VIRTUAL STORAGE
REPLACEMENT POLICY UNDER PAGING

A project submitted in partial satisfaction of the requirements for the degree of Master of Science in Computer Science

by

Phillip Eugene Varner

January, 1981
The Project of Phillip Eugene Varner is approved:

Matthew Ek

Steven Stepanek

Russell Abbott, Committee Chairperson

California State University, Northridge
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>PART I</td>
<td>4</td>
</tr>
<tr>
<td>1.1 Terminology &amp; Clerical Considerations</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Assumptions</td>
<td>7</td>
</tr>
<tr>
<td>1.3 Autofolding and Static Allocation</td>
<td>9</td>
</tr>
<tr>
<td>PART II</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Segmentation</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Paging</td>
<td>24</td>
</tr>
<tr>
<td>PART III</td>
<td>38</td>
</tr>
<tr>
<td>3.1 Replacement and Page Faults</td>
<td>39</td>
</tr>
<tr>
<td>3.2 Replacement Policy</td>
<td>44</td>
</tr>
<tr>
<td>3.3 The Working Set Model</td>
<td>65</td>
</tr>
<tr>
<td>3.4 The Working Set: Implementation &amp; Storage Allocation</td>
<td>76</td>
</tr>
<tr>
<td>3.5 Working Set Summary</td>
<td>83</td>
</tr>
<tr>
<td>PART IV</td>
<td>84</td>
</tr>
<tr>
<td>4.1 Coding, VMSs, and the Structured Approach</td>
<td>85</td>
</tr>
<tr>
<td>4.2 Some Recommendations for Coding</td>
<td>91</td>
</tr>
<tr>
<td>PART V</td>
<td>94</td>
</tr>
<tr>
<td>5.1 SIMU</td>
<td>95</td>
</tr>
<tr>
<td>5.2 Some Utility Programs</td>
<td>106</td>
</tr>
<tr>
<td>5.3 The Simulations</td>
<td>111</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>124</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>126</td>
</tr>
</tbody>
</table>

iii
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1. Address Translation</td>
<td>126</td>
</tr>
<tr>
<td>A2. Secondary Devices</td>
<td>129</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>133</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>140</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.0</td>
<td>19</td>
</tr>
<tr>
<td>2.2.1</td>
<td>25</td>
</tr>
<tr>
<td>2.2.2</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1</td>
<td>50</td>
</tr>
<tr>
<td>3.3.1</td>
<td>67</td>
</tr>
<tr>
<td>3.4.1</td>
<td>80</td>
</tr>
<tr>
<td>4.4.1</td>
<td>86</td>
</tr>
<tr>
<td>5.1.1</td>
<td>96</td>
</tr>
<tr>
<td>5.1.2</td>
<td>103</td>
</tr>
<tr>
<td>5.1.3</td>
<td>105</td>
</tr>
<tr>
<td>5.2.1</td>
<td>107</td>
</tr>
<tr>
<td>5.2.3</td>
<td>109</td>
</tr>
<tr>
<td>5.3.1</td>
<td>114</td>
</tr>
<tr>
<td>5.3.2</td>
<td>115</td>
</tr>
<tr>
<td>5.3.3</td>
<td>117</td>
</tr>
<tr>
<td>A1.1</td>
<td>128</td>
</tr>
<tr>
<td>A1.2</td>
<td>130</td>
</tr>
<tr>
<td>A2.1</td>
<td>132</td>
</tr>
</tbody>
</table>
ABSTRACT

VIRTUAL STORAGE

REPLACEMENT POLICY UNDER PAGING

by

Phillip Eugene Varner

Master of Science in Computer Science

Virtual Memory Systems (VMSs) are an important development in computing systems. A primary means of implementing a VMS has been via a paging mechanism. In any VMS three storage, management policies must be developed: (1) fetch policy, (2) placement policy, and (3) replacement policy. Under paging, (1) and (2) become trivial, leaving the replacement policy as our primary concern. It is the replacement policy that is to be the focus of the paper.

In order to see the importance of the replacement policy, the position it occupies in VMSs is sketched (i.e., dynamic storage allocation via a paging mechanism is outlined as the preferable means of system implementation).
discussion of the replacement policy is then launched upon, culminating in the Working Set Model.

The paper is a collation of information from various sources presented as a tutorial. In addition, a simulator (SIMU) was developed to test various page replacement schemes. SIMU became of limited use due to the nature of reference strings.
INTRODUCTION

Virtual Memory System:
The Replacement Policy Under Paging

Virtual Memory Systems (VMSs) are intended to effectively extend the active storage available on the system beyond the amount provided as main storage. This is, of course, accomplished by use of secondary storage media such as drums or disks. The desired result is a storage system which provides a great deal of space and high speed access while at the same time keeping costs low. One wants the speed of the main memory and the size available on secondary memory at costs approaching that of the secondary storage devices. What results is a storage hierarchy of two or more levels. In its simplest configuration, the hierarchy has main storage interacting with secondary storage. As processor speed becomes greater and greater in comparison to the access speed of main storage (as is the trend), a cache storage becomes more and more desirable. Under such a configuration we have the cache storage as a buffer between main storage and the processor registers. Thus, the hierarchy is (1) Cache storage - (2) Main Storage - (3) Secondary storage. The hierarchy, of course, creates a management problem. Capacity in terms of size increases as we move through the hierarchy from (1) to
(3) (and beyond if necessary), presenting us with the problem of deciding what is to be resident where at any given time, since the smaller storage medial cannot accommodate all jobs active on the system at a given time (or a single job if it is large enough).

Hence, considering the simplest of cases; i.e., where we have a two-level hierarchy of main storage and secondary storage, some portion of the active job(s) (perhaps a very large portion) is at all times on secondary storage media. Some policies are required whereby the appropriate portions of code and/or data can be brought into main storage when required during processing. Of central concern will be the Replacement Policy under a paging VMS mechanism.

Parts I and II provide an orientation towards automatic "folding" (or blocking) or program texts with dynamic storage allocation techniques under a paging mechanism. Part III speaks directly to the replacement policy under a paged VMS (why the replacement policy is of particular concern is developed in Part II and at the beginning of Part III).

Part IV supplies a brief discussion of program coding in the context of VMSs, particularly "structured" programming techniques.

Part V is a discussion of the simulator (SIMU) and of the significance of results of several runs. Particular
attention is given to runs of an anomaly described in Part III. Listings, in Pascal, of SIMU and of several utilities for its use are provided.

The following footnote or reference policy is observed throughout. References appear in "[" and "]": [Al, #] where "Al" denotes the appropriate reference as it appears in the bibliography, and "#" is a page number in the reference Al (the page number is optional as required).

Certain discussions of topics not directly relevant to the central theme have been relegated to Appendices. These discussions seemed necessary, but also seemed to disturb the overall flow of the central theme. They can be referenced when interest or need arise.
PART I

Terminology and some basic assumptions are made explicit. A discussion of manual vs automatic folding as well as dynamic vs static storage allocation follows. The intent is to sketch out why automatic folding and dynamic storage allocation are preferable.
1.1 Terminology and Clerical Considerations

It will prove helpful to establish a certain amount of standard terminology.

A storage block is some number of contiguous words of main storage. The system page is a predefined block size into which user jobs, and main as well as secondary storage are broken; it being some \(ck\) words in extent (where \(c\) is some constant, often \(.5\), \(1\) or \(2\), and \(K = 1024\)). A page frame is a contiguous block of main storage (the size of a system page) into which one page slot fits. A page slot is one block (of size system page) of a user job on a secondary storage device. The reference string of a program is simply a record of references to or accesses of the blocks of the program in the order in which those references occur (note: the reference string for a program can vary from one run to another due to conditional branching; i.e., the reference string is to some extent data driven). Storage fragmentation will be considered the consumption of space in main storage due to needs other than specific specific requirements of jobs on the system (thus, most any VMS will induce some amount of fragmentation). Thrashing is the status that a VMS exhibits when excessive amounts of processor time are taken up with the management of blocks in the storage hierarchy—excessive time is being consumed in performing operations that are not directly relevant to
the processing that the user jobs wish to perform. A descriptor of either a page or a segment, contains pertinent information on the page/segment which is described. This information may involve: (1) whether the block is present in main memory, (2) where the block is in main memory, or where it is located on secondary storage if it is not in main memory, (3) whether the page in main memory has been changed or not, (4) for segments; their length, (5) protection codes, and other information where required. Page or segment tables of such descriptors are kept as maps of main storage.

Paging vs segmentation, if not already clear, will become so as we progress. Generally, a paged VMS exhibits fixed block size; a segmented VMS having variable block size.

A basic insight that made the development of VMSs possible is that the address space available to the user need not be identical with, or limited in size to, that of the physical address space available on the host machine. The virtual space is the user addressable space, irrespective of the size of main storage. The physical space is the actual address space inherent to a given machine's main storage.
1.2 Assumptions

Since the entire scope of virtual memory could not possibly be adequately dealt with here, and thus the intention is to deal with some portion thereof, it is imperative that a few basic assumptions be outlined.

What follows is not intended to argue the case for or against VMSs (virtual memory systems). VMSs' have their worth—whatever it might be.

The focus is to be on the multi-user environment. Thus, what is detrimental to system performance must be defined in terms of the community of users on a system. Hence, while a certain practice might well be to the benefit of a certain user, if the overall result is system degradation, then that practice must be seen as detrimental. Moreover, the concern here is with system operation in an environment of users with varying degrees of skill and expertise. I assume the typical user is not intimately familiar with computer systems, nor does he wish to become so familiar (assumption (le) below).

As the above comments may hint, the thrust of what follows is toward automatic virtual memory techniques. More precisely, and for reasons which will become clear, emphasis will eventually center on paging systems and a policy involved in their implementation (the page replacement policy).
Additionally, the following assumptions are made regarding desirable features of programs and computing systems.

(1a) Programs should be, insofar as possible, machine independent.

(1b) Programs ought to be modular; i.e., written such that the various portions are independently maintainable.

(1c) The handling of dynamic data structures should be facilitated.

(1d) While maintaining some level of efficiency, systems should provide ease of use.

(1e) Given (1d), it is reasonable to require as little system knowledge on the part of the user as possible. (Users interact with some level of virtual machine; thus, the virtual machine ought to be defined with ease of use in mind.)
1.3 Autofolding and Static Allocation

The initial need in any VMS is to map the virtual space onto the physical space. The problem is, of course, one of storage allocation. Available solutions run along two lines: (1) static allocation and (2) dynamic allocation. Concern here will center on the dynamic approach (as mentioned earlier), and with good reason.

As outlined by Denning [D1], the common ground upon which the dispute between static and dynamic allocation turns is with regard to:

(1) prediction of the availability of storage resources,

and

(2) prediction of the character of the reference string.

The static approach assumes that (1) can be given prior to run time (e.g., by the user) or be predicted (e.g., a result of compiler analysis of the program text) prior to execution and that (2) can be gained via preprocessing the program or again via compiler examination. The dynamic approach responds that (1) often cannot be predicted and that (2) can only be properly monitored during execution (since the reference string is to a great extent data driven).
To adjudicate between these two approaches we must consider how information about resource allocation is to be obtained.

The operating system must be able to collect information regarding system resources required by jobs. More particularly, storage requirement information must be collected. Generally, such information can be collected from two sources: the user, or the system and its utilities. Denning [D2] lists several reasons for not relying upon the user for advice on resource requirements. It is not altogether clear exactly what information that a user could provide would be of most use. Even if such information could be isolated, the user normally allocates resources with the intent of optimization for his particular job. What the user "sees" is normally a dedicated virtual machine, and his concern is not with others using the system. Hence, with user supplied information alone, overall system degradation could result. Again, we have focused concern on the system interacting with a community of users (and hence we must define a degradation in overall system performance as detrimental to each user). Even with such problems aside, a user often builds a job by use of procedures or utilities developed by others, the resource requirements of which are either unknown or of no interest to the user.
There remain, however, the system utilities (compilers, assemblers, etc.) which often operate on a job and which might provide information. This might be a more reliable source of information; however, there is again reason to discount the source. With the ever-increasing emphasis on program modularity, one cannot always be sure the appropriate information on the various program segments will be available—particularly at compile time (i.e., program modules may well be compiled at different times). Also, since code is often data dependent, the module requirements for a particular run cannot be known until run time and hence resource allocation must be delayed until that time. In addition, as mentioned by Denning [D1], for a compiler to have the capabilities to project resource requirements as it translates code, will increase the overhead incurred during compilation (and it is not altogether clear that such allocation, particularly storage allocation, could be done at that time (consider dynamic data structures) with enough accuracy to offset the increased overhead being counter productive). When more is required of a compiler, speed is normally lost to some degree, and again overall system degradation can result.

Thus, efficiency is not served by gathering the required information for resource allocation through user advice or compiler projections. The static approach of
resource allocation prior to run time seems inadequate for the above reasons, but additionally there are system reasons [D1] for preferring the dynamic approach.

It is desirable to be able to run partially loaded programs (in multiprogramming and timesharing environments) and to be able to load programs or portions thereof into spaces of arbitrary size. The space required by a program should be allowed to fluctuate during a run, this due to such capabilities as dynamic data structures, among other things. Another desirable feature is the ability to change system equipment without reprogramming and thus the programs should not be tied to specific storage needs ((1a) of section 1.2). The dynamic approach to allocation facilitates these capabilities.

Hence, the static approach has proven mistaken in its responses to (1) and (2), and thus essentially conflicts with the desirable state of affairs.

These considerations seem to indicate the choice of automatic memory management, in a dynamic, as opposed to static, scheme. Not only might the user be a poor source of information on storage needs, but the point is that in many cases the user should not have to be concerned with the resource management policies ((1d) and (1e) of section 1.2). To make the computer system more easily useful to a wider range of prospective users, resource management should be done automatically. However, even with these
considerations aside, Sayre [S1] provides results that indicate that automatic folding of the virtual space is at least as efficient as manual folding (the user defining the blocking of the virtual space, and managing the blocks during the run) if not more so. Thus, experimental evidence corroborates the dynamic approach's claim to superiority. As Denning comments:

... we may conclude that the best automatic folding mechanisms compete very well (and may indeed out-perform) the best manually folded texts. Virtual memory is thus empirically justifiable. [D1,160]

Having extolled the virtues of autofolding program text and dynamically allocating storage, attention can be turned to the basic techniques employed: segmentation, paging, and segmentation with paging.
PART II

Mechanisms and policies required in VMSs are introduced. Segmentation and paging are discussed, with a case for paging being sketched in terms of policy requirements among other things. Segmentation with paging is seen to be essentially a paged VMS mechanism.
2.1 Segmentation

The mapping from virtual space to physical space is accomplished in any VMS by dividing the program(s) to be run into blocks and allocating main storage for the blocks as they are needed (dynamic allocation has been specified here). The mode of blocking used provides the distinction between two basic approaches to implementing a VMS. The block sizes can be allowed to vary, resulting in what will here be termed segmentation, or the blocks can be of a predetermined fixed size, here termed paging.

Again, the center of concern is to be the paged VMS; however, a discussion of segmentation is in order and will serve to introduce basic mechanisms and policies that must be developed in either type of system, as well as supplying a forum from which to describe why the paged system and the replacement schemes therein were chosen for detailed concern. Hence, what follows is a discussion of segmentation, then of paging, and subsequently of segmentation with paging.

Under segmentation, block sizes are allowed to vary, segments being defined by the programmer. Thus, the program is divided into various segments, and when an instruction or data from a segment is required, a contiguous block of space in main storage must be found which is large enough to hold the entire segment. Thus, allocation
and deallocation problems arise, pointing up the need for governing policies.

Denning [D1] characterizes VMSs as requiring for implementation (a) mechanisms and (b) policies for use of the mechanisms. The mechanisms are the chosen technique for creating blocks in storage and in the program text (i.e., segmentation or paging) and the mechanisms required for translation (mapping) of the virtual space into the physical space (see Appendix A for address translation details).

Regarding the policies; where \( f(a) \) is the address mapping function and \( a \) is a member of the virtual space \( \text{vs} \) and \( a' \) is a member of the physical space \( \text{ps} \), the following is the case. For all \( a \in \text{vs} \) \( f(a) = a' \in \text{ps} \) (i.e., the desired word is in main storage) OR \( f(a) = \phi \) (i.e., \( a \) maps to no physical address; the desired word is not in main storage).

\[
\begin{align*}
  f(a) &= \phi, \text{ is termed a miss,} \\
  f(a) &= a', \text{ is termed a hit.}
\end{align*}
\]

When a miss occurs, the needed block must be loaded. If memory is full, a block must be removed to allow the loading. Hence, the following policies must be organized:
fetch rule - when to load a block,
placement rule - where to put the block,
replacement rule - what to remove from storage if placement is required and storage is full.

In both segmented and paged systems fetch and replacement policies develop from essentially the same notions. The fetch policy can be either anticipatory or demand based, although fetch on demand seems to be the widely preferred technique. The anticipatory approach requires a projection concerning which blocks will be needed at specified times in the future. The problem is, of course, accurately predicting block usage, and a major drawback is that when inaccurate predictions are made, one has not only wasted a great deal of time in terms of system overhead to make the prediction (which now turns out to be incorrect), but one has also incurred added time loss in loading blocks which turn out to be unnecessary and storage waste due to the space they have occupied. As an added detraction, it is likely that in loading such superfluous blocks one has had to remove other blocks which may have been recently needed—this can contribute to the development of thrashing. Hence, a fetch on demand scheme is generally preferable. Under the demand scheme, the fetch policy is essentially trivialized since one simply waits
until a block is required during processing to bring that block into main storage.

Regarding the replacement policy, again similar considerations arise in both segmented and paged systems. The replacement policy is invoked when a block must be brought into main storage from a secondary device and no room is available in main storage. Some block currently in main storage must be pushed out and obviously we want to push that block which is the least likely to be referenced again in the near future among all blocks currently resident in main storage. How we are to determine which block to push is, of course, the crucial question in need of an answer for development of a replacement policy. Discussion of answers to this question appear in Part III.

It is with respect to the placement policy that segmented and paged systems differ greatly. Under segmentation, main storage will often appear as in figure 2.2.0—a collection of blocks in use with various unused "holes" throughout. "Holes" are portions of storage not being used by any job. The development of holes is, of course, due to the various sizes that the segments can be. If a 16K segment "leaves" the system a 14K segment from another job may occupy that space, leaving a 2K hole. If a 3K segment requires loading and the 2K hole is the largest available, the hole is, of course, not usable. Either, (1) the job in need of the 3K segment is held up, until space becomes
Main Storage

Figure 2.2.0
available, or (2) a segment somewhere must be pushed to make room (a segment whose size is $\geq 3K$), or (3) segments must be moved in storage in an effort to coalesce a number of small holes into a sufficiently large hole. The problem lies partially in the segment's being one contiguous unit, which can only be loaded and run as such. The problem with this feature of segments becomes obvious when we note that if storage becomes badly fractured (a great many very small holes are scattered throughout storage) there may be no room to immediately load a segment when in actuality the sum of the space taken up by all of the holes is many times the size of the segment. Segments must be treated individually as linear, contiguous chunks of information. Hence, it is imperative that we minimize, not just the space occupied by holes but, more critically, the occurrence of small holes—since they can cause an invocation of the replacement policy even when the total space available is adequate for loading the segment.

