Fast Allocation and Deallocation of Memory Based on Object Lifetimes

David R. Hanson
Department of Computer Science, Princeton University,
Princeton, New Jersey 08544

Research Report CS-TR-191-88

November 1988

ABSTRACT

Dynamic storage management algorithms are based either on object sizes or object lifetimes. Stack allocation, which is based on object lifetimes, is the most efficient algorithm but severely restricts object lifetimes and creation times. Other algorithms based on object sizes, such as first fit and related algorithms, permit arbitrary lifetimes but cost more. More efficient variations capitalize on application-specific characteristics. Quick fit, for example, is more efficient when there are only a few object sizes. This paper describes a simple algorithm that is very efficient when there are few object lifetimes. In terms of instructions executed per byte allocated, the algorithm is almost half the cost of quick fit and less than twice the cost of stack allocation. Space for all objects with the same lifetime is allocated from a list of large arenas, and the entire list is deallocated at once. An implementation in ANSI C is included.

Fast Allocation and Deallocation of Memory Based on Object Lifetimes

Introduction

Most storage management schemes in which storage is allocated and deallocated explicitly are based on the distribution of the sizes of objects. Examples of such schemes are buddy systems, first-fit algorithms [1], and quick-fit [2]. Systems in which storage is deallocated implicitly rely on garbage collection or reference counts to identify inaccessible storage that can be reused [3]. In these systems, as in systems that use stack allocation, the storage management scheme is based on the lifetimes of objects.

Many storage management algorithms are designed to reduce the time overhead. Space overhead may also be a concern, but reducing time overhead is often the first priority. Quick fit [2] is an example of a design that attempts to reduce time overhead. In many applications, most objects are of only few different sizes and quick fit capitalizes on this observation. There are n free lists; `freelist[`k`]` heads a linked list of free k-byte objects. Rare requests for $m > n$ bytes are handled by an alternate scheme, such as first-fit from a list of free blocks of sizes greater than n. This alternate scheme is also used when `freelist[`k`]` is empty.

Quick fit is fast because most allocations can be done in a few lines of in-line code. For example, `k` bytes can be allocated by the following code written in ANSI C [4].

```
if (k <= n && (p = freelist[k]))
        freelist[k] = p->link;
else
        p = allocate(k);
```

The pointer `p` is assigned the first free block on `freelist[k]` if there is one, and the free list pointer is advanced. Otherwise, `allocate` is called to perform an alternate allocation scheme, such as first fit, or to request more memory from the operating system.

Deallocation using quick fit is equally efficient: Deallocating `k` bytes pointed to by `p` is done by the following code.

```
if (k <= n) {
        p->link = freelist[k];
        freelist[k] = p;
} else
        deallocate(p, k);
```

If the size of the object pointed to by `p` is less that or equal to n bytes, the object is added to the front of the appropriate free list. Otherwise, `deallocate`, which implements an alternate deallocation scheme, is called. If `k` is a constant known to be less than or equal to `n`, the test `k <= n` can be omitted in both the allocation and deallocation fragments.

Quick fit and similar algorithms require explicit deallocation, and the cost of explicit deallocation can be significant. In some applications, most deallocations occur at the same time. Window systems are an example. Space for objects that implement a window, such as scroll bars, buttons, etc., are allocated when the window is created and deallocated when the window is destroyed.

Compilers are another example. In compilers for most traditional languages, symbol table entries are allocated in response to declarations, and intermediate representation structures are allocated in response to statements. Both are often deallocated at the end of each compound statement or each procedure. In `lcc`, a recently completed compiler for ANSI C, symbol table entries for local variables and nodes for trees and dags are allocated in just this fashion and deallocated at the end of each function. Deallocation is deferred to the ends of functions so that the code generator can process an entire function, which leads to better code.

In these examples and in similar systems, storage management algorithms that are based on object lifetimes instead of object sizes would be more efficient. Indeed, stack allocation would be most efficient, but can be used only if all object lifetimes are nested properly, which is generally not the case in window systems, compilers, and many other applications. The remainder of this paper describes a simple storage management scheme that is based on object lifetimes. This scheme is designed to be used in applications such as those described above; its use in `lcc`, for example, reduced storage management overhead significantly.

Storage Organization

Objects are grouped according to lifetimes, not sizes. While there can be as many groups as necessary, a few suffice in practice. For example, `lcc` uses only two groups.

