Operating Systems

R.S. Gaines Editor

Buddy Systems

James L. Peterson
The University of Texas at Austin

Theodore A. Norman
Brigham Young University

Two algorithms are presented for implementing any of a class of buddy systems for dynamic storage allocation. Each buddy system corresponds to a set of recurrence relations which relate the block sizes provided to each other. Analyses of the internal fragmentation of the binary buddy system, the Fibonacci buddy system, and the weighted buddy system are given. Comparative simulation results are also presented for internal, external, and total fragmentation.

Key Words and Phrases: dynamic storage allocation, buddy system, fragmentation, Fibonacci buddy system, weighted buddy system

CR Categories: 3.89, 4.32, 4.39

This research was supported, in part, by the National Science Foundation under Grant Number MCS75-16425. Authors' addresses: J.L. Peterson, Department of Computer Sciences, The University of Texas at Austin, Austin TX 78712; T.A. Norman, Department of Computer Science, Brigham Young University, Provo, UT 84601.

1. Introduction

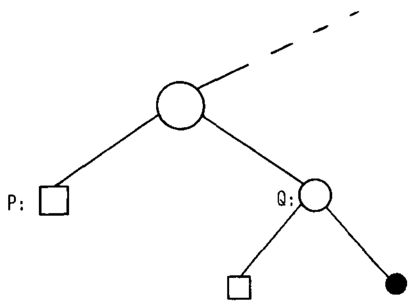
Two dynamic storage allocation algorithms derived from the buddy system have recently been proposed. Knowlton [5] and Knuth [6] described the original buddy system. This memory management scheme allocates blocks whose sizes are powers of 2. (In this paper, we call this system the *binary buddy system* to distinguish it from the other buddy systems considered.) Hirschberg [4], taking Knuth's suggestion [6, problem 2.5.31] has designed a *Fibonacci buddy system* with block sizes which are Fibonacci numbers. Shen and Peterson [12] have described an algorithm for a *weighted buddy system* which provides blocks whose sizes are 2^k and $3 \cdot 2^k$.

These three buddy systems are similar in the overall design of the algorithm, with the major differences being in the sizes of the memory blocks provided and the consequent address calculation for locating the buddy of a released block. The address calculation for the binary and weighted buddy systems is straightforward, but the original procedure for the Fibonacci buddy system was either limited to a small, fixed number of block sizes or a time consuming computation [4]. A recent note by Cranston and Thomas [1] has removed this problem and made the address calculation for the Fibonacci buddy system comparable with the address calculation for the binary or weighted buddy systems.

Another important variation among these three buddy systems is in memory utilization. Buddy systems suffer from both internal and external fragmentation. *Internal fragmentation* is the result of allocating memory only in predefined block sizes. A request for a block of memory which is not one of these specified block sizes must be satisfied by allocating the next larger block size, with a resulting loss in available memory. *External fragmentation* results from breaking available memory into blocks which can be recombined only if they are buddies. Thus a request for memory may have to be rejected because no single block is large enough, although the total amount of available memory (in smaller blocks) may be sufficient to satisfy the request many times over.

The amount of internal and external fragmentation in a buddy system depends upon the distribution of requests for memory which must be satisfied and the block sizes provided. For a particular distribution, one buddy system may have lower fragmentation than the other systems, while the situation may be reversed for another distribution. Since it is generally not easy to change the memory distribution to match the allocation strategy, it would be useful to have available a class of dynamic storage allocation algorithms. For a particular problem, an algorithm could be selected from this class to minimize fragmentation and hence maximize memory utilization. Hirschberg [4] has suggested that such a class of algorithms could be defined to allocate block

Fig. 1. The block at the address of the buddy of the block at P is available, but the buddy of the block at P (at Q) is not available. Squares (□) indicate available blocks; circles indicate blocks reserved by the user (●) or blocks split into buddies (○).



sizes which satisfy the following recurrence relation:

$$L_i = L_{i-1} + L_{i-k}, \quad k > 0.$$

For each value of k, a new buddy system is defined. ($k = 1$ is the binary buddy system; $k = 2$ is the Fibonacci buddy system.) The weighted buddy system does not satisfy the above recurrence relation, however, so this class appears to be too restrictive. In general, a buddy system can be based upon any sequence of numbers which satisfy a set of n recurrence relations with the form,

$$L_i \geq L_{i-1} + L_{\beta(i)}, \quad i = 1, \ldots, n, \; L_0 = 0. \tag{1}$$

where β is any function over the positive integers with $\beta(i) < i$.

In Section 2, we present an algorithm for the request and release procedures which can be used to implement any buddy system whose blocks sizes satisfy the set of recurrence relations (1). In Section 3, we discuss an even more general class of buddy systems and their implementation. Section 4 presents some analytic results on the expected internal fragmentation for a uniform distribution of requests, while Sections 5 and 6 investigate both internal and external fragmentation for several buddy systems by means of simulation. Section 7 summarizes and presents some conclusions concerning the advantages and disadvantages of the buddy systems for dynamic storage allocation.

2. Generalized Buddy Algorithm

Let $L_1, L_2, \ldots, L_n$ be the set of block sizes provided by the buddy system such that these block sizes satisfy the set of n recurrence relations (1) for a function β with $L_1 < L_2 < \cdots < L_n$. For a block size L_i, we call i its *size index*. The generalized buddy system will split a block of size L_i into two blocks of size L_{i-1} and $L_{\beta(i)}$.

The major data structure for the generalized buddy system is its *available space list*. The available space list is an n-vector which is indexed by the size index of a block. The ith element of the available space list is a record with a *HEAD* and *TAIL* field pointing to the front and rear of a doubly linked list of all available blocks of size L_i. Other fields may be present in the available space list elements (such as a field which records the number of available blocks of size L_i). In particular, L_i and $\beta(i)$ may be stored as fields of an element of the available space list. This data structure, or any of its fields, may be implemented as separate parallel n-vectors rather than as a vector of records if necessary for efficient accessing.

The generalized buddy algorithm can now be stated. For a request for a block of size index i, the request procedure is:

Q1. Search up the available space list from the ith entry for the smallest available block (of size index j at location P) such that $j \geq i$. If no block of sufficient size is available, memory overflow has occurred, and the appropriate action must be taken. If such a block exists, remove it from the available space list, and continue to step Q2.

Q2. While $j > i$, split the block at location P of size index j into two buddies: (1) a *left* buddy at location P of size index $\beta(j)$ and (2) a *right* buddy at location $P + L_{\beta(j)}$ of size index $j - 1$. Reset P and j to specify the smaller of the two buddies which is large enough to satisfy the request, and attach the other to the available space list. (i.e. if $i \leq \beta(j)$ then $(P, j) \leftarrow (P, \beta(j))$ else $(P, j) \leftarrow (P + L_{\beta(j)}, j - 1)$.)

