M. Z. Ghanem

Dynamic Partitioning of the Main Memory Using the Working Set Concept

Abstract: An algorithm to divide the main memory among N competing programs with different characteristics, running in a multiprogramming and virtual memory environment, is proposed. The algorithm is based on an optimal allocation policy, which is derived in this paper, using the concept of the working set. A brief description of the hardware implementation of the algorithm is also presented. It is shown that under this optimal allocation policy "the value of a page-frame" (the amount of reduction in the page fault rate if an additional page frame is allocated to that program) to each program is the same.

Introduction

There are two classes of automatic memory management policies, fixed partitioning and variable partitioning. The important advantage of fixed partitioning is the apparent low overhead of implementation, since partition changes occur as infrequently as possible—viz, when the set of active programs changes. This advantage can be very easily offset (even if the memory requirement of each program can be predicted prior to program processing) when one accounts for changing locality [1] in a program. Consider the behavior of a fixed partition when each program of the set of active programs (P_1, $P_2 \cdots P_N$) has a large variance in locality set size as time varies. Assume that P_i has rigidly been allocated m_i page-frames and P_j has rigidly been allocated m_j page-frames. Assume that at a given time, the size of the locality of P_i is less than m_i and the size of the locality of P_j is greater than m_j. Because the partition is fixed, there is no way to reallocate page-frames from P_i to P_j, even though such a reallocation would not degrade the performance of P_i but would improve the performance of P_j. Coffman and Ryan [2] have analyzed this effect by comparing fixed versus variable memory partition strategies in terms of the probability that the memory space which a program demands exceeds the allocated space. Their study suggests that variable partition strategy is better than the fixed partition strategy. Oden and Shedler [3] obtained a similar conclusion using a different approach. For an excellent and interesting discussion about the comparison between fixed and variable partition strategies, the reader is advised to read Denning and Graham [4, 5].

In this paper, we develop a variable partitioning algorithm which divides the main memory among N competing programs with different characteristics running in multiprogramming and virtual memory environments. The algorithm is based on an optimal allocation policy, derived in this paper, by using the concept of the working set. We also describe briefly the hardware implementation of the algorithm.

In the next section, we review some important variable partitioning algorithms and the motivations to develop this algorithm.

The section on optimal memory allocation contains the description of the process invoked in the derivation of the algorithm. The section on algorithm for implementation describes the algorithm proposed in this paper as well as a brief description of a hardware version for its implementation.

The last section contains the concluding remarks and recommendations for further research.

Variable partitioning

Several important algorithms for implementing variable partitioning strategies have been proposed in the past. Examples are "Global LRU", "Global FIFO" [6, 7], "Global FINUFO" (First In Not Used First Out) [8], and the "AC/RT" procedure [9].

Global LRU arranges all the pages of the active programs in a single global LRU list.

Global FIFO arranges all the pages of the active programs in a single FIFO list.

Global FINUFO arranges all the pages of the active programs in a circular list with a positioning pointer. Each page has a usage bit which is set to "zero" if the page has not been referenced and is set to "one", by the hardware, whenever the page is referenced. When a

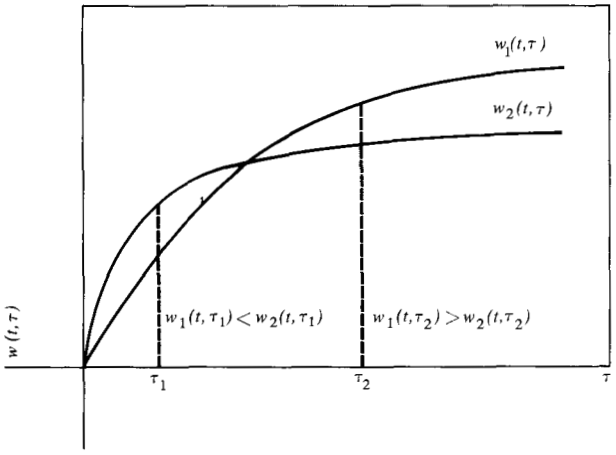


Figure 1 Possible qualitative relation between the working set sizes of two programs.

page fault occurs, the pointer is advanced around the list until the first page with a "zero" usage bit is found. The other usage bits are reset to "zero" when the pointer passes by. The page at which the pointer stops is chosen for replacement. Global FINUFO is the paging algorithm used in the Multics system at MIT [8].

