THE DOUBLE BUDDY·SYSTEM *

by

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 79

THE DOUBLE BUDDY·SYSTEM

DAVID S. WISE

DECEMBER, 1978

* Research reported herein was supported (in part) by the National Science Foundation under a grant numbered MCS77-22325.

Abstract: A new buddy system is described in which the region of storage being managed is partitioned into two subregions, each managed by a fairly standard "binary" buddy system. Like the weighted buddy system of Shen and Peterson, the block sizes are of sizes 2^{n+1} or $3 \cdot 2^{n}$, but unlike theirs there is no extra overhead for typing information or for buddy calculation, and an allocation which requires splitting an extant available block only rarely creates a block smaller than the one being allocated. Such smaller blocks are carved out only when the boundary between the two subregions floats; the most interesting property of this system is that the procedures for allocation and deallocation are designed to keep blocks immediately adjacent to the subregion boundary free, so that the boundary may be moved within a range of unused space without disturbing blocks in use. This option is attained with a minimum of extra computation beyond that of a binary buddy system, and provides this scheme with a new approach to the problem of external fragmentation.

A new buddy system is described in which the region of storage being managed is partitioned into two subregions, each managed by a fairly standard "binary" buddy system. Like the weighted buddy system of Shen and Peterson [11] the block sizes are of sizes 2^{n+1} or $3 \cdot 2^n$, but unlike their's there is no extra overhead for typing information or for buddy calculation, and an allocation which requires splitting an extant available block only rarely creates a block smaller than the one being allocated. Such smaller blocks are carved out only when the boundary between the two subregions floats; the most interesting property of this system is that the procedures for allocation and deallocation are designed to keep blocks immediately adjacent to the subregion boundary free, so that the boundary may be moved within a range of unused space without disturbing blocks in use. This option is attained with a minimum of extra computation beyond that of a binary buddy system, and provides this scheme with a new approach to the problem of external fragmentation.

The remainder of this paper is in six parts. A review of the history of the buddy system is followed by a description of the data structure necessary for the double buddy-system. An example developed in that section is continued in the next which deals with allocation when the boundary is fixed. The next section describes the algorithm for releasing used blocks and is followed by a section on allocation

with boundary movement. The final section anticipates the behavior of the algorithm and improvements available through tuning and generalization.

History

This paper presents an algorithm for the double buddy-system, an alternative to Peterson's weighted buddy system [11] which has been known by this author for five years--antedating his knowledge of that work. Because of the interest in the buddy system, (both for application as the storage manager of various operating systems and, therefore, as a procedure for significant mathematical analysis), the novel properties of this algorithm, and the unliklihood that it will be investigated further in these quarters, it is time that others saw it. Because of the availability of detailed programs the buddy system [5, 6-Appendix] and the difficulty of expressing bit manipulations in higher level languages, the double buddy-system will be described in English with a minimum of code. The invariants given are peculiar to the new features of this algorithm.

The original buddy system, discovered by Knowlton [4] and Markowitz, studied by Knuth [5] and Purdom and Steigler [7,8] managed blocks of size 2^n and has been called the binary buddy system. Hirschberg [3] developed Knuth's idea of a Fibonacci buddy system, which provided blocks roughly of $F_n \approx \phi^n/\sqrt{5}$ (where $\phi = (1+\sqrt{5})/2 = 1.618...$), which complicates the problem of locating buddies [1,2]. Shen and Peterson [11] present a modification to a weighted buddy system, in which a block of size 2^{n+2} is split into buddies of size

2^n and $3 \cdot 2^n$ and a block of size $3 \cdot 2^n$ is split into buddies of size 2^n and 2^{n+1} (just as a Fibonacci block of size F_{n+2} is split into buddies of size F_n and F_{n+1}.) Thus, the ratio of successive block sizes has decreased from 2 (binary) to 1.618 (Fibonacci) to 1.417 (weighted; this is the simple average of the alternating 4/3 and 3/2 ratio) with a corresponding improvement on internal fragmentation.