Q3. When $j = i$, allocate the block at location P.

For a release of a block at location P of size index i, the release procedure is:

L1. Set $j \leftarrow i$.

L2. While the buddy of the block at location P of size index j is available,
  (a) Remove the buddy from its available space list.
  (b) Recombine the block at location P of size index j and its buddy. Reset P and j to specify the block which results from this recombination. (i.e. if the buddy of (P, j) is (Q, l), then $(P, j) \leftarrow (\min(P, Q), 1 + \max(j, l))$.)

L3. When the block at location P, of size index j, cannot be recombined with its buddy (because the buddy is not available), attach the block at location P to the available space list for blocks of size index j.

The determination of the availability of the buddy of a block is the central computation of a buddy system. It involves first calculating the address of the buddy and then determining that the block at that address is (1) available and (2) the desired buddy. Figure 1 illustrates this problem.

To aid in the computation of the availability of the buddy of a block, we define three fields which are stored in each block in the buddy system:

(1) A *TAG* field, a Boolean value which records the available (*TAG* = 0) or allocated (*TAG* = 1) status of the block.

(2) A *TYPE* field, a two bit value (ab) which specifies by its first bit (a) whether this block is a left ($a = 0$) or right ($a = 1$) buddy. The second bit (b) records the first (if $a = 0$) or second (if $a = 1$) bit of the *TYPE* field of the parent block of this block. This allows the *TYPE* field of the parent block to be redefined when this block and its buddy are recombined. (The definition of this field is due to Cranston and Thomas [1].)

(3) An *INDEX* field, specifying the size index of the block. If n block sizes are provided by the buddy system, then $\lceil \log_2 n \rceil$ bits are needed for this field.

On most machines, for many buddy systems, these fields can be packed into the first word of the block.

Using these fields, we can now define the computation of the address, Q, of the buddy of a block at location P with size index j, as

$$\begin{aligned} Q &= P + L_j, & \text{if } TYPE(P) = 0b, \\ &= P - L_{\beta(j+1)}, & \text{if } TYPE(P) = 1b. \end{aligned}$$

The block at Q is the buddy of the block at P only if the buddy of P has not been split into subblocks. If P is a left buddy ($TYPE(P) = 0b$), then the block at Q is the buddy of P if (and only if) the block at Q is a right buddy ($TYPE(Q) = 1b$). All subblocks of Q which are at location Q are left buddies. If P is a right buddy ($TYPE(P) = 1b$), then the block at Q is the buddy of P if and only if the size index field of Q, $INDEX(Q)$, is equal to $\beta(j + 1)$, where j is the value of the $INDEX$ field of the block at location P. The size index of the parent of the block at location P is $j + 1$ and $\beta(j + 1)$ is the size index of the buddy of the block at location P.

This completes the description of the generalized buddy system algorithm. For special cases of the β function, more specific and more efficient algorithms can be defined (such as in [6] for the binary buddy system), but the algorithm just described will work for any β function. The Appendix lists a PASCAL version of the request and release procedures, where *size*[*i*] is L_i and *subbuddy*[*i*] is $\beta(i)$.

The algorithm described was designed to allocate memory from a large initially available block of size M, addressed from 0 to $M - 1$, where $M = L_n$. In this case the available space list is initialized to indicate one available block, of size index n at address 0. If the size of the initial block M is not one of the defined block sizes $L_1, L_2, \ldots, L_n$, then we initialize the available space list to indicate that a set of blocks of sizes $L_{j_1}, L_{j_2}, \ldots, L_{j_l}$ is available at locations $0, L_{j_1}, L_{j_1} + L_{j_2}, \ldots, \sum_{i=1}^{l-1} L_{j_i}$, respectively, with $M = \sum_{i=1}^{l} L_{j_i}$. The $TYPE$ fields are set to indicate all blocks as left buddies ($0b$). This prevents the release routine from trying to combine these blocks when any of them became available. If an address of a buddy is generated which is greater than $M - 1$, it is treated as a buddy which is not available. As an example, an initial block of size 25 would provide initial blocks of size 16, 8, and 1 at locations 0, 16, and 24, respectively, for the binary buddy system, or of sizes 21, 3, and 1 at locations 0, 21, and 24, respectively, for the Fibonacci buddy system.

Block sizes can be in any unit of storage (bytes, halfwords, words, doublewords, etc.) since, if $L_1, L_2, \ldots, L_n$ is a solution to a set of recurrence relations (1), then $\gamma \cdot L_1, \gamma \cdot L_2, \ldots, \gamma \cdot L_n$ is also a solution for constant γ. An initial block whose absolute starting address is L, rather than zero, can be used in a buddy system by considering all addresses to be relative to location L. Other similar minor variations to the basic buddy system are also possible.

3. An Even More General Class of Buddy Systems

The major problem in the algorithm described in Section 2 was the computation of the address of the buddy of a released block. An alternative approach to the solution given in Section 2 is to store the address of the buddy of a block explicitly in the block when the two buddies are created. Thus, if P and Q are buddies, each will contain a pointer pointing to the other. This can be extended to allow any number of buddies to be created from a block by linking them in a circularly linked list. However, now when a block is split into subblocks and these blocks are recombined, we must be able to recreate the pointer to the buddies of the recombined block. One solution would be to break the pointer up into parts which are stored in the header of its subblocks, as was done with the $TYPE$ field of the algorithm presented in Section 2. If sufficient room exists in the header word of each block, this change is a minor variation of the algorithm of Section 2.

Another approach is to retain the header of a block when it is split into subblocks. If μ storage units are needed for a header containing a TAG field (to indicate available/allocated status), an $INDEX$ field (to store a size index), and the pointer to the next buddy in the circularly linked list, then a buddy system can be designed for any set of block sizes $L_1, L_2, \ldots, L_n$ which satisfy a set of recurrence relations to the form

$$L_i \geq \mu + L_{j_{i,1}} + L_{j_{i,2}} + \cdots + L_{j_{i,l(i)}}, \quad i = 1, \ldots, n$$

with the restriction that $j_{i,r} < i$ for all r, $1 \leq r \leq l(i)$. A block of size index i at location P is split into a header word (of length μ at location P) and $l(i)$ buddies, circularly linked through their header words, of size indices, $j_{i,1}, j_{i,2}, \ldots, j_{i,l(i)}$ at locations $P + \mu, P + \mu + L_{j_{i,1}}, \ldots, P + \mu + \sum_{r=1}^{l(i)-1} L_{j_{i,r}}$, respectively. If the pointer is dismembered and stored in the header words of the subblocks as mentioned above, we have $\mu = 0$. It would be necessary to require that $j_{i,r} = i - 1$ for some r (for each i) because of the search policy in the request procedure, but a more complicated search policy might be able to remove this constraint also for some buddy systems. (The problem is; If a block of size index i is requested and not immediately available, where do we look for a block which can be split to produce a block of size index i? In the algorithm of Section 2, we look at $i + 1, i + 2$, etc. If a block of size index $i + 1$ is available, it makes sense to use this to produce the requested block of size index i.)

