having separate allocators is that no single allocator could starve the rest of the kernel of all its available memory and thus a single runaway client could not paralyze the system. By putting limits on each type of memory, the single general purpose memory allocator can provide the same protection against memory starvation.†

Figure 1 shows the memory usage of the kernel over a one day period on a general timesharing machine at Berkeley. The ‘‘In Use’’, ‘‘Free’’, and ‘‘Mem Use’’ fields are instantaneous values; the ‘‘Requests’’ field is the number of allocations since system startup; the ‘‘High Use’’ field is the maximum value of the ‘‘Mem Use’’ field since system startup. The figure demonstrates that most allocations are for small objects. Large allocations occur infrequently, and are typically for long-lived objects such as buffers to hold the superblock for a mounted file system. Thus, a memory allocator only needs to be fast for small pieces of memory.

†One might seriously ask the question what good it is if ‘‘only’’ one subsystem within the kernel hangs if it is something like the network on a diskless workstation.

Memory statistics by bucket size

Size	In Use	Free	Requests
128	329	39	3129219
256	0	0	0
512	4	0	16
1024	17	5	648771
2048	13	0	13
2049−4096	0	0	157
4097−8192	2	0	103
8193−16384	0	0	0
16385−32768	1	0	1

Memory statistics by type

Type	In Use	Mem Use	High Use	Requests
mbuf	6	1K	17K	3099066
devbuf	13	53K	53K	13
socket	37	5K	6K	1275
pcb	55	7K	8K	1512
routetbl	229	29K	29K	2424
fragtbl	0	0K	1K	404
zombie	3	1K	1K	24538
namei	0	0K	5K	648754
ioctlops	0	0K	1K	12
superblk	24	34K	34K	24
temp	0	0K	8K	258

\.in 0
\.ce
*(Lb. *(Lt

5. Implementation of the Kernel Memory Allocator

In reviewing the available memory allocators, none of their strategies could be used without some modification. The kernel memory allocator that we ended up with is a hybrid of the fast memory allocator found in the 4.2BSD C library and a slower but more-memory-efficient first-fit allocator.

Small allocations are done using the 4.2BSD power-of-two list strategy; the typical allocation requires only a computation of the list to use and the removal of an element if it is available, so it is quite fast. Macros are provided to avoid the cost of a subroutine call. Only if the request cannot be fulfilled from a list is a call made to the allocator itself. To ensure that the allocator is always called for large requests, the lists corresponding to large allocations are always empty. Appendix A shows the data structures and implementation of the macros.

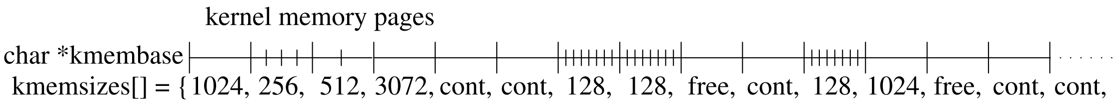
Similarly, freeing a block of memory can be done with a macro. The macro computes the list on which to place the request and puts it there. The free routine is called only if the block of memory is considered to be a large allocation. Including the cost of blocking out interrupts, the allocation and freeing macros generate respectively only nine and sixteen (simple) VAX instructions.

Because of the inefficiency of power-of-two allocation strategies for large allocations, a different strategy is used for allocations larger than two kilobytes. The selection of two kilobytes is derived from our statistics on the utilization of memory within the kernel, that showed that 95 to 98% of allocations are of size one kilobyte or less. A frequent caller of the memory allocator (the name translation function) always requests a one kilobyte block. Additionally the allocation method for large blocks is based on allocating pieces of memory in multiples of pages. Consequently the actual allocation size for requests of size

$2 \times$ *pagesize* or less are identical.† In 4.3BSD on the VAX, the (software) page size is one kilobyte, so two kilobytes is the smallest logical cutoff.

Large allocations are first rounded up to be a multiple of the page size. The allocator then uses a first-fit algorithm to find space in the kernel address arena set aside for dynamic allocations. Thus a request for a five kilobyte piece of memory will use exactly five pages of memory rather than eight kilobytes as with the power-of-two allocation strategy. When a large piece of memory is freed, the memory pages are returned to the free memory pool, and the address space is returned to the kernel address arena where it is coalesced with adjacent free pieces.