Internal fragmentation [9] is an inefficiency in storage management caused by allocating a block satisfying a request which is larger than required. Buddy systems are characterized by a fixed sequence of "legal" block sizes, and so a lower ratio between successive sizes will reduce this internal fragmentation, [6,11] and analytic results [6,10] confirm this. According to Russell's analysis, we may expect 30½% of allocated memory to be wasted in this way for a binary system, versus an expected 22% for the Fibonacci system, and something like 16½% for a weighted buddy system.*

---

*Russell's formula for expected overallocation does not really apply to the weighted buddy system because it is not "simple", but Peterson's minimum and maximum [6 - p. 425] correlate nicely with those Russell's formulae would derive from the simple average above. This 16½% figure is (mis)derived similarly for comparison, but it seems to fit Peterson's Figures 3 and 4.

External fragmentation is an inefficiency in management characterized by the unallocated space being in blocks too small to be useful. It is known that internal fragmentation is the real problem with the binary buddy system [10, 5, 7], and Peterson and Norman showed further that external fragmentation rises when the Fibonacci or weighted buddy system is used to decrease internal fragmentation [6]. The weighted buddy system, however, suffers particularly from external fragmentation; its total fragmentation is worst of any studied and they advise against its use in general.

Perhaps it is so bad because the weighted buddy system routinely splits a block to leave an unallocated block available which is smaller than the one being allocated. Such a proliferation of small blocks, which occurs whenever it is necessary to split to get a block of size $3 \cdot 2^n$, encumbers the storage manager with additional external fragmentation and recombination time when these too-small blocks go unused. A lesser problem with the weighted buddy system is that an additional two type bits are required in every allocated (and unallocated) block. These are necessary to the "buddy address" calculation, which is considerably more complicated than that for the binary buddy system [11 - Corrigendum].

Introduction

The double buddy-system* exhibits none of these handicaps which hamper the weighted buddy system. Nevertheless, it provides the same sizes of blocks as the weighted buddy system so that the discussion of internal fragmentation above applies in general.†

It also exhibits a new behavior by the way it uses both ends of the available space lists which tends to push allocated blocks toward the end of the memory region being managed. This could turn out to be very significant in controlling external fragmentation. The run-time overhead is ordinarily quite comparable to that of the binary buddy system; a request is handled as directly according to two bits from the binary representation of its size, the "buddyaddress" calculation is direct (using binary arithmetic also), and there is no need for additional information in any block. Addresses carry such information implicitly.

The double buddy system is really two binary buddy systems managing adjacent subregions of one large contiguous

---

*The hyphen is only for parsing assistance.

†The exception is when a block of size 2^{n+1} is requested and only one of size $3 \cdot 2^n$ (at an address away from the boundary) is available. Unless special provision is made, such a request is unfillable under the double buddy-system.

region of memory. A boundary address, B, separates the region allocated in blocks of size 2^{n+1} from that allocated in blocks of $3 \cdot 2^{n}$. Each subregion has its own array of doubly-linked lists of available blocks and operates only between its limiting addresses: the boundary and one end of memory. Each binary buddy system also endeavors to allocate blocks far from B whenever possible. Thus, if one subregion is inappropriately large then some of its blocks adjacent to the boundary will be free. So the boundary may be moved into this unused region in order that the other subregion might grow; only available blocks are disturbed by this tuning and the alterations to the available space lists are straightforward.

Data Structure

Let the region addressed from and including address L up to but excluding address H be that to be managed by double buddy-system. L, the lower, should be even and that H, the higher limit, should be a multiple of three; $L+1 < H$. Some B, an even multiple of 6, is selected so that $L \leq B \leq H$; B denotes the boundary between the two subregions each partitioned into blocks of appropriate size. That is,

Invariant 1: A pointer, P, to a block such that $L < P < B$ always refers to a block of size 2^{n+1} where $P+1$ is necessarily a multiple of 2^{n+1} .