The basic problem confronted here is, of course, fragmentation: external fragmentation to be precise. External fragmentation occurs in segmented VMSs, in that the space occupied by a segment is tailored to the size of that segment and thus the holes appear between segments (not within them). Control of external fragmentation is partially a function of the placement policy but moreover
requires the development of added policies directed exclusively at fragmentation control.

Briefly, placement policies under segmentation ordinarily involve searches of hole lists to accommodate a "best fit," "first fit," or some other policy (Knuth outlines results concerning placement policies in segmented VMSs in The Art of Computer Programming, Vol. VI). That is, the placement policy requires some sophistication in order to reduce, or minimize the threat to system performance posed by external fragmentation. Such sophistication can be reduced only at the cost (ordinarily a high cost) of developing some sort of compaction policy for coalescing holes:

Instead of using a sophisticated hole selection policy and no compaction, we may use a sophisticated compaction policy and no hole selection. [D1,167]

Certain other policies can be adopted to try to minimize external fragmentation; for example, allocating no smaller than a certain sized portion of storage, or perhaps allocating only on a preset size boundary (e.g., only on boundaries that are multiples of 64). Schemes such as Dijkstra's Tag Boundary Method and the Buddy System are attempts to limit external fragmentation and eliminate the need for compaction.
The two main points to be gleaned from the discussion of segmentation thus far are:

(1) fragmentation, here the external variety, can have a decidedly degenerative effect on VMS performance,

and

(2) the placement policy under segmentation is anything but trivial.

The above considerations do not speak well for segmentation as the choice of mechanisms. However, the intention has not been to belie segmentation so much as to draw attention to a couple of serious problems with segmented VMSs, one of which a paged system neatly eliminates. Segmenting certainly has its positive features.

Segmentation provides a ready scheme for enhancing program modularity since the various modules of a program can each be defined as distinct segments ((1b) of the Preliminary Assumptions). This also aids in maintaining machine independence ((1a) of the Preliminary Assumptions) and the ability to recompile a given module at will. Of great importance, particularly on multi-programmed and timesharing systems, is provision for protection and sharing of blocks of code or data. It is a simple matter to carry in the segment descriptors flags indicating priority of access. The flags are easily checked at the time
of attempted reference, since the descriptor for the segment must be consulted at the time of reference for virtual address translation to be carried out.

Keeping in mind the advantages present in the use of segmentation, attention can now shift to a consideration of paging.
2.2 Paging

The paging mechanism functions with fixed length blocks. Under such a scheme, certain advantages accrue to the VMS, which were not present under segmentation.

As mentioned earlier, a VMS exhibits a storage hierarchy of two or more levels. For a VMS of \( n \) levels \((S_1, S_2, \ldots S_n : n \geq 1)\) information must be managed among all these levels \((S_1\) being the storage from which the processor accesses information). A typical trilevel hierarchy involves a cache storage between main storage and processor registers (see Figure 2.2.1A). A limited number of pages (blocks) are maintained in cache (essentially, what we will later term the working set). The principles operative at the cache-main storage interface are much the same as those operative at the main-secondary storage interface, save that differences in speed of memory cycle can make some difference to page size—as will be pointed out later.

Since the operative principles are essentially the same in an \( n \)-level hierarchy as in a bilevel hierarchy, we will in the interest of clarity and simplicity, deal here with the 2-level hierarchy. When relevant, comments will be made concerning differences when other than the bilevel hierarchy is involved (e.g., when a cache is involved). Hence, the 2-level storage appears diagrammatically as in Figure 2.2.1B.
Figure 2.2.1 (A)

Figure 2.2.1 (B)

Figure 2.2.1 (C)
Drawing on the techniques of Denning [Dl] and Gecsei, Slutz, Traiger, and Mattson [Gl], the bilevel hierarchy can be defined as follows, in terms of memory states.

In simplest configuration; for one user with a virtual space of $n$ ($n \geq 1$) slots and physical space of $m$ frames ($m \geq 1$): given that

$I_{St}$ = the number of frames committed to use at a time $t$,

$S_t$ is the set of pages resident in frames of main storage, at $t$, (s a subset of all slots),

$r_t$ is the page (block) referenced at time $t$,

under a demand paging scheme (fetch on demand), with

$t = 1, 2, 3, \ldots, l$; the successive reference times,

and

$S_0 = \emptyset$ and $I_{S_0} = 0$

the following requirements govern the paging system:

1. If $r_t \in S_{t-1}$ then $S_t = S_{t-1}$
2. If $r_t \notin S_{t-1}$ and $I_{S_{t-1}} < m$ then $S_t = S_{t-1} \cup r_t$
3. If $r_t \notin S_{t-1}$ and $I_{S_{t-1}} = m$ then $S_t = (S_{t-1} - [x_t \in S_{t-1}]) \cup r_t$

where $x_t$ is determined via the replacement policy.

The definition in no way determines the nature of the replacement policy. (1) essentially says that if the
referenced page \( r_t \) is resident in main storage, the memory state remains unchanged (in terms of pages of the program present from \( t-1 \) to \( t \)). (1) specifies the situation referred to earlier as a "hit." (2) and (3) specify memory state transitions when a miss occurs. If all page frames are not in use at \( t-1 \) (i.e., \( |S_{t-1}| < m \), see (2)) then the page referenced is simply added to \( S \) and \( |S_t| = |S_{t-1}| + 1 \) (note: under paging we care not what specific frame is used during the placement; i.e., any free frame is used during the placement; i.e., any free frame will do). If all frames are in use at \( t-1 \) (i.e., \( |S_{t-1}| \geq m \), see (3)) then some page (i.e., \( x \)) must be removed from \( S \) (a frame must be cleared) so that \( r \) can be added. In the latter case, we care greatly what frame is chosen to have its current page pushed from main storage.

Requirement (2) points up an important distinction between paged and non-paged (segmented) systems. As mentioned near the end of 2.1, under segmentation the placement policy is nontrivial, in that a space of adequate size must be found. (2) above is not really adequate as a requirement in a segmented system because the right side of the conjunct in the antecedent of the conditional only involves the number of blocks (here pages) currently in storage and some maximum number of blocks that could possibly be in storage (i.e., \( m \)). The situation under segmentation is not that simple. One has to determine, not
just if there are any openings but rather if there are any openings with requisite characteristics and if a number of such openings exist, which among all those available should be used. Decision structures and criteria must be provided under segmentation for the placement policy, while with paging the placement policy is essentially trivialized. One does not care which available frame is used, only if there are any available. Moreover, there are a determinant number of frames into which main storage is broken. (2) evinces the ease with which placement is done in a paged system. Hence, problem (2) from section 2.1 is eliminated.

The other problem mentioned in section 2.1, that of fragmentation, while not being eliminated, has had its nature changed in a paged VMS.

Fragmentation in VMSs is of essentially three types: External, Table, and Internal. External fragmentation, as mentioned in section 2.1, occurs in segmented VMSs when holes appear as segments are allocated and deallocated—causing placement policy complication, compaction problems or both. Since compaction is ordinarily a time consuming and inefficient remedy, placement policy complication is usually opted for under segmentation.

Internal fragmentation occurs in paged VMSs when a page frame is allocated to a job and some portion thereof is unused. (See Figure 2.2.1.(C).) Ordinarily, internal
fragmentation is limited to the last page of a job, when the amount of information is not sufficient to completely fill the last frame allocated for the job (as shown in the figure). The worst possible situation occurs when a job is of the size $cx + 1$; where $c$ is an integer constant and $x$ is the number of words in a page. That is, the job requires a page frame for but one word; the remainder being relegated to internal fragmentation. Consideration of page size and coding techniques can help defer the damage done in such cases—as will be seen. The best possible case occurs when a job is of the size $cx$ ($c$ and $x$ as above). Here, the job neatly fills some number of pages; no internal fragmentation occurs. As is apparent, external fragmentation cannot develop in a straight-forwardly paged system since no storage exists outside of the page frames. As will be discussed shortly, careful consideration of page size can aid in limiting internal fragmentation.

Table fragmentation is characteristic of both paged and segmented VMSs, since this amounts to the space in main storage occupied by the address mapping tables (which are required under both VMS mechanisms—see Appendix A).

Thus, under paging, the fragmentation problem is shifted from external to internal (a more predictable beast), and the placement policy is essentially trivialized. The placement policy amounts to using any available frame, and if one is not available, make one available and
use that one. Hence, the onus rides with the replacement policy.

With this, we come to four critical parameters in a paged VMS:

1. Replacement algorithm choice,
2. Memory size,
3. Page size,
4. Transport time.

(1) is to be dealt with later on a grand scale, and (2) relates nicely to (1) in some respects. Thus, (1) and (2) are pointed out here, and left as promissory notes. Hence, we may take up (3) Page Size and related thereto (4) Transport Time.

To determine the appropriate or optimal page size, the influencing factors must be isolated. The above discussion indicates that internal fragmentation is of concern. The loss of storage to fragmentation in the last page of a job, while certainly not in all cases occurring as in the worst possible situation, can none-the-less be cause for concern. Since we cannot predict for programs generally how they will fair in terms of internal fragmentation, we can only randomly project what the fragmentation in the last page will be and thus assume an expected loss of \( x/2 \); where \( x \) is the page size. Now, obviously we can minimize the loss by choosing a page size of one word—i.e., there will be no loss. This, however, as is equally
obvious, will not prove practical, the reason being partially due to the fourth and related parameter, Transport time. With the typical secondary devices upon which the program's pages will reside until required in main storage, one can access an entire chunk of information in nearly the amount of time required to access a single word of information. This characteristic of the secondary devices is due, of course, to read/write head alignment and rotational delay. Even where these problems can be eliminated by use of non-mechanical secondary devices (see Appendix A) the processor registers and main storage memory cycle time are still ordinarily much faster than the secondary devices. Hence, to access one word from the secondary device requires an unacceptable amount of busy wait time on the part of the processor and perhaps of main storage (even with a few words accessed at a time an intolerable wait can occur). Thus, the page size must be greater than one; indeed, somewhat greater than one.

Denning [D1] supplies a formula for generating an optimal page size in terms of table fragmentation cost and internal fragmentation cost, given an average segment size. The optimal page size $x_0$ is:
\[ x_0 = (2cs)^{1/2} \]

where \( c = \frac{c_1}{c_2} \)

- \( c_1 \) = cost of table fragmentation,
- \( c_2 \) = cost of internal fragmentation,

and \( s \) = the average segment size.

Some results are presented in Figure 2.2.2. The results shown are from a program written to allow one or the other (but not both simultaneously) of internal fragmentation cost or average segment size to vary. Figure 2.2.2 B shows results when the average segment size is held constant at 1K, and the cost of internal fragmentation starts at 1 and is allowed to grow. The table shows what is to be expected; for a given segment size, as the cost of internal fragmentation becomes higher and higher in relation to table fragmentation, the optimal page size becomes smaller and smaller. This seems reasonable, since the more the fragmentation costs the more important it becomes to limit such fragmentation, and the smaller the page size the smaller \( x/2 \) becomes (i.e., the expected word loss to internal fragmentation).

Just how the cost of, for instance, internal fragmentation is to be determined is not altogether clear from Denning and others. However, Denning mentions that with \( s \) = 1000 and \( c = 1 \) (the overall cost of fragmentation) we get \( x_0 \leq 45 \) words [D1,169]. The tables of Figure 2.2.2 A and B seem to bear this out. Figure 2.2.2 A shows the
Table fragmentation cost: 1  
Internal fragmentation cost: 1  
Initial power of 2: 2  
P20: 20  
P32: 32768

Cost = Table Fragmentation / Internal Fragmentation

<table>
<thead>
<tr>
<th>Cost</th>
<th>Segment size</th>
<th>Optimal page size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 / 1</td>
<td>4</td>
<td>2.828427</td>
</tr>
<tr>
<td>1 / 1</td>
<td>8</td>
<td>4.000000</td>
</tr>
<tr>
<td>1 / 1</td>
<td>16</td>
<td>5.656854</td>
</tr>
<tr>
<td>1 / 1</td>
<td>32</td>
<td>8.000000</td>
</tr>
<tr>
<td>1 / 1</td>
<td>64</td>
<td>11.313708</td>
</tr>
<tr>
<td>1 / 1</td>
<td>128</td>
<td>15.000000</td>
</tr>
<tr>
<td>1 / 1</td>
<td>256</td>
<td>22.627417</td>
</tr>
<tr>
<td>1 / 1</td>
<td>512</td>
<td>32.000000</td>
</tr>
<tr>
<td>1 / 1</td>
<td>1024</td>
<td>45.294834</td>
</tr>
<tr>
<td>1 / 1</td>
<td>2048</td>
<td>64.000000</td>
</tr>
<tr>
<td>1 / 1</td>
<td>4096</td>
<td>90.593668</td>
</tr>
<tr>
<td>1 / 1</td>
<td>8192</td>
<td>128.000000</td>
</tr>
<tr>
<td>1 / 1</td>
<td>16384</td>
<td>191.019336</td>
</tr>
<tr>
<td>1 / 1</td>
<td>32768</td>
<td>255.000000</td>
</tr>
<tr>
<td>1 / 1</td>
<td>65536</td>
<td>362.038572</td>
</tr>
</tbody>
</table>

Ready.

Figure 2.2.2 (A)
TABLE FRAGMENTATION COST: 1
INTERNAL FRAGMENTATION COST: 1
INITIAL POWER OF 2: 10
C2TERM: 20
PGPTERM: 32768

COST = TABLE FRAGMENTATION / INTERNAL FRAGMENTATION

<table>
<thead>
<tr>
<th>COST</th>
<th>SEGMENT SIZE</th>
<th>OPTIMAL PAGE SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 / 1</td>
<td>1024</td>
<td>45.254834</td>
</tr>
<tr>
<td>1 / 2</td>
<td>1024</td>
<td>32.000000</td>
</tr>
<tr>
<td>1 / 3</td>
<td>1024</td>
<td>26.127891</td>
</tr>
<tr>
<td>1 / 4</td>
<td>1024</td>
<td>22.627417</td>
</tr>
<tr>
<td>1 / 5</td>
<td>1024</td>
<td>20.238577</td>
</tr>
<tr>
<td>1 / 6</td>
<td>1024</td>
<td>18.475209</td>
</tr>
<tr>
<td>1 / 7</td>
<td>1024</td>
<td>17.104719</td>
</tr>
<tr>
<td>1 / 8</td>
<td>1024</td>
<td>16.000000</td>
</tr>
<tr>
<td>1 / 9</td>
<td>1024</td>
<td>15.084945</td>
</tr>
<tr>
<td>1 /10</td>
<td>1024</td>
<td>14.310835</td>
</tr>
<tr>
<td>1 /11</td>
<td>1024</td>
<td>13.644846</td>
</tr>
<tr>
<td>1 /12</td>
<td>1024</td>
<td>13.063945</td>
</tr>
<tr>
<td>1 /13</td>
<td>1024</td>
<td>12.551433</td>
</tr>
<tr>
<td>1 /14</td>
<td>1024</td>
<td>12.094063</td>
</tr>
<tr>
<td>1 /15</td>
<td>1024</td>
<td>11.684748</td>
</tr>
<tr>
<td>1 /16</td>
<td>1024</td>
<td>11.313708</td>
</tr>
<tr>
<td>1 /17</td>
<td>1024</td>
<td>10.975909</td>
</tr>
<tr>
<td>1 /18</td>
<td>1024</td>
<td>10.666657</td>
</tr>
<tr>
<td>1 /19</td>
<td>1024</td>
<td>10.342171</td>
</tr>
</tbody>
</table>

Figure 2.2.2 (B)
optimal page size with cost constant at 1 and the segment size varying by powers of 2.

With the optimal page size results being what they are, it may be asked why do many systems employ page sizes of 512 or 1024 words? The answer lies, of course, with the transport time between the levels of storage. It is here that differences lie between management of, for example, the main-secondary interface and management of the cache-main storage interface. With the latter management problem, where the hierarchy puts cache storage as S1 and main as S2, the speed with which S2 can function in relation to S1 may be such that the optimal page size results can be approached. Hence, with cache to main storage management the page sizes can be much smaller than with main to secondary storage. Generally, as the speed ratio of one storage level in the hierarchy $S_n$ to another $S_{n+1}$ approaches 1, the page size used can shrink correspondingly. ". . . there is a great discrepancy between the page size for maximizing storage utilization and the page size for maximizing page-transport efficiency . . . " [D1,170]

Before turning to memory size and the replacement a couple of additional matters require comment.

A look at paging VMSs evinces certain advantages over strictly segmented VMSs. Paging provides a guaranteed uniform block size, which is instrumental in allowing the trivialization of the placement policy. To add to its
attractiveness, consider that not only can portions of a program be loaded and still run (segmentation provides for this), but portions of what under segmentation would have been a segment can be loaded and run (something for which segmentation did not allow). The importance of this feature in terms of storage space and loading time saved becomes apparent when we realize that a program may well, as often happens, require only a limited (perhaps very small) portion of the code of any given segment. Under paging, with sufficiently small pages, unneeded portions of a segment, or the greater portion thereof, needn't be present, occupying space in main storage for the program to run. Hence, in a sense, compaction is accomplished in a paging system, since code that is superfluous to the run need not occupy space in main storage.

Paging, is not, however, without its detractions. The important features of block protection and block sharing which were facilitated by the segmentation mechanism become more difficult under a strict paging VMS. A segment will typically be broken into several pages, and relations among pages will be difficult to trace. This also shows how the modularity of programs can be destroyed.

The paging mechanism, while more desirable than segmentation in some important respects, seems to violate certain of the basic features found desirable in systems (as did segmentation). The solution to the dilemma lies
close at hand: combine segmentation and paging. Allow variable size segments, but build segments out of the present pages. What results is essentially a paged VMS, but certain of the advantages of segmentation are acquired. (See Appendix A for an outline of the address mapping scheme). Segmentation with paging seems to afford the optimal VMS since we seem to gain many of the previously noted advantages of both, while experiencing little of their detractions. As mentioned, such a VMS will here be considered essentially a paged VMS, since, as under paging, the replacement policy is of primary concern and the basic unit of storage allocation is the system page. Henceforth, discussion of replacement schemes in paging systems is considered applicable to paging systems with segmentation even as the discussion may speak in terms of paging alone. For outlines of systems which actually employ the approach of paging with segmentation, see Appendix C.
PART III

It is time to make good on the promissory note of Part II regarding replacement policy choice and memory size.

The importance of minimizing page faults is discussed as an appropriate criteria for judging replacement algorithms (the Principle of Optimality is introduced). Several classifications of replacement algorithms are developed; realizable vs unrealizable and Belady's three classes of algorithms. Several algorithms are discussed culminating with the working set model (which turns out to be more than a replacement policy).
3.1 Replacement and Page Faults

It will become apparent that the replacement policy can have a great effect on how often blocks must be brought in from (or written to) secondary storage. If we let:

\[ A = \text{access time for information in Main Memory}, \]

and

\[ T = \text{transport time for information moving between main and secondary storage}, \]

the importance of avoiding transports is realized when we realize that ordinarily \( T \gg A \). Since every time a page fault occurs (i.e., \( f(a) = \emptyset \), a miss) transport time is incurred (2\( T \) when \( |S_{t-1}| = m \); \( T \) when \( |S_{t-1}| < m \)) the relation between faults and time expended in waiting for transports to finish is a direct relation. Since the page transport busy wait time is undesirable, the minimization of faults (i.e., misses) is highly desirable (since to minimize faults is to minimize the collective effect of \( T \) over a run). Transports can also have the negative effect of slowing system functions (in terms of real time) since with a DMA, cycle stealing is induced.

