INFORMATION TO USERS

This dissertation was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
A Xerox Education Company
GORDON, Robert Loomis, 1937-
THE ORGANIZATION AND CONTROL OF A SLAVE MEMORY HIERARCHY.
The Ohio State University, Ph.D., 1972
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

Copyright by
Robert Loomis Gordon
© 1972

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED
THE ORGANIZATION AND CONTROL
OF A
SLAVE MEMORY HIERARCHY

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Robert Loomis Gordon, B.Sc., M.Sc.

The Ohio State University
1972

Approved by

[Signature]
Adviser
Department of Computer
and Information Science
PLEASE NOTE:

Some pages may have
indistinct print.
Filmed as received.

University Microfilms, A Xerox Education Company
ACKNOWLEDGMENTS

Very few research efforts resulting in a dissertation originate through the labors of an individual. Various kinds of support, personalized discussion and encouragement from many quarters contribute to this thesis. Particular thanks are due to the Department Chairman, Marshall C. Yovits and to the thesis readers, Professors Clinton R. Foulk and Ming-Tsan Liu for their evaluation and criticism. The author would also like to express his sincere and deep appreciation to the Director of Research and Development of Chemical Abstracts Service, Dr. Ronald L. Wigington who has been a source of inspiration and guidance throughout the entire project. Continual encouragement was provided by Dr. F. John Kingsbury and the Long Term Training Committee of the Naval Underwater Systems Center. Also, the efforts of Mrs. Fran McMillen in the typing of the final copy are clearly evident and are greatly appreciated.

The author is fully as grateful for the financial assistance provided by the Naval Underwater Systems Center and by the Department of Computer and Information Science of The Ohio State University. The use of the computers in The Ohio State University Instruction and Research Computer Center, The Ohio State Department of Highways and The Chemi-
cal Abstracts Service provided essential hardware and software. The cooperation of these organizational units provided unique opportunities to the author.

Finally, and perhaps most importantly, an acknowledgment is due the author's wife, Margaret who for several years has withstood the problems of raising a family and living with a graduate student.
VITA

December 15, 1937 • • • • Born - Columbus, Ohio

1959 • • • • • • • • B.S.M.E., Pennsylvania State University, University Park, Pennsylvania

1959-1966 • • • • Experimental Engineer, Pratt & Whitney, Division of United Aircraft Corporation, East Hartford, Connecticut

1965 • • • • • • M.S. Engr. Science, Rensselaer Polytechnic Institute, Troy, New York

1966 • • • • • • Research Electronic Engineer, Naval Underwater Systems Center, New London, Connecticut

1971-1972 • • • • Graduate Assistant, Department of Computer and Information Science, The Ohio State University, Fall and Winter Quarters

PUBLICATIONS


FIELDS OF STUDY

Major Field: Digital Computer Organization

Minor Fields: Operating Systems

Generalized Information Theory
# TABLE OF CONTENTS

ACKNOWLEDGMENTS................................................. ii
VITA ................................................................. iv
LIST OF TABLES .................................................... vii
LIST OF FIGURES ................................................... viii
SYMBOLOGY .......................................................... x

Chapter

I. INTRODUCTION .................................................. 1
   Background ..................................................... 3
   Method .......................................................... 13

II. LOGICAL ORGANIZATION ...................................... 14
   Global Name Space ............................................. 16
   System Name Space ............................................ 19
   Process Name Space .......................................... 22
   Process Name Allocation .................................... 24
   Reallocation of Process Names .............................. 28
   Partial Segment Allocation ................................ 35
   An Example of Process Name Reallocation ................ 36
   Relationships Between Name Spaces ....................... 39
   Summary ....................................................... 42

