THE AUSTRALIAN NATIONAL UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

TECHNICAL REPORT

EFFICIENT IMPLEMENTATION OF THE FIRST-FIT STRATEGY FOR DYNAMIC STORAGE ALLOCATION

BY

R.P. BRENT

TR-CS-81-05

EFFICIENT IMPLEMENTATION OF THE FIRST-FIT STRATEGY FOR DYNAMIC STORAGE ALLOCATION

Richard P. Brent
Department of Computer Science
Australian National University
Box 4, Canberra, A.C.T. 2600

Abstract

We describe an algorithm which efficiently implements the first-fit strategy for dynamic storage allocation. The algorithm imposes a storage overhead of only one word per allocated block (plus a few percent of the total space used for dynamic storage), and the time required to allocate or free a block is O(log W), where W is the size of the dynamic storage area. The algorithm is faster than commonly used algorithms when many small blocks are allocated, is relatively easy to implement in a low-level programming language, and could be used to provide runtime support for high-level languages such as Pascal.

1. Introduction

The dynamic storage allocation problem is to maintain a region of memory so that requests for the allocation and subsequent liberation of blocks of various sizes can be met as far as possible. The problem arises in operating systems (where the blocks are usually large), in simulation (where they are usually small), and in providing support for the run-time facilities of some programming languages, e.g. the "new" and "dispose" procedures of Pascal [3]. A surprising number of current Pascal systems fail to implement "dispose", implement it inefficiently, or use a stack discipline instead of genuine dynamic storage allocation

It is important to distinguish between a strategy for dynamic storage allocation, and an algorithm (or set of algorithms) designed to implement a particular strategy. Different algorithms implement the same strategy if they always satisfy identical sequences of requests by allocations at identical sequences of memory locations, and differ only in the time and space overhead required to satisfy the requests.

Several dynamic storage allocation strategies have been proposed and compared [4, 5, 8]. The result of such a comparison depends greatly on the assumed distribution of block sizes, block lifetimes, and details of the testing procedure: see [4]. The theoretical worst-case behaviour of several strategies has also been studied [7]. For our purposes it is sufficient to note that the "first-fit" strategy compares well with other strategies, including the "best-fit" and "buddy" strategies, both empirically and in the worst case. In the comparisons the first-fit strategy emerges either as the best strategy or close to the best, depending on the precise assumptions and testing procedure.

This paper is concerned with algorithms for implementing the first-fit strategy. The obvious algorithm [5, Alg. A] maintains a singly-linked list of free blocks and has to search about halfway along this list (on average) to allocate a block, so it is slow if the number of free blocks is large. A common "improvement" [5, ex. 6] avoids this difficulty at the expense of not implementing the first-fit strategy at all: instead it implements a "next-fit" strategy which is inferior to first-fit for certain distributions of block sizes and lifetimes, e.g. distribution (a) of Section 5. In Section 3 we describe an algorithm which implements the pure first-fit strategy, but is much faster than the obvious algorithm when the number of free blocks is large. The worst-case performance of the algorithm is discussed in Section 4, and some empirical results are given in Section 5.

McCreight [6, ex. 6.2.3.30] has devised algorithms, based on balanced binary trees, for the first-fit and best-fit strategies. Our new algorithm is faster and easier to implement than McCreight's algorithms, and has other advantages (mentioned in Section 4) when small blocks are common. McCreight's first-fit algorithm is described briefly in Section 2.

2. Two known algorithms for the first-fit strategy

A simple algorithm, which we call "Algorithm A", is given in [5, Algs. A and B]. Each free block p contains two fields:

size(p) - the number of words in the block, and
link(p) - a pointer to the next free block.

(Here, p and link(p) may be memory addresses, array indices or reference variables. For simplicity we shall assume that they are memory addresses. We shall also assume that a "word" is the basic unit of storage.) If a block of n words is required, we simply scan the list of free blocks from the beginning, until either a block p is found with size(p) $\geq$ n, or the end of the list is reached (when no sufficiently large block is available). If size(p) > n, the block is split into two smaller blocks, of size n and size(p) - n. (There may be a lower bound on the size of a block which can be created by splitting, but we ignore this complication here.) The block of size n is removed from the free list and made available for use. For details see [5, Alg. A].

When a block p is freed it is necessary to add it to the list of free blocks, and to merge it with its left and/or right neighbours if they are free. This is possible if

a) the free list is kept in address order (i.e. link(p) > p if p and link (p) are the addresses of successive blocks on the free list); and

b) the size of a block to be released is known. The simplest way to ensure this is to reserve a size field in allocated blocks as well as in free blocks.