An example of the use of this algorithm is a system which requires blocks of sizes 12, 80, and 132, the first for a control block, the second for a card image buffer, and the third for a line printer image buffer. Figure 2 illustrates how blocks can be split in a system with $\mu =$

2, starting from a block of size 136. Under the binary buddy system, it would be necessary to allocate blocks of sizes 16, 128, and 256 with large internal fragmentation. The weighted buddy system does somewhat better on internal fragmentation with blocks of sizes 16, 96, and 192. (Remember that the header word in these systems takes at least the first word; so to have 12 usable words, at least 13 must be allocated.) The Fibonacci buddy has very low internal fragmentation with block sizes of 13, 89, and 144, but with an initial block of size 144, only 8 blocks of size 13 can be created, while 9 blocks of size 14 can be created from an initial block of size 136 in the pointer buddy system of Figure 2.

4. Internal Fragmentation

Unless the set of requested block sizes is a subset of the set of provided block sizes, it will be necessary to allocate more memory than is requested for some requests. The memory wasted due to this overallocation is internal fragmentation. The amount of internal fragmentation will vary depending upon the set of provided block sizes and the distribution of requests for memory. Thus it can be used as a point of comparison for buddy systems.

A measure of internal fragmentation can be defined in several ways. Several researchers [4, 7, 11] have considered the ratio of average allocated space to average requested space. This measure is difficult to compare with external fragmentation in order to compute total fragmentation, however. Another suggested measure is the ratio of overallocated memory (average allocation minus average request) to total memory size M, but this results in a measure which is a function of both the buddy system and the memory size. We have chosen to use a measure of internal fragmentation which is the ratio of overallocated memory to allocated memory. This measure is a function of the buddy system and the request distribution. By knowing the proportion of total memory which is allocated, the ratio of overallocated memory to total memory can be computed.

Letting $\lambda(i)$ be the size of the allocated block for a request of size i and ρ_i be the probability of a request of size i, we have the internal fragmentation for a request distribution with requests for blocks of memory in the range 1 to m defined by

$$\sum_{i=1}^{m} \rho_i(\lambda(i) - i)\Big/\sum_{i=1}^{m} \rho_i\lambda(i).$$

Notice that since we are interested in comparing the utilization of memory of buddy systems, the probability ρ_i is the probability of finding a block of size i in memory. This probability will, in general, differ from the probability of a request for a block of size i. The probability of a block being allocated in memory is influenced not only by the request distribution and the memory management scheme, but also by the size of memory and the lifetime distribution for requests. If request lifetimes are independent of request sizes and the size of memory is large relative to the maximum request size, however, the difference between these two probabilities will be minimal.

In order to compare the performances of the previously published buddy systems (binary, Fibonacci, and weighted), the internal fragmentation for a uniform probability distribution has been investigated. A uniform distribution of blocks in memory, or even a uniform request distribution, is admittedly not very realistic, but it is mathematically tractable, and the relative performance of buddy systems under this distribution is believed by the authors to be indicative of the performance under other distributions. The simulation results of Section 5 for both a uniform distribution and a (truncated) exponential distribution and of Section 6 for three actual request distributions support this belief, but a major motivation for examining the uniform distribution is its mathematical tractability and the fact that real memory distributions tend to be very system specific and hard to work with analytically.

For a uniform distribution of requests from 1 to m, the average internal fragmentation is

$$(\lambda_m - a_m)/\lambda_m = 1 - a_m/\lambda_m,$$

where

$$a_m = 1 + 2 + 3 + 4 + \cdots + m = \sum_{i=1}^{m} i = (m^2 + m)/2,$$

$$\lambda_m = \lambda(1) + \lambda(2) + \lambda(3) + \lambda(4) + \cdots + \lambda(m),$$

with $\lambda(i)$ being the size of the block which is allocated for a request of size i ($\lambda(i) = L_k$ such that $L_{k-1} < i \leq L_k$). Noting that $\lambda(i)$ is constant for $L_{k-1} < i \leq L_k$, we can express the sum λ_m, for $L_k \leq m \leq L_{k+1}$, as

$$\lambda_m = L_1^2 + \sum_{i=2}^{k} L_i \cdot (L_i - L_{i-1}) + \alpha(L_{k+1} - L_k)\cdot L_{k+1},$$

with $m = L_k + \alpha(L_{k+1} - L_k)$, $0 \leq \alpha \leq 1$. The parameter α indicates how close m is to L_k or L_{k+1}. For buddy systems based on the set of recurrence relations (1), this is then

$$\lambda_m = L_1^2 + \sum_{i=2}^{k} L_i \cdot L_{\beta(i)} + \alpha L_{k+1} \cdot L_{\beta(k+1)}.$$

The Fibonacci-like sequences which were suggested as the basis of a class of buddy systems by Hirschberg [4] have been studied by Harris and Styles [3] and Ferguson [2]. If we define

$$u_i = 0, \quad \text{for } i < 0, \qquad u_0 = 1,$$
$$u_{i+1} = u_i + u_{i-p}, \quad \text{for } i \geq 0,$$

then we have $L_i = u_i$ with

$$\beta(i) = i - p - 1, \quad i > p$$
$$\quad = 0, \qquad i \leq p.$$

Fig. 2. Tree structure for a pointer buddy system with $\mu = 2$, providing usable blocks of sizes 12, 80, 132.

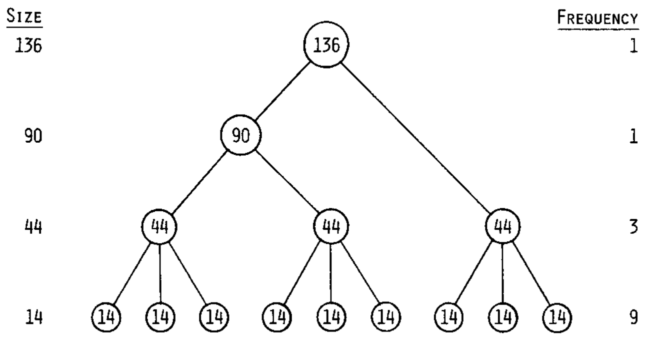


Table I. Asymptotic values of the maximum and minimum internal fragmentation for a uniform distribution of requests from 1 to m (as $m \to \infty$) for buddy systems based on the recurrence relations $u_i = u_{i-1} + u_{i-p-1}$.

