Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation

R. P. BRENT
Australian National University

We describe an algorithm that 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 maximum number of words allocated dynamically. The algorithm is faster than many commonly used algorithms, especially when many small blocks are allocated, and has good worst-case behavior. It is relatively easy to implement and could be used internally by an operating system or to provide run-time support for high-level languages such as Pascal and Ada. A Pascal implementation is given in the Appendix.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*allocation/deallocation strategies*; *main memory*; D.4.8 [**Operating Systems**]: Performance—modeling and prediction; simulation; E.2 [**Data**]: Data Storage Representations—*contiguous representations*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Dispose, dynamic memory management, dynamic storage allocation, first-fit strategy, heaps, new, trees

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, for example, the "new" and "dispose" procedures of Pascal [8]. A surprisingly large 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* designed to implement a particular strategy. A strategy specifies which blocks are allocated, but not how they are allocated. Different algorithms

Author's address: Computer Sciences Laboratory, Australian National University, GPO Box 4, Canberra ACT 2601, Australia.

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 overheads required to satisfy the requests.

Several dynamic storage allocation strategies have been proposed and compared [3, 9, 12, 14, 16]. The result of such a comparison depends greatly on the assumed distribution of block sizes, block lifetimes, and details of the testing procedure. The theoretical worst-case behavior of several strategies has also been studied [15]. 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 [12, 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" [12, ex. 6] avoids this difficulty at the expense of not implementing the first-fit strategy at all: Instead it implements a "next-fit" strategy that is inferior to first-fit for certain distributions of block sizes and lifetimes, for example, distributions (a)–(d) of Section 5 (see also [1] and [3]). In Section 3 we describe an algorithm that 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 Sections 5 and 6. A secure implementation is described in Section 7.

McCreight [13, ex. 6.2.3.30] has devised theoretically good 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.

Stephenson [18] has recently suggested algorithms for the first-fit and related strategies. Stephenson's algorithms use "Cartesian" trees [19] that may become unbalanced, so the worst-case performance of our algorithm is better than that of Stephenson's algorithms. McCreight's and Stephenson's algorithms for the first-fit strategy are described briefly in Section 2.

2. THREE KNOWN ALGORITHMS FOR THE FIRST-FIT STRATEGY

A simple algorithm, which we call "Algorithm A," is given in [12, 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 indexes, or reference variables. For simplicity we assume that they are memory addresses. We also assume that a "word" is the basic unit of storage, where a word is large enough to store an address. 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 sizes n and size(p) − n. (There may be a lower bound on the size of a block that 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 [12, 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 neighbors if they are free. This is possible if

(1) 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
(2) 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 that 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 [12, Alg. C and ex. 19].

McCreight [13, 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 (i.e., an 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 neighboring 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.

Stephenson [18] has suggested an algorithm, which we call "Algorithm S," in which free blocks are maintained as nodes in a Cartesian tree [19]; that is, for each block N in the tree,

(1) address of descendants on left (if any) < address of block N < address of descendants on right (if any), and
(2) size of descendants on left (if any) ≤ size of block N ≥ size of descendants on right (if any).

Stephenson's algorithm works well, with less overhead than McCreight's, so long as the Cartesian tree remains well balanced. Unfortunately, it is not possible to guarantee this. In the worst case, the Cartesian tree could degenerate to a list, and the time required to allocate or free a block would be $O(F)$, as for Algorithm A.

3. A NEW ALGORITHM FOR THE FIRST-FIT STRATEGY

In this section we describe a new algorithm, "Algorithm N," for implementing the pure first-fit strategy. Suppose that W contiguous words are avilable 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; for example, $c = 200$ (see Section 4). The dynamic storage area is split into S segments numbered $0, \ldots, S-1$, each (except possibly the last) containing $\lceil W/S \rceil$ words.