III. PROGRAM BEHAVIOR ........................................... 45
   Program Reference Spaces ................................... 46
   A Global Reference Model .................................. 48
   Reference Pattern Analysis ................................ 49
   Success Functions and Working Set Statistics ............ 54
   Referencing Slave Memory Hierarchies .................... 61
   Experimental Results from the Global Reference Model  68
   Figures of Merit .............................................. 72
   Summary ....................................................... 74
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV.</td>
<td>RESOURCE UTILIZATION AND CONTROL</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>Fixed and Variable Block Size Hierarchies</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>A Four Level Variable Block Size Memory</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>Memory Management and Control</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>Simulation Description and Results</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>Page Service Time Analysis</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>103</td>
</tr>
<tr>
<td>V.</td>
<td>FILE TRAFFIC IN SEVERAL CURRENT SYSTEMS</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>I/O Control in Multiprogrammed Systems</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>Channel Programs and Device Characteristics</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>Sampling Method and Data Collection</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>Data Analysis</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>125</td>
</tr>
<tr>
<td>VI.</td>
<td>CONCLUSIONS</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>Summary of Ideas</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>Technological Assessment</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>Directions for Further Research</td>
<td>132</td>
</tr>
<tr>
<td>Appendix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.</td>
<td>The GPSS/360 Simulation Model</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>Hardware Parameters</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>Software Parameters</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>Derivation of Expected Page Service Times</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>The Program</td>
<td>145</td>
</tr>
<tr>
<td>B.</td>
<td>The Channel Trace Program</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>The Program Listing</td>
<td>154</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>156</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Comparison of Program Behavior Measures</td>
<td>75</td>
</tr>
<tr>
<td>2.</td>
<td>Tape Block Size Summary for Sites 3 and 4</td>
<td>116</td>
</tr>
<tr>
<td>3.</td>
<td>Direct Access Block Size Summary</td>
<td>117</td>
</tr>
<tr>
<td>4.</td>
<td>Software Parameters for the Baseline</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>Configuration of the GPSS Simulation</td>
<td></td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>An n-level Slave Memory Hierarchy</td>
<td>17</td>
</tr>
<tr>
<td>2.</td>
<td>The Format of an External Reference Block</td>
<td>27</td>
</tr>
<tr>
<td>3.</td>
<td>Allocation of a Process Name to a Global Name</td>
<td>29</td>
</tr>
<tr>
<td>4.</td>
<td>An Example of the Linking of Common External References in Different Segments Using a Doubly Linked List</td>
<td>32</td>
</tr>
<tr>
<td>5.</td>
<td>The Initial Linkage Conditions for an Example of the Reallocation of a Process Name &quot;4&quot;</td>
<td>38</td>
</tr>
<tr>
<td>6.</td>
<td>Linkage Changes during the Reallocation of Process Name &quot;4&quot; to the Second Page of Segment ALPHA</td>
<td>40</td>
</tr>
<tr>
<td>7.</td>
<td>Final Linkage Conditions after Reallocating Process Name &quot;4&quot; to the Second Page of Segment ALPHA</td>
<td>41</td>
</tr>
<tr>
<td>8.</td>
<td>An Example of the Modified LRV Stack Processing Algorithm</td>
<td>52</td>
</tr>
<tr>
<td>9.</td>
<td>Stack Distance Functions for the Page Trace of Figure 8</td>
<td>56</td>
</tr>
<tr>
<td>10.</td>
<td>Reference Interval Functions for the Page Trace of Figure 8</td>
<td>58</td>
</tr>
<tr>
<td>11.</td>
<td>Working Set Functions for the Page Trace of Figure 8</td>
<td>59</td>
</tr>
<tr>
<td>12.</td>
<td>A Typical Success Function for Pages using the Global Reference Model</td>
<td>69</td>
</tr>
<tr>
<td>13.</td>
<td>Average Page and Book Working Set Sizes vs. Window Width for the Global Reference Model</td>
<td>70</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>14.</td>
<td>Typical Working Set Distribution for Pages using the Global Reference Model</td>
<td>71</td>
</tr>
<tr>
<td>15.</td>
<td>Data Paths for a &quot;Trickle Through&quot; Storage Hierarchy</td>
<td>79</td>
</tr>
<tr>
<td>16.</td>
<td>A Four Level &quot;Telescope&quot; Slave Memory</td>
<td>83</td>
</tr>
<tr>
<td>17.</td>
<td>Assumed Process Lifetime Function</td>
<td>87</td>
</tr>
<tr>
<td>18.</td>
<td>Baseline Configuration for Memory Managed with a Fixed Partition Rule</td>
<td>94</td>
</tr>
<tr>
<td>19.</td>
<td>System Sensitivity to a Decreased Read/Write Ratio Between Levels 3 and 4</td>
<td>96</td>
</tr>
<tr>
<td>20.</td>
<td>System Sensitivity to a Decrease in the Level 2 Success Function</td>
<td>97</td>
</tr>
<tr>
<td>21.</td>
<td>Resource Utilization for a Primary Memory Managed by a Working Set Policy</td>
<td>99</td>
</tr>
<tr>
<td>22.</td>
<td>The Distribution of Tape Block Size for Site #3</td>
<td>119</td>
</tr>
<tr>
<td>23.</td>
<td>The Distribution of Tape Block Size for Site #4</td>
<td>120</td>
</tr>
<tr>
<td>24.</td>
<td>The Distribution of Disk Block Size for Site #4</td>
<td>121</td>
</tr>
<tr>
<td>25.</td>
<td>Cumulative Block Size Distribution for Disk Data Commands</td>
<td>123</td>
</tr>
<tr>
<td>26.</td>
<td>The Distribution of Drum Block Size for Site #1</td>
<td>124</td>
</tr>
<tr>
<td>27.</td>
<td>The Distribution of Direct Access Head Seek Time used in the GPSS Simulation Model</td>
<td>137</td>
</tr>
<tr>
<td>28.</td>
<td>Level 2 Success Functions used in the GPSS Simulation Model</td>
<td>141</td>
</tr>
<tr>
<td>29.</td>
<td>Program Flow Chart of the Channel Trace Logic</td>
<td>152</td>
</tr>
</tbody>
</table>
SYMBOLS