Hence, a good criteria by which to judge policies generally (and replacement in particular) is minimization of page faults. An advantage lies therein over such criteria as running speed comparisons of algorithms that implement replacement policies. First, the speed with which the algorithm runs is somewhat dependent upon the
system and on the design of the algorithm. Second, the sheer speed with which the algorithm would run is ambiguous between at least two senses.

Sense 1 - speed over a processing run,

and

Sense 2 - speed over a typical invocation of the algorithm

Given 2 algorithms A and B, with individual run times of 2 units and 3 units respectively, A is the preferable algorithm in Sense 2. If, however, upon processing some reference string Z, B (being an algorithm that minimizes page faults) is invoked 3 times, while A is invoked 10 times then over the entire run, time consumption is:

A (2)(10) = 20 units,

B (3)(3) = 9 units.

In Sense 1, algorithm B is minimal. In addition, we must again consider the transport times involved when faults occur.

While an algorithm that consumes excessive amounts of time in performing its function (which, it will be seen, is essentially what is the case with many of the truly optimal algorithms) must be considered detrimental and counter productive, the criteria of primary concern must be that of page faults.
Thus, a guiding principle can be gleaned from these considerations. In order to minimize page faults it is necessary to maximize the time between faults. We must endeavor to replace that page, among those currently in main storage, which has the longest interval from the current reference to that page's next reference (or if complete knowledge is not possible, that page which we expect has the longest such interval.)

The Principle of Optimality (P.O.):

Let \( S = 1', 2', \ldots, m' \) be the memory state at time \( t \), the moment of a page fault, and let \( t(i') \geq t \) be the earliest moment at which page \( i' \) is next referenced. Define \( Tt(i') = t(i') - t \). Replace that page \( i' \) for which \( Tt(i') \) is maximum. If the future is not precisely known, replace that page \( i' \) for which the expected time \( E(Tt(i')) \) is maximum. \([D1,178]\)

The preference for a paged VMS has been sketched. Of the three policies considered, placement has been seen to become trivial under paging and it has been noted that a demand policy seems the preferable route for the fetch policy, considering its simplicity. However, the demand fetch's claim to superiority is open to question unless a proof of that claim is forthcoming. Such a proof is provided by Gecsei, Slutz, Traiger, and Mattson \([G1,109-10]\). The proof shows that for any nondemand (anticipatory) fetch paging scheme there exists a fetch on demand scheme which incurs no more, and perhaps less, page faults than does the anticipatory scheme. The importance of this result is not only that we can deal in terms of demand paging systems
alone (since any nondemand paging system reduces to one
that is demand based), but the demand scheme becomes the
overridingly desirable fetch policy since in its sim-
plicity, in comparison to the anticipatory approach, it
helps relieve system overhead time incurred. Under paging,
the fetch and placement policies have thus both been
trivialized, truly leaving as the major concern under such
a mechanism the replacement policy.

It will be useful to have conventions for referring
to reference strings generally, and to replacement algo-

Let: \( R \) = a reference string, whose elements are
\( r_1 r_2, \ldots, r_n \): \( n \) distinct references to pages of
a program (not necessarily distinct pages),
where the indexes 1 to \( n \) indicate the ordering
of the references,

\[ R^i = \text{the sub-reference string to } R; \text{ consisting of all}\]
\[ r_1 \text{ thru } r_i \text{ elements of } R, \]

\( A \) - be a replacement algorithm which implements a
replacement policy,

and \( m \) = the number of pages of main storage available
(either the size of main storage; or in the case
of multiprogrammed environments, the storage
allotted to the process wishing to run),

such that:

\[ S(A,m,R) \]

is the storage status after \( A \) has processed \( R \) in storage \( m \).
The replacement scheme can thus be seen as defining storage
status transitions when (recalling the definition in section 2.2).

\[ r_t \notin S_{t-1} \text{ and } |S_{t-1}| = m. \]

This situation requires that some frame \( x \) be chosen for replacement, such choosing being done by the replacement algorithm (call it A) that implements the replacement policy.
3.2 Replacement Policy

P.O. implores us to maximize $E(Tt(i'))$. Resolving to do so provides a perspective from which to work, but by no means does it settle what the replacement policy (or the algorithm that implements it) is to be. In one way or another, the behavior of reference strings must be characterized, and this will have a great deal to do with how one ought to go about maximizing $E(Tt(i'))$. Generally, there seems to be three approaches to the development of a replacement policy, based in commitments about the character of reference strings (and thus what must be done to maximize $E(Tt(i'))$).

(A1) Reference strings universally exhibit a fixed and predictable character (i.e., pay no attention to dynamic reference density or reference behavior over an interval of time). One starts with a pre-conceived notion about reference strings.

(A2) If we are at $r_x$ ($1 \leq x \leq n$), monitor reference density or reference behavior over some $y$ instances ($1 \leq y \leq n$) of the string, starting from $r_x$ and moving forward, so that prediction of future need is precise. (An
optimal algorithm here, makes x=1 and y=n; i.e., preprocess R in its entirety).

(A3) If at \( r_x (1 \leq x \leq n) \), monitor reference behavior over some previous interval \( t \), using references in that interval as indicative of future need.

The various approaches to the replacement policy give rise to classifications of algorithms--notably, classifications due to Belady [Bl,Bl] and Denning [D1].

Belady's classification runs as follows:

Class 1: all pages are assumed to have an equal probability of being referenced at any time (thus, no memory use information is kept).

Class 2: pages are classified by the history of most recent use in main storage.

Class 3: pages are classified by their absence and presence in main storage--detailed information is kept about all pages in the entire program.

Class 3, of course, implies a great deal of information is being kept as the reference string is processed. At the extreme, preprocessing of the entire reference string is required in order to afford enough information about page use to allow optimal results in terms of minimizing page faults. Hence, Class 3 algorithms qualify for what Denning
terms the category of unrealizable algorithms. An unrealizable replacement algorithm is one that requires, for a decision at $t$, precise information about reference $s$ at $t + i$ ($i > 0$). That is, look-ahead information is requisite and thus such algorithms are not realizable in environments where extensive preprocessing of programs is not possible (which is ordinarily the case). The truly optimal algorithms are those that take P.O.'s proclamation to "replace that page $i'$ for which $T_t(i')$ is maximum," at its word. To do so, we must have the precise knowledge of future page references that Class 3 hints at.

Certainly, the optimal algorithm for a given program and data set can be obtained by preprocessing the program with the data and recording the resultant reference string ($R$). $R$ can then be used to generate a tailored replacement algorithm for the particular program and data.

This, of course, has the severe drawback that to change either the program or the data is to risk optimality of the algorithm. This technique may sound outrageous, and in some respects is so.

However, if programming techniques could be developed and used that would allow a general analysis of the program control flow for all data sets in its input domain and the program's running environment could be seen as stable (no change of major control altering needed in the near future), the technique of preprocessing could in the
long term save time, and be thus feasible. Yet, the cases where this approach might be possible are extremely limited and will not be dealt with here.

Thus, it is apparent why the final sentence of P.0. refers to the expectation of $T_t(i')$. Considering $E(T_t(i'))$ we get the realizable replacement algorithms, since $E(T_t(i'))$ can be based either upon past page reference activity or upon pre-assumed attributes of the reference string (approaches (A3) and (A1) above respectively). Realizable replacement algorithms are those that base replacement decisions at $t$ only on assumptions (e.g., probabilities) about future references.

Approach (1) leads to the consideration of the replacement algorithms RAND (a random replacement policy) and FIFO (a policy based on the assumption that the least recently paged frame is also the least likely to be next referenced; i.e., the least recently paged frame has maximal $E(T_t(i'))$). These are Class 1 algorithms in that no memory usage information is kept. These algorithms will prove useful for pointing out an important property of reference strings (in the case of RAND) and for dramatically demonstrating the relation of memory size to replacement algorithm (in the case of FIFO). The primary (and perhaps sole) advantage in both cases is their simplicity of implementation.
RAND assumes from the outset, that for any time \( t \) at which a page is referenced \( (r_t) \), all pages of the program (i.e., \( r_1, r_2, \ldots, r_n \)) have an equal probability of being referenced, and hence the choice of frames when a page must be removed from main storage may as well be a random choice. If we let:

\[
s = \text{number of pages in the program,}
\]

and

\[
c = \text{number of frames in main storage, \quad \left( s > c \right) }
\]

the probabilities of hits, misses, and of a page \( x \) being referenced at any given reference become straightforward.

\[
\begin{align*}
\Pr(\text{page } x \text{ is referenced}) &= \frac{1}{s} \\
\Pr(\text{a hit}) &= \frac{c}{s} \\
\Pr(\text{a miss}) &= \frac{(s - c)}{s}.
\end{align*}
\]

(Belady [B1] outlines these probabilities.)

The ratio of hits to misses becomes:

\[
\frac{(c/s)}{((s - c)/s)} = \frac{c}{(s - c)}. \quad [B1,81]
\]

For a simplistic example; where

\[
s = 256,
\]

\[
c = 64 \text{ and } x \text{ is any of the } s \text{ pages},
\]

\[
\begin{align*}
\Pr(x \text{ ref}) &= 1/256 \\
\Pr(\text{hit}) &= 64/256 = 1/4 \\
\Pr(\text{miss}) &= (256 - 64) / 256 = 192/256 = 3/4
\end{align*}
\]

ratio of hits/misses = \( 64/192 = .33 \).
While the simplicity with which these results obtain and with which RAND is implemented may be a happy situation, it is clear that the assumption of randomness for the reference string upon which RAND is founded, is a mistaken one. As Denning points out: "Programs tend to reference certain pages heavily, others lightly, still others rarely." [Dl,179]

The reference string $R$ of a program is not utterly random. It is generated by the effect of data on conditional branches and by rigid control structures. Programs often spend great amounts of time in some few modules and perhaps in a few loops of those modules, while code for errors or exception handling (as in BASIC) is relegated to pages that are rarely if ever accessed. Hence, algorithms that take advantage of these facts can assure a lower probability of fault occurrence, whereas given RAND's assumption $F(Tt(i')) = F(Tt(j'))$ and the Pr(miss); i.e, the fault probability curve, for programs would always appear as shown by the dashed line in Figure 3.2.1 (A).

Figure 3.2.1 (A) is provided by Denning [Dl,179] along with a specification of fault probability, adapted as follows: where $f(A,m)$ is the fault probability of algorithm $A$ on storage of size $m$ pages,
Fig. 3.2.1 (A)

Realizable A
"Random" Programs
Unrealizable A

\[ f(A, m) \]

LRPCOUNTER = (LRPCOUNTER + 1) MODULO m

LRPCOUNTER is the pointer to the frame to be selected for replacement.

Fig. 3.2.1 (B)

Figure 3.2.1
with $\Pr(R)$ as the probability of $R$'s occurrence,

$|R| = \text{length of } R,$

$F(A, m, R) = \text{number of faults generated},$

we have:

$$f(A, m) = \sum_{\text{all } R} \Pr(R) \times \frac{F(A, m, R)}{|R|}.$$  

What is important here is first; that the fault probability for realizable algorithms approaches that of the unrealizable ones (indeed, this approach becomes closer and closer as the number of frames approaches the number of program pages—the importance of storage size is evinced) and second; that reference strings are not randomly generated (this will have repercussions on the simulations of Part V).

If we adopt the assumption about reference strings that the least recently=paged frame (i.e., the page that has been in main storage the longest of all pages currently resident) is also least likely to be needed again very soon, we initiate the FIFO (First-In-First-Out) replacement policy. Such a view of the replacement policy implies an assumption about how reference strings are generated (i.e., how programs execute). The assumption is that programs basically execute in a linear fashion, references tending to "cluster" (as Denning puts it [D2,325]). Given this assumption, the idea of managing available storage as a cyclic queue is quite reasonable. Memory can be seen as a cyclic list of frames into which pages are queued as
required. Thus, when storage fills and replacement is necessary, one simply removes the page in the frame that has been in the list the longest. Implementation is extremely simple since all that is really needed is a counter pointing at the frame which has longest been occupied. Once main storage fills (after m misses) the next miss requires a replacement, which will use frame 0 (see Figure 3.2.1 (B)). The next frame to be used for replacement is frame 1, and so on until we cycle back to 0 and start again. Using LRPCOUNTER as the pointer, and initializing it to 0, it is a simple matter to increment it every time a replacement is required, as follows:

\[
\text{LRPCOUNTER} := (\text{LRPCOUNTER} + 1) \mod \text{m}.
\]

To the detriment of FIFO, it is apparent that the assumption upon which it is based is ill-founded. Program execution, although once it may well have been characterizable as linear in behavior, certainly is no more [Vl and D2]. As mentioned by Denning [D2,385] FIFO is also susceptible to what is termed "overloading." If page demand becomes heavy enough, a replacement cycle of the storage can complete too quickly and pages that may soon be needed can be knocked out of storage; thus, the system begins to self-generate a progressively higher degree of faults. This sort of behavior points out the close relation that can exist between replacement algorithm and
storage size available: most dramatically in the case of an anomaly noticed by Belady [B2].

Intuitively one might expect, as the graph of Figure 3.2.1 (A) suggests, that for replacement algorithms with fault probability graphed vertically against \( m \) (storage size in terms of number of frames) that such algorithms are monotonically decreasing (or at least nonincreasing) on \( m \) (i.e., that given more frames in which to process \( R \), there are certainly no more faults generated, and the expectation being that less will be generated than in the smaller storage). Such intuitions are mistaken in the case of FIFO.

Belady noticed, in a certain running program, a tendency to incur more page faults when FIFO was allowed to process the reference string in a storage of size \( L \) than in one of size \( S \) (where \( L \) and \( S \) are indications of the number of page frames available (for a fixed size frame) and \( L > S \)). Indeed, Belady provides two rules comprised of productions which systematically allow the generation of various \( R \) that exhibit the anomaly. With slight alterations to fit the paging scheme of the simulator in Part V, these rules are as follows: Where \( L \), \( S \) and \( A \) are in terms of number of pages, and:
L = the size of the larger storage,
S = the size of the smaller storage,
A = the number of pages in the program to be processed, if the integers 1, 2, ..., A are used to indicate the pages referenced, we have:

The CLUSTER RULE

for S < L < A < 2S,

concatenate the results of the productions:

(1) \(0, 1, 2, \ldots, L\)
(2) \(0, 1, 2, \ldots, (L-S)\)
(3) \(L, (L+1), (L+2), \ldots, (A-1)\)
(4) Production (2)
(5) \(((L-S) + 1), ((L-S) + 2), \ldots, (2L - A)\)
(6) Production (3)

in the order (1) through (6) to form R.

The CYCLIC RULE is:

for S < L < 2S - 1, and \(A \geq (L + (S/2))\)

\(R = \text{Prefix string (concatenated with) } x(\text{Post string})\) where "\(x(i)\)" means \(x\) occurrences of the string \(i\) (\(x > 0\)), and

Prefix string = \((S-1), S, (S+1), \ldots, (L-1), 0, 1, \ldots, S\)

and the Post string is formed from (a) and (b):

(a) = \((S-1), (S+1), \ldots, (A-1), 0, 1, 2, \ldots, S\)
(b) = \(0, 1, 2, \ldots, (A-1)\)

by starting with the first entry of (a) followed by the first entry of (b), concatenating these entries a with b, removing them from their lists and continuing thus until (a) and (b) are exhausted.
The generation of $R$ may seem contrived yet Belady reports having noticed this phenomenon in at least one running program, and suspects that it, at times, goes unnoticed as a substring of $R$ for jobs running under FIFO. In PART V, simulation results of $R$ generated using the Cluster and Cyclic rules are presented.

The importance of these results is that such degenerative cases can occur and spawns the desire for replacement algorithms that assure us that $f(A,m)$ (the fault probability) is nonincreasing, and preferably decreasing on $m$. The behavior does not appear so anomalous once we carefully consider how FIFO functions. The reference string need only be such that a configuration of frame usage is reached where the need for replacement is causing pages still in use to be removed--the algorithm in a sense cycles on itself and begins to "eat up" pages it has brought in. The need for the restriction regarding how much larger $L$ can be than $S$ is obvious when we notice that if $L$ is allowed to be greater than twice the size of $S$, the cycle through storage will not complete fast enough and the anomaly will not result. Hence, storage size is an important parameter in conjunction with the replacement algorithm.

These considerations have prompted the desire for algorithms which assure that the fault probability function for such replacement algorithms is nonincreasing on $m$. That
is, the need is for a criteria to match algorithms against, which if an algorithm meets we are assured that $f(A,m+i) \leq f(A,m)$ ($i \geq 0$). This will not tell us what size storage is best used with algorithm A, but will at least assure us that for a choice of $m$, when opportunity arises to use an $m+1$ or larger storage there need be no concern for increasing the number of faults incurred when processing R. Such a criteria is at hand in what Gecsei, Slutz, Traiger and Mattson [Gl] term the class of "stack" algorithms.

If we examine algorithms (for instance LRU (Least Frequently Used)) or LFU (Least Frequently Used) that monitor reference density in one fashion or another, it will be noticed that the replacement choice is made on the basis of priorities assigned to the different pages resident in main storage (they form a prioritized list). How the priorities are assigned and maintained, of course, varies with the algorithm used, but a brief look at, for instance LRU, where the $m$ most recently used page are maintained in main storage (prioritized within this list from most to least recently used), tells us that the page contents of the frames, using LRU to process R in an $m$ frame storage, is a subset of the identical situation save for storage being $m+1$ frames (i.e., $S(LRU,m,R) \subseteq S(LRU,m+1,R)$ --to recall an earlier method of specifying a storage configuration for an algorithm having processed R in a
space of \( m \) frames). To notice this is to notice what is
termed the "inclusion property" [Gl,84]. The inclusion
property holds for an algorithm \( A' \) if and only if:

\[
S(A',1,R) \subseteq S(A',2,R) \subseteq \ldots \subseteq S(A',m-1,R) \subseteq S(A',m,R)
\]

for all \( R \).

Thus, in cases of this sort everything in one size main
storage is contained in a larger storage for every \( R \) that
\( A' \) processes. As such, algorithms which preserve this
property have storage state traces that can be described as
"stacking-up" on one another—which the name. Algorithms
which, in an \( m \) frame storage, maintain the most recently
used pages are certainly inclusion property preserving
since the \( m \) most recently used pages are surely among the
\( m+1 \) most recently used pages. Thus, if \( x = r_i \) (a page
reference) results in a hit for the size \( m \) storage (i.e.,
\( x \in S(A',m,R_i) \) ) it will also result in a hit in an \( m+1 \)
size storage (\( x \in S(A',m+1,R_i) \) ). Thus, algorithms like LRU
and LFU are stack algorithms—a highly desirable feature.
As such, hits that occur in the size \( m \) storage cannot be
translated to misses in an \( m+1 \) storage. Thus, such algo-

To show that FIFO is not a stack algorithm it will
suffice to point out storage configurations which do not
preserve the inclusion property, for some \( R \). If we employ
the Cluster Rule with \( L=7, \ S=6 \) and \( A=11 \), generate \( R \) and
trace the storage contents as FIFO processes \( R \) in storage \( L \)
and then in storage size $S$, a configuration is arrived at where,

for $L$ frames: $S(\text{FIFO}, 7, R^{17}) = \{8, 9, 10, 0, 1, 2, 3\}$

for $S$ frames: $S(\text{FIFO}, 6, R^{17}) = \{7, 8, 9, 10, 2, 3\}$

and $S(\text{FIFO}, 6, R^{17}) \neq S(\text{FIFO}, 7, R^{17})$: page 7 is the aberration. For a more complete trace of this, see Figure 3.2.2. Thus, FIFO is not a stack algorithm.