The algorithm maintains two arrays

```
PA: array [0 ··· S − 1] of integer;      {"pointer array"}
ST: array [0 ··· 2S − 1] of integer;     {"segment tree"}
```

so that the following relations hold:

$$\mathrm{PA}[i] = \begin{cases} (\text{address of the first block starting in segment } i) - 1, & \text{or} \\ W \quad \text{if there is no such block.} \end{cases}$$

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

Thus, ST[1], ..., ST[2S − 1] is a "heap" in the sense of [13, Sect. 5.2.3], though we shall avoid using the word *heap* as 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 for the case $S = 4$ in Figure 1.

There is one "control" word of overhead for each block. We adopt the convention that, for a block of size s starting at address $p + 1$, word p is the control word for the block, and words $p + 1, \ldots, p + s$ are available for use; the control word for the next block (if any) is word $p + s + 1$. By "block p" we mean the block whose control word is at address p. We say that a block starts in a given segment if its control word is in that segment.

The control word for a block contains a signed integer; the sign is positive if the block is free and negative if the block is allocated. The absolute value of the control word is the number of words occupied by the block and its control word, that is, $s + 1$. Thus, if $V[p]$ denotes the contents of word p, a block starting at address $p + 1$ has size $\mathrm{abs}(V[p]) - 1$ and is free if $V[p] > 0$, and the next block

