Programming Techniques
C. Manacher, S.L. Graham Editors

A Buddy System Variation for Disk Storage Allocation

Warren Burton
University of East Anglia

A generalization of the buddy system for storage allocation is described. The set of permitted block sizes $\{SIZE_i\}_{i=0}^{n}$ must satisfy the condition $SIZE_i = SIZE_{i-1} + SIZE_{i-k(i)}$ where k may be any meaningful integral-valued function. This makes it possible to force logical storage blocks to coincide with physical storage blocks, such as tracks and cylinders.

Key Words and Phrases: buddy system, dynamic storage allocation

CR Categories: 3.89, 4.32, 4.39

Author's present address: Department of Mathematics, Michigan Technical University, Houghton, MI 49931.

A number of recent papers [1–4] have discussed variations of the buddy system [5, 6] for allocating blocks of storage. However, in each of these the set of block sizes $\{SIZE_i\}_{i=0}^{n}$ must satisfy the condition $SIZE_i = SIZE_{i-1} + SIZE_{i-k(i)}$ where k is a simple, usually constant, function. These methods tend to be unsuitable for disk storage allocation since logical blocks often overlap the boundaries of physical blocks (e.g. sectors, tracks, cylinders and disk packs).

By selecting a suitable function k, it is possible to prevent any logical block from overlapping the boundary of any physical block which is larger than the logical block. For example, some ICL disks have 128 words per sector (page), 1024 words per track and 10240 words per cylinder. Table I shows one sequence of block sizes which insures that every physical block is also a logical block. A binary-like scheme ($k(i)$ often 1) could also be used, with a block of size 10240 splitting into blocks of size 8192 and 2048.

Virtually any sequence of block sizes may be allowed (as a subsequence of the permitted block sizes) by use of a suitable function k. Table II illustrates how blocks of size 50 and 150 could be provided while using the ICL disks of the previous example. (This might be useful if frequent requests for blocks of size 50 or 150 were superimposed on a random request distribution.)

If blocks are split and recombined in the usual way, with the larger block resulting from an uneven split always being the left block, then the only new problem is determining the size of the buddy of a right block. (In Table I a right block of size 2048 may have a left buddy of size 2048 or 4096.) The method described by Cranston and Thomas [1] can be extended to solve this problem.

In [1], for each block of storage a control area contains information about the block. A new field, called the *I*-field, can be added to maintain information about the sizes of the buddies of right blocks. For a right block, the *I*-field will contain the index of the size of the block's buddy. (Alternatively, the difference between the indices may be stored, saving several bits.) For a left block, the *I*-field will hold the former contents of the *I*-field of its parent block. In this way the information about the size of a right block's buddy is preserved even when the block is split.

Due to the frequent updating of control areas, it may be advantageous to keep these in primary store. The control areas could be hashed on the first word addresses of their associated blocks (see [7]).

While problems of fragmentation in storage allocation are not well understood, the relative merits of buddy system variations are known to vary with the distribution of requested block sizes (see [4]). The flexibility of the method described here may prove increasingly useful as knowledge in this area increases.

This new variation of the buddy system has been developed for allocating storage for geometrical data

Table I. A Fibonacci-like Scheme ($k(i)$ often 2).

i	$SIZE_i$	$k(i)$
0	16	—
1	32	1
2	48	2
3	80	2
4	128	2
5	256	1
6	384	2
7	640	2
8	1024	2
9	2048	1
10	4096	1
11	6144	2
12	10240	2

Table II. An Irregular Scheme.

i	$SIZE_i$	$k(i)$
0	22	—
1	28	—
2	50	2
3	78	2
4	106	3
5	128	5
6	150	6
7	256	3
8	384	3
9	640	2
10	1024	2
11	2048	1
12	4096	1
13	6144	2
14	10240	2

items (polygons, sets of points, etc.) in *BEAUTIFUL* (Burton's East Anglia University Topographical Information Facility Utilizing Location).

Received July 1975, revised October 1975

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. 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 16*, 10 (Oct. 1973), 615–618.
4. Shen, K.K. and Peterson, J.L. A weighted buddy method for dynamic storage allocation. *Comm. ACM 17*, 10 (Oct. 1974), 558–562.
5. Knowlton, K.C. A fast storage allocator. *Comm. ACM 8*, 10 (Oct. 1965), 623–625.
6. Knuth, Donald E. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968, pp. 442–445.
7. Bobrow, D.G. A note on hash linking. *Comm. ACM 18*, 7 (July 1975), 413–415.

Short Communications
Management Science/Operations Research

Heaps Applied to Event Driven Mechanisms

Gaston H. Gonnet
University of Waterloo

Key Words and Phrases: discrete event simulation, event-scanning mechanisms, priority queues, heaps
CR Categories: 4.34, 8.1

I read with great interest F.P. Wyman's paper "Improved Event-Scanning Mechanisms for Discrete Event Simulation" [10] which, I found, makes a careful study of some techniques for event-list scanning.

However, all models analyzed present a data structure that is basically an ordered list. The primitive and most frequent operations that are performed over the set of future events are only two: add an event to the set and extract the next (minimum time) event. There is a large class of structures that allow these kinds of primitives, are more efficient and have some important advantages in their programming. These structures are commonly called "heaps." Their first appearance was related to sorting routines (Williams [9], Floyd [2] and Knuth [5]), but lately they have been applied to a wide variety of problems as a priority queue (Malcolm [6], Gentleman [3], Aho [1], etc.).

Other operations, such as finding the predecessor or successor of a given event, or any operation that finds an event based on an order relation over the keys, can be accomplished in $O(N)$ operations ($\frac{1}{2}n$ on the average). These operations are performed so infrequently that $O(N)$ time is acceptable.

Vaucher and Duval [8] discard the priority queues because they do not preserve the FIFO property. This may be easily overcome, if necessary, by constructing a compound key that contains the event creation time concatenated to the event time.

Using this kind of data structure, the maximum time required to add or extract one element from the

Author's address: Department of Computer Science, Faculty of Mathematics, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.