F. J. Corbató
Massachusetts Institute of Technology

19 A Paging Experiment with the Multics System

Paging was first introduced in the Atlas computer[1] and promptly had great influence on system designers, for paging allows the solution of two problems in a systematic way: (1) It is possible to execute programs that are only partially loaded into primary memory or that require primary memory space larger than that available, and (2) It is possible to interrupt and remove programs from primary memory and later restore them with minimal storage allocation difficulties. These issues become especially important when one designs large-scale multiple-access systems in which multiplexing among user programs, which are competing for primary memory, is generally the case. Despite the obvious benefits that can result from paging, there is also a danger of extreme degradation of system performance whenever excessive paging activity occurs.[2,3] System degradation as a result of paging is, of course, caused by a mixture of overall system design (*e.g.*, a relatively small core memory) and the particular paging strategy employed. Since there still is not a great deal of insight into paging strategies, it is valuable to examine them so as to isolate this component of the general problem.

Introduction

Paging strategies and algorithms have been studied previously, for example, by Belady.[4] From these results and others, it is clear that there

* Work reported herein was supported (in part) by Project MAC, and M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102 (01).

is considerable statistical fluctuation in performing experiments, and more importantly, that there is a dependence on the characteristics of the program samples chosen and the computer environment in which the programs are operated. The strength of this dependence is still not well understood and is one reason for the present study. Moreover, because the Multics system[5] contains many radical departures from previous systems, it is a good candidate for investigating the invariance of results. Some of the departures that could easily affect paging behavior are: the use of recursive and pure procedures, the use of segmentation, the employment of a PL/I subset as a language for system programming, the homogeneous paging of much of the supervisor on the same basis as user programs, as well as the relatively large amount of pageable memory, and the particular page size.

With the above motivation for paging studies, it is important to recognize the goals and limitations that are being considered. It is, in principle, possible to design computing systems where all paging sequences are specified by the programmers. In this way, one would obtain more effective operation with near optimum sequencing of programs and data to and from secondary and primary memory. However, such a procedure is in g eneral impractical in the Multics sytem for a variety of reasons; one immediate obstacle is that the system is designed for the simultaneous multiplexing of multiple users and processors who in turn share a single copy of all common procedures or data. Thus a procedure may be simultaneously a member of several processes (*i.e.*, programs) each with different paging specifications. But even beyond this fundamental difficulty, there is the inconvenience to the programmer of determining and analyzing program behavior. Even granting that he were able to do it accurately and that he could specify data-dependent variations, there is still a problem that such specification represents unwanted clerical tedium. Thus, in large systems such as Multics, there is an important need for an automatic paging strategy that adapts to all situations, changes of programming habits, and even variations in the programming style or sharing patterns among the user population.

Having established that an automatic paging strategy is desirable, it also follows from the above considerations that such a strategy is constrained to using past performance and events to predict the future. It is only to the extent that there is a correlation of past and future behavior that a paging algorithm can have any effect. A key strategic question that a paging algorithm must answer whenever a new page is needed is: "Which page should be removed from core memory?" Two

possible strategies of page removal are those of first-in-first-out (FIFO) and least-recently-used (LRU). In the case of the FIFO algorithm, the justification is that it is easy to implement; it is particularly valid in those cases where pages are brought in for a brief initial burst of activity and then abandoned for long periods. In the case of the LRU algorithm, it is not obvious how to implement it precisely without expensive hardware design assistance or without incurring paging overhead several orders of magnitude greater than can be tolerated; nevertheless, the LRU paging strategy is probably a good one since one would expect a high correlation of least-recently-used pages with pages unneeded by a program in the immediate future.

The Multics Paging Algorithm

In the Multics system a paging algorithm has been developed that has the implementation ease and low overhead of the FIFO strategy and is an approximation to the LRU strategy. In fact, the algorithm can be viewed as a particular member of a class of algorithms which embody for each page a shift register memory of length k. At one limit of $k = 0$, the algorithm becomes FIFO; at the other limit as $k \to \infty$, the algorithm is LRU. The current Multics system is using a value of $k = 1$, and it is the purpose of the present study to explore the effect of variations of k.