These results indicate that in estimating $E(T_t(i'))$ monitoring reference behavior dynamically could prove far more profitable (Approach (A3) above) than attempts at isolating preassumed attributes of reference strings generally (as RAND and FIFO attempted). Choosing the page for replacement from a prioritized list maintained by dynamically monitoring reference behavior provides us with stack algorithms. The first such dynamic monitor that suggests itself is numeric reference density—i.e., sheer number of references to pages resident in main storage.

LFU (Least Frequently Used) basically operates by maintaining counters which indicate the number of times a page has been referenced. When replacement becomes necessary that page, of those in main storage, which has been referenced the least number of time is chosen (clearly, the assumption being that $E(T_t(i'))$ is maximized via a choice based on number of references). Implementation is not particularly difficult since the counters for the pages can be maintained in the page descriptors, which must be
Using the Cluster rule; with \( S = 6 \), \( L = 7 \), and \( A = 11 \), we get:

\[
R = 0 1 2 3 4 5 6 / 0 1 / 7 8 9 10 / 0 1 / 2 3 / 7 8 9 10
\]

\[
i=15 \quad i=17
\]

(slashes separate what is generated by each of the productions (1) - (6))

where "(i)::" is read as: "referencing page i, transition to storage status,"

we can trace \( S(FIFO,m,R) \) from \( R^{15} \) to \( R^{17} \) for both \( m = 6 \) and \( m = 7 \):

\[
S(FIFO,m,R^{15}) \quad \cdots \cdots \cdots \quad S(FIFO,m,R^{17})
\]

\[
L/\ 6,7,8,9,10,0,1\ (2): 7,8,9,10,0,1,2\ (3): 8,9,10,0,1,2,3
\]

\[
S/\ 0,1,7,8,9,10\ (2): 1,7,8,9,10,2\ (3): 7,8,9,10,2,3
\]

Thus; \( S(FIFO,6,R^{17}) \subsetneq S(FIFO,7,R^{17}) \) and FIFO is not inclusion property preserving.

FIFO is not a stack replacement algorithm.

Figure 3.2.2
referenced at each storage access anyway. There are at least two ways that the count can be maintained; (1) only on pages while they are in main storage, or (2) on all pages used during a run (i.e., pages carry their reference counts to secondary storage when they are pushed out, and back in with them if they are referenced again). It is difficult to decide which method is preferable, but in either case LFU seems to have some failings.

While being assured that increasing available storage from \( m \) to \( m+1 \) will not induce added page faults on processing \( R \), we are left to try and decide the storage size that would be best for a given job. LFU does not speak to the question of the memory size required. One must simply specify some number of frames that a job is to be allotted. There is, however, a more serious flaw.

Noting that, particularly with current trends in programming style, often a number of modules are created in a program to perform some specific duty (e.g., initializations) we can be sure that a page(s) of those modules are going to be heavily referenced when a module is invoked to perform its function. Thus, such pages quickly develop high reference densities. When, as with initialization loops, such a module is exited, often the page or pages will no longer be used. However, for a long period of time such pages will remain resident, occupying frames, since a high numeric reference density will give them a
high priority among all pages resident. Thus, the priority in such cases can be miss placed—especially if a new module has been entered to, for example, perform calculations and the new module in a sense cycles on a number of pages. The old pages can be wasting space, in that they may never be referenced again, or are referenced infrequently. If the new module's pages are not too numerous to fit in \( m \) frames, but are too numerous to fit in \((m-1)\) or \((m-2)\)...etc. frames, and if enough of the old module's pages remain to reduce the space available to \( m-i \) frames, a great many unnecessary page faults can occur. Indeed, the old, useless pages may remain in storage for great amounts of time if their reference densities are quite high. This sort of problem, together with the former problem (having to specify an amount of storage to commit to a job) points out how storage space (a precious commodity under multiprogramming) can be wasted. We essentially incur what could be called "hidden" fragmentation.

To abrogate the second problem with numeric reference density measure, the measure can be changed to density in terms of a time interval. Pages are prioritized by a sort of chronology of reference—the least recently used (referenced) over a given time interval being chosen for replacement. Thus, we have the LRU (Least Recently Used) algorithm. The first question that arises is, of course, over what time interval? The most obvious choice for time
interval is that between page faults. Only when \( |S_{t-1}| = m \) and a miss occurs for \( x \) does replacement become necessary, so while maintaining the priority list of pages in use, why not wait until need arises to choose a page to be pushed from main storage? Thus, LFU's second problem is solved, since sheer number of references to the page (numeric reference density) has nothing to do with how long it remains in storage, so long as the space it occupied is needed. That is, when page faults occur, even if a page has the highest numeric reference density, if it has bottom priority on the temporal reference density list (i.e., it among all pages resident has not been referenced "lately"), it will be pushed from the frame it occupies. LRU maintains, essentially, a list of \( m \) pages (the \( m \) pages occupying frames) which list is always ordered such that, starting from the "first" page (\( p_1 \)) and moving "down" the list towards \( p_m \) the following relation is maintained:

for any time \( k \),

\[ P_i \text{ has been referenced more recently than } P_{i+1} \quad (1 \leq i \leq m). \]

Maintenance of this relation on members of the list, of course, implements LRU when \( p_m \) is always chosen for replacement.

There remains, however, the pivotal problem mentioned first in the discussion of LFU. What size \( m \) ought a job \( X \) get? LRU fails to speak to this problem—as did LFU.
Additionally, LRU does not completely escape the problem of "hidden" fragmentation, although it goes a long way toward eliminating it. The remaining portion of the hidden fragmentation problem is related to the problem of specifying \( m \). Once \( m \) is specified for LRU we are not guaranteed that "dead" pages (ones never referenced again, or not referenced again for a very long period of time) will quickly be detected and removed, since this can only occur when a page fault occurs. If, e.g., the pages on which \( X \) is currently executing occupy \((m-2)\) frames and there are 2 dead pages occupying frames, the dead pages will not be removed until \( X \) begins to execute on pages other than the \((m-2)\) pages it is currently using. This may not sound like a problem, and in a single user environment where \( m = \) the number of frames in main storage, it isn't so much of a problem. But as pointed out in section 1.2, interest here lies with the multi-user environment. In the multiprogrammed environment where jobs compete for resources, notably storage space, the situation as described has frames actually not in use (in the example, 2 frames) which, but for the policy restrictions could be used. The main problem here, is, of course, the prefixed size of storage available to the job (since \( X \) does not really need the 2 dead frames anyway). If the number of frames committed to a job \( X \) could vary (i.e.,
if \( m \) could somehow fluctuate) the dead frames could be made available for allocation to other jobs that perhaps need space.
3.3 The Working Set Model

To date, several algorithms have been considered which manifest various problems. One desires a realizable algorithm developed from the perspective of Approach (A3), which is a stack algorithm and allows a running analysis of the storage needs of jobs during execution. LRU seems desirable on all but the latter count. As discussed, LRU had as its time interval for monitoring reference behavior the intervals between page faults (i.e., the interval between the need for replacements). This interval will, of course, vary as R is processed, and the number of frames m (the storage size) allocated to a job is fixed as the job is initiated. A solution to the allocation problem lies in fixing the time interval and allowing m to vary (under certain restrictions).

LRU introduced the notion of a replacement algorithm parameter. This parameter is the interval over which LRU checks page usage. To facilitate storage allocation this parameter can be fixed at other than the interval between page faults. This makes it truly a time parameter and allows storage allocation to vary while time remains fixed. At its heart, this is the tack taken by the Working Set Model (or the working set replacement policy).

The working set replacement policy is actually a type of LRU replacement scheme with its parameter w fixed at some time interval.
Roughly speaking, a working set of pages is the minimum collection of pages that must be loaded in main memory for a process to operate efficiently without "unnecessary" page faults. [D2,324]

A page \( x \) is a member of the working set of some program at a time \( t \) if and only if \( x \) is referenced in the interval \( (t-w \text{ to } t) \) inclusive (see Figure 3.3.1(A) ). A working set at some time \( t \) for an interval \( w \) is denoted:

\[ W(t,w), \]

and includes all and only those pages referenced during the interval \( t-w \) to \( t \) (\( w > 0 \) units). The size of the working set \( W(t,w) \) is the cardinality of that set.

\[ |W(t,w)| = \text{the number of elements in } W(t,w). \]

This will indicate the number of frames required to hold \( W(t,w) \). This measure of working set size will, of course, provide a measure of the space needed for a program to run. For instance, in a timesharing environment when job \( X \)'s quantum period comes due, \( |W(t,w)| \) for \( X \) can be an indication of the number of frames required by \( X \)—a dynamic measure, as will be seen. The working set model provides a dynamic monitor of program behavior and will provide a sound basis for storage management (beyond being simply a replacement policy). But first, the model and its characteristics must be presented.

Being a species of LRU policy, the working set model is a stack algorithm. Denning lists four important properties of \( W(t,w) \):
The working set parameter $w$, a reference interval, and 2 working sets, both of interval $w$: one at $t$, the other at $t-w$.

**Fig. 3.3.1 (A)**

Page residency time and $w$.

**Fig. 3.3.1 (B)**

**Fig. 3.3.1 (C)**

**Figure 3.3.1**
Size and Predictability will be discussed here; however, with regard to Denning's discussions of Reentry Rate and w-Sensitivity, suspicion arises [D2,327].

In trying to estimate three reentry rate functions (one each for process-time reentry, real-time reentry and total return traffic), Denning is trying to model the general case for reference strings. Reference strings have interreference intervals which amount to the time between a reference of a page \( p \) at some time \( t_{n-1} \) and the next occurrence of a reference of \( p \) at \( t_n \). Where \( x_n \) is the \( n \)-th interreference interval, \( x_n = t_n - t_{n-1} \). In developing the analysis, Denning announces the assumption that "the interreference intervals \( \{x_n\}_{n \geq 1} \) are independent, identically distributed random variables." [D2,327 (my underscoring)]. The analysis from that point turns on this assumption. The contention here is that this assumption is ill-advised, and does not hold for typical programs.

Interreference intervals are dependent upon reference distribution. As such, to get randomly distributed interreference intervals requires randomly distributed references. As pointed out by Denning at other times [D1,179] and above in section 3.2.2, reference strings are
not randomly generated. Hence, simulations on random reference strings and assumptions of randomness for interreference intervals occupy the same doomed vessel. If we reject the former, as we should, we must reject the latter as well.

The point of the Reentry Rate analysis was to allow an estimation of channel capacity required on a system, under a working set storage allocation scheme. Thus, rejection of the derivations there presented does not reflect on the core features of the working set model. The w-Sensitivity discussion must be viewed with suspicion as well since it is based on certain results of the Reentry Rate discussion.

Regarding size; obviously, \( |W(t,0)| = 0 \) at the one extreme, and \( |W(t,d)| = (\text{the number of distinct pages referenced}) \) when \( d = \text{the interval from } t \text{ back to the first page reference} \). Thus, \( |W(t,w)| \) is surely a nondecreasing function of \( w \), since as \( w \) increases (the interval becomes larger) more pages are very likely to enter \( W(t,w) \)--and surely no less. Indeed, the function will often be increasing, but with a concave-down graph. Denning asserts that on the average:

\[
(1) \quad |W(t,2w)| \leq 2|W(t,w)|.
\]

To justify this claim, it is pointed out that:

\[
(2) \quad W(t,2w) = W(t,w) \cup W(t-w,w),
\]

which entails:
Only if $W(t, w) \cap W(t-w, w) = \emptyset$ will $|W(t, 2w)| = |W(t, w)| + |W(t-w, w)|$. So, where $W(t, w) \cap W(t-w, w) \neq \emptyset$ the inequality in (3) holds. If we assume statistical regularity, $W(t-w, w)$ behaves on average like $W(t, w)$ behaves (D2,326). Thus, on the average, replacing in (3):

$$|W(t, 2w)| \leq |W(t, w)| + |W(t, w)|.$$ 

Hence, we get (1). That is, while increases in $w$ normally increase $|W(t, w)|$, $2|W(t, w)|$ is an upper limit on the size of the set for a doubled interval $w$—thus, the concave-down nature of the curve.

Of more interest is the relevance of certain of the above results for the Predictability of working sets. Since "references to the same page tend to cluster in time" [D2,326] some important results obtain about the probability that:

$$W(t, w) \cap W(t-w, w) = \emptyset.$$ 

(See Figure 3.3.1(A) for graphic representation of these two working sets on a time line.) Given the cluster effect of references $Pr((4))$ must be very small if $w$ is very small. This, indeed, is the working assumption behind LRU replacement schemes generally. Since page references cluster in time (i.e., random referencing is extremely rare), over two small adjacent intervals ($t-2w,t-w$ and $(t-w,t)$) the pages referenced in each will often be quite
the same. That is, $W(t,w)$ is often little different from $W(t-w,w)$—so long as the intervals are small.

To approach this another way, à la Denning: Let: $z$ be a small time interval, and $z < w$.

Thus:

$$W(t+z,w) = W(t+z,z) + W(t,w-z)$$

and the probability is very small that:

$$W(t+z,z) \cap W(t,w) = \emptyset.$$  (See Figure 3.3.1(C)).

Hence, when we realize that:

$$W(t,w-z) \subseteq W(t,w) \cap W(t+z,w),$$

it can be seen that certain pages of $W(t,w)$ (perhaps a great many or all of them) will be in use after time $t$.

[D2,326]

Thus, for small intervals $z$, $W(t,w)$ gives us a good indication of $W(t+z,w)$. The working set approach seems to allow a good prediction of need for the near future, by monitoring an interval of past use. This predictability will provide not only an indication of which pages are needed in the near future, but, more importantly, as it will turn out, an easy method of dynamically specifying a job's storage needs (in that the size of $W(t,w)$ can be monitored). The basic policy approach of the working set model is to keep only the pages of a program's working set in main storage—since the working set is the method of defining needed pages. That is, we want the **needed** pages
occupying frames (and only the needed pages, to avoid storage waste) and \( W(t,w) \) defines this need.

To clarify how the model works, the mode of implementation must be sketched, but before this can be done, how one is to make the critical choice of a value for \( w \) must be discussed.

In the above discussion, emphasis was put on "small" intervals since only over small interals is the first a good indication of page use in the second. What "small" amounts to was left largely to the imagination. However, choice of \( w \) (i.e., how "small" the intervals are) is, of course, critical to efficient operation. If \( w \) is too large, \( W(t,w) \) will not be a good predictor of future page need in addition to wasting space. If \( w \) is too small, pages that the program actually needs will not be in \( W(t,w) \) (i.e., will be defined as unneeded) -- resulting in excessive page faults.

It is Denning's contention that choice of \( w \) must be a function of \( T \), the transport time between two levels of the storage hierarchy mentioned earlier [D2,328]. If we define the residency of a page as "the fraction of time it is potentially available in . . . memory" [D2,328], some interesting properties of interreference intervals can be noted. The "fraction" is \( w/V \), where \( V = \) some estimate of the time the page occupies a frame plus transfer time where necessary. This measure seems somewhat appropriate in that
on the extremes, when \( w = 0 \), the residency for \( d \) is 0% (as is expected). Also, if \( w \) = the interval from the reference to the beginning of the reference string, residency for \( d \) becomes 100% (as expected). Again, an interreference interval \( x \) is a value representing a time interval \( t - t_{-1} \), where \( t_{-1} \) is a time at which page \( d \) is referenced and \( t \) is the next time at which \( d \) is referenced. Thus, letting \( x \) be the interreference interval for any page \( d \), \( T \) be the storage transfer time, and assuming that pages are only removed from main storage when they are no longer members of \( W(t, w) \), three situations can be noted. Concerning \( x \), either:

Case 1: \( x \leq w \), or
Case 2: \( w < x \leq (w + T) \), or
Case 3: \( x > (w + T) \).

In Case 1 the page has 100% residency, since it is in \( W(t, w) \) at the beginning of the interval \( w \) and \( x \) (its interreference interval) terminates (\( d \) is referenced again) before or as \( w \) expires (i.e., residency = \( w/z = 100\% : z \leq w \)).

In Case 2, however, \( d \) is resident for \( w \) time units and then (having not been referenced and thus no longer being part of \( W(t, w) \)) is pushed to secondary storage. However, with \( x \leq (w + T) \), \( x - w = T \) and during the transporting of \( d \) to secondary storage (or just as the transport terminates) \( d \) is again referenced. Thus, \( d \) must be
immediately transported back to main storage. Thus, for a value of \( V \) we have \( w \) plus two transport times \( T \).

\[
\text{residency} = \frac{w}{w + 2T}.
\]

In Case 3 \( d \) occupies storage for the entire interval \( w \) and is then pushed to secondary storage (not having been referenced). \( d \) occupies a slot in secondary storage for some interval being then referenced again, requiring a transfer time of \( T \). Thus, \( d \) reappears in a frame \((w + T)\) after its previous reference; hence, \( V = w + T \) and

\[
\text{residency} = \frac{w}{w + T}
\]

Denning supplies a graph of residency as seen in Figure 3.3.1(B) [D2,328].

To maximize efficiency it is desirable to maintain the residency percentages at as high a level as practical—while realizing that residency for all \( d \) cannot with practicality be maintained at 100%. To use the equations for Cases 1 through 3 it is only required that the proportions of \( w \) to \( T \) to \( x \) be inspected. Example 1: Let \( w = T = 3 \)

Case 2 - if \( x = 5 \) (i.e, \( x \) is 2/3 of the way from \( w \) to \((w+T)) \)

\[
\text{residency} = \frac{w}{w+2T} = \frac{3}{9} = 33\%
\]

Case 3 - if \( x = 8 \) \((x = (w+T) + 1/3(w+T))\)

\[
\text{residency} = \frac{w}{x+T} = \frac{3}{11} = 27\%
\]

Compare these results where \( w = T \), with what results when \( w = 2T \).
Example 2: Let $T = 3$, $w = 2T = 6$

**Case 2** - if $x = 8$ (i.e., $x$ is $2/3$ of the way between $w$ & $(w+T)$)
residency = $6/(6+6) = 1/2 = 50\%$

**Case 3** - if $x = 12$ ($x = (w+T) + 1/3(w+T)$)
residency = $6/(12+3) = 6/15 = 40\%$.

Notice, that at $w = 2T$ residency in Case 2 is at $50\%$, whereas with smaller $w$ (i.e., $w = T$) the residency in Case 2 is much smaller. From the graph in Figure 3.3.1(B) the residence in terms of the 3 cases can be seen, and it is apparent that quite a drop occurs as $x$ becomes greater than $w$ (i.e., a drop to $w/(w+2T)$). It can be seen that the drop to $50\%$ in example 2 is no mere random occurrence. Where $w = 2T$ we have:

$$\frac{w}{w+2T} = \frac{2T}{4T} = 50\%,$$

while for $w = T$ (as in example 1) we have:

$$\frac{w}{w+2T} = \frac{T}{3T} = 33\%.$$

Thus, as Denning recommends, to avoid a drop to less than $50\%$ residency from $100\%$ when $x > w$, we must have $w \approx 2T$.