		Internal fragmentation	
p	ϕ	max	min
0	2.000	0.333	0.250
1	1.618	0.236	0.191
2	1.466	0.189	0.159
3	1.380	0.160	0.138
4	1.325	0.140	0.123
5	1.285	0.125	0.111
6	1.255	0.113	0.102
7	1.232	0.104	0.094
8	1.213	0.096	0.088
9	1.197	0.090	0.082
10	1.184	0.084	0.078

(For $p = 0, u_i = 2^i$; for $P = 1, u_i = F_{i+1}$). Then

$$\lambda_m = 1 + \sum_{i=0}^{k-p-1} u_i \cdot u_{i+p+1} + \alpha\, u_{k+1} \cdot u_{k-p}$$
$$m = u_k + \alpha\, u_{k-p},\ 0 \le \alpha \le 1.$$

Harris and Styles [3] have investigated the sequences u_i and proved many useful summation and product formulas. They also show that the limit, as $k \to \infty$, of u_{k+1}/u_k is the largest real root of the equation $x^{p+1} - x^p - 1 = 0$. Ferguson [2] gives numerical values for these roots.

If we let ϕ be the largest real root of $x^{p+1} - x^p - 1 = 0$ $(1 \le \phi \le 2)$, then u_i may be approximated by $c \cdot \phi^i$ for an appropriate choice of c. (For $p = 0, u_i = 1 \cdot 2^i$; for $p = 1, u_i = F_{i+1} \simeq \phi^{i+1}/\sqrt{5}$.) The approximation is quite good even for small i owing to the form of the zeros of $x^{p+1} - x^p - 1$[2]. With this approximation,

$$\begin{aligned}\lambda_m &\simeq 1 + \sum_{i=0}^{k-p-1} c^2\phi^{2i+p+1} + \alpha c^2\phi^{2k-p+1} \\ &= 1 + c^2\,(\phi^{2k-p+1} - \phi^{p+1})/(\phi^2 - 1) \\ &\quad + \alpha c^2\phi^{2k-p+1} \\ &= 1 + [c^2/(\phi^2 - 1)]\cdot[(1 + \alpha(\phi^2 - 1))\cdot\phi^{2k-p+1} \\ &\quad - \phi^{p+1}].\end{aligned}$$

$$\begin{aligned}a_m &= \tfrac{1}{2}(m^2 + m) \\ &= \tfrac{1}{2}\,[c^2\,\phi^{2k} + 2\alpha c^2\,\phi^{2k-p} + \alpha^2\,c^2\,\phi^{2k-2p} + c\phi^k \\ &\quad + \alpha c\phi^{k-p}].\end{aligned}$$

And internal fragmentation is

$$1 - a_m/\lambda_m =$$

$$1 - \frac{(\phi^2-1)[\phi^{2k}+2\alpha\phi^{2k-p}+\alpha^2\phi^{2k-2p}+(\phi^k+\alpha\phi^{k-p})/c]}{2(\phi^2-1)/c^2+[1+(\phi^2-1)\alpha]\cdot\phi^{2k-p+1}-\phi^{p+1})},$$

and, as $k \to \infty$,

$$\begin{aligned}&= 1 - \frac{(\phi^2 - 1)(\phi^{2p} + 2\alpha\phi^p + \alpha^2)}{2[1 + \alpha(\phi^2 - 1)]\phi^{p+1}} \\ &= 1 - \frac{(\phi^2 - 1)(\phi^p + \alpha)^2}{2\phi(\phi^p + \alpha[\phi^{p+2} - \phi^p)]} \\ &= 1 - \frac{(\phi^2 - 1)(\phi^p + \alpha)^2}{2\phi(\phi^p + \alpha\phi + \alpha)},\quad 0 \le \alpha \le 1.\end{aligned}$$

This function of α has a maximum value of $(\phi - 1)/(\phi + 1)$ when $\alpha = 1/(\phi + 1)$, and a minimum value of $(\phi - 1)/2\phi$ when $\alpha = 0$ or $\alpha = 1$. Table I lists the values of ϕ and the minimum and maximum internal fragmentation for $p = 0, \ldots, 10$. The binary buddy system (corresponding to p = 0) has an internal fragmentation of 25 to 33 percent of allocated memory. The Fibonacci buddy system (p = 1) suffers from 19 to 23.6 percent wasted memory due to internal fragmentation. This agrees with the computations of Knuth [6] (for the binary buddy) and Russell [11] (for the Fibonacci buddy).

The block sizes of the weighted buddy system do not correspond to any of the systems whose internal fragmentation is given in Table I. Internal fragmentation for the weighted buddy system requires the analysis of two cases:

(1) $2^k \le m \le 3 \cdot 2^{k-1},\ m = \alpha 2^k,\ 1 \le \alpha \le \frac{3}{2}$,
(2) $3 \cdot 2^{k-1} \le m \le 2^{k+1},\ m = \alpha 2^k,\ \frac{3}{2} \le \alpha \le 2$.

As $m \to \infty$ in these cases, internal fragmentation becomes

(1) $1 - 2\alpha^2/(6\alpha - \frac{11}{3})$
$= (6\alpha^2 - 18\alpha + 11)/(11 - 18\alpha),\ 1 \le \alpha \le \frac{3}{2}$

(2) $1 - 2\alpha^2/(8\alpha - \frac{20}{3})$
$= (3\alpha^2 - 12\alpha + 10)/(10 - 12\alpha),\ \frac{3}{2} \le \alpha \le 2.$

Internal fragmentation for the weighted buddy system has a maximum value of $\frac{5}{27}$ (0.185) at $\alpha = \frac{11}{9}$ and a minimum value of $\frac{1}{7}$ (0.143) at $\alpha = 1$ or $\alpha = 2$. A local maximum of $\frac{1}{6}$ (0.167) occurs at $\alpha = \frac{5}{3}$, and a local miniminum of $\frac{5}{32}$ (0.156) occurs at $\alpha = \frac{3}{2}$.

From these calculations, we see that the weighted buddy system always has lower internal fragmentation than the Fibonacci buddy system, which always has lower internal fragmentation than the binary buddy system. Because of the similar block sizes for the binary and the weighted buddy systems, we can compare their internal fragmentation directly to show that the binary buddy system has from 2.08 ($m = 3 \cdot 2^k$) to 1.72 ($m = 1.08 \cdot 2^k$) times more internal fragmentation than the weighted buddy system. Clearly the weighted buddy system performs much better than either the binary or Fibonacci buddy systems in terms of internal fragmentation.

It must be emphasized, however, that these results are valid only for the particular theoretical distribution considered here. For a real distribution, internal fragmentation may be considerably different depending upon the "fit" of the provided block sizes to the requested block sizes.