Another technique to improve both the efficiency of memory utilization and the speed of allocation is to cluster same-sized small allocations on a page. When a list for a power-of-two allocation is empty, a new page is allocated and divided into pieces of the needed size. This strategy speeds future allocations as several pieces of memory become available as a result of the call into the allocator.



Legend:
free − unused page
cont − continuation of previous page

Usage:

```
memsize(addr)
        char *addr;
{
        return(kmemsizes[(addr − kmembase) − PAGESIZE]);
}
```

\.in 0 \.ce *(Lb. *(Lt

Because the size is not specified when a block of memory is freed, the allocator must keep track of the sizes of the pieces it has handed out. The 4.2BSD user-level allocator stores the size of each block in a header just before the allocation. However, this strategy doubles the memory requirement for allocations that require a power-of-two-sized block. Therefore, instead of storing the size of each piece of memory with the piece itself, the size information is associated with the memory page. Figure 2 shows how the kernel determines the size of a piece of memory that is being freed, by calculating the page in which it resides, and looking up the size associated with that page. Eliminating the cost of the overhead per piece improved utilization far more than expected. The reason is that many allocations in the kernel are for blocks of memory whose size is exactly a power of two. These requests would be nearly doubled if the user-level strategy were used. Now they can be accommodated with no wasted memory.

The allocator can be called both from the top half of the kernel, which is willing to wait for memory to become available, and from the interrupt routines in the bottom half of the kernel that cannot wait for memory to become available. Clients indicate their willingness (and ability) to wait with a flag to the allocation routine. For clients that are willing to wait, the allocator guarrentees that their request will succeed. Thus, these clients can need not check the return value from the allocator. If memory is unavailable and the client cannot wait, the allocator returns a null pointer. These clients must be prepared to cope with this (hopefully infrequent) condition (usually by giving up and hoping to do better later).

†To understand why this number is $2 \times$ *pagesize* one observes that the power-of-two algorithm yields sizes of 1, 2, 4, 8, ... pages while the large block algorithm that allocates in multiples of pages yields sizes of 1, 2, 3, 4, ... pages. Thus for allocations of sizes between one and two pages both algorithms use two pages; it is not until allocations of sizes between two and three pages that a difference emerges where the power-of-two algorithm will use four pages while the large block algorithm will use three pages.

6. Results of the Implementation

The new memory allocator was written about a year ago. Conversion from the old memory allocators to the new allocator has been going on ever since. Many of the special purpose allocators have been eliminated. This list includes *calloc*(), *wmemall*(), and *zmemall*(). Many of the special purpose memory allocators built on top of other allocators have also been eliminated. For example, the allocator that was built on top of the buffer pool allocator *geteblk*() to allocate pathname buffers in *namei*() has been eliminated. Because the typical allocation is so fast, we have found that none of the special purpose pools are needed. Indeed, the allocation is about the same as the previous cost of allocating buffers from the network pool (*mbuf*s). Consequently applications that used to allocate network buffers for their own uses have been switched over to using the general purpose allocator without increasing their running time.

Quantifying the performance of the allocator is difficult because it is hard to measure the amount of time spent allocating and freeing memory in the kernel. The usual approach is to compile a kernel for profiling and then compare the running time of the routines that implemented the old abstraction versus those that implement the new one. The old routines are difficult to quantify because individual routines were used for more than one purpose. For example, the *geteblk*() routine was used both to allocate one kilobyte memory blocks and for its intended purpose of providing buffers to the filesystem. Differentiating these uses is often difficult. To get a measure of the cost of memory allocation before putting in our new allocator, we summed up the running time of all the routines whose exclusive task was memory allocation. To this total we added the fraction of the running time of the multi-purpose routines that could clearly be identified as memory allocation usage. This number showed that approximately three percent of the time spent in the kernel could be accounted to memory allocation.