The working set parameter $w$ can thus be seen as a simple function of the transport time $T$ on a given system:

$$w \approx 2T.$$
3.4 The Working Set: Implementation & Storage Allocation

The working set model of program behavior, implemented as a replacement policy has certain definite advantages; indeed, it turns out to provide more than simply a replacement policy. The working set meets all of the needs mentioned at the outset of section 3.3. Being a type of LRU replacement scheme, it preserves the inclusion property and is thus a stack algorithm, assuring us that an increase in the amount of storage in which it operates will not allow any increase in page faults. This feature becomes ever more important when we consider the multi-programmed environment (particularly timesharing) in which the amount of storage (number of frames) a job is allotted may vary during a run (or on different runs). Hence, under the working set model there need be no concern for increases in storage allotment that may arise.

The bottom line is not a search for an algorithm that has minimal faults (in terms of $E(T_t(i'))$) on all $R$. Certainly, for any realizable algorithm $A$ implementing a replacement policy, if we know how it operates, we could devise a method for generating at least one $R$ such that $A$ does a poor job of processing $R$. What is sought, and in the case of the working set model found, is a reasonable,
well thought-out A. Such A, is one that manifests such desirable properties as being a stack algorithm, does not incur excessive overhead, is realizable, and allows for efficient storage management. The working set model supplies this, and to its implementation and storage management features, attention now turns.

The size of storage allotted to a job X has been a running concern (storage size is one of the 4 important parameters mentioned in Section 2.2). The working set model helps in answering both when to allocate and how much to allocate. It provides an LRU scheme for replacement and allows an easy monitoring of \( W(t,w) \)--the working set size. Since \( W(t,w) \) defines what pages are needed at a time \( t \), the number of frames allocated should be \( W(t,w) \):

"The basic assumption in memory allocation is that a program will not be run unless there is space in memory for its working set" [D2,329]. This is why the emphasis was put on monitoring \( W(t,w) \) as more important than tracking which pages comprise \( W(t,w) \). When, for example, a job X blocks for a period of time, \( W(t,w) \) frames must be available for X to begin executing again (where \( t \) is the time of the blocking). There is, of course, the temptation here to consider preloading X's working set prior to again initiating execution. However, there is good reason not to do so, but rather to simply be sure that there are \( W(t,w) \) frames available for X to use. The probability of X again
blocking before using all pages in $W(t,w)$ is a consideration; again, as earlier discussed, loading pages that turn out to be superfluous is a terrible waste of time in terms of system overhead. Furthermore, in an interactive environment, when $X$ is reactivated after having blocked, the working set may well change (perhaps radically) due to user response to queries. Since under these circumstances $W(t,w)$ often changes, to preload pages would again waste resources. [D2, 329] Thus, it is Denning's claim that, "Knowledge of only $W(t,w)$! with demand paging suffices to manage memory well." [D2, 329]

There do seem, however, some problems that must be resolved. When $X$ is first brought onto the system and is ready to run, how many frames is it to be allotted (since to know $W(t,w)$! $X$ has to have been run for a sufficient interval of time)? A prespecified allotment is to be suggested allowing $W(t,w)$! to develop with time and $X$'s storage needs to stabilize. This, however, is again to request information from users or compilers. Another problem is posed with $W(t,w)$ changing radically after blocking since the space required after unblocking might be much less than that required prior to blocking and some storage waste results. Hopefully, occurrences of the latter sort are rare.
The working set model provides some useful parameters for storage management of running programs. The implementation of this policy is now addressed.

The preferable mode of implementation is, of course, via hardware, where possible. Timers could be used in conjunction with each frame of main storage to mark the w interval and a flag to mark frames whose pages are no longer in W(t,w) and are thus eligible for removal. Denning describes such a scheme [D2,332-33]). The more interesting question is how to implement the working set scheme on existent systems where auxiliary hardware for memory management is not available; i.e., implementation in software.

Software implementation requires the addition of bits to the page descriptors [D2,328-29] and a way of monitoring intervals of process time (rather than real time due to such occurrences as blocking). The working set interval w is divided into some K sampling intervals of length q to allow monitoring of page use for the determination of W(t,w) : q=w/K. Thus, W(t,qK) is the working set in which we are interested. A method is needed for maintaining W(t,qK) over any K adjacent sampling intervals q. Ideal for this purpose would be a right-shifting (or left-shifting) end-off shift register, or a word that could be used in that way (see Figure 3.4.1). Assuming such a word or register with K bits; identifying the bits as
Use Bits

Use bit shift, and 0 entry at $b_1$.

Figure 3.4.1
\(b_1, b_2, \ldots, b_k\), let each page descriptor have such a word. Let the following procedure be carried-out concerning page references, the use bits and replacements. Where:

- \(a\) is a page identifier,
- \(\oplus(a, b_i, v)\) means that \(b_i\) (the \(i\)th use bit) of page \(a\) is assigned the value of \(v\) (\(v\) is either, one of the constants 0 or 1, or the value of another \(b_j\)).
- \(+(a)\) takes the logical sum of \(b_1, \ldots, b_k\) of \(a\),

we have the policy as follows:

1. When \(a\) is referenced: \(\oplus(a, b_1, 1)\).
2. Before a new sampling interval \(a\) is begun:
   - for all \(a\), for all \(b_i\) of each \(a\),
     - \(\oplus(a, b_i+1, b_i)\) \((1 < i < K)\)
     - \(\oplus(a, b_1, 0)\)
   - (shift the use bits and insert 0 at \(b_1\)).
3. To determine \(W(t, qK)\) for any \(t\) occurring at the end of a sampling interval;
   - for all \(a\),
     - \(+(a)\),
     - \(W(t, qK) = \{a \mid +(a) = 1\}\).
4. All \(a \notin W(t, qK)\) are eligible for removal from the frames they occupy.

The policy in (1) - (4) does not commit one to removing pages which are resident and found to no longer belong to \(W(t, qK)\) until the space is needed. However, given what \(W(t, qK)\) represents (i.e., \(W(t, qK)\) defines the page need for \(X\) at \(t\)), there is good reason to remove such pages immediately. Since \(W(t, qK)\) defines page need, to keep in main storage pages not members of \(W(t, qK)\) is in a
real sense to keep unneeded pages; i.e., it is to waste space. This is particularly the case in multiprogrammed environments where the space may be quite valuable. Also, with the pages removed the frames are ready to accept a page placement when a fault occurs. Thus, storage use can be trimmed to a minimum.

As is apparent, the policy specified in (1) - (4) monitors for any page \( a \), whether \( a \) has been referenced in the last \( K \) sampling intervals (i.e., in the interval \( w \)). When \( a \) is referenced \( b_1 \) of \( a \) is set to "1" (it may indeed already have a value of 1, if it has already been referenced during the current sampling interval). When the current sampling interval is over, all bit values are shifted to the right, \( b_k \)'s value being lost and \( b_1 \) being assigned 0 (see Figure 3.4.1). Since there are \( K \) use bits for each page, and since if a page is referenced even once during one of the \( K \) sampling intervals it will be flagged "used" over an interval \( qK \) (due to \( +c(a) \) at the end of every \( q \) units of time), any page used during \( w \) will be flagged as in \( W(t,w) \) (since \( w = qK \)). Obviously, \( |W(t,w)| \) is easily monitored.
3.5 Working Set Summary

The working set model essentially defines a relation between processor need and storage allocation. To execute a program a certain number of pages are required to be present in main storage at a time $t$. Which pages, and the number required can vary for different $t$, and $W(t,w)$ provides a method for monitoring these requirements. For a program to unblock and continue execution a certain number of frames must be available for use. $W(t,w)$ provides a measure of this number.

Obviously, situations can occur where the working set model fairs poorly. The situations where $W(t,w)$ measures well the page needs are those situations where the program is executing in a definite locality. That is, execution is for a period of time isolated among some few pages (a subset, ordinarily small, of all pages of the program). But this is precisely how most programs reference their pages. A reference string will often have numerous (perhaps hundreds) successive references to the same page. The working set model banks on first the locality property holding the overwhelming majority of the time and second, that the set of pages needed (i.e., $W(t,w)$) changes slowly (the first being the more critical requirement). That programs behave in this fashion "is an experimentally observed phenomenon" [Dl,180].
PART IV

A brief discussion of how coding techniques can affect VMS performance is provided. Particular attention is paid to the structured or modular approach to programming. Some recommendations (mostly due to Myers and J. G. Rogers) for effective coding for VMSs are outlined.
4.1 Coding, VMSs, and the Structured Approach

In section 2.2 it was noted that the damage due to internal fragmentation in a program text could to an extent be minimized by observing certain rules of thumb in coding. These considerations prompt the question of how coding can generally effect VMS performance. Hence, some brief comments on coding are in order.

Of particular interest are the current techniques of "structured" or modular programming. Defining a problem in terms of successively smaller and smaller modules to handle different portions of a problem certainly has the advantage of dividing the problem into subproblems that can be more easily grasped and handled. Indeed, having a number of small modules seems made to order for a VMS. The sheer position of modules in the source code can affect storage requirements and page fault frequency. Consider the diagrams in Figure 4.1.1. The shaded areas indicate four modules that at a time \( t \) are required for program execution (a working set). Due to the layout of the program; i.e., where the modules appear in the code, (A) evinces the "need" for 7 pages (i.e., \( \|W(t,w)\| = 7 \)). However, as seen in (B), if the modules are grouped differently the page need is greatly decreased (\( \|W(t,w)\| = 3 \)). Figure 4.1.1 illustrates an extreme case, yet even if the grouping of modules were not as efficient as demonstrated
Figure 4.1.1

(A) 7 pages "needed"

(B) 3 pages "needed"
in 4.1.1(B), the page requirement might be reducible to 4 or 5—a significant saving if not optimal. The sheer size of the modules is also important. Large modules which perform various functions may well require a number of frames of main storage, but more importantly, they can limit the amount of efficient ordering of modules. If module A performs a number of functions, one of which functions (call it A') makes numerous calls to E, F, and G, the size of A may make it impossible to get E, F and G close to A'. If, as may happen, A will just fit on a page, and A', E, F, and G will also fit on one page; efficiency may be served by the latter grouping rather than the former. The point here is that small modules properly grouped can save main storage space and reduce page fault frequency. What, then, might be some rules-of-thumb for grouping the modules? Myers [M1,127-36] describes what he terms Packaging of modules for VMSs. Rules for grouping modules of a program are of the following priority: (1) Group together modules that iteratively call one another. (2) Group together sets of modules which are called as a group by other modules. (3) Group modules by the frequency with which they call one another. (4) Group together modules which tend to execute sequentially. These groupings are most efficiently done when the system page size is known, and the module sizes are known; i.e., several modules, which are related as noted above, can sometimes be fitted
Hence, the techniques generally attributed to structured programming seem to lend themselves to coding for virtual storage systems. While this is to some extent the case, there is room for dispute. The notion of a module has not been clearly specified. Myers tends to define modules as blocks in programs which perform a function and relates modules via what functions they perform. Thus, modularity is based in similarity of function. Yet, for efficiency in a VMS environment, J. G. Rogers [R1] maintains that "... modularity should be by proximity of usage rather than by similarity of function." [R1,391] Indeed, even the grouping rules outlined above indicate a grouping by usage rather than function. The conflict here is to an extent superficial.

Myers' method of successive refinement on the basis of functional strength is a way of defining what a module is, and does not really speak to the ordering of the modules in the code. When Myers speaks of modules, he is indicating the functional units of a program. Rogers seems to be thinking in terms of what is to be internal to a module. Myers' modules are small units, singular in function. Thus, these small units, although developed on a functional basis, can be grouped along the lines of usage. Rogers seems to be thinking in terms of a larger unit, one with multiple functions (a high-level module, if you will).
In terms of Myers' modules, we might say that one gets a higher-level module by combining several of the functional modules to perform some function that is more general in character. Thus, given Myers' approach, one is not excluded from creating modules (at a higher-level) by grouping on the basis of proximity of usage. Indeed, as Myers has pointed out, the method of functional decomposition lends itself to such a procedure, since the modules which result from application of his methods are small and singular in function.

In addition, the general character of the block-structured languages (e.g., Pascal or SIMULA 67) lends itself well to the VMS environment. In such languages context is well defined due to the encouragement of blocking. Moreover, multiple exists from a context (block) are discouraged or prohibited when the control structures built into these languages are used (rather than unconditional jumps, etc.). Thus, the control structures in a program are well defined and predictable in nature. When a program executes in a VMS, efficiency is served when a certain locality is maintained (execution on a certain number of pages) and when that locality changes slowly. The structured programming approach and the block structured languages by their nature encourage such locality in execution (which may not be the case when the flow of control is left completely to the programmer's discretion).
Other features of the structured approach to programming seem to lend themselves to programming for VMSs. For instance, non-selfmodifying code is not only more easily understood and modified, but helps to improve the efficiency of VMSs since blocks of such code needn't be transported out of main storage when the frame containing such code is chosen for replacement. Since nothing in such a block has changed the frame can simply be overlayed, saving a transport time T.

A lack of side effects in function can also be a plus in a VMS. When side effects are prohibited it is far easier to assess what other modules a function uses, and locality is again served (the function is not as likely to be indiscriminantly referencing various pages to access data).

As mentioned earlier, the trend toward so-called GOTOless programming, aids in keeping flow of control from becoming excessively complex—again locality is served.
4.2 Some Recommendations for Coding

No straightforwardly applicable rules govern coding for VMSs. The nature of the VMS in use and of the language translators used among other things, greatly affect optimal coding of a program. However, without putting undue strain on designers and programmers, there are some rules-of-thumb that, if followed, will generally improve a program's run time and storage utilization under a VMS.

Rogers [R1] mentions several considerations which, he claims, are compatible with structured programming techniques. First, error handling or exception handling code should appear in its own modules; separate from the modules which may invoke it. This, of course, keeps such little-used code from occupying storage space unnecessarily. It might be added here that it is a good policy to keep little-used codes at the bottom of the program or at the bottom of segments that might be defined. Since internal fragmentation in a paging VMS occurs in the last page of a program and/or segment, such a policy helps to minimize the occurrence of internal fragmentation, since the pages upon which it can occur are those least likely to appear in main storage.

Second, an effort should be made to reference data in the order in which it is stored and vice versa (particularly for large portions of data). Doing so,
avoids having to bring large masses of data into main storage to access only small portions thereof. Obviously, for a large array $A$ indexed from 1 to 10,000 which is stored in consecutive words, to jump around accessing locations: $A[3]$, $A[9000]$, $A[500]$ ... etc, will require more pages to be referenced (assuming typical page sizes) than if locations are referenced in a more orderly fashion. However, this example points up how annoying system and/or language dependencies can be in these matters. Consider a two-dimensional array. Such a data structure can be stored in at least two distinct ways; by column or by row. Assuming the former, when we store data, in order to observe the just-mentioned policy, we would store by column incrementing the row index each time until a column is full. This would serve, so far as possible, to group successive pieces of data that we need on the same page, whereas storing by row might well place successive pieces of data on separate pages. The problem here, as with other techniques of coding for VMSs, is that it requires the programmer to be aware of the particular idiosyncrasies regarding how variables are stored, for the system and/or language which he/she is using. Whether the programmer should have to be aware of such details is, of course, moot; yet, as outlined in Part I, since we are often defining virtual machines with which users will work, it
seems undesirable to require of them such detailed knowledge of the system.

Third, and related to the second point, data that is to be used together should be stored together. Again, we are obviously trying to get page references to cluster to as great an extent as possible.

Rogers also recommends the avoidance of elaborate search strategies for large data sets. Presumably, such techniques jump too violently among various pages on which the data is stored, the time they save in searching being lost due to excessive page faults (which may cause a great number of transports to be incurred) [R1,401].

In addition, particularly for large data structures, run time initializations are best avoided; i.e., if the language used has the facilities for compile time initialization of data areas.

Clearly, coding style can have an effect on VMS performance. Since language definition influences how one can code using the language, the language used is another consideration. However, structured programming techniques as mentioned by Myers (M1), seem largely applicable to the VMS environment. Functional decomposition of a problem lends itself well to developing efficient and readable software for VMSs.
PART V

The simulator SIMU and some accompanying utility programs (FIFOANO, DUMPREF, RANDREF) are discussed. Results of runs are presented, with particular interest in properties of replacement policies and reference strings alluded to previously.
5.1 SIMU

The replacement algorithm simulator SIMU is implemented entirely in Pascal 6000 Release 3, running under NOS (NETWORK OPERATING SYSTEM) on the California State University, C D C Cyber 174. As such, documentation for the Pascal 6000 implementation can be found in the form of the PASGUID available through the California State University Northridge Computer Center. The only external subprograms used are the system pseudo-random number generator RANDOM and SETRANDOM used to seed RANDOM. These routines are described in the PASCLIB documentation also available through the Computer Center. A SIMU listing is provided in Figure 5.1.1. Logic charts outlining control flow appear in Figure 5.1.2.