In the "AC/RT" algorithm [9], each program has associated with it two variables, AC and RT, whose values are updated at each page fault of that program. The "activity count" AC_i, of the P_i program, registers the fraction of its allocated memory which has been referenced since its last page fault. The "round trip frequency" RT_i, registers the fraction of the last K page faults (K is a parameter) of P_i which caused the recall of the most recently replaced page. A high value of AC_i indicates that the P_i program is making effective use of its allocated memory. A high value of RT_i indicates that many mistakes are being made in the replacement decisions. The operation of AC/RT triggered by a page fault of P_i can be summarized as follows.

If the RT_i value is low, steal a page from the program itself. If the RT_i value is high, replace a page from the memory allocated to the program with the lowest AC_j value.

In spite of the apparent advantage of these variable partitioning techniques over the fixed paritioning policies, the reason for their success has not been formulated analytically. The analytical formulation will enable us to maximize that success. Chamberlin, Fuller, and Liu [10] have proposed an algorithm which distributes the main memory among the programs that are resident in the main memory such that the page fault rate of each of these programs is approximately the same. That is an intuitively appealing algorithm but no analytical proof to explain the expected success of the algorithm was given. A similar algorithm has been proposed independently by Chu and Opderbeck [11].

Another example of variable memory partitioning is the working set algorithm which takes into account the varying memory requirements during execution of a program. Denning [12, 13] introduced the concept of the working set to describe program behavior in virtual memory environments. The working set $W(t, \tau)$ of a program at time t is defined as the set of distinct pages which are referenced during the execution of the program over the interval $(t\text{-}\tau, t)$ where τ is called the window size. The working set size $w(t, \tau)$ is the size (or cardinality) of the set $W(t, \tau)$. The working set size as an indicator of program behavior may be sometimes misleading for the following reason: Two programs may have the same working set size characteristics for a certain window size, but their working set size characteristics may differ significantly for another window size. In a certain range of window sizes, the working set size for one program may be smaller than the working set size for the other program; but the reverse may be true in another range of window sizes (see Fig. 1). In other words, the working set size as an indicator for program behavior is a local indicator and highly dependent on τ. Thus, in order to better characterize program behavior, we need to introduce an indicator which is global (e.g., a working set size with dynamically changing parameter τ or the integration of the working set size curve with respect to τ).

In this paper, we transform the problem of partitioning the memory space into that of choosing an optimal (and varying) set of window sizes (one window size for each program) for which the working sets are defined. Then, we derive an optimal policy to allocate the main memory dynamically among many programs. Our optimization criterion is to minimize the sum of the page fault rates of all the programs which are residing in the main memory. For each program residing in the main memory, we derive *the value of a page frame* (the amount of reduction in the page fault rate that occurs when an additional page frame is allocated to that program). We show that for the optimal partition of the main memory, these values are equal. So we propose an algorithm that desitributes main memory among the active programs in such a way that these values (the value of a page frame to each of these programs) lie near each other.