The new allocator is difficult to measure because the usual case of the memory allocator is implemented as a macro. Thus, its running time is a small fraction of the running time of the numerous routines in the kernel that use it. To get a bound on the cost, we changed the macro always to call the memory allocation routine. Running in this mode, the memory allocator accounted for six percent of the time spent in the kernel. Factoring out the cost of the statistics collection and the subroutine call overhead for the cases that could normally be handled by the macro, we estimate that the allocator would account for at most four percent of time in the kernel. These measurements show that the new allocator does not introduce significant new run-time costs.

The other major success has been in keeping the size information on a per-page basis. This technique allows the most frequently requested sizes to be allocated without waste. It also reduces the amount of bookkeeping information associated with the allocator to four kilobytes of information per megabyte of memory under management (with a one kilobyte page size).

7. Future Work

Our next project is to convert many of the static kernel tables to be dynamically allocated. Static tables include the process table, the file table, and the mount table. Making these tables dynamic will have two benefits. First, it will reduce the amount of memory that must be statically allocated at boot time. Second, it will eliminate the arbitrary upper limit imposed by the current static sizing (although a limit will be retained to constrain runaway clients). Other researchers have already shown the memory savings achieved by this conversion [Rodriguez88].

Under the current implementation, memory is never moved from one size list to another. With the 4.2BSD memory allocator this causes problems, particularly for large allocations where a process may use a quarter megabyte piece of memory once, which is then never available for any other size request. In our hybrid scheme, memory can be shuffled between large requests so that large blocks of memory are never stranded as they are with the 4.2BSD allocator. However, pages allocated to small requests are allocated once to a particular size and never changed thereafter. If a burst of requests came in for a particular size, that size would acquire a large amount of memory that would then not be available for other future requests.

In practice, we do not find that the free lists become too large. However, we have been investigating ways to handle such problems if they occur in the future. Our current investigations involve a routine that can run as part of the idle loop that would sort the elements on each of the free lists into order of increasing address. Since any given page has only one size of elements allocated from it, the effect of the sorting would be to sort the list into distinct pages. When all the pieces of a page became free, the page itself could be released back to the free pool so that it could be allocated to another purpose. Although there is no guarantee that all the pieces of a page would ever be freed, most allocations are short-lived, lasting only for the duration of an open file descriptor, an open network connection, or a system call. As new allocations would be made from the page sorted to the front of the list, return of elements from pages at the back would eventually allow pages later in the list to be freed.

Two of the traditional UNIX memory allocators remain in the current system. The terminal subsystem uses *clist*s (character lists). That part of the system is expected to undergo major revision within the next year or so, and it will probably be changed to use *mbuf*s as it is merged into the network system. The other major allocator that remains is *getblk*(), the routine that manages the filesystem buffer pool memory and associated control information. Only the filesystem uses *getblk*() in the current system; it manages the constant-sized buffer pool. We plan to merge the filesystem buffer cache into the virtual memory system's page cache in the future. This change will allow the size of the buffer pool to be changed according to memory load, but will require a policy for balancing memory needs with filesystem cache performance.

8. Acknowledgments

In the spirit of community support, we have made various versions of our allocator available to our test sites. They have been busily burning it in and giving us feedback on their experiences. We acknowledge their invaluable input. The feedback from the Usenix program committee on the initial draft of our paper suggested numerous important improvements.

9. References

Korn85 David Korn, Kiem-Phong Vo, ‘‘In Search of a Better Malloc’’ *Proceedings of the Portland Usenix Conference*, pp 489-506, June 1985.

McKusick85 M. McKusick, M. Karels, S. Leffler, ‘‘Performance Improvements and Functional Enhancements in 4.3BSD’’ *Proceedings of the Portland Usenix Conference*, pp 519-531, June 1985.

Rodriguez88 Robert Rodriguez, Matt Koehler, Larry Palmer, Ricky Palmer, ‘‘A Dynamic UNIX Operating System’’ *Proceedings of the San Francisco Usenix Conference*, June 1988.

Thompson78 Ken Thompson, ‘‘UNIX Implementation’’ *Bell System Technical Journal*, volume 57, number 6, pp 1931-1946, 1978.