To understand more clearly the significance of k, a simplified view of the Multics algorithm is first given. The algorithm that is called upon whenever a new page is needed keeps track of all potentially removable pages with a pointer into a circular list, an example of which is shown in Figure 1, and cyclically evaluates each page for either retention or replacement by the new page. Each page has associated with it a usage bit in its page table entry and the usage bit is turned on (*i.e.*, set to 1) by the processor hardware whenever the page is referenced (*i.e.*, read or written). Whenever the algorithm examines a page entry, it extracts the associated usage bit and enters it into the high-order position of a k-bit shift register after shifting the contents of the register one bit-position lower. Then if the shift register is nonzero, the page is retained; if the shift register is zero, the page is replaced by the new page. In either case the usage bit for the page is turned off and the circular list pointer is advanced.

Some properties of this class of algorithms can be deduced from the above description. It is clear that the case of $k = 0$ is nothing but the FIFO algorithm. Further, as k increases, the pointer of the circular list must on the average revolve k times as far as it does in the case of $k = 1$

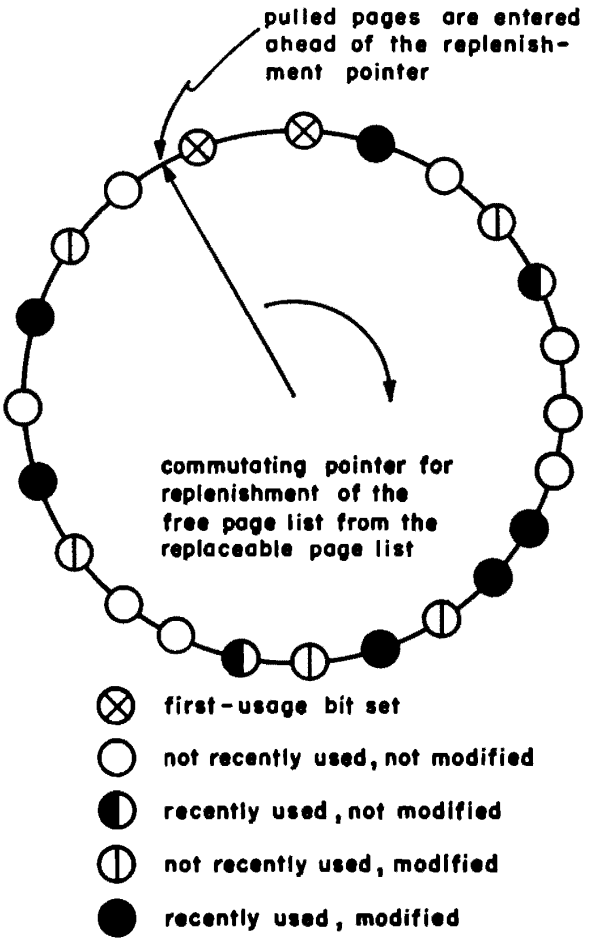

Figure 1. An example of the replaceable page list.

to yield as many page replacements. For large values of k, the mechanism takes on the character of a periodic sampled-data system where the number of leading (high-order) zeros in the shift register of a page is a direct measure of the number of cycles through the list since the page was used and hence the approximate time; in this way the LRU algorithm is reached in the limit.

An important aspect of this class of algorithm is that the overhead of operation increases with increases of k. It is therefore possible to engineer a practical algorithm by adjusting k to a value such that the reduction in page replacements due to the algorithm's improved predictive properties is just balanced by the increased overhead.

A critical aspect in the design of a paging strategy is the maintenance of stable operation under transient program behavior. Many of the algorithms studied by Belady have the property of cyclic performance

variations in that periodically all pages have their usage bits reset simultaneously. Similarly, an early version of the Multics algorithm suffered from a periodic sampling of page usage bits which was only loosely coupled with page replacement frequency. To consider this issue further and to examine the stability of the present Multics algorithm, the example of $k = 1$ is considered. In this case, the usage bits are distributed on the average such that half are on and half are off. For if the circular pointer sweeps through the usage bit 1, it will leave behind it the usage bit 0. Conversely, when usage bit 0 is swept through, usage bit 1 is left behind since the replacing page brought in will in the case of demand paging certainly be used. Furthermore, with periodic page replacements, if there is an excess of zero usage bits, the pointer rotation rate will slow down and more one bits will be created; conversely, for the opposite case, the pointer rotation rate will speed up. Thus, the algorithm's decision making mechanism, because it is synchronized to the traffic of page replacements and by its design contains negative feedback, is quite stable in its performance.