SIMU interactively requests information regarding the current run, as shown in Figure 5.1.3. Four basic replacement policy choices are available: RAND (Random policy), FIFO, LRU, and LFU. For more details on SIMU, see the listing in Figure 5.1.1.
0010 PROGRAM SIMU(INPUT/, OUTPUT, REF, SR, STOUMP, QUIKSH);
0015  (* DOCUMENTATION *)
0020  (* THE FOLLOWING CODE IS PASCAL 6000 RELEASE 3, FOR C D C EQUIPMENT. *)
0025  SIMU IS A REPLACEMENT POLICY SIMULATOR THAT MAKES AVAILABLE A
0030  CHOICE OF FOUR REPLACEMENT POLICIES: RAND (RANDOM), FIFO, LRU
0034  (LEAST RECENTLY USED) AND LFU. FULLY ASSOCIATIVE MAPPING OF THE
0038  VIRTUAL SPACE INTO THE PHYSICAL SPACE IS ASSUMED. SIMU IS ONLY
0042  INTENDED TO WORK A REPLACEMENT POLICY GIVEN A REFERENCE STRING.
0046  TO ADJUST THE NUMBER OF PAGES AVAILABLE TO A USER, THE NUMBER OF
0050  FRAMES OF MAIN STORAGE AVAILABLE AND THE NUMBER OF USERS. THE
0054  APPROPRIATE CONSTANTS MUST BE ALTERED IN THE DECLARATIONS. THE
0058  VARIOUS POLICIES ARE IMPLEMENTED AS SUBPROGRAM CALLS.
0062  IMPORTANT ARRAYS:
0067  MAPTARL - THE PAGE TABLES FOR ALL USERS. TYPE PAGET.
0071  MAINFR - THE MAIN STORAGE. TYPE MAINFR.
0075  LRU LIST - USED WITH LRU REPLACEMENT POLICY TO MAINTAIN
0079  THE LRU PRIORITY RELATIONS ON STORAGE FRAMES.
0083  FIRST - USED WITH LFU REPLACEMENT POLICY TO MAINTAIN THE
0087  APPROPRIATE PRIORITIZATION ON PAGE USE.
0091  IMPORTANT VARIABLES:
0096  JOBID - HOLDS JOB IDENTIFICATION.
0101  REFMAX, REFCOUNT - THE MAXIMUM NUMBER OF REFERENCES TO BE
0105  PROCESSED, AND THE CURRENT NUMBER OF REFERENCES THAT
0109  HAVE BEEN PROCESSED, RESPECTIVELY.
0113  UN, PN - USER NUMBER AND PAGE NUMBER RESPECTIVELY.
0117  THEFRAME - THE FRAME WHERE AN INCOMING PAGE IS TO BE PUT
0121  (EITHER AS RESULT OF A PLACEMENT OR A REPLACEMENT).
0125  DISPLUN, DISPLEN - THE USER AND PAGE TO BE DISPLACED DURING
0129  A REPLACEMENT.
0133  MISS - A MISS (FAULT) COUNTER.
0137  SELECT - SELECTS THE REPLACEMENT POLICY TO BE RUN.
0141  LRP - THE LIST OF EMPTY, CURRENTLY UNUSED FRAMES.
0145  LRP - AN INTEGER POINTER, USED BY FIFO, TO TRACE THE
0149  NEXT FRAME TO BE USED FOR REPLACEMENT.
0153  PROCEDURES & FUNCTIONS: (IN ORDER OF DECLARATION)
0157  RANDOM, SETRANDOM - SYSTEM FUNCTION FOR RANDOM NUMBER
0161  GENERATION.
0165  CHOOSE - PROMPTS SELECTION OF A REPLACEMENT POLICY.
0169  DORENRT - PROMPTS DECISION AS TO REWRITING THE FILES

Figure 5.1.1
SP AND STDUMP.

DUMPSTOR - CAUSES MAINSTOR TO BE DUMPED TO THE FILE STDUMP.

INSERT - INSERTS TO THE EMPTY LIST.

DELETE - WITHDRAWS AND DELETES FROM THE EMPTY LIST.

RAND - THE RANDOM REPLACEMENT POLICY.

FIFO - THE FIRST-IN-FIRST-OUT REPLACEMENT POLICY.

ADJUST - USED BY LRU TO MAINTAIN THE PRIORITIZED LIST.

LRU - THE LEAST RECENTLY USED REPLACEMENT POLICY.

LFU - THE LEAST FREQUENTLY USED REPLACEMENT POLICY (NUMERIC REFERENCE DENSITY BASED).

FILES USED: (ALL OF TYPE TEXT)

<table>
<thead>
<tr>
<th>FILE NAME</th>
<th>INPUT/OUTPUT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP</td>
<td>INPUT</td>
<td>HOLDS REFERENCE STRING.</td>
</tr>
<tr>
<td>SR</td>
<td>OUTPUT</td>
<td>DETAILED INFORMATION ON RESULTS;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WHEN FAULTS (MISSES) OCCURRED, WHAT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WAS REPLACED, &quot;THE FRAME WHERE REPLACEMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OCCURRED, AND THE MISS RATIO AT THAT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIME.</td>
</tr>
<tr>
<td>STDUMP</td>
<td>OUTPUT</td>
<td>HOLDS ANY STORAGE DUMPS THAT ARE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REQUESTED. IT WILL ALWAYS CONTAIN A</td>
</tr>
</tbody>
</table>
|           |              | STORAGE MAP OF MAIN STORAGE WHEN PRO-
|           |              | CESSING TERMINATED. |
| QUIKSR    | OUTPUT       | GIVES QUICK ACCOUNT OF MISS RATIO AND |
|           |              | STORAGE SIZE USED. |

(* DECLARATIONS *)

0540 CONST (* MAXIMUM NUMBER OF PAGES A USER HAS = MAXPAGE + 1. *)

0545 MAXPAGE=10;

0550 (* THE NUMBER OF FRAMES IN MAIN STORAGE IS: *)

0552 NUMFRAMS=6;

0554 (* THE NUMBER OF USERS IS: *)

0555 NUMUSERS=1;

0550 TYPE MAINFRAM = RECORD USER,PAGE,USED : INTEGER; END:

0550 LINK = "NODE;"

0559 NODE = RECORD FRAME : INTEGER;

0600 NEXT : LINK;

0510 END;

0620 (* NOTE: ALFA IS A PACKED ARRAY OF 10 CHAR. *)

0630 PAGE = RECORD PRESEN : INTEGER; (* PRESENCE BIT *)

0640 LOC : INTEGER; (* FRAME NUMBER *)

0645 LFUCOUNT : INTEGER; (* REF.DENSITY COUNTER *)

0546 (* FOR LFU. *)

END;

0650 VAR MAINFR = ARRAY[1..NUMUSERS,0..MAXPAGE] OF PAGE;

0700 MAINSTOR = ARRAY[0..NUMFRAMS] OF MAINFR;

Figure 1.1.1 con't
0715  LRULIST : ARRAY[1..NUMFRAMS] OF INTEGER;
0717  FRST : ARRAY[0..MAXPAGE] OF INTEGER;
0720  JID : ALFA;
0730  JID : CHAR;
0740  T1,T2,T3 : INTEGER; (* INDEXES *)
0745  SELECT,MISS,DISPLUN,DISPLPN,THEFRAME : INTEGER;
0750  EMPTY,ENT : LINK;
0755  DUMP : INTEGER;
0760  LRPCOUNT,SEED1,SEED2 : INTEGER;
0765  REFMAX,UN,PN,REFCOUNT : INTEGER;
0770  SELECT,MISS,DISPLUN,DISPLPN,THEFRAME,EMPTY,ENT : LINK;
0775  (* INDEXES *)
0780  LRUCOUNT,SEED1,SEED2 : INTEGER;
0785  TEXT;
0790  WRITE(STDUMP, 'FRAME USER PAGE IN-USE'); Writeln(STDUMP);
0795  FOR I := 1 TO 16 DO Writeln(STDUMP,' '); Writeln(STDUMP):
0800  WRITE(STDUMP); Writeln(STDUMP);
0805  FOR I := 0 TO (NUMFRAMS - 1) DO WITH MAINSTORE[I] DO
0810  WRITE(STDUMP, I:4, USER:5, PAGE:7, USED:4);
0815  END; (* DUMPSTORE *)
0820  PROCEDURE DUMPSTORE; VAR I : INTEGER; BEGIN
0825  Writeln(STDUMP, 'MAINSTORE REFERENCE: ', REFCOUNT:6,' ', JOBSID);
0830  Writeln(STDUMP, 'FRAME USER PAGE IN-USE'); Writeln(STDUMP);
0835  FOR I := 0 TO (NUMFRAMS - 1) DO WITH MAINSTORE[I] DO
0840  WRITE(STDUMP, I:4, USER:5, PAGE:7, USED:4);
0845  Writeln(STDUMP); Writeln(STDUMP);
0850  END; (* DUMPSTORE *)
0855  FUNCTION INSEXT(VAL : INTEGER); VAR P : LINK; BEGIN
0860  NEW(P);
0865  IF EMPTY = NIL THEN BEGIN
0870  EMPTY := P; EMPTY.FRAME := VAL;
0875  EMPTY.NEXT := EMPTY; END
0880  ELSE BEGIN
0885  P.NEXT := EMPTY; EMPTY.NEXT := P;
0890  EMPTY := P; P.FRAME := VAL; END;
0895  END; (* INSEXT *)
0900  FUNCTION DELETE : INTEGER;

Figure 5.1.1 con't
(* RETURNS THE NUMBER OF THE EMPTY FRAME. *)

VAR FRONT : LINK;
BEGIN
IF EMPTY = NIL THEN
  WRITELN('ATTEMPT TO DELETE FROM EMPTY, EMPTY LIST.');
ELSE BEGIN
  FRONT := EMPTY^.NEXT; DELETE := FRONT^.FRAME;
  IF EMPTY^.NEXT = EMPTY THEN EMPTY := NIL
ELSE BEGIN
  EMPTY^.NEXT := FRONT^.NEXT; END;
END;
END:

(* DELETE *)

FUNCTION RAND : INTEGER; BEGIN
  RAND := TRUNC(RANDOM * (NUMFRAMS - 1));
END:

FUNCTION FIFO(VAR LRP : INTEGER) : INTEGER; BEGIN
  FIFO := LRP;
  LRP := (LRP + 1) MOD NUMFRAMS;
END:

PROCEDURE ADJUST(NUM : INTEGER);
BEGIN
  IF LRULIST[1] <> NUM THEN BEGIN
    I := 1;
    WHILE LRULIST[1] <> NUM DO I := I + 1;
    FOR IND := I DOWNTO 2 DO
      LRULIST[IND] := LRULIST[IND-1];
    LRULIST[1] := NUM;
  END;
END:

FUNCTION LRU : INTEGER;
BEGIN
  TEMP := LRULIST[NUMFRAMS];
  ADJUST(TEMP); LRU := TEMP;
END:

FUNCTION LFU : INTEGER;
BEGIN
  TEMP := LRULIST[NUMFRAMS];
  ADJUST(TEMP); LFU := TEMP;
END:

(* LRU (LEAST RECENTLY USED) IS THE REPLACEMENT POLICY WHERE BY
THE LRU PAGE OCCUPYING A FRAME IS REPLACED. *)

(* LFU (LEAST FREQUENTLY USED) IS THE REPLACEMENT POLICY WHERE BY
THE LFU PAGE OCCUPYING A FRAME IS REPLACED, LFU MAINTAINS THE

Figure 5.1.1 con't
PRIORITY LIST BY MONITORING NUMERIC REFERENCE DENSITY. *)
VAR I1,NLOW,REPL,LOWFRM,CLO : INTEGER;
BEGIN
NLOW:=0; CLO:=MAXINT;
FOR I1 := 0 TO MAXPAGE DO BEGIN
WITH MAPTABL[UN,I1] DO BEGIN
IF (PRES=1) AND (LFUCOUNT <= CLO) THEN BEGIN
NLOW:=NLOW + 1; CLO := LFUCOUNT; LOWFRM := I1;
IF LFUCOUNT = CLO THEN FIRST[NLOW-1] := I1;
END;
(* THE IF *)
END;
(* THE WITH *)
END;
(* THE FOR *)
IF NLOW <> 1 THEN BEGIN
REPL := TRUNC(RANDOM * NLOW);
REPL := FIRST[REPL];
END;
(* LFU *)
ELSE
REPL := LOWFRM;
LFU := MAPTABL[UN,REPL].LOC;
END;
(* LFU *)
BEGIN
EMPTY := NIL; LRPCOUNT := 0; RESET(REF);
FOR I1 := 0 TO (NUMFRAMS - 1) DO WITH MIINSTOR[I1] DO BEGIN
USER:=0; PAGE:=0; USED:=0; LRULIST[I1+1]:=I1; INSERT(I1); END;
(* INITIALIZE FRAMES, LRU PRIORITY LIST & EMPTY LIST. *)
FOR I1 := 1 TO NUMUSERS DO FOR I2 := 0 TO MAXPAGE DO BEGIN
WITH MAPTABL[I1,I2] DO BEGIN
PRES := 0; LOC := -1; LFUCOUNT := -1; END;
END;
WRITELN; WRITELN('ENTER RUN NAME:');
FOR I1 := 1 TO 10 DO REIID(JID) := TOI3ID[I1];
WRITELN(SR,'RUN NAME: ',JOBID); 
WRITELN(SR,'WITH RANDOM SEEDS: ',SEED1:6,SEED2:8); 
WRITELN(SR,'STORAGE SIZE IN NUMBER OF FRAMES = ',NUMFRAMS:6); 
WRITELN(SR,'WITH RANDOM SEEDS: ',SEED1:6,SEED2:9); 
WRITELN(SR,'STORAGE SIZE IN NUMBER OF FRAMES = ',NUMFRAMS:6); 
BEGIN
WRITE(SR,'REF#/NRI'ER (SR,'SLOT REPLACES AT FRAME MTSS RATIO'); 
FOR I1 := 1 TO 67 DO WRITE(SR,' '); 
Figure 5.1.1 con't
5140 WRITELN(SR); WRITELN(SR);
5190 WRITELN('INPUT THE NUMBER OF REFERENCES TO BE PROCESSED: ');
5500 READLN; READ(REFMAX); REFCOUNT := 0;
5510 MISS := 0;
5510 WHILE NOT(EOF(REF)) AND (REFCOUNT < REFCOUNT) DO
5520 BEGIN
5530 READ(REF,UN); READ(REF,PN);
5990 IF MAPTABL[UN,PN].PRES <> 1 THEN (* A MISS *)
6000 BEGIN
6010 REFCOUNT := REFCOUNT + 1;
6020 MISS := MISS + 1;
6030 IF EMPTY <> NIL THEN BEGIN
6040 THEFRAME := DELETE; DISPLUN := 0; DISPLPN := 0;
6047 CASE SELECT OF
6048 1 : BEGIN (* NULL *) END;
6049 2 : BEGIN (* NULL *) END;
6050 3 : ADJUST(THEFRAME);
6051 4 : BEGIN (* NULL *) END;
6052 END;
6053 END;
6054 ELSE BEGIN (* INVOKE REPLACEMENT POLICY. *)
6055 CASE SELECT OF
6056 1 (* RAND *) BEGIN (* NULL *) END;
6057 2 (* FIFO *) BEGIN (* FIFO(LRPCOUNT) *) END;
6058 3 (* LFU *) BEGIN (* LFU(LRPCOUNT) *) END;
6059 4 (* LFU *) BEGIN (* LFU(LRPCOUNT) *) END;
6060 END;
6061 DISPLPN := MAINSTOR[THEFRAME].PAGE;
6140 WITH MAPTABL[UN,PN] DO BEGIN (* MARK THAT PAGE *
6150 PRES:=O; LFUCOUNT:=1; LOC:=-1; END; (* NOT PRESENT. *)
6165 END; (* THE IF *)
6175 WITH MAINSTOR[THEFRAME] DO (* THE NEW PAGE IS *
6190 BEGIN USER:=UN; PAGE:=PN; USED:=1; (* ENTERED IN THE FRAME. *)
6195 END;
6200
e200 WRITE(SR,REFCOUNT:6,' ',UN:2,PN:7);
6510 WRITE(SR,'':11);
6520 WRITE(SR,DISPLUN:2,DISPLPN:7,THEFRAME:11);
6530 WRITE(SR,' ',MISS:4,' '/,REFCOUNT:6);
6540 WRITELN(SR);
6545 WITH MAPTABL[UN,PN] DO BEGIN (* MARK THAT PAGE AS
6550 LFUCOUNT:=1; LOC:=THEFRAME; PRES:=1; END; (* PRESENT AT LOC. *)
7990 END (* MISS SERVICE *)
8000 ELSE BEGIN (* A HIT HAS OCCURRED *) (* A HIT *)
8010 WITH MAPTABL[UN,PN] DO
8020 CASE SELECT OF
8022 1 : BEGIN (* NULL *) END;
8024 2 : BEGIN (* NULL *) END;
8030 3 : ADJUST(LOC);
8040 4 : LFUCOUNT := LFUCOUNT + 1;

Figure 5.1.1 con't
8050       END;
8060       END;
8070     IF (REFCOUNT MOD DUMP) = 0 THEN DUMPSTOR;
8080       READLN(REF);
8090     END; (* THE WHILE LOOP *)
8500     WRITELN('END PROCESSING');
9000     WRITELN(SR);
9010     WRITELN(SR,'FINAL MISS RATIO: ',MISS:4,' /',REFCOUNT:6);
9012     WRITE(QUIKSR,'THE FINAL MISS RATIO FOR ',JOIID,' ');
9016     WRITELN(QUIKSR,'IS ',MISS:4,' /',REFCOUNT:6);
9018     WRITELN(QUIKSR,'THE STORAGE SIZE IS: ',NUMFRAMS:4);
9019     WRITELN(STDUMP,'--------FINAL STATUS OF STORAGE FRAMES--------');
9020       DUMPSTOR;
9999     END. (* MAIN *)

READY.

Figure 5.1.1 con't
**SIMU**

Initialize EMPTY list, Counters: reset file REF.

<table>
<thead>
<tr>
<th>FOR I1 := 0 TO number of frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize Main Storage and LRU priority list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FOR I1 := 1 TO number of users</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR I2 := 0 TO number of pages</td>
</tr>
<tr>
<td>Initialize mapping tables</td>
</tr>
</tbody>
</table>

READ job identification.

Collect run information from user.

WHILE NOT (end-of-file) AND references remain to be processed

<table>
<thead>
<tr>
<th>READ a record from file REF</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the Page is not present</td>
</tr>
<tr>
<td>THEN</td>
</tr>
<tr>
<td>A MISS</td>
</tr>
<tr>
<td>ELSE</td>
</tr>
<tr>
<td>A HIT</td>
</tr>
</tbody>
</table>

Issue "END PROCESSING" message.

Write final miss ratio and storage status to the files SR, STDUMP, and QUIKSR.

---

**A HIT**

Determine Replacement Policy:

1: RAND : Null
2: FIFO : Null
3: LRU : Adjust the frame in the priority list.
4: LFU : Increment the LFU counter for that page.

Figure 5.1.2
A MISS

<table>
<thead>
<tr>
<th>Increment miss counter</th>
<th>If empty frames exist</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>THEN</strong></td>
<td><strong>ELSE</strong></td>
</tr>
<tr>
<td>Select a frame</td>
<td>Determine Replacement Policy:</td>
</tr>
<tr>
<td></td>
<td>1 : RAND : Null</td>
</tr>
<tr>
<td></td>
<td>2 : FIFO : Null</td>
</tr>
<tr>
<td>Determine Replacement Policy:</td>
<td>3 : LRU : Null</td>
</tr>
<tr>
<td>1 : RAND : Null</td>
<td>4 : LFU : Null</td>
</tr>
<tr>
<td>2 : FIFO : Null</td>
<td>Record Displaced page and user numbers.</td>
</tr>
<tr>
<td>3 : LRU : Adjust the frame in the priority list.</td>
<td>Mark the removed page as not present.</td>
</tr>
<tr>
<td>4 : LFU : Null</td>
<td></td>
</tr>
</tbody>
</table>

Enter the incoming page in the selected frame.

Record the miss data on file SR.

Mark the new page as present: i.e. set the present bit in the page's map table.

Figure 5.1.2 (Continued)
ENTER RUN NAME:
? input a 9 character identification
DUMP MAINSTORAGE EVERY;
? integer indicating when to cycle storage dumps
SELECT ONE (1) OF THE FOLLOWING REPLACEMENT POLICIES;
RAND = 1
FIFO = 2
LRU = 3
LFU = 4
? enter selection
INPUT 2 RANDOM SEEDS:
? input 2 integer seeds for random number generation
REWRITE FILES Y OR N
/ Y to rewrite files SR and STDUMP, N to preserve them
INPUT THE NUMBER OF REFERENCE TO BE PROCESSED;
? enter an integer indicating the number of references -
SIMU will terminate execution if an end-of-file is
encountered prior to processing the number of
references indicated
when processing is complete, the response is:
END PROCESSING
READY

For example:
ENTER RUN NAME:
? TEST ITNO
DUMP MAINSTORAGE EVERY;
? 100
SELECT ONE (1) OF THE FOLLOWING REPLACEMENT POLICIES
RAND = 1
FIFO = 2
LRU = 3
LFU = 4
? 2
INPUT 2 RANDOM SEEDS:
? 45 8
REWRITE FILES Y OR N
? Y
INPUT THENUMBER OF REFERENCE TO BE PROCESSED;
? 21
END PROCESSING
READY

An interactive session running SIMU, with an example.