The following notation is used throughout this paper with the indicated meaning.

- \( x^y \): exponentiation of \( x \) by \( y \)
- \( <x,y> \), \( \{x\} \): two-dimensional address
- \( x \in y \): \( x \) is an element of \( y \)
- \( x_y \): subscription of \( x \) by \( y \)
- \( \{x_1,x_2,...\} \): countably infinite set
- \( \{x_1,x_2,...,x_n\} \): finite set of \( n \) elements
- \( \Pr[x] \): probability of the event \( x \)
- \( \Pr[x|y] \): probability of the event \( x \) given \( y \)
- \( x \mod y \): \( x \) modulus \( y \)
- \( x[i] \): \( i \)th member of a list
- \( x/y \): \( x \) divided by \( y \)
- \( f(x) \): function of \( x \)
- \( \sum_{i=1}^{k} \): summation operator
- \( \min(x,y) \): minimum value of \( x \) and \( y \)
- \( [x] \): the ceiling of \( x \) or greatest integer
- \( [x] \): the floor of \( x \) or least integer
- \( \bar{x}, E(x), E[x] \): expected value of \( x \)

...
the approximation of y by x

the implication of y by x

logical union operator
CHAPTER I
INTRODUCTION

The intrinsic problem of any storage system is to store the items in such a way that frequently used items are close at hand, and to distribute the rest of the items among the storage system according to their probability of usage. The problem results from a basic desire to minimize the work involved in moving items between the various parts of a storage system. Computer memory systems are unique compared to other kinds of storage systems because the time duration between the use of a single item of information may vary over twelve orders of magnitude. This single fact is sufficient to make computer memory organization a key consideration in computer design. In fact the overall performance of many computer systems is directly related to the successful solution of the basic storage problem.