5. External Fragmentation, Total Fragmentation, and Execution Time

Internal fragmentation is not the only measure of memory utilization, however. External fragmentation can also decrease the effective size of available memory. Unlike internal fragmentation, which occurs continuously in a buddy system, it is a matter of definition whether external fragmentation can be said to occur before a request for memory must be rejected because all available blocks are of insufficient size (i.e. before memory overflows). A measure of external fragmentation is the proportion of total memory which is available when overflow occurs. This measure depends upon the specific sequence of requests and releases which precede overflow, and is therefore difficult to deal with analytically, although attempts to analyze other measures of external fragmentation have been made [10].

Internal and external fragmentation result from different properties of the buddy system, but both decrease the effective size of the available memory which is being managed by making portions of that memory unusable. We define *total fragmentation* of a buddy system to be the total amount of memory which is unusable due to either internal or external fragmentation (normalized by dividing by the total memory size). Since our definition of internal fragmentation is the proportion of allocated memory which is unusable, while external fragmentation is a proportion of total memory, total fragmentation is not a simple sum of internal and external, but rather,

$$\begin{aligned}\text{total} &= (1 - \text{external}) \cdot \text{internal} + \text{external} \\ &= \text{internal} + \text{external} - \text{internal} \cdot \text{external}.\end{aligned}$$

Another important property for a buddy system is its running time. The original advantage of the buddy system over first-fit or best-fit memory management schemes was its reduction in search time to find and allocate an available block of the appropriate size. Three statistics are important in a buddy system. In the algorithm description of Section 2, these are the number of times that steps Q1 (the number of searches), Q2 (the number of splits), and L2 (the number of recombinations) are executed. In equilibrium, the number of splits will be equal to the number of recombinations; so only two statistics are needed. Also, the number of splits is always less than or equal to the number of searches, and if, as would be hoped, the number of searches is normally 0 or 1, these two statistics will also be equal. (The discrepancy between the number of searches and the number of splits occurs when a block of size L_j is split into subblocks of sizes L_{j-1} and $L_{\beta(j)}$, and the block of size $L_{\beta(j)}$ is used to continue the splitting. In the binary buddy system, $\beta(j) = j - 1$, and the two statistics are equal.)

A simulation of four buddy systems (binary, Fibonacci, weighted, and the F-2 buddy system based on the recurrence relation $L_{i+1} = L_i + L_{i-2}$) was used to obtain comparative values of internal, external, and total fragmentation as well as the average number of searches, splits, and recombinations. Request and release procedures for a general buddy system were programmed. Since we wished to obtain both internal and external fragmentation figures, the memory management schemes were driven at overflow in the simulation. A 31-bit uniform pseudo-random number generator was used to produce an unbounded sequence of requests according to either a uniform distribution from 1 to m or a (truncated) exponential distribution with mean $m/2$. Associated with each request was a uniformly distributed block lifetime. If a block was allocated at time τ, then it was queued for release at τ plus the lifetime of the block. Requests were made until overflow occurred. Then blocks were released (and the simulation timer incremented as needed) until the block which caused overflow could be successfully requested; the system then returned to making requests until overflow occurred again. This process of alternately requesting and releasing blocks continued until a fixed number of (simulated) time units had elapsed. At regular intervals, statistics on internal, external, and total fragmentation were taken. The number of searches, splits, and recombinations was also recorded. Identical request sequences were given to all buddy systems.

For our simulations, a memory of 10,000 words was simulated. The block lifetimes were uniformly distributed from 1 to 10. The simulations continued for 4000 time units at first, and, as our budget became tighter, for 2000 time units (with no significant change in results). The uniform request distributions were investigated as m, the maximum request size, varied from 40 to 1000 in steps of 10.

For the exponential request distribution, the distribution was truncated to generate requests which were less than 1000 with a mean which varied from 50 to about 400 (to match the mean of the uniform distribution) by steps of 25 (due to budget pressures). The actual mean did not vary by steps of 25, but increased more slowly owing to the effect of discarding requests greater than 1000. Thus the mean of the last generating

Fig. 3. Internal fragmentation for a uniform request distribution.

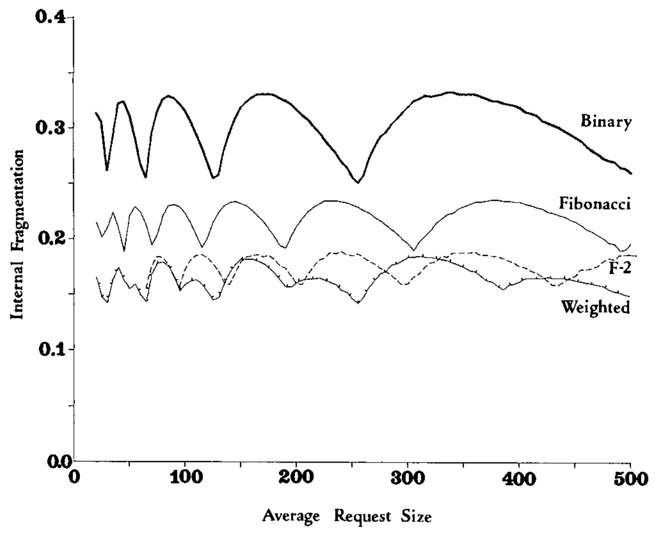


Fig. 5. External fragmentation for a uniform request distribution.

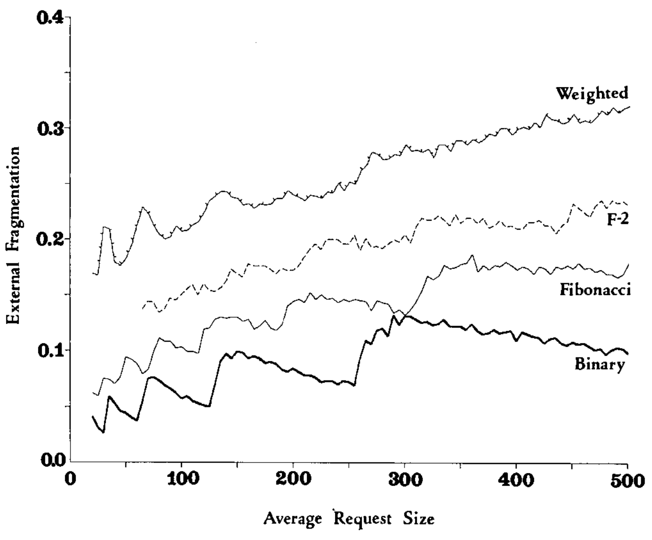


Fig. 4. Internal fragmentation for a (truncated) exponential request distribution.