Figure 5.1.3
5.2 Some Utility Programs

Three utility programs have been created due to needs that arose.

The file containing reference strings (REF) does not have the references numbered. As a convenience for checking this file against results of having processed it via SIMU, DUMPREF (see Figure 5.2.1) creates the file PRNT from REF by adding a header and numbering the references. Thus, PRNT can be queued for printing and is much easier to read.

RANDREF (see Figure 5.2.2) fills the need to be able to generate random reference strings. Results of generation are written to the file REF. RANDREF employs the system random number routines RANDOM and SETRANDOM.

The FIFO anomaly, with the accompanying Cluster and Cyclic Rules, brought about the need to be able to generate reference strings which exhibited the anomaly. FIFOANO (see Figure 5.2.3) provides the capability of generating such R, via one or the other of the rules. Results of generation are, of course, written to the file REF.
010 PROGRAM DUMPREF(INPUT/,OUTPUT,REF,PRNT);
015 (* DOCUMENTATION *)
020 (* DUMPREF creates a file PRNT from the file REF by adding a header
025 identifying the various fields of records in REF, and numbering
030 the entries. *)
035 PROCEDURES & FUNCTIONS:  none.
040 FILES REQUIRED:
045 REF - provides input (the references to be numbered).
050 PRNT - results are written to this file.*)
055 (* DECLARATIONS *)
060 VAR COUNT,I,USER,PAGE : INTEGER;
065 REF,PRNT : TEXT;
070 (* MAIN PROGRAM *)
075 BEGIN
080 RESET(REF); REWRITE(PRNT);
085 WRITELN(PRNT,'THE REF FILE');
090 WRITELN(PRNT,'REF & USER PAGE');
095 FOR I := 1 TO 15 DO WRITE(PRNT,'_'); WRITELN(PRNT);
100 COUNT := 1;
105 READ(REF,USER,PAGE);
110 READLN(REF);
115 UNTIL EOF(REF)
120 END. (* DUMPREF *)
125 READY.

Figure 5.2.1
PROGRAM RANDREF(INPUT/,OUTPUT,REF);

(* DOCUMENTATION *)

(* RANDREF SERVES TO ALLOW THE GENERATION OF RANDOM REFERENCE STRINGS. IT REQUIRES TWO (2) SEEDS FOR RANDOM NUMBER GENERATION, THE NUMBER OF THE HIGHEST REFERENCEABLE PAGE (I.E. IF THERE ARE 10 PAGES, THE HIGHEST PAGE NUMBER IS 9), THE NUMBER OF USERS FOR WHICH TO GENERATE REFERENCES, AND THE NUMBER OF REFERENCES TO GENERATE. THE SYSTEM RANDOM NUMBER ROUTINES RANDOM AND SETRANDOM ARE USED. *)

PROCEDURES & FUNCTIONS:

RANDOM - SYSTEM RANDOM NUMBER GENERATOR.
SETRANDOM - USED TO SEED RANDOM (BOTH ARE SYSTEM ROUTINES).

FILES REQUIRED:

REF - HOLDS RESULTS OF GENERATION.

(* DECLARATIONS *)

VAR SEED1,SEED2,PAGEMAX,NUMUSERS,GEN,GENCOUNT,USER,PAGE : INTEGER;
REF : TEXT;

FUNCTION RANDOM : REAL; EXTERN;
PROCEDURE SETRANDOM(SEED1,SEED2 : INTEGER); EXTERN;

(* MAIN PROGRAM *)

BEGIN
WRITELN('ENTER: ');
WRITELN('SEED1,SEED2,PAGEMAX,NUMUSERS,REFS TO GENERATE');
READLN; READ(SEED1,SEED2,PAGEMAX,NUMUSERS,GEN);
REWRITE(REF);

FOR GENCOUNT := 1 TO GEN DO BEGIN
  USER := TRUNC(RANDOM * NUMUSERS) + 1;
  PAGE := TRUNC(RANDOM * PAGEMAX);
  WRITELN(REF,USER:4,PAGE:6);
END;
END.

(* RANDREF *)

READY.

Figure 5.2.2
010 PROGRAM FIFOANO(input/, output, ref);
015 (* FIFOANO serves to generate reference strings that exhibit the
020 FIFO anomaly. In terms of pages, "L" is the size of the larger
025 storage, "S" the size of the smaller storage, "A" the number
030 of pages that can be referenced (size of the program).
035 PROVocêURES & FUNCTIONS:
036 CL2 - PROVIDES FOR PRODUCTION #2, OF THE CLUSTER RULE.
037 CL3 - PROVIDES FOR PRODUCTION #3, OF THE CLUSTER RULE.
038 POSTSTR(I) - GENERATES I INSTANCES OF THE POST STRING.
040 FILES REQUIRED:
041 REF - HOLDS RESULTS OF STRING GENERATION.
042 (* DECLARATIONS *)
110 VAR GEN, L, S, A, ONE, I1 : INTEGER;
140 PROCEDURE CL2:
210 VAR I1 : INTEGER; BEGIN
220 FOR I1 := 0 TO (L - S) DO WRITELN(ref, one:4, i1:6);
230 END; (* CL2 *)
240 PROCEDURE CL3:
260 VAR I1 : INTEGER; BEGIN
270 FOR I1 := L TO (A - 1) DO WRITELN(ref, one:4, i1:6);
280 END; (* CL3 *)
300 PROCEDURE POSTSTR(NUM : INTEGER);
330 VAR I1, I2, I3 : INTEGER;
350 BEGIN
370 FOR I3 := 1 TO NUM DO BEGIN
380 I1 := 0; I2 := S - 1;
390 REPEAT
400 WRITELN(ref, one:4, i2:6);
410 I1 := I1 + 1; I2 := I2 + 1;
420 UNTIL (I2 > (A - 1));
430 I2 := 0;
440 REPEAT
450 WRITELN(ref, one:4, i2:6);
460 I1 := I1 + 1; I2 := I2 + 1;
470 UNTIL (I2 > (S - 2));
480 END; (* THE FOR LOOP *)
490 END; (* POSTSTR *)
510 (* MAIN PROGRAM *)
520 Figure 5.2.3
BEGIN
READLN; READ(GEN);
WRITELN('ENTER: PAGES IN L, S (THE SMALLER), # OF PAGES REFERENCEABLE. ');
READLN; READ(L,S,A);
ONE:=1; REWRITE(REF);
IF GEN = 1 THEN BEGIN (* CLUSTER RULE *)
FOR I1 := 0 TO (L - 1) DO WRITELN(REF,ONE:4,I1:6); (* PROD. 1 *)
CL2;
CL3;
CL2;
(* PROD. 4 *)
FOR I1 := ((L-S)+1) TO ((2*L)-A) DO WRITELN(REF,ONE:4,I1:6); (* PROD. 5 *)
CL3;
(* PROD. 6 *)
END
ELSE BEGIN (* CYCLIC RULE *)
WRITELN('HOW MANY REPETITIONS OF THE POST STRING? ');
READLN; READ(GEN);
FOR I1 := (S - 1) TO (L - 1) DO WRITELN(REF,ONE:4,I1:6); (* THE PREFIX STRING *)
FOR I1 := 0 TO (S - 2) DOWRITELN(REF,ONE:4,I1:6); (* IS GENERATED. *)
POSTSTR(GEN); (* GENERATE THE APPROPRIATE *)
(* NUMBER OF ITERATIONS OF THE *)
(* POST STRING. *)
END;
END. (* FIFOANO *)
READY.

Figure 5.2.3 con't
5.3 The Simulations

As originally conceived, the simulator would have provided a means of testing various replacement policies on random reference string data. However, as became clear in Part III, randomly generated reference strings do not simulate reference strings of actual running programs that a system encounters. Thus, to some extent SIMU has been, as it were, disarmed. Nonetheless, the simulator has proved to be of some value beyond a programming exercise.

In section 3.2 it was pointed out that where the reference string is randomly generated the hit-and-miss probabilities are straightforward. If $R$ has this random nature and we are at reference $r_t$, the $Pr(r_{t+1} = x) = 1/s$ for all $x$ (where $s$ = the number of page slots). That is, past reference activity is of no help in predicting reference activity to come. Since the various replacement policies essentially differ with regard to how they use past reference activity to project future reference activity, where past reference activity is of no help (i.e., with randomly generated $R$) not only are the hit-and-miss probabilities straightforward, but these various replacement policies should yield essentially the same results. These expectations are borne out in several runs of SIMU on randomly generated $R$. 
RANDREF was used with seeds of 10 and 20 to generate an R of 1000 references, for a virtual space of 64 pages. SIMU processed the resultant reference string in 8 frames of main storage. Results for the various replacement policies were as follows. With the miss ratio as number of misses to number of references:

<table>
<thead>
<tr>
<th>Policy</th>
<th>Miss Ratio</th>
<th>% Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAND</td>
<td>888 / 1000</td>
<td>89%</td>
</tr>
<tr>
<td>FIFO</td>
<td>877 / 1000</td>
<td>88%</td>
</tr>
<tr>
<td>LRU</td>
<td>882 / 1000</td>
<td>88%</td>
</tr>
<tr>
<td>LFU</td>
<td>884 / 1000</td>
<td>88%</td>
</tr>
</tbody>
</table>

Recalling the probabilities developed in section 3.2, the probability of a miss with 64 pages and 8 frames is:

\[
\frac{(64 - 8)}{64} = \frac{56}{64} = 87.5%.
\]

Thus, the estimated probability of a miss is borne out in the above run. In addition, it obviously made no difference which policy was employed.

Very similar results obtained with 5000 randomly generated references.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Miss Ratio</th>
<th>% Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAND</td>
<td>4400/5000</td>
<td>88%</td>
</tr>
<tr>
<td>FIFO</td>
<td>4376/5000</td>
<td>87.52%</td>
</tr>
<tr>
<td>LRU</td>
<td>4381/5000</td>
<td>87.62%</td>
</tr>
<tr>
<td>LFU</td>
<td>4427/5000</td>
<td>88.54%</td>
</tr>
</tbody>
</table>
Thus, these runs corroborate the expectations outlined earlier.

SIMU was also useful for demonstrating the FIFO anomaly. The interest, of course, is not in sheer number of faults a policy incurs, but in whether a policy preserves the inclusion property (see section 3.2). Indeed, as can be seen from the results presented below, certain of the more preferable policies (e.g., LRU) incur a very high number of page faults when processing R that produce the anomaly under FIFO. However, stack algorithms (like LRU) are inclusion property preserving which, as mentioned earlier, has definite advantages.

Figure 5.3.1 shows results of processing under FIFO an R generated via the Cluster Rule where there are 11 pages referenceable, in a 6-frame storage (5.3.1(A)) and in a 7 frame storage (5.3.1(B)). Provided is: the reference number when the fault occurred, the incoming slot (the user number and page number), followed by the slot to be replaced (double "0" indicates an unoccupied frame), the frame number chosen for replacement, and finally the miss ratio at the time of the fault. The final miss ratios evince the anomalous behavior. The reference string, with the references numbered, appears in Figure 5.3.2. A successive trace, reference by reference, of main storage in
**Figure 5.3.1**

**RUN NAME:** FIFOAN3=6  
**WITH RANDOM SEEDS:** 3 7  
**STORAGE SIZE IN NUMBER OF FRAMES =** 6

<table>
<thead>
<tr>
<th>REF # / SLOT</th>
<th>REPLACES</th>
<th>SLOT AT FRAME</th>
<th>MISS RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

**FINAL MISS RATIO:** 15 / 21  

(A)

**RUN NAME:** FIFOANL=7  
**WITH RANDOM SEEDS:** 3 7  
**STORAGE SIZE IN NUMBER OF FRAMES =** 7

<table>
<thead>
<tr>
<th>REF # / SLOT</th>
<th>REPLACES</th>
<th>SLOT AT FRAME</th>
<th>MISS RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

**FINAL MISS RATIO:** 19 / 21  

(B)  

READY.
<table>
<thead>
<tr>
<th>REF #</th>
<th>USER PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>

Figure 5.3.2
Figure 5.3.3 shows the development of the anomaly by a comparison of main storage contents for size 6 and size 7 main stores.

By contrast with FIFO, the following results obtain for LRU and LFU (stack algorithms) when processing the R of figure 5.3.2.

<table>
<thead>
<tr>
<th>Policy</th>
<th># of Frames</th>
<th>Miss Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>6</td>
<td>19/21</td>
</tr>
<tr>
<td>LRU</td>
<td>7</td>
<td>17/21</td>
</tr>
<tr>
<td>LFU</td>
<td>6</td>
<td>16/21</td>
</tr>
<tr>
<td>LFU</td>
<td>7</td>
<td>13/21</td>
</tr>
</tbody>
</table>

Obviously, as is generally the case with stack algorithms, increasing the number of frames does not increase the number of faults.

The anomaly become even more dramatic when FIFO processes R generated via the Cyclic Rule. For long reference strings generated thus, the difference in number of faults from the larger storage to the smaller approaches 2 to 1. This occurs due to the reference cycle of the Post string employed by the Cyclic Rule (see section 3.2). Again, where 11 pages are referenceable, and where 10 iterations of the Post string are generated, we get 227 references in R. The results of processing R are as follows:
<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>FRAME USER PAGE IN-USE</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>FRAME USER PAGE IN-USE</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>FRAME USER PAGE IN-USE</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 1 2 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>FRAME USER PAGE IN-USE</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 1 2 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 1 3 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>FRAME USER PAGE IN-USE</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 1 2 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 1 3 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 1 4 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

(A) Figure 5.3.3

(B)
<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE: 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
</tr>
<tr>
<td>0  1  0  1</td>
</tr>
<tr>
<td>1  1  1  1</td>
</tr>
<tr>
<td>2  1  2  1</td>
</tr>
<tr>
<td>3  1  3  1</td>
</tr>
<tr>
<td>4  1  4  1</td>
</tr>
<tr>
<td>5  1  5  1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE: 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
</tr>
<tr>
<td>0  1  6  1</td>
</tr>
<tr>
<td>1  1  0  1</td>
</tr>
<tr>
<td>2  1  2  1</td>
</tr>
<tr>
<td>3  1  3  1</td>
</tr>
<tr>
<td>4  1  4  1</td>
</tr>
<tr>
<td>5  1  5  1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE: 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
</tr>
<tr>
<td>0  1  6  1</td>
</tr>
<tr>
<td>1  1  0  1</td>
</tr>
<tr>
<td>2  1  1  1</td>
</tr>
<tr>
<td>3  1  3  1</td>
</tr>
<tr>
<td>4  1  4  1</td>
</tr>
<tr>
<td>5  1  5  1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE: 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
</tr>
<tr>
<td>0  1  6  1</td>
</tr>
<tr>
<td>1  1  0  1</td>
</tr>
<tr>
<td>2  1  1  1</td>
</tr>
<tr>
<td>3  1  3  1</td>
</tr>
<tr>
<td>4  1  4  1</td>
</tr>
<tr>
<td>5  1  5  1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
</tr>
<tr>
<td>0  1  6  1</td>
</tr>
<tr>
<td>1  1  0  1</td>
</tr>
<tr>
<td>2  1  1  1</td>
</tr>
<tr>
<td>3  1  7  1</td>
</tr>
<tr>
<td>4  1  4  1</td>
</tr>
<tr>
<td>5  1  5  1</td>
</tr>
</tbody>
</table>

(A) Figure 5.3.3 cont'
Figure 5.3.3 con't
<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
<td></td>
</tr>
<tr>
<td>0 1 10 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 2 1</td>
<td></td>
</tr>
<tr>
<td>2 1 1 1</td>
<td></td>
</tr>
<tr>
<td>3 1 7 1</td>
<td></td>
</tr>
<tr>
<td>4 1 8 1</td>
<td></td>
</tr>
<tr>
<td>5 1 9 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
<td></td>
</tr>
<tr>
<td>0 1 10 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 2 1</td>
<td></td>
</tr>
<tr>
<td>2 1 3 1</td>
<td></td>
</tr>
<tr>
<td>3 1 7 1</td>
<td></td>
</tr>
<tr>
<td>4 1 8 1</td>
<td></td>
</tr>
<tr>
<td>5 1 9 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
<td></td>
</tr>
<tr>
<td>0 1 10 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 2 1</td>
<td></td>
</tr>
<tr>
<td>2 1 3 1</td>
<td></td>
</tr>
<tr>
<td>3 1 7 1</td>
<td></td>
</tr>
<tr>
<td>4 1 0 1</td>
<td></td>
</tr>
<tr>
<td>5 1 9 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
<td></td>
</tr>
<tr>
<td>0 1 10 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 2 1</td>
<td></td>
</tr>
<tr>
<td>2 1 3 1</td>
<td></td>
</tr>
<tr>
<td>3 1 7 1</td>
<td></td>
</tr>
<tr>
<td>4 1 8 1</td>
<td></td>
</tr>
<tr>
<td>5 1 9 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME USER PAGE IN-USE</td>
<td></td>
</tr>
<tr>
<td>0 1 10 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 2 1</td>
<td></td>
</tr>
<tr>
<td>2 1 3 1</td>
<td></td>
</tr>
<tr>
<td>3 1 7 1</td>
<td></td>
</tr>
<tr>
<td>4 1 8 1</td>
<td></td>
</tr>
<tr>
<td>5 1 9 1</td>
<td></td>
</tr>
</tbody>
</table>

(A) (B)

Figure 5.3.3 con't
**Figure 5.3.3 con't**

---

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>FRAME USER PAGE IN-USE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17</td>
</tr>
<tr>
<td>0 1 10 1</td>
<td></td>
</tr>
<tr>
<td>1 1 2 1</td>
<td></td>
</tr>
<tr>
<td>2 1 3 1</td>
<td></td>
</tr>
<tr>
<td>3 1 7 1</td>
<td></td>
</tr>
<tr>
<td>4 1 8 1</td>
<td></td>
</tr>
<tr>
<td>5 1 9 1</td>
<td></td>
</tr>
</tbody>
</table>

---

**FINAL STATUS OF STORAGE FRAMES---**

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>FRAME USER PAGE IN-USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 FIFOANS=6</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 10 1</td>
<td></td>
</tr>
<tr>
<td>1 1 2 1</td>
<td></td>
</tr>
<tr>
<td>2 1 3 1</td>
<td></td>
</tr>
<tr>
<td>3 1 7 1</td>
<td></td>
</tr>
<tr>
<td>4 1 8 1</td>
<td></td>
</tr>
<tr>
<td>5 1 9 1</td>
<td></td>
</tr>
</tbody>
</table>

\[\text{READY.}\]

---

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>FRAME USER PAGE IN-USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>0 1 3 1</td>
<td></td>
</tr>
<tr>
<td>1 1 7 1</td>
<td></td>
</tr>
<tr>
<td>2 1 9 1</td>
<td></td>
</tr>
<tr>
<td>3 1 10 1</td>
<td></td>
</tr>
<tr>
<td>4 1 0 1</td>
<td></td>
</tr>
<tr>
<td>5 1 1 1</td>
<td></td>
</tr>
<tr>
<td>6 1 2 1</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>MAINSTORE AT REFERENCE:</th>
<th>FRAME USER PAGE IN-USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>0 1 3 1</td>
<td></td>
</tr>
<tr>
<td>1 1 7 1</td>
<td></td>
</tr>
<tr>
<td>2 1 8 1</td>
<td></td>
</tr>
<tr>
<td>3 1 9 1</td>
<td></td>
</tr>
<tr>
<td>4 1 0 1</td>
<td></td>
</tr>
<tr>
<td>5 1 1 1</td>
<td></td>
</tr>
<tr>
<td>6 1 2 1</td>
<td></td>
</tr>
</tbody>
</table>
MAINSTORE AT REFERENCE: 21 FIFOANL=7
FRAME USER PAGE IN-USE

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

--------FINAL STATUS OF STORAGE FRAMES---------
MAINSTORE AT REFERENCE: 21 FIFOANL=7
FRAME USER PAGE IN-USE

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

READY.