An optimal memory allocation policy

The problem is to divide the main memory with a fixed size of M pages among N competing programs so that the sum of page fault rates is minimized. Let τ_i be the window size parameter associated with program i at a given time t. Let us further assume that a partition of

$w_i(t, \tau_i)$ page frames of main memory (hereafter we write $w_i(\tau_i)$ by suppressing t) is allocated to this program. The derivative of $w_i(\tau_i)$ with respect to τ_i is called the "missing-page rate" [13] and this quantity is approximately equal to the amount of increase in the working set size if we increase the window size by one. We assume that the sequence of working set sizes is locally stationary. Our problem is equivalent to finding a set of $\{\tau_i\}$ which solves the following problem

$$\text{minimize} \sum_{i=1}^{N} w_i'(\tau_i) \tag{1}$$

$$\text{subject to} \sum_{i=1}^{N} w_i(\tau_i) = M. \tag{2}$$

Combining the constraint $\Sigma_{i=1}^{N} w_i(\tau_i) = M$ with the cost function $[\Sigma_{i=1}^{N} w_i'(\tau_i)]$ by the appropriate Lagrange multiplier constant λ, we obtain

$$L = \sum_{i=1}^{N} w_i'(\tau_i) + \lambda\left[\sum_{i=1}^{N} w_i(\tau_i) - M\right] \tag{3}$$

which implies that for the optimal set $\{\tau_1, \tau_2, \cdots, \tau_n\}$ we must have

$$\frac{w_1''(\tau_i)}{w_i'(\tau_i)} = \lambda. \tag{4}$$

Let us define the quantity by

$$k_i(\tau_i) = -\frac{w_i''(\tau_i)}{w_i'(\tau_i)} = -\frac{dw_i'(\tau_i)}{dw_i(\tau_i)}\ ; \tag{5}$$

we see that $k_i(\tau_i)$ represents the amount of reduction in the missing page rate, if an additional page frame is allocated to program i. It is also clear that $k_i(\tau_i)$ represents the increase in the missing page rate, if one page frame is taken away from program i. So if there is an additional page frame available in main memory, it should be given to the program that has the largest $k_i(\tau_i)$; and if we want to steal a page frame from some program, then it should be taken from the program that has the smallest $k_i(\tau_i)$. For the optimal partitioning, therefore, all the k_i must be equal. That is to say, an optimal policy is to vary the τ_i to keep all the k_i as close to each other as possible. Note that the common value of all the k_i at the optimal solution is equal to λ, the Lagrange multiplier. On the other hand λ is equal to $\partial L/\partial M$ for the optimal partition; i.e., it is the amount of reduction in the minimized total missing-page rate, when the space available to these programs is increased by one page frame.

The cost function, which is also called the objective function $\Sigma_{i=1}^{N} w_i'(\tau_i)$, has been considered by assuming that the CPU is distributing its time equally among the N programs. We now demonstrate how our technique can be applied under different assumptions: If the CPU time assigned to program i is linearly proportional to the main memory space allocated to it, then the objective function is $\Sigma_{i=1}^{N} w_i(\tau_i) w_i'(\tau_i)$, and $k_i(\tau_i)$ is found to be

$$k_i(\tau_i) = -\left[w_i'(\tau_i) + \frac{w_i(\tau_i)\, w_i''(\tau_i)}{w_i'(\tau_i)}\right]. \tag{6}$$

In other cases, it may be better to maximize the summation of the lifetimes [7] of all the programs rather than to minimize the summation of the missing page rates. In this case the objective function is

$$\sum_{i=1}^{N} \frac{1}{w_i'(\tau_i)}, \text{ and } k_i(\tau_i) = \frac{w_i''(\tau_i)}{[w_i'(\tau_i)]^3}. \tag{7}$$

For a more complicated function such as the one shown in [14], it is suggested that the relation between the working set size and the lifetime function be used to derive the objective function. The result will be an expression dependent on the working set sizes and their derivatives with respect to the window sizes. Finally an algorithm can be developed by determining the value of the page frame.