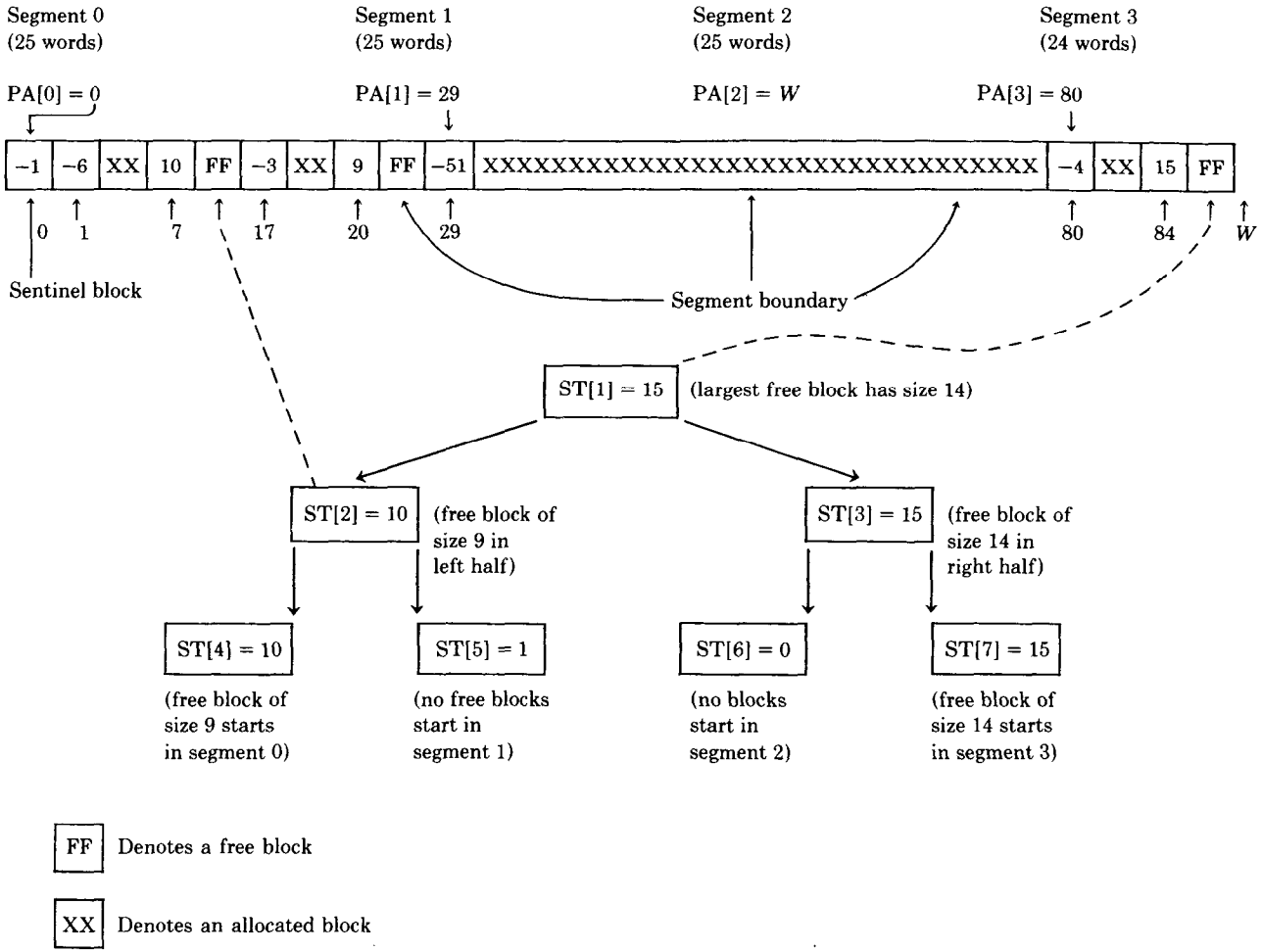

Fig. 1. An example with $W = 99$, $S = 4$ (not drawn to scale).

(if any) has its control word at address p + abs ($V[p]$). To find the first free block of size at least $n - 1$ 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 − S];   {the required block starts in segment i − S, and p is the
                   address of the control word of the first block in segment
                   i − S}
while V[p] < n do p := p + abs(V[p]);
                  {scan until block found in segment i − S}
{now the required block starts at address p + 1}
```

Before allocating a block p of $n - 1$ words, it is necessary to split it into two blocks 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, so we omit the details and assume that $V[p] = n$. To allocate a block p in segment $i - S$, we set

$$V[p] := -V[p]$$

and update the array ST as follows:

```
{compute new value mx for ST[i]}
  q := PA[i − S];   {address of control word of first block in segment i − S}
  if q ∈ segment (i − S) then mx := 1 else mx := 0;
  while q ∈ segment (i − S) do        {look for largest free block in segment i − S}
    begin
    mx := max(mx, V[q]);
    q := q + abs(V[q])
    end;
  {now update ST[i] and its ancestors in the segment tree}
  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 $i - S$ is freed, it must be merged with its left and/or right neighbors if they are free, and PA and ST must be updated appropriately. There are several cases, depending on whether the neighbors are in segment $i - S$ 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 the

following algorithm:

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

The algorithm assumes that block p has a predecessor, so we allocate a "sentinel" block of length 0 at the start of segment 0. This is illustrated in Figure 1. Note that a similar algorithm can be used to find the first word of a block, given the address of an arbitrary word within the block.

A Pascal implementation of Algorithm N is given in the Appendix. The Pascal implementation includes some refinements that were not mentioned in the description above. For example, S is a variable (initially 1), so the overheads of initializing and searching large segment trees are avoided if only a small fraction of the W words available are actually used. S is doubled when necessary, until it attains its maximum value $S_{\max}$ (which corresponds to S in the description above; i.e., $W \leq cS_{\max} < 2W$). The procedures of interest to users are blnew (which allocates a block), bldisp (which frees a block), blsize (which returns the size of a block), and blinit (which performs initialization).

4. WORST-CASE ANALYSIS OF ALGORITHM N

In this section we consider the worst-case space and time requirements of Algorithm N and compare Algorithm N with Algorithms A, M, and S (see Section 2). When comparing the space overheads of different algorithms, we count any space used outside the W words reserved for the dynamic storage area (e.g., the arrays PA and ST used by Algorithm N), as well as any reserved fields in the dynamic storage area (e.g., the control words used by Algorithm N). We do not count the space made available in allocated blocks (since this is common to all dynamic storage allocation algorithms) or the space occupied by free blocks (even if the free blocks are incorporated in some data structure, e.g., a tree or linked list). Thus, we are counting "internal" but not "external" fragmentation [16]. The external fragmentation depends on the dynamic storage strategy but not on the algorithm that implements it.