In the area of algorithms used by system programs, it is easy for misunderstanding to arise and for nontrivial aspects to be overlooked. For this reason a more detailed view of the precise mechanism of the Multics algorithm is given in Figures 2 and 3. As will be noted in the following explanation, several additional refinements over what has already been discussed are needed to create a working strategy.

Figure 2 gives the flow of control which occurs when a program finds a page needed from secondary storage and a page fault occurs. The initial test in Box 1 normally takes the "no" branch. Because its purpose is easier to understand after the main logic of the algorithm is established, an explanation is given below rather than here. In Box 2 a test is made to ascertain if the pool of empty blocks of primary storage is nearing exhaustion. (A free block pool is used so as to be able to service a page fault more rapidly without waiting for a page to be removed from primary core memory.) If the free block pool is low, Box 3 is entered to replenish the pool. (Figure 3 represents an elaboration of Box 3.) Control next passes to Box 4 where a free block is designated for the new page and a pointer to this block is entered into a list of pages being read (*i.e.*, "pulled") into core memory. In Box 5, the location on secondary storage of the new page is determined from a master data base called the Active Segment Table (AST) and the secondary storage drum controller is directed to transmit the page into core memory. However, before this transmission is complete the program in Boxes 6 and 7 proceeds to do several bookkeeping steps for all

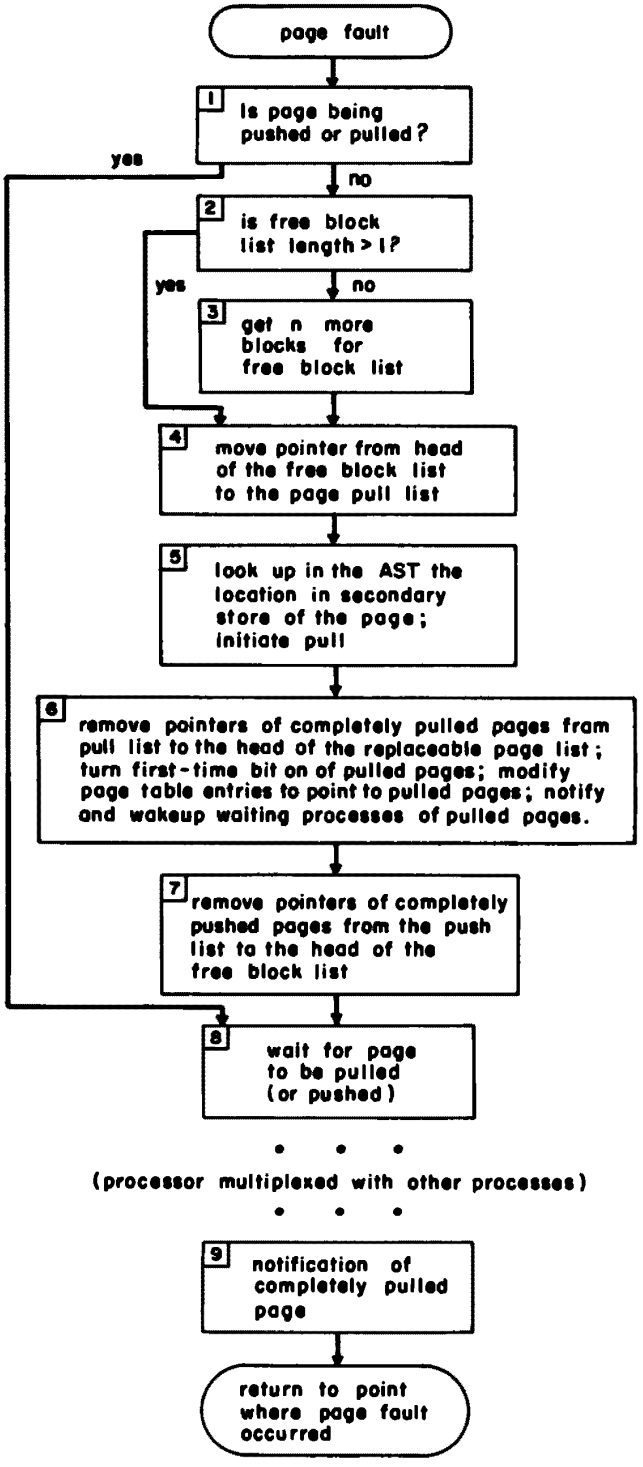

Figure 2. The Multics page fault algorithm.