Following is another example that indicates how to apply our technique under other assumptions. Let program i have a processing rate requirement that is different from the processing rate required by program j (due to the difference in their priorities). We assume that we can choose a vector $\alpha \equiv [\alpha_1, \alpha_2, \cdots, \alpha_N]$ such that α_j represents a relative figure of merit describing the importance of program j. In this case the problem can be formulated as the following:

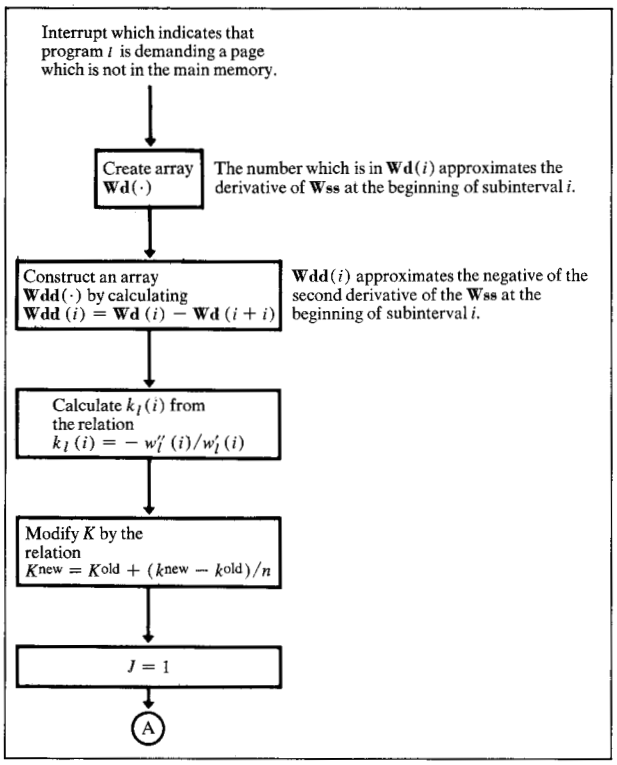
$$\text{minimize} \sum_{i=1}^{N} \alpha_i w_i'(\tau_i) \tag{8}$$

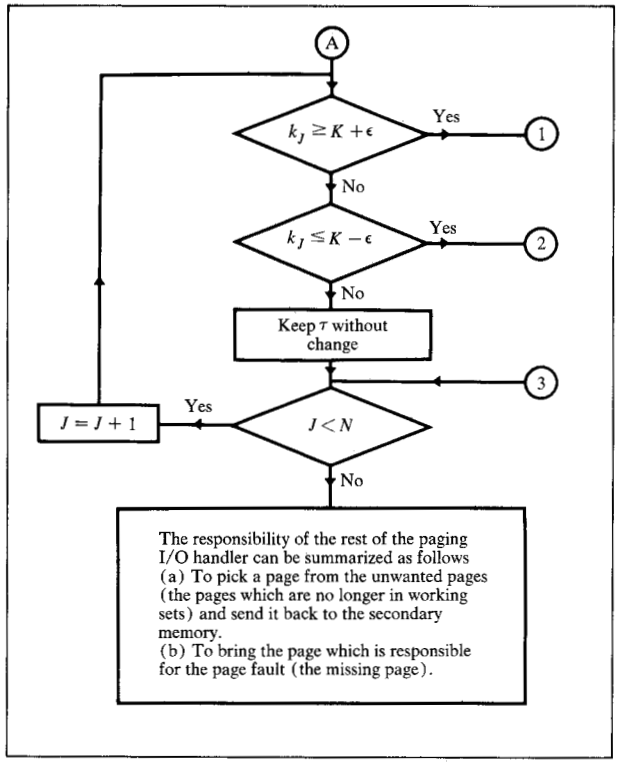
$$\text{subject to} \sum_{i=1}^{N} w_i(\tau_i) = M. \tag{9}$$