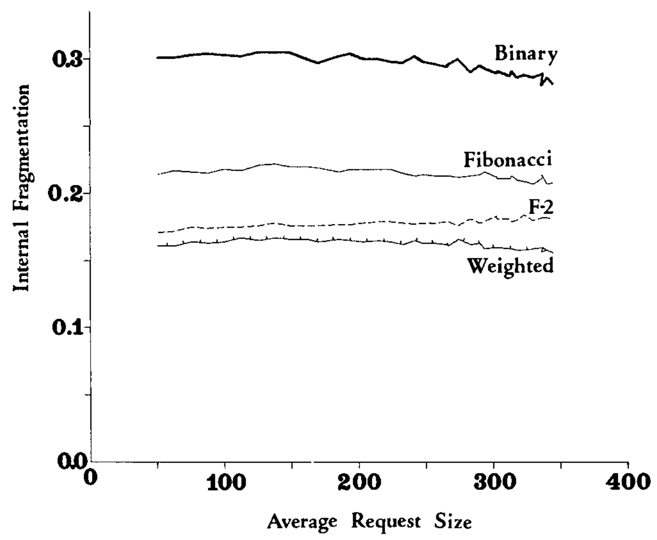
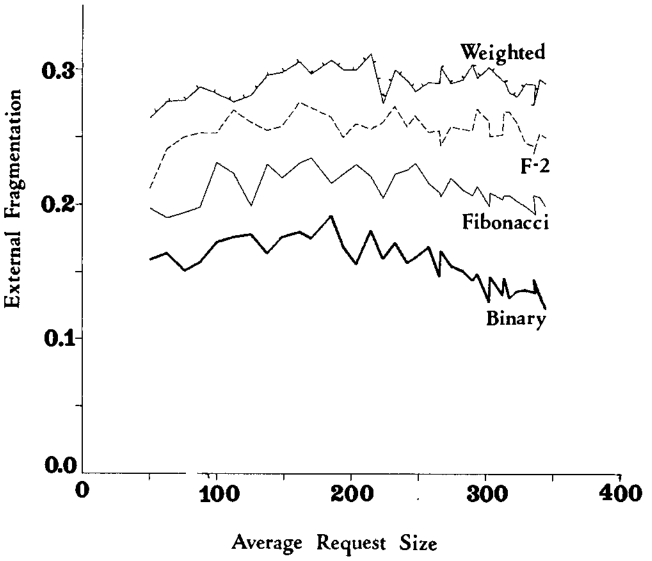


Fig. 6. External fragmentation for an exponential request distribution.



exponential distribution was 400, but the mean of the truncated distribution was only 345. All results are presented in terms of the true mean of the truncated distribution.

The results of our simulations for internal fragmentation are presented in the graphs of Figure 3 (for a uniform distribution of requests) and Figure 4 (for an exponential distribution of request). The curves of Figure 3 agree with the computations of Section 4 to within 1 percent, lending support to the validity of our simulations. Notice that both the relative position and the average internal fragmentation of the four buddy systems do not change radically as a function of the two distributions presented.

Figures 5 and 6 present our simulation results for external fragmentation. We notice that although our measure of external fragmentation is not directly comparable to the measures of other studies, our results are compatible with the previous observations of Knuth [6] and Purdom and Stigler [10]. The values obtained for the buddy systems also seem reasonable if we consider that the lower internal fragmentation values of Figures 3 and 4 were obtained because of the increased number of different block sizes which are available in the weighted buddy and F-2 buddy systems over the number available in the binary and Fibonacci buddy systems. With a smaller intrablock difference $(L_i - L_{i-1})$, a better fit to the requested block size can be made, yielding lower internal fragmentation. However, this also produces a smaller available buddy (if a block is split), and this smaller block is less likely to be as useful as the larger buddies provided by the binary and Fibonacci buddy systems. These small unusable but available blocks contribute to higher external fragmentation.

These considerations also lead us to the conclusion that these smaller blocks will (being unused) be available when their buddies are released and hence will be recombined immediately, requiring the resultant parent block to again be split if the just released block size is requested again. This should result in an average number of searches, splits, and recombinations which parallels the external fragmentation of a buddy system. Figure 7 presents the average number of searches for a uniform request distribution. The graphs for the number of splits and recombinations are identical to each other and similar to the graph of Figure 7. The savings in the number of splits due to using the smaller of $\beta(j)$ and $j - 1$ are minor. The largest savings are for the Fibonacci buddy system where, for example, for a uniform distribution from 1 to 1000, the average number of searches is 0.44 while the average number of splits is 0.35. The standard deviations of these performance measures increase as the external fragmentation increases also (being on the order of 1.00 for the weighted buddy system).

The total fragmentation for the four buddy systems investigated is plotted in Figure 8 (for a uniform distribution of requests) and Figure 9 (for an exponential distribution). While we apologize for the difficulty in reading these graphs, one very important conclusion can be drawn from this exact problem: *The total amount of usable space in a buddy system is relatively independent of the buddy system used.* The total fragmentation for all these buddy systems lies in a band, with the difference between the best and the worst total fragmentation being only 5 to 10 percent of total memory. The standard deviation of the points on the total fragmentation curves is in the same range (5 to 10 percent).

6. Simulations of Actual Request Distributions

In addition to the simulations using the theoretical distributions, simulations were performed for each of the four buddy systems with each of three actual request distributions. The actual request distributions, listed in Table II, were the distribution of buffer requests on the UNIVAC 1108 Exec 8 system at the University of Maryland [4], the distribution of memory requests on an IBM 360 CP-67 system [8], and the distribution of partition size requests on the IBM 360/65 OS MVT system at Brigham Young University.

Note that the University of Maryland and Brigham Young University distributions are labeled "continuous." This is because their tables consist of points on the cumulative distribution function between which the probability is equally distributed over the integers. Consider, for example, the first two entries in the University of Maryland table. This implies zero probability for a request of size 1 or 2 and a probability of 0.06 for each of sizes 3–8.

Fig. 7. Average number of searches per request for a uniform request distribution.

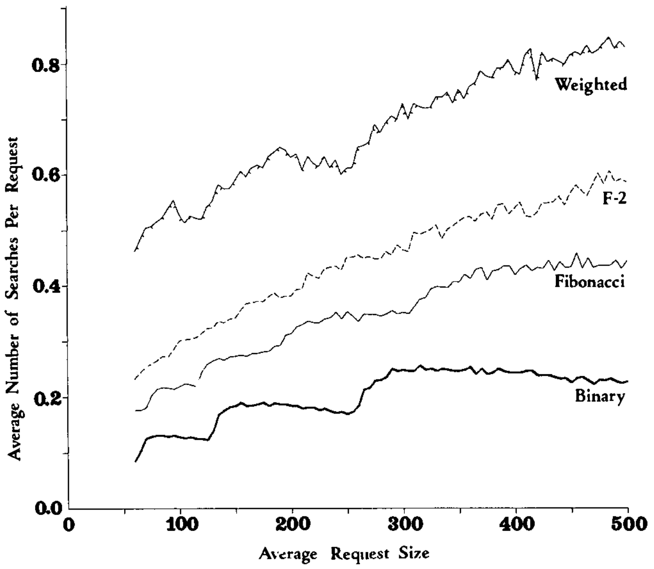


Fig. 8. Total fragmentation for a uniform request distribution.