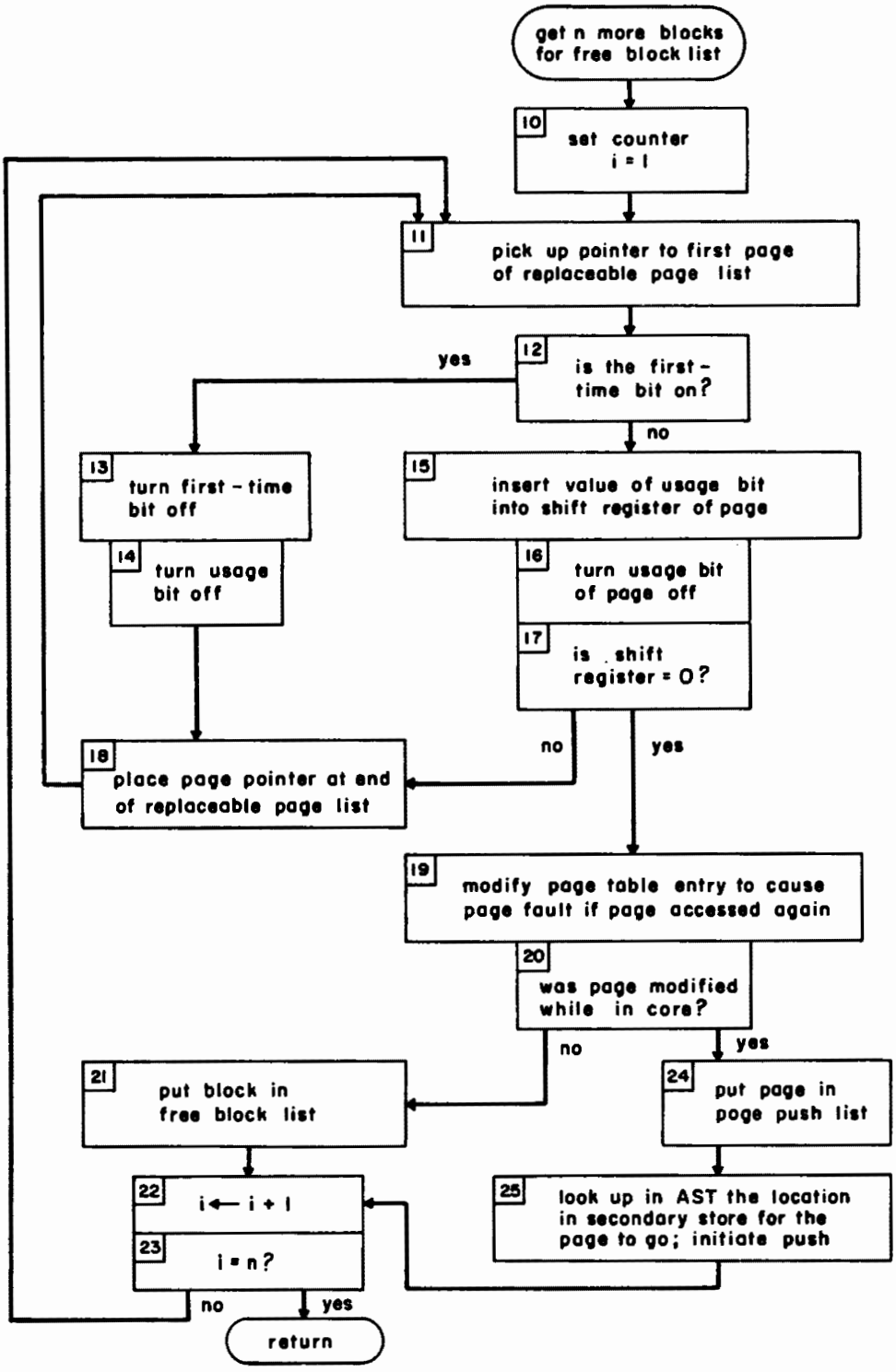

Figure 3. Replenishment of the free block list.