Suppose that R blocks have been allocated by Algorithm N and that S is as in Section 3. The space required by Algorithm N is $3S + R$ words ($3S$ for the arrays PA and ST, and one control word per allocated block). Recall that $cS < 2W$, so if $c = 200$ the space required for the arrays PA and ST is less than 3 percent of the space (W words) reserved for the dynamic storage area.

Let $s_{\min}$ be the size of the smallest block allocated (excluding the initial sentinel block of size 0). We can assume that $s_{\min} \geq 1$. Thus, the space overhead caused by the control words for each block is $R \leq W/(s_{\min} + 1) \leq W/2$ 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 some applications, e.g., allocation of nonvariant records in Pascal, the size can be determined at compile time.)

We now consider the time required by Algorithm N to allocate or free a block. The number of blocks starting in any segment is at most $\lceil \lceil W/S \rceil / (s_{\min} + 1) \rceil$, which is bounded by $\lceil c/(s_{\min} + 1) \rceil \leq \lceil c/2 \rceil$. To allocate or free a block requires at worst a small number of scans along the linked list of blocks in a segment and a smaller number of traversals of a branch of the segment tree. Thus, the number of operations is $O(\log S) + O(1) = O(\log W)$. (In fact, if S varies as in the implementation given in the Appendix, this bound may be reduced to $O(\log W_{\max})$, where $W_{\max} \leq W$ is the maximum number of words actually used in the dynamic storage area.)

For Algorithms A and S, the worst-case number of operations required to allocate or free a block is $O(F)$, 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. The average number of operations required by Algorithm A in most circumstances is not much better than the worst case, but for Algorithm S the average behavior may be appreciably better (see [18]).

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 hidden by the "O" notation is considerably larger than for Algorithm N (see Section 5).

5. IMPLEMENTATION AND COMPARISON OF ALGORITHMS N AND A

Algorithm N has been implemented in both Pascal and FORTRAN. The Pascal implementation is given in the Appendix. It has been used as part of a module that provides dynamic storage allocation (more efficiently than "new" and "dispose") for Pascal programs on a VAX® running under VMS (see [6]). The FORTRAN implementation was written as part of a package intended to make dynamic storage allocation readily available to FORTRAN library routines.

The motivation for the development of the FORTRAN dynamic storage allocation package was that it was needed to provide storage management for a multiple-precision interval arithmetic package and a single stack, as used in [7], was insufficient.

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 [9], on Univac 1100/82 and VAX 11/750 computers. 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 slightly 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 about 50 free blocks; note the "fifty percent" rule [12, 17]). Some statistics are given in Table I. Results for various other distributions of block sizes and lifetimes were similar to those given in Table I.

® VAX is a trademark of Digital Equipment Corporation.

Table I. Comparison of Algorithms A and N

Distribution[a]	Average number of free blocks F	Average time to allocate and free a block[b]		Ratio of times: Algorithm N/Algorithm A
		Algorithm A	Algorithm N	
(a)	30	0.94	1.17	1.24
(b)	120	2.40	1.53	0.64
(c)	480	8.09	1.73	0.22
(d)	1960	32.08	1.92	0.06

[a] 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; and blocksize uniform in 100 . . 1000, lifetime in 100 . . 200, with probability 0.1. Distribution (b) is the same as (a) except that lifetimes are multiplied by 4. Similarly for distributions (c) and (d), with factors of 16 and 64, respectively. In all cases the block sizes are in bytes but are rounded up to the next multiple of 4 to give an integral number of words (each of 4 bytes).

[b] Times are in milliseconds, based on 1,000,000 trials on a VAX 11/750. For Algorithm N we used the Pascal implementation given in the Appendix with $W = 2^{19} - 1$ words (each of 4 bytes) and $S = 2^{12}$ segments so each segment except the last was 128 words (i.e., 512 bytes or 1 page). In fact, $W_{max} < 2^{18}$, so at most 2^{11} segments were used.