Let F denote the average number of free blocks. (We assume that an equilibrium has been reached, so it makes sense to talk about averages.) Algorithm A requires, on average, the inspection of about F/2 blocks when a block is allocated or freed. Algorithms which use tag fields or doubly linked lists may be slightly faster than Algorithm A, but they still require time O(F) on average to allocate a block [5, Alg. C and ex. 19].

McCreight [6, ex. 6.2.3.30] has given a (theoretically) more efficient first-fit algorithm. His algorithm, which we call "Algorithm M", uses a height-balanced binary tree (AVL tree) with each free block corresponding to a node in the tree. A field is reserved in each node to indicate the size of the largest free block corresponding to a node in the left subtree attached to the given node. A disadvantage of Algorithm M is that the smallest block must be large enough to hold at least five fields (two pointers to left and right descendants, a balance factor indicating the difference in height between the left and right subtrees, and two size fields). A practical implementation would probably maintain three additional fields (two pointers to left and right neighbouring free blocks, and an "up" pointer to avoid the need for a stack when traversing the tree). Thus, Algorithm M is not suitable in applications where small blocks are common or where, to avoid the need for "actual" and "requested" size fields, allocated blocks must be exactly the size requested.

The time required by Algorithm M to allocate or free a block is O(log F), theoretically better than the O(F) of Algorithm A. However, the constant hidden in the "O" notation is rather large (see Section 5) and the implementation of Algorithm M is not a trivial task. The algorithm described in Section 3 avoids these difficulties while retaining a logarithmic worst-case time bound.

3. A new algorithm for the first-fit strategy

In this section we describe a new algorithm, "Algorithm N", for the pure first-fit strategy. Suppose that W contiguous words are available for the dynamic storage area. Choose S to be a power of two in the range $W \le cS < 2W$ for some suitable constant c (e.g. c = 200: see Section 4). The dynamic storage area is split into S segments, each (except possibly for the last one) of $\lceil W/S \rceil$ words. For a reason which will soon be apparent, we number the segments S, S+1, ... , 2S-1.

The algorithm maintains two arrays:

```
PA: array [S .. (2S-1)] of integer; {"pointer array"}
ST: array [0 .. (2S-1)] of integer; {"segment tree" }
```

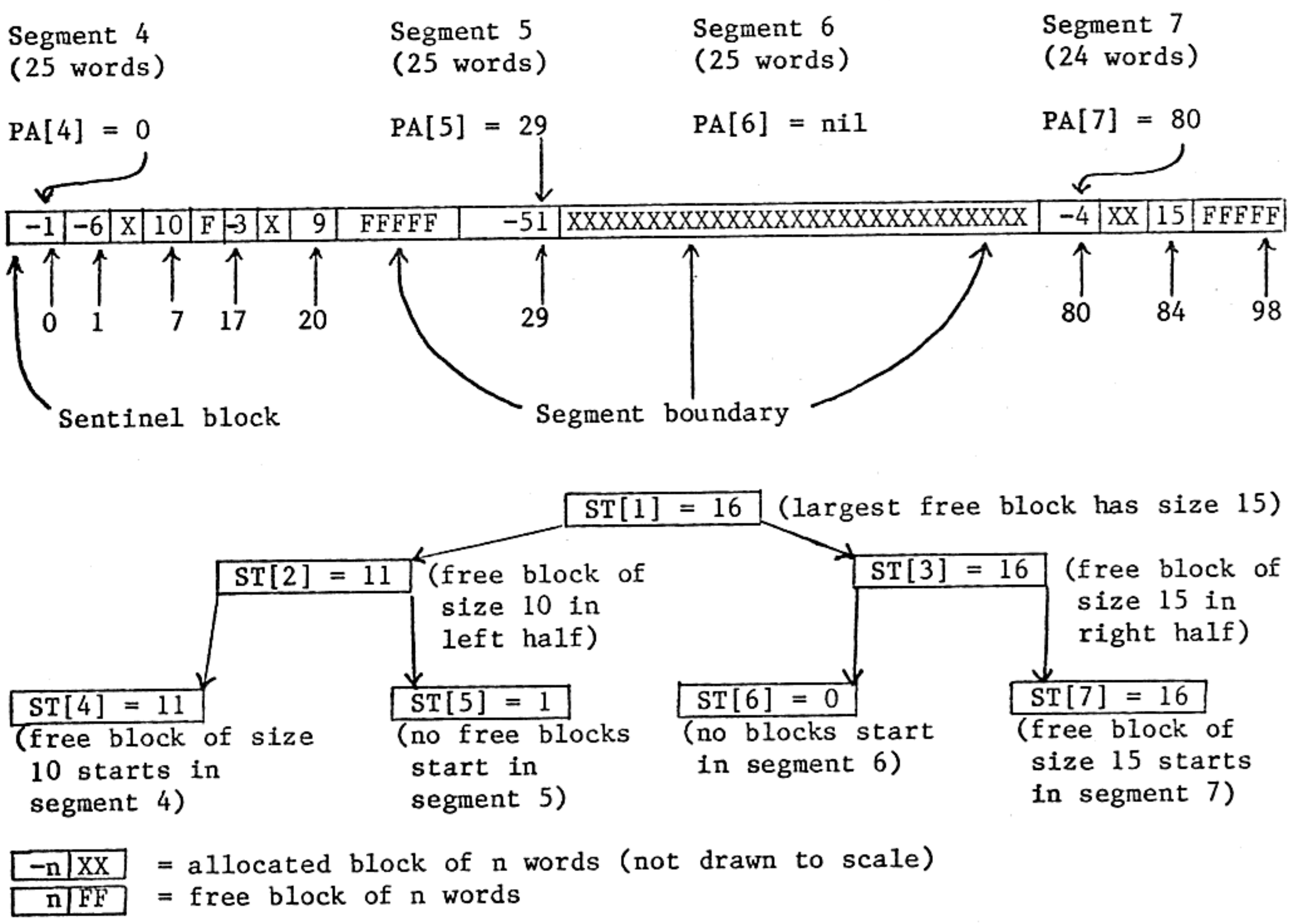
so that the following relations hold:

$$PA[i] = \begin{cases} \text{address of the first block starting in segment } i \text{, or nil if there is no block starting} \\ \text{in segment } i \text{ ("nil" is some address outside the allowable range for the dynamic storage area);} \end{cases}$$

$$ST[i] = \begin{cases} \max(ST[2i], ST[2i+1]) \text{ if } 0 < i < S, \\ 0 \text{ if no block starts in segment } i,\ S \le i < 2S, \\ 1 \text{ if some block but no free block starts in segment } i,\ S \le i < 2S, \\ 1 + (\text{size of largest free block starting in segment } i) \text{ if some free block starts in segment } i,\ S \le i < 2S. \end{cases}$$

Thus, ST[1], ..., ST[2S-1] is a "heap" in the sense of [6, Sec. 5.2.3], though we shall avoid using the word "heap" because it has a different meaning in the context of dynamic storage allocation. We may think of ST[1], ..., ST[2S-1] as a perfectly balanced binary tree of 2S-1 nodes, with implicit links. This is illustrated in Figure 1 for S = 4.

Figure 1: An example with W = 99, S = 4



Segment 4
(25 words)
PA[4] = 0
Segment 5
(25 words)
PA[5] = 29
Segment 6
(25 words)
PA[6] = nil
Segment 7
(24 words)
PA[7] = 80
-1
-6
X
10
F
-3
X
9
FFFFF
-51
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
-4
XX
15
FFFFF
0
1
7
17
20
29
80
84
98
Sentinel block
Segment boundary
ST[1] = 16
(largest free block has size 15)
ST[2] = 11
(free block of size 10 in left half)
ST[3] = 16
(free block of size 15 in right half)
ST[4] = 11
(free block of size 10 starts in segment 4)
ST[5] = 1
(no free blocks start in segment 5)
ST[6] = 0
(no blocks start in segment 6)
ST[7] = 16
(free block of size 15 starts in segment 7)
-n XX
= allocated block of n words (not drawn to scale)
n FF
= free block of n words

The first word in each block (free or allocated) is reserved for a signed size field. The sign is positive if the block is free, negative if it is allocated. Thus, if V[p] denotes the content of memory location p, a block starting at location p has size abs(V[p]), is free if V[p] > 0, and the next block (if any) starts at location p + abs(V[p]). We essentially have a singly-linked list of blocks with a one-bit field to indicate whether a block is free or allocated. To find the first free block of size at least n words, we have (in pseudo-Pascal):

```
if ST[1] ≤ n then   {error exit: there is no free block
                     large enough};
i := 1;
while i < S do      {descend segment tree, keeping left
                     where possible}
    if ST[2*i] > n then i := 2*i else i := 2*i + 1;
p := PA[i];         {the required block starts in segment i,
                     and p is the address of the first
                     block in segment i}
while V[p] < n do p := p + abs(V[p]); {scan until block
                     found in segment i}
{now the required block starts at address p}
```