Objects with the same lifetime t are allocated from a large *arena*. An arena is defined by the C structure

```
struct arena {          /* storage allocation arena: */
        struct arena *next;     /* link to next arena */
        char *limit;            /* address of one past end of arena */
        char *avail;            /* next available location */
};
```

The space immediately following the arena structure up to the location given by the `limit` field is the allocatable portion of the arena. `avail` points to the first free location; space below `avail` has been allocated and space beginning at `avail` and up to `limit` is available.

The arenas for objects with lifetime t are linked together using the `next` field in a list beginning with the arena `first[t]`. `first[t]` is a zero-length arena. It has an arena header, but no allocatable space; it serves as a list head. `arena[t]` points to the last arena in this list from which space for allocation requests is taken. Arenas are added to the list dynamically during allocation, as detailed below. Figure 1 shows an example of the state of the arenas for lifetime group 1 after three arenas have been allocated. Shading indicates allocated space. The unused space at the end of the first full-sized arena in Figure 1 is explained below.

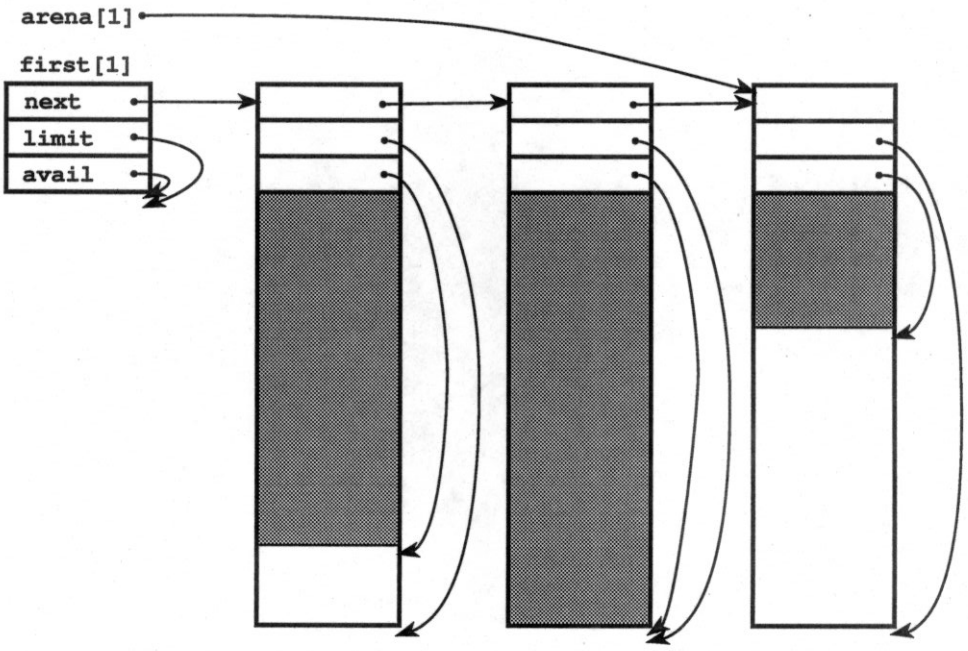

Figure 1. Arena list for lifetime group 1.

Allocation and Deallocation

Allocation is simple and, like quick fit, most allocations can be done with in-line code. To allocate `k` bytes from `arena[t]`, `arena[t]->avail` is simply incremented:

```
p = arena[t]->avail;
if ((arena[t]->avail += k) > arena[t]->limit)
        p = allocate(k, &arena[t]);
```

In the fragment above, `t` is usually a constant so the array reference is reduced to a reference to a global pointer. If `k` bytes do not remain in the current arena, `allocate` is called with `k` and a pointer to the appropriate arena pointer.

All objects with lifetime `t` are deallocated at once by calling `deallocate`:

```
void deallocate(int t) {
        if (arena[t] = first[t].next)
                arena[t]->avail = (char *)arena[t] + sizeof *arena[t];
        else
                arena[t] = &first[t];
}
```

`deallocate(t)` resets `arena[t]` to point to the first zero-length arena, `first[t]`. The list of arenas is *not* modified; thus, subsequent allocations will cause `arena[t]` to reuse the existing arenas. The code in `deallocate` anticipates this use and advances `arena[t]` past the initial zero-length arena, if possible; otherwise it reinitializes `arena[t]`.