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 are evident from the entries in Table I. Because of its complexity, Algorithm M has not been implemented, but timing of a priority queue implementation using "leftist trees" [13, Sect. 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). For similar reasons we would expect Algorithm S to be no faster than Algorithm N, even if the Cartesian tree remained well balanced. In the worst case, Algorithm S could be about as slow as Algorithm A.

6. A REALISTIC TEST OF ALGORITHM N

The block size and lifetime distributions used in Section 5 are artificial. Recently Bozman et al. [3] compared several dynamic storage allocation strategies using distributions obtained from monitoring large time-sharing systems. For a realistic test of Algorithm N, we used two of Bozman's empirical distributions:

(1) CAMBRIDG, using data obtained on an IBM 158 UP serving an average of 40–50 logged users at the IBM Cambridge Scientific Center in Cambridge, Massachusetts; and
(2) YKTVMV, using data obtained on an IBM 3033 MP serving an average of 450–540 logged users at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York.

In both cases the block sizes are multiples of 8 bytes in the range 8 to 4096 bytes. For each block size $8s$, the interarrival times and lifetimes are modeled by independent exponentially distributed random variables with empirically determined means IAT(s) and HT(s), respectively. The values IAT(s) and HT(s) are given in Tables 9 and 10 of [3]. For the CAMBRIDG data, the mean number of

Table II. Performance of Algorithm N with Bozman's distributions

Distribution	Algorithm	Mean items visited per request	Mean items visited per release	Storage efficiency
CAMBRIDG	Algorithm N	12.4	9.9	0.86
	Algorithm A[a]	213.3	212.1	0.84
YKTVMV	Algorithm N	12.5	9.7	0.89
	Algorithm A[a]	1678.9	1591.1	0.93

[a] Results for Algorithm A are from Bozman et al. [3].

requests per second for a block of storage is 198, the mean total size of current requests is 306K bytes, and the mean number of allocated blocks is 2356. For the YKTVMV data, these statistics are 1034, 2995K bytes, and 27,334, respectively.

To facilitate comparison with the results in Tables 3 and 4 of [3], we give in Table II the mean number of items visited per request for a block, the mean number of items visited per release of a block, and the storage efficiency. These terms are defined in [3]; in counting items visited, we have counted the number of references to control words in the dynamic storage area (i.e., the array V) but not the (comparable) number of references to the pointer array PA and the segment tree ST since the latter are likely to be present in the cache on the computers considered. "Storage efficiency" is the ratio of space requested to space used in the array V (including internal fragmentation due to control words and external fragmentation due to gaps between allocated blocks); the space used for the arrays PA and ST is not counted but is approximately 2.5 percent for a segment size of 512 bytes.

It is clear from Table II that Algorithm A is quite impractical for use in a large system. As Bozman et al. point out, the overhead per request increases roughly in proportion to the size of the system. However, Algorithm N is practical because the overhead per request increases only logarithmically with the size of the system. An even faster algorithm, discussed in [6], is a combination of Algorithm A and "subpooling," that is, keeping a "lookaside" list [2] of free blocks of a few popular sizes and resorting to Algorithm A for blocks of other sizes. However, the increase in speed may be at the expense of a decrease in storage efficiency (see [3, Tables 1–3]).

7. A MORE SECURE IMPLEMENTATION OF THE FIRST-FIT STRATEGY

Algorithm N is insecure in the sense that it fails if the block control words are accidentally or maliciously overwritten. To avoid this, we might keep the control words in a separate address space that could be protected from modification by the user's process. Algorithm N can easily be modified so that, instead of preceding the blocks to which they refer, the control words are kept in a separate singly linked list. The modified algorithm uses an additional array

```
CWP: array [0 .. S − 1] of integer;  {control word pointer array}
```

that is maintained so that CWP[i] points to the control word for the first block (if any) starting in segment i. To find the control word associated with a block in segment i, we search the linked list of control words, starting from the control

word at address CWP[i]; the addresses of the blocks in segment i are easily computed from PA[i] and the control words as they are scanned.