Before allocating a block p it is necessary to split it into two blocks, of size n and V[p] - n, if V[p] > n. To do this we set

```
V[p+n] := V[p] - n;
V[p] := n;
```

and update the arrays PA and ST. The actions required are a combination of those described below (although some optimizations are possible), so we omit the details and assume that V[p] = n. To allocate a block p starting in segment i we set

```
V[p] := -V[p]
```

and update the array ST as follows:

```
{compute new value mx for ST[i]}
mx := 0;
q := PA[i];                {first block in segment i}
while q ∈ segment i do     {look for largest free block in
                            segment i}
     begin
     mx := max (mx, V[q]);
     q := q + abs(V[q])
     end;
mx := mx + 1;
{now update ST[i] and its ancestors up the segment tree
                           as far as necessary}
ST[0] := 0; {sentinel to ensure that the while loop
                           terminates}
while ST[i] > mx do
     begin
     ST[i] := mx;
     i := i div 2; {ancestor}
     mx := max (ST[2*i], ST[2*i+1])
     end;
```

When a block p in segment i is freed, it must be merged with its left and/or right neighbours if they are free, and PA and ST must be updated appropriately. There are several cases, depending on whether the neighbours are in segment i or not, but everything is straightforward once the block q preceding block p is found. This may be done in O(log S) operations by:

```
j := i;
if PA[i] = p then {the difficult case:  block p is the
                         first block in segment i}
    begin
    while ST[j-1] = 0 do j := j div 2; {ascend tree}
    j := j - 1;  move left
    while j < S do {descend tree again, keeping right
                         if possible}
         begin
         j := 2*j;
         if ST[j+1] > 0 then j := j + 1
         end
    end;
{now the predecessor of block p lies in segment j}
qn := PA[j]; {first block in segment j}
repeat {scan segment j until predecessor of p is found}
    q := qn;
    qn := qn + abs(V[qn]) {qn is the successor of q}
until qn = p
{now q is the predecessor of p}
```

This works provided that a "sentinel" block is allocated at the start of segment S and never freed (so that each block except the sentinel actually has a predecessor). This is illustrated in Figure 1.

4. Worst-case analysis of Algorithm N

When comparing the space requirements of different algorithms we count any space used outside the W words reserved for the dynamic storage area, as well as any reserved fields in allocated blocks. We do not count any space used in free blocks.

The space required by Algorithm N is 3S + R words (3S for the arrays PA snd ST, and one word per allocated block for the signed size field). Recall that $cS < 2W$, so if $c = 200$ the space required for PA and ST is less than 3 percent of the space reserved for the dynamic storage area.

Let n_{min} be the size of the smallest block ever allocated. (We ignore the initial sentinel block here.) Since only $n_{min}-1$ words are actually available in such a block (one word being reserved for the size field), we can assume that $n_{min} \geq 2$, and probably $n_{min} \geq 3$. Thus, the space overhead caused by the size fields is at most $W/n_{min} \leq W/2$ words. Although substantial if there are many small blocks, this overhead is common to all the first-fit algorithms considered - they all need to know the size of a block when it is released, so a size field is generally necessary. (In Pascal the size of a record without variants can be determined at compile-time, but a size field is necessary to determine the size of a record with variants at run-time, unless the maximum size over all variants is allocated, which may waste more space than the size fields.)

The number of blocks starting in any segment is at most $\lceil \lceil W/S \rceil / n_{min} \rceil$, which is bounded by $\lceil c/n_{min} \rceil$. To allocate or free a block requires at worst a small number of scans along the chain of blocks in a segment and a small number of traversals of a branch of the segment tree. Thus, the number of operations is $O(\log S) + O(1) = O(\log W)$.

For Algorithm A the worst-case (and average) number of operations is of order F, where F is the number of free blocks. F is of order W if the average block size is small and the loading is heavy.

For Algorithm M the worst-case (and average) number of operations required to allocate and/or free a block is $O(\log F) = O(\log W)$. However, the constant implied by the "O" notation is considerably larger than for Algorithm N (see Section 5).