Invariant 2: When such a pointer P' satisfies $B \leq P' < H$ then P' refers to a block of size $3 \cdot 2^n$ where P' is necessarily a multiple of $3 \cdot 2^n$.

We shall use the convention of P pointing to a "two-block" (of size $2 \cdot 2^n$) and P' referring to a "three-block" (of size $3 \cdot 2^n$) below to suggest these invariants.

It is straightforward to show that a two-block of size 2^{n+1} at address P may be split into two adjacent two-blocks of size 2^n at addresses P and $P-2^n$ respectively (when $n>1$), and Invariant 1 above will still be preserved. Similarly, a three-block size $3 \cdot 2^n$ at address P' may be split into two adjacent three-blocks of size $3 \cdot 2^{n-1}$ at ad-

dresses P' and $P' + 3 \cdot 2^{n-1}$, respectively, whenever $n>1$ without violating Invariant 2. Such splits may be rejoined, of course, and still preserve these invariant relations.

Unused blocks of available space are maintained in doubly linked lists of similarly sized blocks. Like the binary buddy system described by Knuth [5], these lists have headers in an array indexed by the exponents, n above. In this case the array of headers is partitioned into two arrays; ARRAY2 contains the headers for lists of unused two-blocks and ARRAY3 contains the headers for lists of unused three-blocks. Let ARRAY2 be indexed from 0 up to s and ARRAY3 be indexed from 0 up to t *; then s should equal t (alternatively, t+1), and $(H-L)/2^s \geq 3$ (respectively, 2).

Associated with the address of every block in the system is a bit which indicates whether that block is in use; we shall use the terms <u>used</u> and <u>free</u> (denoted * and ° in the figures) to indicate the two values of that bit. (Knuth [5-prob. 29] suggests that this bit may be located in a separate bit table, a good idea here since the bit which be in a different position within a used block depending on whether is is a two-block or a three-block.) No other information is needed for used blocks. Free blocks contain three other

---

*The letters s and t are suggested by the second and third ordinals. A lower bound for these arrays above zero is possible, but it would restrict our choice of L, H, B, and r_{min} below.

pieces of information at their block address: two pointers for double linking to the appropriate available space list and a value of n (between 0 and s) where either the block at P is of size 2^{n+1} stretching from address $(k-1)2^{n+1}$ up through $k \cdot 2^{n+1} - 1 = P$, or the block is at P' of size $3 \cdot 2^n$ and stretches from address $3k \cdot 2^n = P'$ up through $3(k+1)2^n - 1$. This places the address of any block P (alternatively, P') at the rightmost (respectively, leftmost) end closest to the boundary address, B.

Figure 1 illustrates an example initialization of a double-buddy system managing a region of one-hundred words. The addresses are arbitrary, relative to a zero discussed in the last section L=44, H=144, s=5, and t=4. Initially we consider B=96 so that the single available three-block of size 48 is located at 96 and is linked onto AVAIL3[4]. The remaining AVAIL3[i] are empty for $0 \le i \le 3$. There are initially three available two-blocks of size 4, 16, and 32 located at 47, 63, and 95, respectively. These are doubly linked onto AVAIL2[1], AVAIL2[3], and AVAIL2[4] with the other AVAIL2[i] being empty. The figure indicates how two-blocks are addressed by their highest (rightmost) word and how three-blocks are addressed by their lowest (leftmost).