The price paid for security is an increase in the space overhead of the algorithm. We showed in Section 4 that the space overhead of Algorithm N is $3S + R$ words. The modified algorithm requires an additional S words for the array CWP, F control words associated with free blocks (since it is not secure to store these control words in the free blocks), and $F + R$ words for the links in the linked list of control words, making $4S + 2(F + R)$ words of overhead in all. Since $F + R \leq W/s_{\min}$, the additional overhead is not very significant unless $s_{\min}$ is small.

8. 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; and
(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 behavior of first-fit is usually about as good as that of best-fit, and the worst-case analysis clearly favors first-fit [15]. First-fit also appears to make better use of the available storage space than does the buddy system [3, 10–12, 14], unless the block sizes are restricted to favor 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 that grows downward. This facility has been included in the implementation [5]. Since allocation of space on a stack requires only constant time, it is desirable to use a stack where possible.

APPENDIX. Pascal Implementation of the First-Fit Strategy

```
{Pascal procedures implementing first-fit strategy.
 The procedures of interest to users are blnew, bldisp, blsize, and blinit.}
  const  w = 524287;          {size of dynamic storage area in words, may be in-
                               creased or decreased as desired}
         smax = 4096;         {power of 2, preferably about w/100}
         s21 = 8191;          {2 * smax - 1}
  type   seg = 0 .. smax;
         exseg = 0 .. s21;
         address = 0 .. w;
         dtype = record
                   v:  array [address] of integer;   {main DS area}
                   pa: array [seg] of address;       {pointer array}
                   st: array [exseg] of address;     {segment tree}
                   s:  seg;                          {segments in use}
                   wordsperseg: integer
                 end;
         dptr = ^dtype;
```

```
var    d: dptr;  {Global variable, used by first-fit procedures.
                  Note: d could be made an argument of each dynamic storage routine
                  to avoid use of a global variable.}

function blseg (p: address): exseg;

{Returns d^.s + (index of segment containing address p)}

  begin  {blseg}
  with d^ do blseg := s + (p div wordsperseg)
  end  {blseg};

procedure bldouble;

{Doubles number of segments actually in use, assuming current number is
 s < smax/2}

var  i, k: seg;

  begin  {bldouble}
  with d^ do
    begin
    for i := s to 2 * s - 1 do pa[i] := w;
    k := s;
      repeat
      for i := 0 to k - 1 do
        begin
        st[2 * k + i] := st[k + i];
        st[3 * k + i] := 0
        end;
      k := k div 2
      until k = 0;
    st[1] := st[2];
    s := 2 * s
    end
  end  {bldouble};

procedure blfix1 (p: address);

{Does housekeeping necessary if a block with control word v[p] will be allocated or
 merged with a block on its left in a different segment}

var  sister, mx, pj: address;
     pn: integer;
     j: exseg;

  begin  {blfix1}
  with d^ do
    begin
    j := blseg (p);
    if (v[p] <= 0) or (st[j] <= v[p]) then
      begin
      pj := pa[j - s];                    {index of first block in segment}
      pn := (j + 1 - s) * wordsperseg;    {start of next segment}
      if pn > w then pn := w;             {handle last segment}
      if pj < pn then
        begin
        mx := 1;  {there is a block starting in this segment}:
          repeat
          if mx < v[pj] then mx := v[pj];
          pj := pj + abs(v[pj])
          until pj >= pn
        end
```

```
        else mx := 0;  {no block starting in this segment}
        st[0] := 0;    {sentinel}
        while st[j] > mx do
          begin  {update segment tree}
          st[j] := mx;
          if odd(j) then sister := st[j - 1] else sister := st[j + 1];
          if mx < sister then mx := sister;
          j := j div 2  {go up branch of segment tree}
          end
        end
      end
    end  {blfix1};

  procedure blfix2 (p: address);

  {Does housekeeping necessary after a block with control word v[p] is freed or merged
   with a block on its right or created by splitting (with p on the right of the split)}