5. Implementation of Algorithm N

Algorithm N has been implemented as part of package intended to make dynamic storage allocation readily available to Fortran library routines [1]. The package includes routines for operations on multiple stacks, deques and priority queues. Often a single stack is sufficient [2], but in some applications more general facilities are useful. (In fact, the motivation for development of the dynamic storage allocation package was that it was needed to provide storage management for a multiple-precision interval arithmetic package.) It would not be difficult to translate the basic dynamic storage allocation routines into another low-level language.

To provide a benchmark we also implemented Algorithm A. The implementations were tested with several distributions of block sizes and lifetimes, using a "must keep going" testing procedure [4]. As expected, the algorithms both implemented the same (pure first-fit) strategy, and differed only in their space and time overheads. We found that Algorithm A was faster than Algorithm N if there were few blocks allocated, but Algorithm N was faster if there were more than about 100 allocated blocks (or 50 free blocks: note the "fifty percent" rule [5]). Some statistics are given in Table 1.

Table 1: Comparison of Algorithms A and N

Distribution(1)	Average number of free blocks F	Average time to allocate and free a block (2)		Ratio Alg.N/Alg.A
		Alg.A	Alg.N	
(a)	30	287	321	1.14
(b)	60	446	391	0.88
(c)	120	738	484	0.66
(d)	240	1182	559	0.47

(1) Distribution (a) has

blocksize uniform in 1..10, lifetime in 1..100, with probability 0.8,
blocksize uniform in 10..100, lifetime in 1..100, with probability 0.1,
blocksize uniform in 100..1000, lifetime in 100..200, with probability 0.1

Distribution (b) is the same as (a) except lifetimes are doubled and blocksizes halved. Similarly for (c) and (d), with factors of four and eight respectively. (Non-integral blocksizes are rounded up to the next integer.)

(2) Times are in micro-seconds, based on 100,000 trials on a Univac 1100/82. For Algorithm N we used W = 15,000 and S = 128.

The times given for Algorithm N in Table 1 could be decreased, at the expense of using more space for the arrays PA and ST, by increasing S, the number of segments. The almost linear time required by Algorithm A (as a function of F, the number of free blocks), and the logarithmic time required by Algorithm N, is evident from the third and fourth columns of Table 1.

Algorithm M has not yet been implemented fully, but timing of a priority queue implementation using "leftist trees" [6, Sec. 5.2.3], which are easier to update than AVL trees, indicates that our implementation of Algorithm N would be at least twice as fast as a similar implementation of Algorithm M. This is plausible because the tree used by Algorithm N is always perfectly balanced and links do not need to be maintained between its nodes (since they are implicit).

6. Conclusion

We have shown that it is possible to implement a good dynamic storage strategy, the first-fit strategy, so that:

1. Only one word per allocated block (plus a few percent of the total space) is required for "housekeeping" purposes.

2. The time required to allocate or free a block does not increase linearly with the number of free blocks, but only as O(log W), where the dynamic storage area has size W.

3. The algorithm is straightforward and relatively easy to implement, even in a low-level language.

It is not clear whether a similar implementation of the best-fit strategy is possible. However, the average behaviour of first-fit is usually about as good as that of best-fit, and the worst-case analysis clearly favours first-fit [7]. First-fit also appears to make better use of the available storage space than does the "buddy" system [5], unless the block sizes are restricted to favour the "buddy" system. Another advantage of the first-fit strategy is that it tends to leave a large free block at the high end of the dynamic storage area, and this space may be used for a stack which grows downwards. This facility has been included in the implementation [1]. Since allocation of space on a stack is much simpler than general dynamic storage allocation, it is desirable to use a stack where possible (e.g. for procedure activation records).

References

1. R.P. Brent, A portable dynamic storage allocation package, Tech. Report, Dept. of Computer Science, Australian National University (in preparation).

2. P.A. Fox, A.D. Hall & N.L. Schryer, The PORT mathematical subroutine library, ACM Trans. Math. Software 4 (1978), 104-126.

3. B.J. Gerovac, An implementation of new and dispose using boundary tags, Pascal News 19 (1980), 49-59.

4. J.B. Hext, A storage management laboratory, Austral. Comp. Sci. Communications 2, 1 (Jan. 1980), 185-193.

5. D.E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms (2nd edition), Addison-Wesley, Reading, Mass., 1973, Section 2.5.

6. D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

7. J.M. Robson, Worst case fragmentation of first fit and best fit storage allocation strategies, Comp. J. 20 (1977), 242-244.

8. J.E. Shore, On the external storage fragmentation produced by first-fit and best-fit allocation strategies, Comm. ACM 18 (1975), 433-440.