Allocation-Fixed Boundary

Let us pursue the example above for a moment and see how the initialized system of Figure 1 might allocate space. Consider its response to a request queue of successively Fibonacci-sized blocks illustrated in Figure 2. The requests of size 1 and 2 may be satisfied with blocks of size 2^1 at addresses 45 and 47, created by splitting the initial block of size 2^2 at 47. The request of size 3 is met by fragmenting three-blocks until a block of size $3 \cdot 2^0$ at 141 is attained. The request for 5 is then satisfied by the extant three-block of size $3 \cdot 2^1$ at 132. The requests for 8 and 13 each require a two-block which is fragmented from an available block too large; the 8-request is satisfied by a perfectly sized block at address 55 and the 13-request is filled with a block of size 2^4 at address 79. (That leaves available two-blocks of size 2^3 at 63 and 2^4 at 95.) The request for a block of size 21 is met by a then-extant three-block of size $3 \cdot 2^3$ at 96. (The unused three-blocks are of size $3 \cdot 2^2$ at 120 and $3 \cdot 2^0$ at address 138.)

The allocation procedure reflected in Figure 2 is a direct modification of the procedure for other (notably the binary) buddy systems. When a request of size r is received, the value of r-1 is analyzed to determine how the request should be handled. (There is always some

for the smallest $n \le j \le s$ ($n \le j \le t$) such that AVAIL2[j] (AVAIL3[j]) is not empty. A block of size 2^{j+1} ($3 \cdot 2^j$) at address P (P') is removed from the head of that list (This choice will become more complicated when j=s (respectively j=t) below), and successively split into buddies of exponents i where $n \le i < j$. In each case the buddy whose address is closest to B is freed and returned to AVAIL2[i] (or AVAIL3[i]) becoming the only block on that list. The allocated block will therefore be located at address $P - 2(2^j - 2^n)$ (respectively $P' + 3(2^j - 2^n)$).

If no such j exists then the request will be considered unfillable--for the moment. This allows for the possibility that there is, for instance, a three-block which could satisfy r, but since $3 \cdot 2^{n-1} < r \le 2^{n+1}$ we are looking for a two-block and there just isn't anything on AVAIL2[n] ... AVAIL2[s]. We shall make no provision for this sort of failure; the alternative is obvious.

A different alternative which we shall consider in detail below, however, is moving the boundary. Anticipating this feature, we introduce here two system variables which will be maintained in order to indicate when boundary moving is possible.

Invariant 3: D always denotes the address of the highest two-block which is used. U similarly contains the address of the lowest used three-block. When no space is in use then D = L-1 and U = H .

The memory properly between D and U is available and could belong to either subregion. Thus D and U always point down and up from B delimiting a "no-man's-land" of memory within which B is free to move without disturbing blocks in use. In Figure 2, D = 79 and U = 96 = B, and B could therefore be relocated to 84 or 90.

The reason that D and U are introduced here is that they must be maintained, even on simple allocation. Because an available block selected for splitting or for direct allocation might be this in no-man's-land, we must make one last test after allocating a block. If the address of a newly allocated two-block, P is such that P > D then D is reset up to P . (Similarly if the address of a newly allocated three-block, P' , is such that P' < U then U is set back to P' .)
This convention preserves Invariant 3 over the simple allocations described above.

Release

Upon release of a used block the first thing to notice is whether that block borders upon no-man's-land. When the two-block at D (respectively the three-block at U) is released then D will be decreased (U will increase) after all possible recombination in order to preserve Invariant 3.

The recombination algorithm is the same as for a conventional binary buddy system restricted to the sub-region boundaries [5,6]. The "buddyaddress" calculation is particularly straightforward with binary arithmetic. The buddy of a newly freed two-block of size 2^{n+1} at address $P = k \cdot 2^{n+1} - 1$ may be computed as the "exclusive-or" of P and 2^{n+1}. The buddy of a three-block of size $3 \cdot 2^n$ at address P' is found by testing whether P' mod 2^n is even or odd (a mask and test instruction); if it's even then the buddy is at $P' + 3 \cdot 2^n$ and otherwise it's at $P' - 3 \cdot 2^n$. The buddy of a block is always of the same size as that block.

After all recombination has been performed then the resultant free block is returned to available space. If the originally released block was not at D or U then the recombined block is placed at the head of the appropriate available spacelist and the release procedure terminates. If it was at D (for two-blocks) or at U

(for three-blocks) then there is more to do to preserve Invariant 3.