  var  vp: address;
       j: exseg;

    begin  {blfix2}
    with d^ do
      begin
      j := blseg (p);
      while j >= 2 * s do
        begin
        bldouble;
        j := blseg (p)  {s has changed}
        end;
      if pa[j - s] > p then pa[j - s] := p;
      vp := v[p];
      st[0] := vp;  {sentinel}
      while st[j] < vp do
        begin  {go up branch of segment tree}
        st[j] := vp;
        j := j div 2
        end
      end
    end  {blfix2};

  function blpred (p: address): address;

  {Returns predecessor of block p (always exists because of dummy first block);
   p and blpred are indexes of control words}

  var  q, qn: address;
       j: exseg;

    begin  {blpred}
    with d^ do
      begin
      j := blseg (p);
      if pa[j - s] = p then
        begin  {find rightmost nonempty segment to left}
        while st[j - 1] = 0 do j := j div 2;
        j := j - 1;
        while j < s do if st[2 * j + 1] > 0 then j := 2 * j + 1 else j := 2 * j
        end;
```

```
    qn := pa[j − s];
      repeat  {follow chain until reach p}
      q := qn;
      qn := qn + abs(v[qn])
      until qn = p;
    blpred := q
    end
  end  {blpred};

function blnew (size: address): address;

{Returns index of a block of at least size words, or 0 if no such block exists. Note
 that size excludes control word for block and may be zero. blinit must be called to
 perform initialization before calling blnew.}

var  j: exseg;
     n, p, vp: address;
     fix1: Boolean;

  begin  {blnew}
  with d^ do
    begin
    if st[1] <= size then
      blnew := 0   {size too large}
    else
      begin  {find first segment containing large enough free block}
      n := size + 1;   {n is length including control word}
      j := 1;
      while j < s do
        if st[2 * j] >= n then j := 2 * j else j := 2 * j + 1;
      p := pa[j − s];   {index of control word of first block in segment j}
      while v[p] < n do p := p + abs(v[p]);
      {Now p is index of control word of required block}
      vp := v[p];
      v[p] := −n;              {flag block as allocated}
      fix1 := (vp = st[j]);  {blfix2 may change st[j]}
      if vp > n then
        begin                  {split block}
        v[p + n] := vp − n;
        if blseg (p + n) > j then
          blfix2 (p + n)   {necessary as p + n in another segment}
        end;
      if fix1 then blfix1 (p);   {necessary to update st here}
      blnew := p + 1   {return index of first word after control word}
      end
    end
  end  {blnew};

procedure bldisp (va: address);

{Disposes of a block obtained using blnew. va is the index that was returned by
 blnew.}

var  p, pr: address;
     j, jn: exseg;
     pn, vp: integer;

  begin {bldisp}
  p := va − 1;      {index of control word}
  vp := −d^.v[p];   {block already free if d^.v[p] > 0}
  if vp > 0 then with d^ do
```

```
      begin
      v[p] := vp;  {flag as free}
      j := blseg (p);
      pn := p + vp;
      if pn < w then if v[pn] >= 0 then
        begin  {next block free, so merge}
        v[p] := vp + v[pn];
        jn := blseg (pn);
        if jn > j then
          begin
          pa[jn - s] := p + v[p];
          blfix1 (pn)
          end
        end;
      pr := blpred (p);  {preceding block}
      if v[pr] >= 0 then
        begin  {preceding block free, so merge}
        v[pr] := v[pr] + v[p];
        if pa[j - s] = p then
          begin  {adjust pointer to first block in segment j}
          pa[j - s] := pr + v[pr];
          blfix1 (p)
          end;
        blfix2 (pr)
        end
      else if v[p] > st[j] then blfix2 (p)
      end
    end  {bldisp};

  function blsize (va: address): integer;

  {Returns size of a block allocated by call to blnew}

    begin  {blsize}
    with d^ do
      {Error if v[va - 1] <= 0}
      blsize := -(v[va - 1] + 1)
    end  {blsize};

  procedure blinit;

  {Does initialization of segment tree, etc.
   Must be called before calling any other bl ... routine}

  var  dummy: address;

    begin  {blinit}
    new (d);  {get space for dynamic storage area}
    with d^ do
      begin
      s := 1;  {may be increased by bldouble}
      st[1] := w;
      pa[0] := 0;
      v[0] := w;  {one large free block v[1 ... w - 1]}
      wordsperseg := (w div smax) + 1;
      dummy := blnew (0)  {create dummy block as sentinel}
      end
    end  {blinit};
```