When an allocation request cannot be satisfied in the current arena, `allocate` is called to advance to the next arena, if it exists, or to allocate a new one.

```
char *allocate(int n, struct arena **p) {
        struct arena *ap;

        for (ap = *p; ap->avail + n > ap->limit; *p = ap)
                if (ap->next) {         /* move to next arena */
                        ap = ap->next;
                        ap->avail = (char *)ap + sizeof *ap;
                } else {                /* allocate a new arena */
                        int m = ((n + 3)&~3) + MEMINCR*1024 + sizeof *ap;
                        ap->next = (struct arena *) morecore(m);
                        ap = ap->next;
                        ap->limit = (char *)ap + m;
                        ap->avail = (char *)ap + sizeof *ap;
                        ap->next = 0;
                }
        ap->avail += n;
        return ap->avail - n;
}
```

`allocate` is slightly more general than necessary; it handles allocation requests for non-full arenas (by simply incrementing `ap->avail`) as well as full ones. It can thus be called where the details of arenas are best hidden from the caller.

If the request cannot be satisfied from the current arena, `allocate` attempts to advance to the next arena on the list. If this arena exists, its `avail` field is reset. This arena might be too small to accommodate the request, which explains the use of the `for` loop.

If the end of the arena list has been reached, `allocate` requests more memory from the system by calling `morecore`. Enough memory is requested to accommodate the current request rounded to the next multiple of four bytes (`(n + 3)&~3`), plus the size of a nominal arena (`MEMINCR*1024`), plus the size of the arena structure itself (`sizeof *ap`). `MEMINCR` is defined to be a value that is appropriate for the specific application; in `lcc`, for example, `MEMINCR` is 10. The newly acquired arena is initialized and appended to the arena list.

After `ap` has been advanced to the next arena, either an existing one or a new one, the arena pointer is set to point to the new arena. A pointer to this pointer is passed to `allocate` as shown above in the allocation fragment.

Improvements

The algorithm described above is fast because most allocations take only a few instructions and deallocations are nearly free. This speed may come at the cost of space, however. Objects are grouped into a few lifetime classes; two or three is typical. Short-lived objects may be grouped with longer-lived objects just to keep the number of lifetime groups small, which ties up memory longer than strictly necessary. For example, `lcc` no longer needs abstract syntax trees after statements are parsed, but the space is not deallocated until the end of the function. To do otherwise increases deallocation time.

The arena lists may also waste memory. As shown above, once an arena is linked into an arena list, it remains devoted to that list. Thus, for example, if initial allocations in some lifetime group require three arenas, but subsequent allocations require only one arena, two arenas are wasted after their initial use.

The impact of this problem can be reduced by returning the arenas to a list of free arenas in **deallocate** instead of leaving them linked in an arena list. If **freearenas** heads a list of free arenas linked via their **next** fields, **deallocate** becomes

```
void deallocate(int t) {
        arena[t]->next = freearenas;
        freearenas = first[t].next;
        first[t].next = 0;
        arena[t] = &first[t];
}
```

Likewise, **allocate** attempts to get a free arena from **freearenas** before requesting more space from the operating system. This modification adds the following case to the body of the loop in **allocate**.

```
if (ap->next) {         /* move to next arena */
        ap = ap->next;
        ap->avail = (char *)ap + sizeof *ap;
} else if (ap->next = freearenas) {
        freearenas = freearenas->next;
        ap = ap->next;
        ap->avail = (char *)ap + sizeof *ap;
        ap->next = 0;
} else {                /* allocate a new arena */
        ...
}
```

This addition to **allocate** is written so that the arena lists can be preserved for some lifetime groups or added to the free list for others.

When a request cannot be filled in the current arena, **allocate** moves to, or allocates, the next arena, wasting the space at the end of the previous arena. This is illustrated in the first full-size arena in Figure 1. This loss to internal fragmentation is insignificant if the size of arenas is much larger than the average size of allocation requests. For example, **lcc**'s average allocation request is less than 100 bytes and its arenas are at least 10K bytes.

In some applications, the arena list for one lifetime group might grow much larger than the others. In **lcc**, for example, there are two lifetime groups, one for permanent objects such as symbol table entries for global variables and constants, and one for 'transient' objects, such as symbol table entries for local variables and nodes for trees and dags. The transient arena list tends to be much larger than the permanent list. Reducing the length of the arena list by enlarging each arena would reduce fragmentation and reduce the number of calls to **allocate**. This can be accomplished by using different arena sizes for different lifetime groups, or by making the nominal size of an arena (**MEMINCR** in the code above) a function of its position in an arena list.