This thesis is a study of the logical and physical requirements for one type of computer memory system, a slave memory hierarchy, as part of a larger multiplexed computer system. It represents a method of completely automating the storing and moving of information in a computer system in order to eliminate the programming problems that many users encounter in attempting to solve the storage problem on their
own. At this point it is reasonable to consider why the
design of computer memory systems represents an important
problem in computer science.

First is the fact that the effectiveness of a modern
computer system is in direct proportion to the data that is
accessible to it. This effectiveness is in turn due to the
fact that data represents an investment by an organization
that is costly to duplicate and therefore demands sharing of
the data among the community of users belonging to that
organization. Such sharing, through its division of labor
in creating and maintaining the data, and by the cross
fertilization of ideas between users, enhances the value and
validity of the data. This in turn increases the effective­
ness of the computer system thereby accelerating the demand
for even larger systems. A doubling of demand for such
systems in a few years is not uncommon.

Second, the capacity of a computer memory system must be
capable of growing continuously without substantial modifica­
tion of the logical organization of the data to suit current
 technological methods for implementing the memory. The
history of computing is filled with examples of data bases
completely reorganized for the "next" generation computer
system. The achievement of this open ended capability can
be met by a logical and physical system that will automati­
cally handle the allocation and storage of a users data
among the various forms of memory. Such a system frees the
individual user from the concerns of a particular technological implementation at any point in time.

Third, the consolidation of data in one system realizes economies in the maintenance of the software and hardware needed to support large data bases and the associated operating system. The integrity and reliability of data stored may be also enhanced by the inclusion of suitable automated data storage management functions as part of the overall operating system.

Fourth, and finally, the memory hierarchy problem is an interesting problem in its own right since the complete solution touches on all aspects of computer science from man-machine interaction to applied mathematics and computer architecture. It is for this reason that the basic storage problem has motivated much research in many allied areas.

Background

The viewpoint of computers as primarily a data storage system, rather than the earlier viewpoint of computers as primarily a computational system, is developed by tracing the way that data was referenced for arithmetic and logical operations in early computing machines. The development of such a viewpoint is useful since the same concepts that explain data movement between primary memory and the central processor unit can in turn be applied to that of data movement between primary memory and other forms of data storage.
Originally designed for arithmetic computations, most early computers employed zero and one address instructions. If a zero address instruction such as:

ADD

was to be executed, an implicit reference to the location of the augend, the addend, and the result was assumed. The ADD instruction was ordinarily preceded by another instruction that moved the implied data from primary memory into the register locations assumed by the ADD instruction. This in turn must have been preceded by an input/output instruction that placed the data in primary memory. The fact that a zero address instruction, like the ADD, could with the aid of other instructions make references in a global sense was not widely recognized. Instructions or sequences of instructions that aid in making global references have two basic properties. The first property is that of "data movement." A one address instruction such as:

LOAD 1000

that might have preceded the ADD instruction, would have caused the movement of an implied amount of data (usually a computer word) from memory location 1000 to a particular register used in the ADD operation. The second property is that of "address extension." It is derived from the observation that the LOAD instruction may be used to extend the range of addressability of the ADD operation to any pair of words in primary memory. The above properties are separated
for clarity but are in fact intrinsically tied together. In the preceding example the actual result of the LOAD instruction could have either stored the data at location 1000 into a register accessible by the arithmetic unit, or made the address 1000 available to the arithmetic unit to be used in the ADD operation. The point is that either by itself, or in conjunction with other instructions the LOAD instruction causes data movement. Instructions that have the two preceding properties are called "data movement instructions" and are present in every machine since they provide the means by which the central processor makes global storage references.