In this case D (respectively, U) is decreased (increased) by the size of the block resulting from recombination; let it be of size 2^{n+1} located at address P ($3 \cdot 2^n$ at address P'). When $n < s$ ($n < T$ respectively) that block is spliced at the tail of the appropriate available space list, where it will always be last or second-last. (AVAIL2[s] and AVAIL3[t] do not exhibit this property, and so the released block is spliced on directly at the tail.) It will be last precisely when there is no other free block of that size already in no-man's-land closer to B. This is easily tested as we splice it on by checking whether AVAIL2[n].TAIL > P (respectively, AVAIL3[n].TAIL < P')*; if so then the new block is spliced in just ahead of that one which is already available.

Theorem: There are at most two-blocks of size 2^{n+1} where $n < s$ or of size $3 \cdot 2^n$ where $n < t$ in memory between D and U.

---

*These inequalities suggest that the array AVAIL2 be located beneath address L and that AVAIL3 be located above H; then the empty circular lists are not exceptions.

The proof derives from the fact that if there were three, then two of them would be available buddies and hence already recombined. We arrange for such blocks to dwell at the bottom of the available space lists so that blocks closest to B (i.e. in no-man's-land) are allocated last, so that U-D will tend to be large. (For n=s or n=t this goal is obtained on allocation where AVAIL2[s] or AVAIL3[t] should be searched from the head for the first block outside no-man's-land or, failing that, for the block furthest from B.)

Invariant 4: Every block smaller than 2^{s+1} and $3 \cdot 2^t$ with storage address between D and U is at the bottom or next-to-bottom of its respective available space list.

Finally, after the recombined block is placed at the tail of its list, D may be decreased (U may be increased) further as long as it refers to a free block of, say, size 2^{n+1} and $D > L$ ($3 \cdot 2^n$ and $U < H$).
If so then AVAIL2[n].TAIL is checked (unless n=s) to see that the block at D is in its appropriate position on that list (either last or second-last); if not then it is excised directly from wherever it was on that list and spliced back appropriately at the tail. Then $D := D - 2^{n+1}$ and the next block at D is checked. (Similarly for AVAIL3,

t, and $U:=U + 3 \cdot 2^n$.) This restores Invariants 3 and 4 on release.

Allocation--Floating Boundary

If, say, a three-block is required to meet a request and there are no three-blocks available of the appropriate size or larger, then the request might still be filled by locating the desired block within no-man's-land: by moving B down to or beyond that block to some address (still a multiple of 6) above D and reconfiguring AVAIL2 and AVAIL3. A two-block may be located similarly by moving B up toward U.

The procedure for moving the boundary, outlined in this section for locating such a three-block, is complicated by a division by three. On a binary computer with slow integer division this increases overhead and so we shall consider how to use tuning to avoid this procedure in the next section. The procedure for locating a two-block by moving B is similar to that described below for a three-block.

Suppose that we require a block of size $3 \cdot 2^n$ which cannot be satisfied by AVAIL3. First we must see if a legal block is available in the memory from D up to U. Consider the integer part of $U/2^n = X$ and its integer quotient $X/3$. Just one of X, $X-1$, or $X-2$ is evenly divisible by three; let Y be the product of it and 2^n. (When $n=0$, $Y=U$.) We have determined that the block from $Y - 3 \cdot 2^n$ up through $Y-1 < U$ is the first legal three-block below U which satisfies the

request. But is it available and attainable? It is available if that block lies entirely in no-man's-land -- if $D < Y - 3 \cdot 2^n$. It is attainable if there is a largest, legal (multiple of 6) boundary value, B', in no-man's-land beyond the new block; when $n > 0$ then $B' = Y - 3 \cdot 2^n$ is legal and if $n=0$ then $B' = U-6$ is the next legal boundary below B. Thus we locate $D < B' \leq Y-3 \cdot 2^n < Y \leq U$; if no such B' and Y exist then allocation is impossible! The choice of Y is necessary to preserve Invariant 2, and this choice of B' is necessary to maintain Invariants 1 and 2 when B is reset to B' later. (In allocating a two-block we search for $D < Y < Y + 2^{n+1} \leq B' \leq U$; shifting, addition, and comparison is sufficient to determine availability, but a division-by-three is necessary to assure attainability.)