the drum-core transfers that have been completed since the drum status was last checked. One step is the switching of the page pointers from the pull list to the *head* of the replaceable page list. The reason for selecting the head rather than the tail of the list is that in addition a special "first-time" 1-bit switch is turned on for each of the newly pulled pages. As will be seen later in Figure 3, this switch makes it possible for the algorithm to ignore for a few page faults the initial burst of references to a page before it begins to monitor usage. (Otherwise a new page would always appear used after the first pointer rotation since the attempt to reference it is what triggered the pull.*) After setting the switches, the page table entries for the pages are adjusted so that attempts to reference the pages will succeed and the other processes which had been waiting for the arrival of pages now in primary storage are scheduled for processor service. Similarly in Box 7, the blocks corresponding to the pages which have been removed (*i.e.*, "pushed") from core memory to the drum are transferred to the free block list. At this point the process is unable to proceed until its needed page is pulled; hence in Box 8, control is transferred to the processor multiplexing section of the supervisor to await notification from another process. Finally at some later time when the page is pulled, another process schedules the original process for a processor, and in Box 9 control is returned to the point where the page fault occurred so that processing may resume as if uninterrupted.

The initial test in Box 1 only takes the "yes" branch in those relatively rare instances when a desired page is either being pulled into or pushed from primary memory. This situation can come about from two cases. In one case the needed page was in primary memory but became sufficiently inactive that a previous page fault has initiated its removal and the push is not yet complete. In the second case another process has developed a need for the page and has already initiated a pull. In either case, however, the flow of control is to Box 8 where the processor is released until the page transit is completed. At this time the process is restarted at the faulting location. If the page was being pulled, the process proceeds; if the page was being pushed, a page fault occurs and a fresh attempt is made to obtain the page.

* It should be noted that for the "first time" bit mechanism of the algorithm to be effective, the size of the free block pool should be somewhat larger than the number of processes being multiprogrammed; if this is not the case, the process which causes a page fault will probably not have an opportunity to run before the usage bit of the page in question is reset during free block pool replenishment. Under such circumstances the "first-time" bit mechanism can be removed with little effect on the overall algorithm.

Figure 3, which is largely self-explanatory, shows the flow of control required to replenish the free block list with n more blocks. (In Multics, n is currently set to a value of 3.) A few items are of note however. One is that the test in Box 12 is where the first-time switch allows the initial burst of usage of a pulled page to be ignored. Another item of note is in Box 20 where a test is made to avoid pushing pages which have not been modified while in core. (For this purpose the processor hardware assists by setting a modified bit in the page table entry whenever a page is modified.) In any case, however, control returns from the replenishment section when n blocks are selected for the free block list.

Finally it should be noted in passing that the algorithm presented in Figures 2 and 3, although basically correct, lacks many critically important features which are present in the version in the Multics system: These ignored features are needed for proper treatment of several complications. Some are

1. There are additional mechanisms needed for segment descriptor tables and for page tables. (In Multics these pages are either of two sizes: 64 or 1024 words.) In general there must be machinery for segment management.
2. Mechanisms are needed for the dynamic "wiring down" and "unwiring" of pages in core memory for purposes such as page tables and preprocess tables.
3. Data base interlock mechanisms and protocols must be established to allow multiple processors simultaneously to operate and take page faults.
4. Special attention must be paid in the details of the algorithm so as to allow simultaneous sharing of the pages of a segment by different processes.
5. Refinements are needed to allow for special situations such as pages which are created (with zeros) in growing segments.
6. In rare instances, the free block pool can become empty due to statistical fluctuations and special consideration must be made.
7. In the interest of coding efficiency certain mechanisms are not used exactly as described. For example, the push and pull lists are not formally created but are effected by the use of indicators in the elements of the replaceable page list.
8. In the case where the paging drum has completed several pulls but no page faults have occurred in other processes for an inordinate period of time, special provision must be made to reschedule for execution the processes which can proceed.

The Experiment and the Results

One of the purposes of this paper is to describe an experiment with the Multics paging strategy in which the value of k, the shift register length, was varied. To perform the experiment it was necessary to modify the system programs slightly but not in a way that seriously perturbs the results.

To serve as a test load, two cases were selected. The first case was the standard initialization computation which the system always performs to bootstrap itself in from the system tape, generate the configuration-dependent system environment, establish the paging and segmentation machinery, and then proceed to load the remainder of the system into secondary storage. Although there is a great deal of paging in this case, an objective is to establish whether or not the computation is typical.

The second case of the paging test was a short program written for the purpose which proceeded to call automatically a sequence of about ten basic, noninteractive commands. It was not felt that any more careful choice of sample was warranted since the Multics system is still undergoing rapid development and evolution. Moreover, past experience indicates that the commands chosen represent typical operations of a user population.