The fundamental motivation for data movement, as well as the movement of everyday items in the marketplace, is that of efficiency. That is; items are moved to the location of their use. The moving of data items from one storage location to another, programmed via data movement instructions, has a cost associated with each move. This cost is roughly proportional to the amount of time used for the entire move operation. It may be separated into a fixed cost and an amount proportional to the number of items moved as a result of executing the data movement instruction. With this type of cost structure the cost per item may be decreased by moving as many items as possible. However, items moved and never used will increase the cost per item thereby giving rise to an "optimum" number of items for each
data movement instruction. In the LOAD instruction example the number of items (bits) moved from memory to the arithmetic unit is usually fixed and equal to that of a computer word. Even though the arithmetic and logical operations might use one item (bit) at a time, as early machines did, the fixed cost involved in moving individual items from memory to a special register of the arithmetic unit was too high to move a single item. The classical memory paradox of large capacity storage devices having slow access speeds (high fixed cost) and small capacity storage devices having fast access speeds (low fixed cost) as mentioned by Katzan (1) contributes to the dilemma of deciding what is the "optimum" number of items for each data movement instruction. For example, the fixed cost of one class of data movement instructions, the input/output instructions, is much higher than the fixed cost of the class of memory to arithmetic unit instructions. This increases the "optimum" number of items that are usually moved by input/output instructions compared to the memory to arithmetic unit instructions.

As the size and variety of both computational problems and storage devices increased the additional burden of deciding not only what items to move, but also how many, greatly increased the programming and execution time involved in problem solving. This situation was aggravated by the small and primitive set of data movement instructions that resulted from the use of early machines primarily for
arithmetic computation. The easiest and most obvious improvements in the data storage problem were:

- Develop high level languages and their associated data forms and Format statements to aid in programming data movement through the memory hierarchy.
- Increase the number and variety of "data movement instructions" directly executable by the hardware.

An example of the second type of improvement is found in the development of two address instructions such as:

```
ADD 1000,2000
```

that would cause data movement from both storage locations to the arithmetic unit, thus saving the use of a LOAD instruction. Still the range of addresses directly addressable in the early machines by such an instruction did not increase as fast as the range needed by programs because of a proliferation of computer applications and no concurrent breakthrough in storage technology. In addition the mechanisms and patterns of data movement between primary and secondary storage devices were not well understood.

In order to put into perspective the early architectural developments in computer memory systems, it is worthwhile to discuss in detail the relationship between data structures and data storage. As Wegner (2) points out, the fundamental element of data storage is a cell which exists independently of the value that it contains at any given instant. Storage cells are referred to by name, the distinction allowing the
variability between names and values to be factored so that
relations among cells may be expressed as relations among
names independent of their values. Such relations are of
utmost importance in any memory system design since their
structure must be preserved by the memory system. A
computation, or execution of an algorithm may be viewed as
changing the structural relationships between cells and
their values. Certain computer operations alter the value of
a cell, others the name of a cell, and still others both the
value and the name. The most familiar operation, that of
value assignment, results in the change of a cell value as
in the PL/1 assignment statement:

\[ X = 2; \]

A more subtle operation results in a change of the name of
a storage cell, as for example when the name X is reassigned
the name (address) 1000 so that it may be referenced by the
central processing unit. The class of "data movement
instructions" previously defined have this cell renaming
capability by virtue of their address extension property.

If the number of available names that may be referenced
during the course of a computation is not enough to complete
the computation the only choice is to continually rename
cells by using "data movement instructions." This renaming
can either be done automatically as required by the computer
system or explicitly preplanned and controlled by the
programmer/operator, a fact that was recognized as the key
The latter approach represents the conventional file system method as a solution to the renaming problem and is detailed below.