ACKNOWLEDGMENTS

I wish to thank J. B. Hext, D. E. Knuth, and J. M. Robson for their comments on a preliminary version [4] of this paper.

REFERENCES

1. BAYS, C. A comparison of next-fit, first-fit and best-fit. *Commun. ACM 20*, 3 (Mar. 1977), 191–192.
2. BOZMAN, G. The software lookaside buffer reduces search overhead with linked lists. *Commun. ACM 27*, 3 (Mar. 1984), 222–227.
3. BOZMAN, G., BUCO, W., DALY, T. P., AND TETZLAFF, W. H. Analysis of free-storage algorithms—revisited. *IBM Syst. J. 23*, 1 (1984), 44–64.
4. BRENT, R. P. Efficient implementation of the first-fit strategy for dynamic storage allocation. *Australian Comput. Sci. Commun. 3*, 1 (May 1981), 25–34.
5. BRENT, R. P. Efficient implementation of the first-fit strategy for dynamic storage allocation. Rep. CMA-R33-84, Centre for Mathematical Analysis, Australian National University, Aug. 1984.
6. BRENT, R. P. Dynamic storage allocation on a computer with virtual memory. Rep. CMA-R37-84, Centre for Mathematical Analysis, Australian National University, Sept. 1984.
7. FOX, P. A., HALL, A. D., AND SCHRYER, N. L. The PORT mathematical subroutine library. *ACM Trans. Math. Softw. 4*, 2 (June 1978), 104–126.
8. GEROVAC, B. J. An implementation of new and dispose using boundary tags. *Pascal News 19* (Sept. 1980), 49–59.
9. HEXT, J. B. A storage management laboratory. *Aust. Comput. Sci. Commun. 2*, 1 (Jan. 1980), 185–193.
10. KAUFMAN, A. Tailored-list and recombination-delaying buddy systems. *ACM Trans. Program. Lang. Syst. 6*, 1 (Jan. 1984), 118–125.
11. KNOWLTON, K. C. A fast storage allocator. *Commun. ACM 8*, 10 (Oct. 1965), 623–625.
12. KNUTH, D. E. *The Art of Computer Programming. Vol. 1: Fundamental Algorithms.* (2nd edition). Addison-Wesley, Reading, Mass., 1973, Sect. 2.5.
13. KNUTH, D. E. *The Art of Computer Programming. Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
14. PETERSON, J. L., AND NORMAN, T. A. Buddy systems. *Commun. ACM 20*, 6 (June 1977), 421–431.
15. ROBSON, J. M. Worst case fragmentation of first fit and best fit storage allocation strategies. *Comput. J. 20*, 3 (Aug. 1977), 242–244.
16. SHORE, J. E. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Commun. ACM 18*, 8 (Aug. 1975), 433–440.
17. SHORE, J. E. Anomalous behavior of the fifty-percent rule in dynamic memory allocation. *Commun. ACM 20*, 11 (Nov. 1977), 812–820.
18. STEPHENSON, C. J. Fast fits—New methods for dynamic storage allocation. In *Proceedings of the 9th Symposium on Operating System Principles* (Bretton Woods, N.H., Oct. 11–13, 1983). ACM, New York, 1983, pp. 30–32.
19. VUILLEMIN, J. A unifying look at data structures. *Commun. ACM 23*, 4 (Apr. 1980), 229–239.

Received August 1984; revised September 1985; accepted October 1985