10. Appendix A - Implementation Details

```
/*
 * Constants for setting the parameters of the kernel memory allocator.
 *
 * 2 ** MINBUCKET is the smallest unit of memory that will be
 * allocated. It must be at least large enough to hold a pointer.
 *
 * Units of memory less or equal to MAXALLOCSAVE will permanently
 * allocate physical memory; requests for these size pieces of memory
 * are quite fast. Allocations greater than MAXALLOCSAVE must
 * always allocate and free physical memory; requests for these size
 * allocations should be done infrequently as they will be slow.
 * Constraints: CLBYTES <= MAXALLOCSAVE <= 2 ** (MINBUCKET + 14)
 * and MAXALLOCSIZE must be a power of two.
 */
#define MINBUCKET       4                   /* 4 => min allocation of 16 bytes */
#define MAXALLOCSAVE    (2 * CLBYTES)                              MAXALLOCSAVE

/*
 * Maximum amount of kernel dynamic memory.
 * Constraints: must be a multiple of the pagesize.
 */
#define MAXKMEM         (1024 * PAGESIZE)                          MAXKMEM

/*
 * Arena for all kernel dynamic memory allocation.
 * This arena is known to start on a page boundary.
 */
extern char kmembase[MAXKMEM];

/*
 * Array of descriptors that describe the contents of each page
 */
struct kmemsizes {
        short       ks_indx;        /* bucket index, size of small allocations */
        u_short     ks_pagecnt;     /* for large allocations, pages allocated */
} kmemsizes[MAXKMEM / PAGESIZE];

/*
 * Set of buckets for each size of memory block that is retained
 */
struct kmembuckets {
        caddr_t kb_next;            /* list of free blocks */
} bucket[MINBUCKET + 16];
```

MINALLOCSIZE
BUCKETINDX

```
/*
 * Macro to convert a size to a bucket index. If the size is constant,
 * this macro reduces to a compile time constant.
 */
#define MINALLOCSIZE        (1 << MINBUCKET)
#define BUCKETINDX(size) \
        (size) <= (MINALLOCSIZE * 128) \
                ? (size) <= (MINALLOCSIZE * 8) \
                        ? (size) <= (MINALLOCSIZE * 2) \
                                ? (size) <= (MINALLOCSIZE * 1) \
                                        ? (MINBUCKET + 0) \
                                        : (MINBUCKET + 1) \
                                : (size) <= (MINALLOCSIZE * 4) \
                                        ? (MINBUCKET + 2) \
                                        : (MINBUCKET + 3) \
                        : (size) <= (MINALLOCSIZE* 32) \
                                ? (size) <= (MINALLOCSIZE * 16) \
                                        ? (MINBUCKET + 4) \
                                        : (MINBUCKET + 5) \
                                : (size) <= (MINALLOCSIZE * 64) \
                                        ? (MINBUCKET + 6) \
                                        : (MINBUCKET + 7) \
                : (size) <= (MINALLOCSIZE * 2048) \
                        /* etc ... */
```

MALLOC

```
/*
 * Macro versions for the usual cases of malloc/free
 */
#define MALLOC(space, cast, size, flags) { \
        register struct kmembuckets *kbp = &bucket[BUCKETINDX(size)]; \
        long s = splimp(); \
        if (kbp->kb_next == NULL) { \
                (space) = (cast)malloc(size, flags); \
        } else { \
                (space) = (cast)kbp->kb_next; \
                kbp->kb_next = *(caddr_t *)(space); \
        } \
        splx(s); \
}
```

FREE

```
#define FREE(addr) { \
        register struct kmembuckets *kbp; \
        register struct kmemsizes *ksp = \
                &kmemsizes[((addr) - kmembase) / PAGESIZE]; \
        long s = splimp(); \
        if (1 << ksp->ks_indx > MAXALLOCSAVE) { \
                free(addr); \
        } else { \
                kbp = &bucket[ksp->ks_indx]; \
                *(caddr_t *)(addr) = kbp->kb_next; \
                kbp->kb_next = (caddr_t)(addr); \
        } \
        splx(s); \
}
```