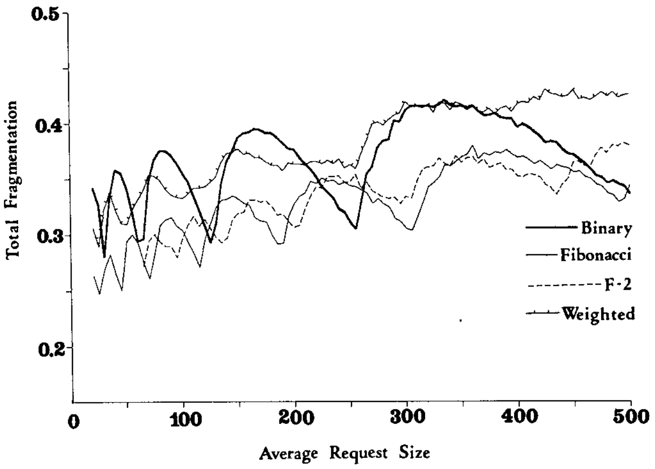
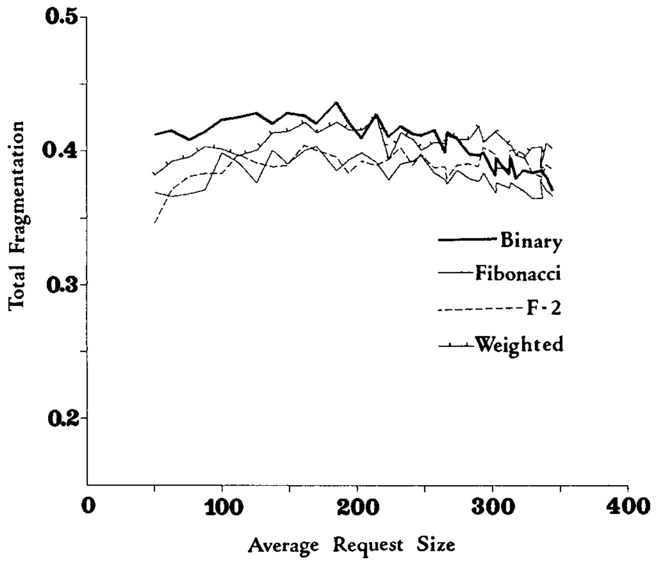


Fig. 9. Total fragmentation for an exponential request distribution.



The CP-67 distribution is a discrete distribution. There is zero probability of request sizes not shown in the table.

The simulations were run in the same way as the simulations of Section 5, with the exception that the actual distributions listed in Table II were used to generate the sequence of requests, and because of the smaller average request for these distributions, a memory of only 1000 words was used.

The results of these simulations are given in Tables III–V. The measured internal, external, and total fragmentation are presented as well as the expected internal fragmentation for each distribution and buddy system. The expected internal fragmentation was computed directly from the request distribution as defined in Section 4. The results of these simulations compare very favorably with the results obtained in Section 5 with the theoretical distributions. Between 28 and 43 percent of available memory is being wasted owing to internal and external fragmentation.

7. Summary and Conclusions

We have, in this paper, considered a number of properties of dynamic storage allocation schemes based upon the buddy system. We have presented two general algorithms which can be used to implement a wide variety of buddy systems. Then, using these algorithms, we investigated, first analytically and then by simulation, the fragmentation characteristics of several buddy systems. These results, presented in Figures 3–9 and Tables III–V, indicate that as internal fragmentation decreases (owing to more block sizes) external fragmentation increases (owing to more small blocks). Total fragmentation remains relatively constant, with from 25 to 40 percent of memory being unusable owing to either internal or external fragmentation. The execution time of the request and release procedures increases with external fragmentation.

Some general comparisons can be made, however, for the binary, Fibonacci, and weighted buddy systems. The total fragmentation of the weighted buddy system is generally worse than that of the Fibonacci buddy system. The total fragmentation of the binary buddy system varies widely because of its internal fragmentation characteristics. Still the variation among these buddy systems is not great, and the lower execution time of the binary buddy would therefore seem to recommend it for general use, although the execution time of the Fibonacci buddy system is not much greater. The weighted buddy system seems to be less desirable than either the binary or the Fibonacci systems owing to its higher execution time and greater external fragmentation.

In conclusion then, we would recommend that the memory management module of a system be constructed as either a binary or Fibonacci buddy system

Table II. Actual request distributions.

University of Maryland		Brigham Young University		CP-67	
Size	CDF (%)	Size	CDF (%)	Size	PDF (%)
2	0.0	3	0.0	1	11.1
8	36.0	16	6.4	2	0.2
10	44.0	32	16.8	3	3.7
15	54.0	48	27.6	4	24.8
25	84.0	64	40.0	5	21.9
30	94.0	80	45.8	6	0.3
35	96.5	96	62.7	7	0.6
40	97.5	112	82.6	8	11.2
50	98.5	128	94.9	9	2.0
70	99.3	144	95.3	10	4.1
100	99.6	160	95.7	11	0.2
200	100.0	176	96.1	12	0.2
		192	96.4	17	0.9
		208	97.0	18	1.9
		224	98.3	21	0.2
		256	99.4	23	0.3
		272	99.6	27	0.1
		304	99.8	29	15.6
		352	99.9	31	0.4
		511	100.0	50	0.3
("continuous" distribution) average request = 15.99		("continuous" distribution) average request = 80.26		(discrete distribution) average request = 9.34	

Table III. Simulation results using University of Maryland request distribution [4].

Buddy system	Expected internal fragmentation	Internal fragmentation	External fragmentation	Total fragmentation
binary	.276	.276	.179	.406
Fibonacci	.198	.199	.217	.373
F-2	.155	.154	.265	.378
weighted	.137	.137	.305	.400

Table IV. Simulation results using CP-67 request distribution [8].

Buddy system	Expected internal fragmentation	Internal fragmentation	External fragmentation	Total fragmentation
binary	.182	.188	.114	.281
Fibonacci	.131	.136	.189	.300
F-2	.210	.216	.230	.397
weighted	.103	.107	.239	.321

Table V. Simulation results using Brigham Young University request distribution.