The environment of the experiment was a reproducible one-user, time-sharing system with a uniform page size* of 1024 36-bit words and a replaceable page pool of about 170 pages. Five distinct runs were made with different values of k and the values obtained are given in Tables 1 and 2. The page fault counts contain both those occurring explicitly as page faults and those contained implicitly within segment faults. (A segment fault occurs when there are neither pages nor segment page table in primary memory.) The Central Processor Unit (CPU) times given are those required to service the combined page and segment faults and do not include either the user terminal typing time or the delay due to the rotation time of the secondary storage drum.

It should be noted the average fault service times given are particularly large because of the present state of the Multics system at this time (July 1968). A soon-to-be-implemented strategy of reducing the number of effective segments by binding related groups together is expected to lower the average fault service time by at least a factor of four to the

* The Multics system potentially could have other effective page sizes as well as nonuniform page sizes. Future experiments may explore these directions although none are now contemplated.

neighborhood of a few milliseconds; the effect of this change will be to reduce the differences between the results for different values of k.

Conclusions

In examining the results given in Tables 1 and 2, several conclusions

Table 1. Page Faults *vs.* CPU Fault Service Time in System Initialization (total CPU time = 564.2 sec for $k = 1$)

k	Page Faults	Total CPU Fault Service Time (sec)
0	8309	184.5
1	4250	107.9
2	4098	106.5
4	4205	112.5
7	4317	120.2

Table 2. Page Faults *vs.* CPU Fault Service Time in a Sample Set of Basic Commands (total CPU time = 74.7 sec for $k = 1$)

k	Page Faults	Total CPU Fault Service Time (sec)
0	3628	72.9
1	1659	36.4
2	1635	37.5
4	1598	38.7
7	1725	44.3

can be drawn. First it is clear that the two cases give similar effects and that there is a dramatic improvement in going from $k = 0$ to $k = 1$. Second, a value of k in the range of 2 to 4 appears to give a paging strategy with the number of page faults down to a level where possible further improvements are small compared to statistical fluctuations. Third, when one examines the total fault service times, it is clear that as k increases so does the computational overhead of the paging algorithm. Since this latter basis is the pertinent one for comparison, the optimum value of k lies in the range of 1 to 2. Fourth, with so little difference between the results in the range of k from 1 to 4, a value of $k = 1$ is indicated. The reason for the latter choice is because a lower value of k

should produce an algorithm which is more stable and adaptive to transient changes in paging behavior. Fifth and finally, there should be some caution in extending the present results to other circumstances since: (1) the results are based on a small sample, (2) the Multics system may have properties which are uniquely its own, and (3) the system is still evolving and changing rapidly.

Acknowledgements

The Multics system is being developed on a cooperative basis by members of the Bell Telephone Laboratories, the General Electric Company, and Project MAC of M.I.T. The development and implementation of the paging strategy has been principally done by a team led by P. G. Neumann and R. C. Daley and including M. R. Wagner and G. F. Clancy. In addition, A. J. Goldstein made early background contributions, and F. J. Corbató assisted in developing the particular method of synchronizing page usage monitoring with page fault frequency.

The author would like to thank P. J. Denning, E. G. Coffmann, C. T. Clingen, and R. C. Daley for helpful discussions. Appreciation is given to J. W. Gintell and D. R. Vinograd who implemented the procedures used to meter the fault processing data. Finally, special and warm gratitude is extended to T. H. Van Vleck for his expert knowledge of Multics and his assistance in performing the experiments in the face of the extreme complexities which are present in the development of a large system.

References

1. T. Kilburn, " One-Level Storage System," *IRE Trans. Electronic Computers*, Vol. EC-11, No. 2 (April 1962).
2. P. J. Denning, " The Working Set Model for Program Behavior," *Commun. ACM*, Vol. 11, No. 5 (May 1968).
3. P. J. Denning, " Resource Allocation in Multiprocess or Computer Systems," (Ph. D. Thesis), *Project MAC Report MAC-TR-50* (May 1968).
4. L. A. Belady, " A Study of Replacement Algorithms For a Virtual Storage Computer," *IBM Systems Journal*, Vol. 5, No. 2 pp. 78–101 (1966).
5. F. J. Corbató and V. A. Vyssotsky, " Introduction and Overview of the Multics System," *AFIPS Conference Proceedings*, Vol. 27, Fall Joint Computer Conference (1965).