After locating the block about to be allocated we must remove all blocks which are currently occupying its space from AVAIL2 and AVAIL3. Starting from the available two-block at P=B-1 and moving sequentially down memory according to the size of blocks encountered, we visit every two-block with address $P > B'$. Each of these is excised from the AVAIL2 list into which it was doubly linked. The space beneath B' and above the address of the final P (< B') belonged to the final block of size 2^{j+1} removed from AVAIL2, but that space must remain available in the subregion managed by AVAIL2 and Invariant 3 requires that it be returned--in smaller

blocks--to AVAIL2. Starting, then, with $Q = B'-1$ and moving Q down towards P, the largest $i < j$, such that $Q-2^{i+1} \geq P$ and 2^{i+1} evenly divides $Q+1$, is chosen; a free two-block of size 2^{i+1} is established at Q, and linked onto the tail of AVAIL2[i]; then Q is decremented by 2^{i+1}. The value of i is strictly monotone increasing during this reassembly of two blocks, which is repeated until Q reaches P after at most $j \leq s$ iterations. Invariant 4 for two-blocks requires that these "new" free blocks be linked at the tail of the AVAIL2 lists.

A similar procedure is now used to clean up AVAIL3. Starting from the three-block at B and moving sequentially up memory according to the size of the blocks encountered P' visits every three-block until $P' \geq Y$. Each of these is excised directly from AVAIL3, but we need to return any portion above Y from the last block of size $3 \cdot 2^j$ located at $P'-3 \cdot 2^j$ (where P' is the first address at or above Y so encountered). So starting with $Q=Y$ and moving Q up towards P', we create an available block of size $3 \cdot 2^i$ at Q for the largest $i < j$ such that $Q+3 \cdot 2^i \leq Y$ and $3 \cdot 2^i$ evenly divides Q*. That block is doubly linked onto the tail of AVAIL3[i] and Q is

---

*The quotient $X/3$ from above is useful in finding the first i without division, and thereafter the strict monotonicity of i obviates the need to test divisibility of Q ever again.

increased by $3 \cdot 2^i$. The reassembly of three-blocks continues, with i again strictly monotone until Q reaches P' after at most $j \leq t$ iterations.

Now B may be set to B' and U may be changed to be $Y-3 \cdot 2^n$, the address of the new used block. Invariant 3 is restored but Invariant 4 may not be satisfied yet. The new value of U will be either B' or, when $n=0$, B' + 3 . In the latter case a free block of size $3 \cdot 2^0$ must be created at B' and linked onto the tail of AVAIL3[0]. (This tiny three-block is very unlikely to be used, but it is necessary to restore Invariant 2.) At last all invariants are again satisfied.

The net effect of this procedure is to move B in order to satisfy an order for a three-block. The procedure for allocating a two-block is quite similar, using nearly identical procedures for cleaning up AVAIL2 and AVAIL3; the only real differences have to do with locating Y, B', and D, and with the final additions to AVAIL2[0] or AVAIL2[1]. In the case detailed above we have converted a series of available two-block-triplets (six words of memory) at B into three-block-pairs; when enough were changed to fill the request we restored the invariants of the buddy system and the four of this paper. Going the other way, a two-block may be created with a move of the boundary by converting a series of three-block-pairs at the boundary into two-block-triplets and then restoring the invariant properties of both binary buddy systems.

Behavior and Conclusions

In this section we raise several points which should be studied before the double buddy-system is put into production. These issues should also be considered by those who would simulate or analyze the algorithm, because minor alterations to the implementation will be seen to have noticeable effects in performance.