Buddy system	Expected internal fragmentation	Internal fragmentation	External fragmentation	Total fragmentation
binary	.227	.227	.151	.343
Fibonacci	.222	.222	.212	.387
F-2	.163	.162	.318	.429
weighted	.134	.132	.323	.413

before any information concerning the actual distribution of block sizes is obtained (assuming of course that a buddy system is to be used at all). With these systems, there is a reasonable assurance that no better buddy system can be chosen without knowledge of the actual request distribution. With the system in actual use, statistics on the actual request distribution can be obtained and, if deemed appropriate, a new buddy system can be tailored [9] to that distribution by use of the algorithms of either Section 2 or Section 3. The new system can then replace the original buddy system to improve memory utilization and execution speed.

Appendix. A General Algorithm for Buddy Systems

The following PASCAL procedures implement the algorithm of Section 2. The constants *null, n,* and *m* are: a special address indicating that overflow has occurred in the *request* procedure, or that the buddy of the block at *p* is not available in the *buddyaddress* function; the number of different block sizes which are provided by the buddy system; and the size of the memory block which is being managed by the buddy system, respectively. The *attachtoasl* and *removefromasl* procedures are standard doubly linked list insertion and deletion routines. The *buddyaddress* function returns *null* or the address of the buddy of the block at *p* if the buddy is available for recombination.

```
CONST  null = -1;
TYPE   address:    null . . m;
       sizeindex:  1 . . n;
VAR    size:       array [sizeindex] of 1 . . m;
       subbuddy:   array [sizeindex] of 0 . . n;
       memory:     array [0 . . m]
                     of packed record
                       tag:         (available, allocated);
                       a,b:         (left, right);
                       index:       sizeindex;
                       forward:     address;
                       backward:    address;
                     end;
       asl:        array [sizeindex]
                     of record
                       head, tail:  address;
                     end;

procedure attachtoasl (p: address);
begin
      with memory [p], asl [index]
        do  begin
              backward := null;
              forward := head;
              head := p;
              if tail = null then tail := p;
            end;
end;

procedure removefromasl (p: address);
begin
      with memory [p], asl [index]
        do begin
             if backward = null
               then head := forward
               else memory [backward].forward := forward;
             if forward = null
               then tail := backward
               else memory [forward].backward := backward;
               end;
end;
function buddyaddress (p: address): address;
var  q: address;
     j: sizeindex;
begin
     if memory [p].a = left
        then begin
                 q := p + size [memory[p].index];
                 if (q ≥ m) ∨ (memory[q].a = left) ∨
                 (memory[q].tag=allocated)
                 then q := null;
             end
        else begin
                 j := subbuddy [1 memory[p].index];
                 q := p - size [j];
                 if (memory[q].index≠j)
                 ∨ (memory[q].tag=allocated)
                 then q := null;
             end
       buddyaddress := q;
end;
procedure request (var p: address; i: sizeindex);
var  j: sizeindex;
     q: address;
begin
     j := i;
     while (j<n) ∧ (asl [j].head=null)
       do j := j + 1;
     if asl [j].head=null
        then  p := null
        else  begin
                p := asl [j].head;
                removefromasl (p);
                while j > i
                  do begin
                       q := p + size [subbuddy[j]];
                       with memory [q]
                         do begin
                           tag := available;
                           b := memory [p].b;
                           a := right;
                           index := j - 1;
                         end;
                       with memory [p]
                         do begin
                           tag := available;
                           b := memory [p].a;
                           a := left;
                           index := subbuddy [j];
                         end;
                       if i ≤ subbuddy [j]
                         then attachtoasl (q)
                         else begin
                           attachtoasl (p);
                           p := q;
                         end;
                     j := memory [p] .index;
                   end;
                memory [p].tag := allocated;
              end;
end;
```

```
procedure release (p: address; i: sizeindex);
var  q,r:  address;
begin
  q := buddyaddress (p);
  while q ≠ null
    do begin
        removefromasl (q);
        if p > q
          then begin r := p; p := q; q := r end;
        with memory [p]
          do begin
              tag := available;
              a := memory [p].b;
              b := memory [q].b;
              index := 1 + memory [q].index;
            end;
        q := buddyaddress(p);
      end;
  attachtoaslp(1)
end;
```

Received October 1974; revised May 1976

References

1. Cranston, B., and Thomas, R. A simplified recombination scheme for the Fibonacci buddy system. *Comm. ACM 18*, 6 (June 1975), 331-332.

2. Ferguson, H.R.P. On a generalization of the Fibonacci numbers useful in memory allocation schema. *The Fibonacci Quart., 14*, 3 (Oct. 1976), 233-243.

3. Harris, V.C., and Styles, C.C. A generalization of Fibonacci numbers. *The Fibonacci Quart. 2*, 4 (Dec. 1964), 227-289.

4. Hirschberg, D.S. A class of dynamic memory allocation algorithms. *Comm. ACM 16*, 10 (Oct. 1973), 615-618.

5. Knowlton, K.C. A fast storage allocator. *Comm. ACM 8*, 10 (Oct. 1965), 623-625.

6. Knuth, D.E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968, pp. 435-455.

7. Lewis, T.G., Smith, B.J., and Smith, M.Z. Dynamic memory allocation systems for minimizing internal fragmentation. Proc. ACM Annual Conf., Nov. 1974, pp. 725-728.

8. Margolin, B.H., Parmelee, R.P., and Schatzoff, M. Analysis of free-storage algorithms. *IBM Systems J. 10*, 4 (1971), 283-304.

9. Norman, T.A. Tailored buddy systems for dynamic storage allocation. *Proc. Fourth Texas Conf. Comptg. Systems*, Nov. 1975, pp. 2B-3.1-2B-3.5.

10. Purdom, P.W., and Stigler, S.M. Statistical properties of the buddy system. *J. ACM 17*, 4 (Oct. 1970), 683-697.

11. Russell, D.L. Internal fragmentation in a class of buddy systems. Tech. Note 54, Digital Systems Lab., Stanford U., Stanford, Calif., Jan. 1975.

12. Shen. K.K., and Peterson, J.L. A weighted buddy method for dynamic storage allocation. *Comm. ACM 17*, 10 (Oct. 1974), 558-562. Corrigendum, *Comm. ACM 18*, 4 (April 1975), 202.

Programming Techniques

G. Manacher, S.L. Graham Editors

A Bounded Storage Algorithm for Copying Cyclic Structures

J. M. Robson
University of Lancaster, England

A new algorithm is presented which copies cyclic list structures using bounded workspace and linear time. Unlike a previous similar algorithm, this one makes no assumptions about the storage allocation system in use and uses only operations likely to be available in a high-level language. The distinctive feature of this algorithm is a technique for traversing the structure twice, using the same spanning tree in each case, first from left to right and then from right to left.

Key Words and Phrases: copying, shared subtrees, cyclic structures

CR Categories: 4.49, 5.25

Author's address: Computer Studies Department, University of Lancaster, Bailrigg, Lancaster LA1 4YN England.