A simpler approach joins physically adjacent arenas that are also adjacent on the same arena list. Lifetime groups that grow quickly lead to this situation; when space for a new arena is requested by calling **morecore**, it is likely that the newly allocated arena immediately follows the last arena on the arena list. This case can be detected in **allocate**:

```
if (ap->next) {          /* move to next arena */
        ...
} else if (ap->next = freearenas) {
        ...
} else {                 /* allocate a new arena */
        int m = ((n + 3)&~3) + MEMINCR*1024 + sizeof *ap;
        ap->next = (struct arena *) morecore(m);
        if ((char *)ap->next == ap->limit) /* extend previous arena? */
                ap->limit = (char *)ap->next + m;
        else {                   /* link to a new arena */
                ap = ap->next;
                ap->limit = (char *)ap + m;
                ap->avail = (char *)ap + sizeof *ap;
        }
        ap->next = 0;
}
...
```

If the newly allocated arena is physically adjacent to the last one, the `limit` field of the last arena is adjusted accordingly, and allocation continues from that arena with no fragmentation.

Discussion

Storage management algorithms trade flexibility for cost per byte allocated. This cost includes both the allocation and deallocation costs. Stack allocation is at one end of the spectrum. For example, the in-line stack allocation fragment

```
p = avail;
if ((avail += k) > limit)
    p = allocate(k);
```

requires about 5 instructions on a VAX. This instruction count and those below are taken from the code generated by `lcc`, which is typical of C compilers. Other compilers may produce different instruction counts, but the relative costs between the algorithms will remain approximately the same. Deallocation cost is negligible; in the best case, the space for many objects can be deallocated in 1 instruction. Thus, for N allocations of an average of k bytes each, stack allocation costs about $5N/k$ instructions executed per byte allocated. The price of this efficiency is reduced flexibility; stack allocation can be used only when object lifetimes nest properly.

Garbage collection [3] represents the other end of the spectrum. While an allocation costs no more than with stack allocation, deallocation can be very costly and yield a high overall cost per byte. Garbage collection is very flexible, however, and imposes no restrictions on object lifetimes and does not require explicit deallocations. Similar comments apply to algorithms based on reference counting, which incur additional costs in maintaining reference counts when pointers are created or destroyed. (Using very large physical memories can dramatically reduce the cost of deallocations and make garbage collection competitive with stack allocation [5].)

Algorithms based on object sizes fall in the middle of this spectrum. First fit, for example, typically requires a search of a list of free blocks, and deallocations require a similar search if a sorted free list is used. Deallocation, and thus first fit as a whole, normally costs less than garbage collection. First fit imposes no lifetime restrictions on objects, but it does require explicit deallocations; for example, `lcc` would have to loop over local symbols at the end of each function and deallocate each symbol. On the VAX, allocation of k bytes costs about $3 \times 6 + 8$ instructions assuming the search loop examines 3 blocks on average as reported in Reference 1. Deallocations cost about $3 \times 6 + 13$ instructions plus the cost of the code in which the deallocations appear. Thus, for N allocations of k bytes each, first fit costs about $57N/k$ instructions per byte on the average. One disadvantage of first fit and similar algorithms is that it is difficult to put the code for most allocations in-line.

As described above, quick fit can cost significantly less than first fit in some applications. The in-line allocation fragment shown in the introduction costs about 8 instructions on a VAX when `freelist[`k`]` points to a free block. Most deallocations cost about 7 instructions using the code shown in the introduction, plus

the cost of the code in which the deallocations occur. (If k is a compile-time constant, these costs can be reduced to 4 and 2 instructions, respectively.) Thus, in the best case, N allocations of k bytes each cost at least $15N/k$ instructions per byte. While quick fit costs about three times more than stack allocation, it is significantly more flexible and costs less than a third the cost of first fit.

The arena-based algorithm presented in this paper retains quick fit's flexibility, but is almost as efficient as stack allocation by avoiding per-object deallocations. This improvement comes at the cost of extra memory. Using the in-line allocation code shown in the third section, allocations cost about 8 instructions on the VAX, when there is room in the current arena and t is a compile-time constant. Deallocation cost, as in stack allocation, is negligible. Thus, the best-case cost for N allocations of k bytes each is $8N/k$ instructions per byte—almost half the cost of quick fit and less than twice the cost of stack allocation.

Acknowledgements

Chris Fraser's comments helped clarify several sections of this paper.

References

1. D. E. Knuth, *The Art of Computer Programming: Volume 1, Fundamental Algorithms*, Addison Wesley, Reading, MA, Second Edition, 1973.
2. C. B. Weinstock and W. A. Wulf, Quick Fit: An Efficient Algorithm for Heap Storage Management, *SIGPLAN Notices* **23**, 10 (Oct. 1988) 141–148.
3. J. Cohen, Garbage Collection of Linked Data Structures, *Computing Surveys* **13**, 3 (Sep. 1981) 341–367.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ, Second Edition, 1988.
5. A. W. Appel, Garbage Collection Can be Faster than Stack Allocation, *Inf. Proc. Letters* **25**, 4 (June 1987) 275–279.