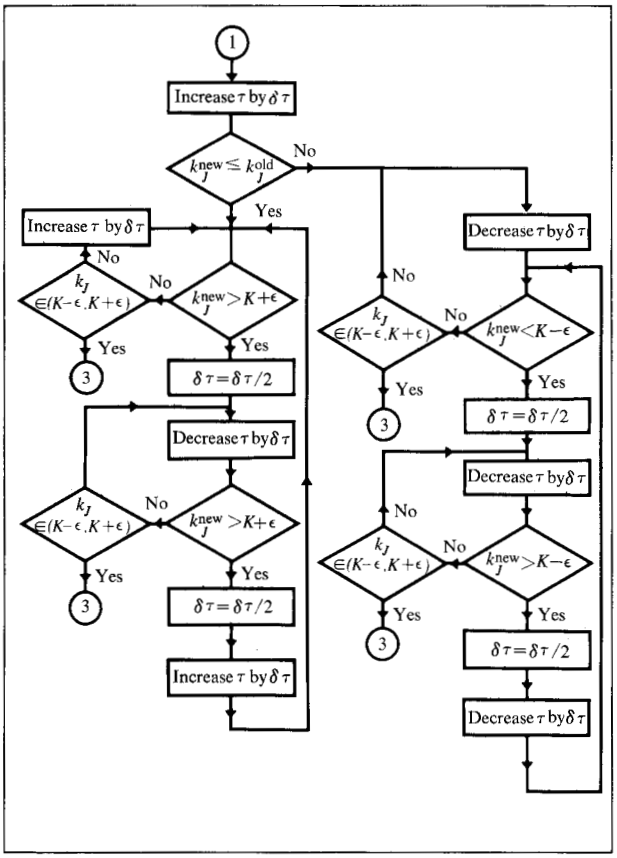
The interpretation of this result is as follows: Since program i is an important program, we conclude that for the optimal solution, the amount of reduction in the page fault rate (if we give this program an additional page-frame) can be less than the corresponding values for the other programs.

Algorithm for implementation

The block diagram of an algorithm to implement the optimal policy derived above is shown in Fig. 2. In the algorithm, we calculate (for each program j) an approximation of the value of a page frame of the program for 32 different values (Δ, 2Δ, 3Δ, $\cdots$,) of τ, where Δ = max. window size/2^p, where p is an integer. In other words, $k_j(\tau_j)$ is approximated for 32 values of τ_j. These calculations are carried out by calculating approximations of $w_j'(\tau_j)$ and $w_j''(\tau_j)$ for the same 32 values of τ_j







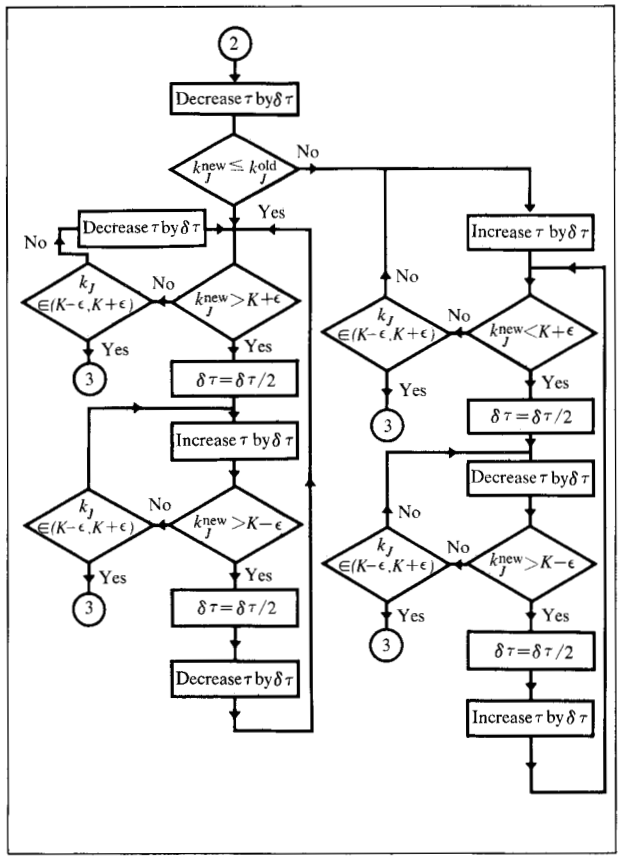


Figure 2 Block diagram for the proposed algorithm.

mentioned above. The algorithm determines the values of τ_j, such that all elements of the set $\{k_j(\tau_j)\}$ are within a small range 2ϵ around their mean K. Therefore, the algorithm distributes the page frames among the N competing active programs such that all values of k are in the neighborhood of their mean.