(B)

Figure 5.3.3 con't
<table>
<thead>
<tr>
<th>Policy</th>
<th># misses for 6 frames</th>
<th># misses for 7 frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>117</td>
<td>220</td>
</tr>
<tr>
<td>LRU</td>
<td>226</td>
<td>223</td>
</tr>
<tr>
<td>LFU</td>
<td>149</td>
<td>132</td>
</tr>
</tbody>
</table>

Under FIFO, increasing the number of frames by one, greatly increases the number of faults.
REFERENCES


APPENDIX A

A1. Address Translation

Any VMS must have facilities for address translation from virtual to physical space. To maintain efficiency this must be done via hardware or in microprogramming. In what follows, concern is with the address mapping in a paging system; however, the mapping technique applies equally well to segmented systems.

To define the mapping, tables must be kept which indicate where any given block of the program is being kept and what its status is (i.e., a table of descriptors for every block/page of the program). When a page is brought into main storage from secondary it is associated with some frame of main storage and the descriptor in the mapping (page) table for that page is marked to indicate such association. The association of program page \( x \) with main storage frame \( m \) can be as simple as a direct association in that when \( x \) is resident in main storage, it is always at \( m_1 \). This, of course, trivializes the process of finding \( x \) when it is resident, but also creates tremendous contention problems. If the virtual space is large enough in comparison to the physical space, several pages will map to \( m_1 \). Thus, if a number of those pages are required at one time a great deal of swapping must occur. To avoid this, one can,
of course, associate a page with more than one frame; hence, reducing contention. Contention is minimized in a fully associative mapping, where any page can be placed in any frame.

Figure A1.1 shows a typical mapping scheme. The high-order \( p \) bits of the virtual address index into the appropriate descriptor in the page table which holds the address of the top of the frame where the page is resident (i.e., a pointer to the frame). The remaining bits are the displacement within the page--hence, adding \( D \) to the pointer value, allows access to the desired word.

Thus, at each reference a descriptor must be consulted. To access storage twice to get the desired word (three times if segmentation with paging is used) can, of course, slow operation significantly. Hence, associative registers are introduced.

A set of associative registers (usually numbering close to 16) can be searched in parallel on a given key yielding a desired value in a fraction of the time required to search the tables. The descriptors of pages currently "needed" can be kept in the associative registers, thus allowing quick access to the required pointers. In this way the address translation is greatly speeded-up. This becomes of increasing importance when segmentation with paging is used.
Arrows indicate possible mapping in a fully associative mapping.

Figure A1.1
Under segmentation with paging another level of table mapping is introduced and thus three accesses per reference are required (see Figure A1.2).

A2. Secondary Devices

The optimal page size results of section 2.2. point out the conflict in page size desired to (1) maximize efficiency in reduction of transport time (larger page size), and (2) maximize efficiency in use of main storage space (small page size to limit internal fragmentation). The larger page size is primarily due to the lack of speed (in comparison to main storage and processor registers) on the part of secondary storage devices. The faster slots on secondary devices can be accessed and transported, the smaller the page size can be with the intent of approaching the optimal size. With this end in view, various devices for secondary storage (or the outer levels of the hierarchy) have been developed.

Drums and Disks have been mentioned as secondary devices. However, where these devices are employed in paging they are ordinarily specially designed for that application. Denning, discussed the organization of a paging drum [D1,173-76] as well as alternative non-mechanical secondary devices. Alternatives to mechanical secondary devices include the LCS (Large Capacity Storage) which is a form of slow core storage, having a cycle time of around 10
Mapping scheme for Paging with Segmentation.

Figure A1.2
μsec. A special type of core storage to enhance transmission capabilities is the ECS (Extended Core Storage) having an access time of typically 3 μsec and cycle time of approximately 1 μsec, with the capability of transmitting a small block of words in one main memory cycle.

The point of these alternative devices is that without the positioning delays involved in mechanical devices transmission can be speeded-up and thus overall system performance enhanced. A basic problem in storage hierarchies of any number of levels is dealing with differences in speed (often great differences) between two levels of the hierarchy L and L+1 — where L is ordinarily much faster than L+1 (see Figure A2.1).

Main storage is not necessarily the level from which the processor directly accesses information. L1 can be a cache storage—essentially a high speed buffer between the CPU and main storage. Cache is useful where the speed of processor registers is much greater than main storage cycle time and hence to allow the processor to run at speeds of which it is capable requires a high speed buffer between it and main storage.
Six Level Storage Hierarchy

From [T1,264]

Figure A2.1
APPENDIX B

What follows is a listing of OPT, a program to compute optimal page size when cost of fragmentation is the overwhelming consideration (see section 2.2.). Several additional runs of OPT for various segment sizes and fragmentation cost ratios follow the listing.
PROGRAM OPT(INPUT/,OUTPUT,RESULT);

(* DOCUMENTATION *)

OPT GENERATES DATA ON OPTIMAL PAGE SIZE WITH EITHER FRAGMENTATION COST OR AVERAGE SEGMENT SIZE VARYING (BUT NOT BOTH DURING THE SAME RUN). IF VARYC2 IS "TRUE" COST VARIES, IF "FALSE" SEGMENT SIZE VARIES. THE EQUATIONS USED ARE DUE TO DENNING'S ARTICLE ON THE WORKING SET MODEL FOR PROGRAM BEHAVIOR.

PROCEDURES & FUNCTIONS:
PGSPESEG - MANAGES THE AVERAGE SEGMENT SIZE WHEN IT VARIES OVER A RUN - IT VARIES BY POWERS OF TWO.
FILES REQUIRED:
RESULT - HOLDS RESULTS GENERATED.

(* DECLARATIONS *)

VAR 7,NUM,C1,C2,C2TERM,PGPSTERM,PGPSVAL : INTEGER;
VARYC2 : BOOLEAN;
OPS : REAL;
RESULT : FILE OF CHAR;

FUNCTION PGSPESEG : INTEGER;
FUNCTION RIPOW(I2 : INTEGER) : INTEGER;

VAR I,VAL : INTEGER;
BEGIN
VAL := 1;
FOR I := 1 TO I2 DO
VAL := 2 * VAL;
RIP := VAL;
END;

BEGIN
PGSPESEG := RIPOW(NUM);
IF NOT(VARYC2) THEN NUM := NUM + 1;
END; (* OF PGSPESEG *)

(* MAIN PROGRAM *)

BEGIN (* MAIN *)
C1 := 1;
VARYC2 := FALSE;
Z := 1;
WRITELN('INPUT: COST NUM C2TERM PGPSTERM'); READLN;
READ(C2,NUM,C2TERM,PGPSTERM);
WRITELN(RESULT,'TABLE FRAGMENTATION COST: ',C1:6);
WRITELN(RESULT,'INTERNAL FRAGMENTATION COST: ',C2:6);
WRITELN(RESULT,'INITIAL POWER OF 2: ',NUM:6);
434  WRITELN(RESULT, 'C2TERM: ', C2TERM: 6);
436  WRITELN(RESULT, 'PGPSTERM: ', PGPSTERM: 6);
438  WRITELN(RESULT); WRITELN(RESULT);
451  WRITE(RESULT, 'COST = TABLE FRAGMENTATION / ');
455  WRITELN(RESULT, 'INTERNAL FRAGMENTATION');
457  WRITELN(RESULT); WRITELN(RESULT);
460  WRITE(RESULT, 'SEGMENT ');
470  WRITELN(RESULT, 'OPTIMAL ');
480  WRITE(RESULT, 'COST SIZE');
490  WRITELN(RESULT, 'PAGE SIZE');
500  WRITELN(RESULT); WRITELN(RESULT);
510  REPEAT
520  PGPSVAL := PGPSEG;
540  OPS := SQRT((2 * (C1/C2)) * (PGPSVAL * 2));
550  WRITE(RESULT, C1:2, '/' , C2:2, ' ');
560  WRITE(RESULT, PGPSVAL: 20, ' ', OPS : 18 : 6);
570  WRITELN(RESULT); WRITELN(RESULT);
580  IF VARYC2 THEN C2 := C2 + 1;
590  UNTIL (C2 = C2TERM) OR (PGPSVAL > PGPSTERM);
600  END. (* MAIN *)
READY.
TABLE FRAGMENTATION COST: 1
INTERNAL FRAGMENTATION COST: 1
INITIAL POWER OF 2: 9
C2TERM: 20
PGPSTERN: 32768

COST = TABLE FRAGMENTATION / INTERNAL FRAGMENTATION

<table>
<thead>
<tr>
<th>COST</th>
<th>SEGMENT SIZE</th>
<th>OPTIMAL PAGE SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 / 1</td>
<td>512</td>
<td>32.000000</td>
</tr>
<tr>
<td>1 / 2</td>
<td>512</td>
<td>22.627417</td>
</tr>
<tr>
<td>1 / 3</td>
<td>512</td>
<td>18.475209</td>
</tr>
<tr>
<td>1 / 4</td>
<td>512</td>
<td>16.000000</td>
</tr>
<tr>
<td>1 / 5</td>
<td>512</td>
<td>14.310335</td>
</tr>
<tr>
<td>1 / 6</td>
<td>512</td>
<td>13.063945</td>
</tr>
<tr>
<td>1 / 7</td>
<td>512</td>
<td>12.094863</td>
</tr>
<tr>
<td>1 / 8</td>
<td>512</td>
<td>11.13708</td>
</tr>
<tr>
<td>1 / 9</td>
<td>512</td>
<td>10.666667</td>
</tr>
<tr>
<td>1 /10</td>
<td>512</td>
<td>10.199289</td>
</tr>
<tr>
<td>1 /11</td>
<td>512</td>
<td>9.648363</td>
</tr>
<tr>
<td>1 /12</td>
<td>512</td>
<td>9.237604</td>
</tr>
<tr>
<td>1 /13</td>
<td>512</td>
<td>8.875203</td>
</tr>
<tr>
<td>1 /14</td>
<td>512</td>
<td>8.552360</td>
</tr>
<tr>
<td>1 /15</td>
<td>512</td>
<td>8.262364</td>
</tr>
<tr>
<td>1 /16</td>
<td>512</td>
<td>8.000000</td>
</tr>
<tr>
<td>1 /17</td>
<td>512</td>
<td>7.761140</td>
</tr>
<tr>
<td>1 /18</td>
<td>512</td>
<td>7.542472</td>
</tr>
<tr>
<td>1 /19</td>
<td>512</td>
<td>7.341303</td>
</tr>
</tbody>
</table>

READY.
TABLE FRAGMENTATION COST: 1
INTERNAL FRAGMENTATION COST: 1
INITIAL POWER OF 2: 11
C2TERM: 20
PGPSTERM: 32768

COST = TABLE FRAGMENTATION / INTERNAL FRAGMENTATION

<table>
<thead>
<tr>
<th>COST</th>
<th>SEGMENT SIZE</th>
<th>OPTIMAL PAGE SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 / 1</td>
<td>2048</td>
<td>64.000000</td>
</tr>
<tr>
<td>1 / 2</td>
<td>2048</td>
<td>45.254834</td>
</tr>
<tr>
<td>1 / 3</td>
<td>2048</td>
<td>36.950417</td>
</tr>
<tr>
<td>1 / 4</td>
<td>2048</td>
<td>32.000000</td>
</tr>
<tr>
<td>1 / 5</td>
<td>2048</td>
<td>28.621670</td>
</tr>
<tr>
<td>1 / 6</td>
<td>2048</td>
<td>26.127891</td>
</tr>
<tr>
<td>1 / 7</td>
<td>2048</td>
<td>24.189726</td>
</tr>
<tr>
<td>1 / 8</td>
<td>2048</td>
<td>22.627417</td>
</tr>
<tr>
<td>1 / 9</td>
<td>2048</td>
<td>21.333333</td>
</tr>
<tr>
<td>1 /10</td>
<td>2048</td>
<td>20.23577</td>
</tr>
<tr>
<td>1 /11</td>
<td>2048</td>
<td>19.296726</td>
</tr>
<tr>
<td>1 /12</td>
<td>2048</td>
<td>18.475209</td>
</tr>
<tr>
<td>1 /13</td>
<td>2048</td>
<td>17.750406</td>
</tr>
<tr>
<td>1 /14</td>
<td>2048</td>
<td>17.104719</td>
</tr>
<tr>
<td>1 /15</td>
<td>2048</td>
<td>16.524729</td>
</tr>
<tr>
<td>1 /16</td>
<td>2048</td>
<td>16.000000</td>
</tr>
<tr>
<td>1 /17</td>
<td>2048</td>
<td>15.522280</td>
</tr>
<tr>
<td>1 /18</td>
<td>2048</td>
<td>15.034945</td>
</tr>
<tr>
<td>1 /19</td>
<td>2048</td>
<td>14.682607</td>
</tr>
</tbody>
</table>

READY.
TABLE FRAGMENTATION COST: 1
INTERNAL FRAGMENTATION COST: 3
INITIAL POWER OF 2: 2
c2tem: 20
PGPITEM: 32768

COST = TABLE FRAGMENTATION / INTERNAL FRAGMENTATION

<table>
<thead>
<tr>
<th>COST</th>
<th>SEGMENT SIZE</th>
<th>OPTIMAL PAGE SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 / 3</td>
<td>4</td>
<td>1.632393</td>
</tr>
<tr>
<td>1 / 3</td>
<td>8</td>
<td>2.309401</td>
</tr>
<tr>
<td>1 / 3</td>
<td>16</td>
<td>3.265985</td>
</tr>
<tr>
<td>1 / 3</td>
<td>32</td>
<td>4.618302</td>
</tr>
<tr>
<td>1 / 3</td>
<td>64</td>
<td>6.531973</td>
</tr>
<tr>
<td>1 / 3</td>
<td>128</td>
<td>9.237604</td>
</tr>
<tr>
<td>1 / 3</td>
<td>256</td>
<td>13.063945</td>
</tr>
<tr>
<td>1 / 3</td>
<td>512</td>
<td>18.475209</td>
</tr>
<tr>
<td>1 / 3</td>
<td>1024</td>
<td>26.127891</td>
</tr>
<tr>
<td>1 / 3</td>
<td>2048</td>
<td>36.950417</td>
</tr>
<tr>
<td>1 / 3</td>
<td>4096</td>
<td>52.255781</td>
</tr>
<tr>
<td>1 / 3</td>
<td>8192</td>
<td>73.900834</td>
</tr>
<tr>
<td>1 / 3</td>
<td>16384</td>
<td>104.511562</td>
</tr>
<tr>
<td>1 / 3</td>
<td>32768</td>
<td>147.831669</td>
</tr>
<tr>
<td>1 / 3</td>
<td>65536</td>
<td>209.023125</td>
</tr>
</tbody>
</table>

READY.
TABLE FRAGMENTATION COST: 1
INTERNAL FRAGMENTATION COST: 5
INITIAL POWER OF 2: 2
C2TERM: 20
PGPSTERM: 32768

COST = TABLE FRAGMENTATION / INTERNAL FRAGMENTATION

<table>
<thead>
<tr>
<th>COST</th>
<th>SEGMENT SIZE</th>
<th>OPTIMAL PAGE SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 / 5</td>
<td>4</td>
<td>1.264911</td>
</tr>
<tr>
<td>1 / 5</td>
<td>8</td>
<td>1.788854</td>
</tr>
<tr>
<td>1 / 5</td>
<td>16</td>
<td>2.529022</td>
</tr>
<tr>
<td>1 / 5</td>
<td>32</td>
<td>3.577709</td>
</tr>
<tr>
<td>1 / 5</td>
<td>64</td>
<td>5.059644</td>
</tr>
<tr>
<td>1 / 5</td>
<td>128</td>
<td>7.155418</td>
</tr>
<tr>
<td>1 / 5</td>
<td>256</td>
<td>10.119289</td>
</tr>
<tr>
<td>1 / 5</td>
<td>512</td>
<td>14.310835</td>
</tr>
<tr>
<td>1 / 5</td>
<td>1024</td>
<td>20.238577</td>
</tr>
<tr>
<td>1 / 5</td>
<td>2048</td>
<td>28.621670</td>
</tr>
<tr>
<td>1 / 5</td>
<td>4096</td>
<td>40.477154</td>
</tr>
<tr>
<td>1 / 5</td>
<td>6192</td>
<td>57.243340</td>
</tr>
<tr>
<td>1 / 5</td>
<td>16384</td>
<td>80.954308</td>
</tr>
<tr>
<td>1 / 5</td>
<td>32768</td>
<td>114.486680</td>
</tr>
<tr>
<td>1 / 5</td>
<td>65536</td>
<td>161.903616</td>
</tr>
</tbody>
</table>

READY.
APPENDIX C

Two excellent examples of VMSs are MULTICS and the IBM System/370 running VM 370.

The IBM 370 has Dynamic Address Translation hardware, implemented via a paging with segmentation mechanism. The maximum number of segments available is 256. Page sizes available are 2K or 4K bytes with the maximum segment size being 1024K bytes. To cut address translation overhead to below ten percent the hardware utilizes an associative memory (termed Translation Lookaside Buffer (TLB)). The IBM 370 virtual storage mechanism has a somewhat more limited virtual space than does the MULTICS system, due to the 370's 24 bit address and resultant limitations on number and size of segments and pages. [M2]

MULTICS (Multiplexed Information and Computing System) as implemented on the Honeywell 6180 carries to a logical conclusion the concepts of virtual storage and paging with segmentation. MULTICS offers as many as 256K (i.e., $2^{18}$) segments of 64K words ($2^{16}$) per process. The hardware allows segments to be divided into 64 word or 1024 word pages, although 1024 word pages are the norm (thus a maximum of 64 pages per segment is the norm). With the possibility of 256K segment descriptors the mapping tables reside in a segment and are paged in and out of storage.
Thus, the virtual space offered under MULTICS is incredibly large; which also offers advantages beyond being able to write huge programs. With the large virtual space, there is little need to maintain a distinction between the virtual address space and the files a user may require. Files are simply assigned a segment and addressing them is as straightforward as addressing other data the program needs. One widely recognized source of information on MULTICS is Organick [Ol].