Although the floating boundary allows the double-buddy system to be initialized quite easily, it is rather expensive to allow the boundary to thrash back and forth. Initialization is possible with L = D + 1 (an odd number), B = U = H (a multiple of 6), AVAIL3 empty, and AVAIL2 initialized as in a binary buddy system [5-Problem 25]. Thence, the system may be run directly as described above or it may be simulated until the behavior of the boundary B has stabilized.

After initial study of the behavior of B in a particular application, several alterations might constrain the boundary from minor but expensive vibrations. We might simply fix B and eliminate all code involving D and U and maintaining order at the tails of the available space lists; if B were stable then this cut makes sense. If not then a better tactic would be to allow the boundary to move only under extreme demand. For instance, we might establish a lower limit on the request size which could be filled through a boundary move until a trend in one direction were established by a census of unfilled

requests. Rather then letting B float freely we could then treat the moving boundary as a system tuning parameter, which may be fixed when the storage management system is installed or which may track the dynamic performance of the system and tune itself conservatively, like the governor on an engine.

The choice of the address base for the region being managed may also need tuning. Figures 1 and 2 illustrate an example of a region managed by the double-buddy system which is addressed relative to a zero not even in the region. (Any even multiple of $6 \cdot 2^s$ may be considered as a zero for this discussion.) The choice of the zero for addressing within this buddy system is important because it may encourage or restrain the moving boundary. While negative addresses are perfectly reasonable addresses for this scheme, no block has buddyaddress on the other size of a zero from itself; any zero becomes an absolute boundary between all blocks. If B happened to fall at a zero of the address scheme, then any allocation made by moving B would require that the entire new block be allocated from no-man's-land in the other sub-region. Thus, moving the boundary becomes a rarely usable allocation strategy for large blocks, but it is exactly the large requests which will not be fillable from extant free blocks in a loaded system. At the other extreme, when $B = 2(4^n - 1)$ then we are assured of a plethora of small blocks in both subregions on either side of B (of sizes 2^{i-1} and $3 \cdot 2^i$

for i ranging from 1 on up). Since such blocks will be discriminated against by being at the tails of available space lists and since there is usually an adequate supply of small blocks at the heads of these lists, such blocks will very likely go unused. In the first case the selection of zero made a particular choice of B artificially stable; in this latter case the choice of B creates a contiguous but unusable chunk of free space in no-man's-land and B becomes artificially mobile.

For example, the Fibonacci requests of Figure 2 could be satisfied (quite compactly) using a moving boundary allocation scheme on only sixty words if we were fortunate enough to choose zero so that $L = 20$ and $U = 81$; ultimately $B = 48$ and all those requests are filled. The same sequence of requests: 1, 2, 3, 5, 8, 13, 21; could not be filled by the same algorithm managing a region of seventy-eight words if we were so unfortunate as to choose $L = 24$ and $U = 102$. Figure 3 illustrates the breakdown of the system on the last request. Of course, these are pathologies which might be avoided if we chose $s << \log_2(H - L)$, but they illustrate the significance of the choice of a zero.

Before B is fixed, therefore, we should experiment with alterations to the whole addressing scheme; the stability of B might change suddenly by adding a small constant to L, to H, and to all other addresses in the system without changing the

size of the managed region. It may turn out that it is more important to have a correct zero for the addressing scheme than to have the pointers be of minimal size; Figures 1 and 2 illustrate pointers of eight bits (to range from 44 to 144) when seven bits would suffice if the zero were moved. After a desirable value of B is found relative to L and H (not to zero) and when we decide to fix B at this value, then it is most appropriate that this B be made a zero of the addressing scheme.

The double buddy-system generalizes naturally to other ratios between the blocks in the two subregions. For instance, it would also work with blocks of size $5 \cdot 2^n$ and $7 \cdot 2^n$ although computation overhead would be a bit more complicated. (e.g. requests of size r would be filled with a "five-block" if the leading bits of r-1 were 111... or 100... and with a "seven-block" if 101... or 110...; buddyaddress computations would be messier.) Blocks of this size, however, have a much more uniform alternating ratio (1.40 and 1.43) between successive blocks.