• *Remarks*

1. The small positive value ϵ is determined so as to avoid unnecessary oscillation of the window size between two values.
2. To determine the appropriate window sizes, a binary search is used due to considerations of convenience in the hardware implementation of the algorithm.

The process of generating an approximation of the derivative of the working set size with respect to the window size is the most critical part of our algorithm. All the other steps use well known operations such as addition, multiplication, etc. Let us now describe the process of generating the approximation of w'.

Each program has an array which has 2^p entries for some integer p (assume $p = 5$). This array is called **Wd**. **Wd**$(\cdot)$ is constructed to approximate the derivative of the working set size with respect to the window size. The steps to construct **Wd**$(\cdot)$, for the running program, are:

1. Every page frame in the main memory has a counter, RR, of p bits (see Fig. 3). In a sense RR is used as an aging register. If we assume that the RR entry corresponding to a certain page frame equals 4, then we are indicating that the instruction generating the last reference to that page frame occurred recently enough to be included among the last 5Δ instructions, but not recently enough to be found within the last 4Δ instructions.
2. For every Δ executed instructions, the counters RR of the page frames which belong to the running program are incremented.
3. Every time a page is referenced, the counter RR of the page frame which is accommodating that page is reset to zero.
4. Every time a page is brought to the main memory, the counter of the page frame which will accommodate that page is reset to zero.
5. In the event of a page fault, a microprogram (see Fig. 4) is initiated, which loads the program I.D. into register A and resets register B to 00000. Registers A. and B can now be used as argument registers for the associative memory C. The associative memory is equipped with a counter (register D) to count the number of matches. Associative operations continue using values in register B varying from 00001-11111. After each associative operation the content of register D is stored in the appropriate entry of the Wd array [e.g., if register B contains 00101, the content of register D is stored in **Wd**(5)].

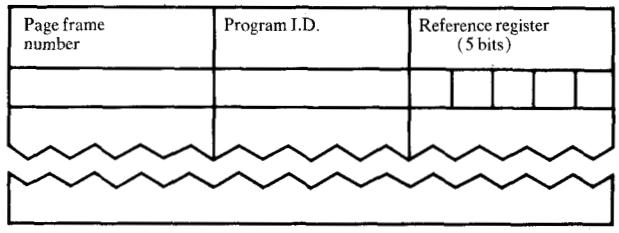


Figure 3 Information to be kept in associative memory for each page frame.