Suppose for example that a user would like to examine data contained in a magnetic tape file with the name JOHN.DOE.REEL.20. The conventional computing system is not aware of this name and hence cannot find it or establish a linkage to it. The usual procedure is for the operator to mount the tape on a tape drive (unit xx) and to inform the program that unit xx is the same as JOHN.DOE.REEL.20. Thus the system is made aware of the existence of an already existing block of information. The reason behind the procedure is that unit xx is the only kind of name that can be referenced by the system, and belonging to a small set of names it must take on different meanings at different times. There are a plethora of examples where a reference was made to a unit of storage that was thought to have a certain name when in fact it had another. For example, if unit xx does not have the correct tape mounted at the correct time, a common human error, catastrophic program results will occur. In the conventional approach the requirement that a storage cell have the right name at the right time is a **user responsibility** and represents a sizable task in programming both large and small computing systems. It is the **central problem** in conventional file systems.
In 1961, two significant accomplishments in storage system architecture took place. Both of the accomplishments were in the area of developing techniques to automate the renaming of storage cells. The first was a software technique introduced by Holt (4) that dynamically allocated program "segments" to main core as needed. The second was the development of hardware to directly increase the range of addressability by instructions. This was the One-Level Storage concept introduced by Kilburn (5) at Manchester University.

The "segmented name space" introduced by Holt represents a software technique for mapping a segment name in the form of a character string into a temporary name (physical address) for use by existing computational machinery. Such mappings were to be carried out by the system without human intervention thus avoiding the central problem in the conventional approach, human error during storage allocation. Character string names inherently belong to large name spaces. For example, strings having a length of n characters composed of m different characters allow for the possibility of m^n distinct names. Such a large name space is of value for two main reasons:

1) Data is most naturally referenced by people in character string fashion.

2) With a large name space the renaming problem may be
avoided by the programmer since all the data used by the system can be explicitly referenced.

The most serious problem with this approach is the inefficiency of current computer systems to handle data referenced in this manner. Nevertheless engineering of such systems has been underway for some time (6).

The introduction of the One-Level Storage concept resulted in the design of hardware that would automatically reassign a name (virtual address) in an expanded name space to another kind of name (physical address) that could be referenced by the arithmetic and logical units. Such systems are now known as linear virtual memory systems, due to the continuous memory like structure of their expanded name space. The approach has merit for two basic reasons.

1) The concept is easy to understand and use.

2) The concept is easy to implement in either hardware or software and could make use of past developments.

The most serious problem with linear virtual memory systems lies in its chief virtue. Since the approach simply extends the available linear name space, it cannot forever avoid the naming problem that was its raison d'être, unless of course technology always allows virtual memory sizes greater than our needs. Another serious, and subtle disadvantage of the approach is that all information potentially common to different programs must be assigned a specific location in
virtual space if a single copy is to be shared, thus increasing virtual name space requirements drastically.

Since 1961 further developments in both the preplanned and automated approaches have resulted in changes to system architecture. The preplanned or conventional approach is chiefly responsible for the software separation of data organization and data access methods as well as the new hardware aids (i.e. programmable I/O) that are available to the user. These tools enable the user to more efficiently plan and execute the management of his storage and represents the most common approach to the storage problem. The "segmented name space" effort has been by and large limited to a family of computers developed by Burroughs (7), and more recently adopted by the MULTICS project (8) at the Massachusetts Institute of Technology. The main contributions have been in the establishment of a methodology for the controlled sharing and dynamic allocation of information among a community of users. The changes in machine architecture influenced by the "virtual memory" approach are due mainly to an increase in the understanding of program behavior gained in the course of implementing the concept. Such changes are not only reflected by virtual machines such as the IBM System 360/67 (9) and the XDS-940 (10) but also in the "cache" or buffer memory design of the IBM System 360/85 (11) and IBM System 370 series. The design of both
conventional and segmented systems will be further enhanced as operational data is collected from these new designs.

Method

In Chapter II the role that names play in large computing systems is discussed and a logical organization for a completely automated slave memory hierarchy is presented. The performance of an automated memory system depends directly on the pattern of references to the stored information that is generated by the community of users. Therefore, Chapter III discusses the measurement of address reference patterns and their role in the design of a slave memory hierarchy. These patterns relate directly to the flow of information between individual storage devices in particular physical organizations, a subject that is treated in Chapter IV. In Chapter V the data traffic in a conventional two-level memory hierarchy is studied by analyzing measurements made in several present day systems. The final chapter reviews the requirements for a slave memory hierarchy and evaluates them with respect to current and future technological prospects. Additional areas of research in memory system design are also discussed in this chapter.
CHAPTER II
LOGICAL ORGANIZATION

Names are the common denominator by which information of all types is referenced. In everyday life people discuss names of books, television programs, letters, buildings, people, time, etc., as a means of selecting a subspace of discourse. Like people, computers must use names in the referencing and processing of information. Thus the role of names provides the central link between man and machine in any system organization for the storing and moving of information.