If the five/seven ratio of the previous paragraph is computationally tractable and the boundary B can be fixed without excessive external fragmentation but internal fragmentation is still a problem, then there is another generalization of the double-buddy system which may be investigated: more than two subregions. In the extreme this becomes a best-fit strategy

[5] when there are as many subregions as possible requests. With B fixed we would free each subregion to divide into two "sub-subregions" so that there might be blocks of size $7 \cdot 2^n$ and $4 \cdot 2^n$ (in the region that was all two-blocks above) and of size $5 \cdot 2^n$ and $6 \cdot 2^n$ (in the region that was formerly composed of three-blocks). The same requests would be handled in each subregion, but each subregion would be managed by its own double buddy-system. Russell's estimate [10] would predict internal fragmentation of something like only 8½% for this scheme. Without a suitable fixed B (quite a challenge if external fragmentation remains the major problem) this proposal is not at all attractive, however; four subregions are too many for simple tuning by floating boundaries [5-page 242].

Finally, we make an observation about the cost of maintaining D and U, the boundaries of no-man's-land. This overhead must be endured on every allocation and release even when boundary movement is restrained, but we argue here that that overhead is minimal; so the noticeable increase in running time of the double buddy-system is really due to boundary movement. There is no particular bother in maintaining D and U on releasing nodes because in most cases released nodes will not border on no-man's-land. When they do, the adjustment of D or U after recombination might require some additional traversal of adjacent free blocks, but that

is no worse than the potential recombination effort for a block of that size. The allocation overhead is trivial when the boundary remains stationery, with an apparent exception when allocating from AVAIL2[s] or AVAIL3[t]. Those lists ought to be searched to find a block not in no-man's-land or to find the one furthest from B if all such blocks are between D and U. This apparently lengthy search is likely to be trivial in practice. There is never more than one block on both these lists when the system is loaded, because any outstanding request may be serviced in that case. If all requests have been satisfied then the storage manager ought to be allowed some leisure to perform the prescribed search; it's doing a fine job!

There is, therefore, much to be learned from the implementation, simulation, and analysis of the double buddy-system. Properly tuned, it should prove to be quite useful because it provides the good behavior with respect internal fragmentation like the weighted buddy system and H has the characteristic ability of the binary buddy system to "learn" what size blocks are needed (by always leaving an equal free block when a split is necessary) which reduces external fragmentation.

REFERENCES

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

2. Hinds, J. A. An algorithm for locating adjacent storage blocks in the buddy system. Comm. ACM 18, 4 (April, 1975), 221-222.

3. Hirschberg, D. S. A class of dynamic memory allocation algorithms. Comm. ACM 10 (October, 1973), 615-618.

4. Knowlton, K. C. A fast storage allocator. Comm. ACM 8, 10 (October, 1965), 623-625.

5. Knuth, D. E. The Art of Computer Programming I, Fundamental Algorithms (2nd ed.), Addison-Wesley, Reading, MA (1973), 435-455, 460-461, 596-606.

6. Peterson, J. L., and Norman, T. A. Buddy systems. Comm. ACM 20, 6 (June, 1977), 421-431.

7. Purdom, P. W. Jr., Stigler, S. M. Statistical properties of the buddy system. J. ACM 17, 4 (October, 1970), 683-697.

8. Purdom, P. W. Jr., Stigler, S. M., and Cheam, T. O. Statistical investigation of three storage allocation algorithms. BIT 11 (1971), 187-195.

9. Randell, B. A note on storage fragmentation and program representation. Comm. ACM 12, 7 (July, 1969), 365-369/372.

10. Russell, D. L. Internal fragmentation in a class of buddy systems. SIAM J. Comput. 6, 4 (December, 1977), 607-621.

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