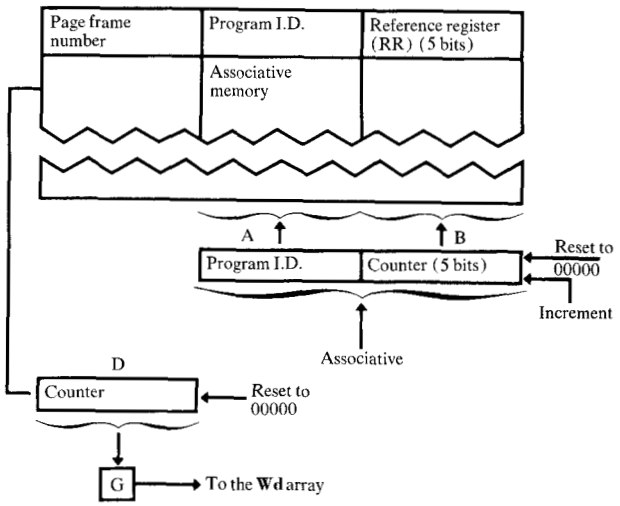


Figure 4 The **Wd** array

Tò show how the array **Wd**$(\cdot)$ approximates the derivative of the working set size with respect to the window size [**Wd**(i) approximates $w'(i\Delta)$], assume without loss of generality that $i = 4$. **Wd**(4) contains the number of page frames that have 4 in their RR registers (from step 5). Thus **Wd**(4) contains the number of page frames which have not been referenced within the last 4Δ instructions but have been referenced within the last 5Δ instructions. Therefore **Wd**(4) approximates $w'(4\Delta)$.

Concluding remarks

We have proposed a methodology for developing algorithms which efficiently and dynamically partition main memory among N competing programs. This methodology is based on expressing the optimization criterion analytically as a function of the working set sizes (of the N competing programs) and their derivatives (with respect

to their window sizes). The case for which the optimization criterion is the sum of the page fault rates was considered in detail. A brief description of a hardware implementation was also presented.

In this paper, we have assumed that there are N competing programs. The question of finding the optimal n (optimal degree of multiprogramming) is still an important question which has to be answered. Also we have proposed that the algorithm should be initiated after every page fault, but the question of how frequently such an algorithm should be initiated is another important question. The answer to this question depends on the effectiveness of this algorithm compared with the overhead imposed when it is used.

Acknowledgment

The author thanks R. Fagin and W. D. Frazer for their valuable comments.

References

1. P. J. Denning, "On Modelling Program Behavior," *Proc. Spring Joint Computer Conference* (AFIPS), Spartan Books, New York 1972.
2. E. G. Coffman, Jr. and T. J. Ryan, Jr., "A Study of Storage Partitioning Using Mathematical Model of Locality," *Comm. ACM* **15**, 185 (1972).
3. P. H. Oden and G. A. Shedler, "A Model of Memory Contention in a Paging Machine," *Comm. ACM* **15**, 761 (1972).
4. P. J. Denning and G. S. Graham, "Multiprogramming and Memory Management," *Proc. IEEE* (Interactive Computer Systems) **63**, 924 (1975).
5. G. S. Graham and P. J. Denning, "Multiprogramming and Program Behavior," *Report Number CSD-TR 122*, Computer Science Department, Purdue University, 1974.
6. L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Syst. J.* 5, 78 (1966).
7. L. A. Belady and C. J. Kuehner, "Dynamic Space Sharing in Computer Systems," *Comm. ACM* **12**, 282 (1969).
8. F. J. Corbato, "A Paging Experiment with the Multics System," *Report MAC-M-384*, Project MAC, Massachusetts Institute of Technology, Cambridge, July 1968.
9. L. A. Belady and R. F. Tsao, "Memory Allocation and Program Behavior under Multiprogramming," *Research Report RC 3469*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 1971.
10. D. D. Chamberlin, S. H. Fuller and L. Y. Liu, "An Analysis of Page Allocation Strategies for Multiprogramming Systems with Virtual Memory," *IBM J. Res. Develop.* **17**, 404 (1973).
11. W. W. Chu and H. Opderbeck, "The Page Fault Frequency Algorithm," *Proc. Fall Joint Computer Conference* (AFIPS), Spartan Books, New York 1972.
12. P. J. Denning, "The Working Set Model for Program Behavior," *Comm. ACM* **11**, 323 (1968).
13. P. J. Denning and S. C. Schwartz, "Properties of the Working Set Model," *Comm. ACM* **15**, 191 (1972).
14. M. Z. Ghanem, "Study of Memory Partitioning for Multiprogramming Systems with Virtual Memory," *IBM J. Res. Develop.* **19**, 451 (1975, this issue).

Received September 10, 1974; revised April 24, 1975

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.