Because names are used to refer to entities and ideas there is associated with each name a set of attributes, the attributes serving to distinguish and classify the various names. The assignment of a name to an entity causes the implicit association of the name and the attributes of that entity. This association, known as binding, is not always easily reversed since an interlocking of entities with similar attributes forces an interlocking of their names. The nature of this binding process, its reversibility, and its role in the design of a hierarchial data storage system must be carefully examined.

It is possible to encode information of all forms into
a string of binary digits. Such strings, known as "data structures" in computer science, are used to represent many different things. The semantics of the information in these data structures is purposely disregarded in order to concentrate on the attributes of length, location, ownership, date of creation, method of coding, etc., that are important to data storage systems. Names are commonly associated with data structures in order to allow the composition of other data structures and the selective referencing of their various components by both people and computers. It is the business of the data storage system to move, store, and keep track of data structures, while it is the business of the central processor to interpret and transform them.

Although the physical structure of a computer system may take many forms, as technology dictates, the logical structure seen by the user should present a consistent and invariant interface to the system. Collections of information naturally grow in size, so it is necessary that separate open ended physical and logical structures coexist within the finite limitations of any system design in order to achieve an invariant interface. For these reasons three separate and related name spaces are introduced in this chapter as the basis of the logical organization. The first name space is constructed as a natural logical interface for the users. The other two name spaces are internal to the system (i.e. the user need not be aware of these names) and are
required for the efficient storing and accessing of the information.

Pervading the design sketched in this chapter is the underlying assumption of a slave memory hierarchy as shown in Figure 1, and the concept of a segment. A segment is considered to be a data structure (2) with the minimal attributes of length and location. Examples of segments are such diverse items as payroll files, a book in a library, or a compiler. Some of these forms may be in the last level of the memory hierarchy (off-line), and may have to be transcribed into machine readable form before being used in a computation, but this does not void their consideration as a segment. Other attributes such as read only, execute only, etc., admittedly will be necessary in an environment where segments are to be shared (8) since they are used to control the copying, interpretation and modification of the segments, but they do not contribute to an understanding of the logical organization presented in this chapter.

Global Name Space

If the objective of growth is to be met, as outlined in Chapter I, the data storage system must be capable of referencing data stored off-line of all kinds. This demands a wider view than is commonly held of the addressing capability of a computer. It demands the view that computers can reference any segment that is contained in a master index, so that entering a segment name in an index is the only action
Figure 1
An n-level Slave Memory Hierarchy
necessary on the part of the user to make that segment available for computational purposes. This action constitutes the binding of a data structure to a global name.

Since the people that use the computer should refer to segments by a natural character string name, segment names as seen by the users are assumed to be a variable length character string. Each is composed of \( m \) characters and delimited by a special \( m+1^{st} \) character. The special character, \$ in the examples of this chapter, is of course outside of the alphabet used to compose a segment name. The set of all possible variable length names constitutes a variable size name space that is defined as the global name space \( N_g \). The variable size is crucial to the logical organization since it allows unlimited growth for the global name space and hence does not force the reuse of a global name.

However, at any instant in time it has a finite size of \( m^n \) names, where \( m \) is the number of allowable characters in the alphabet, and \( n \) is the length of the longest name in the characters. The main purpose of the global name space is to provide a consistent and invariant interface between the system and the users since the renaming of information is not necessary.

The most natural way for people to logically organize a collection of names is in a hierarchial structure of directories. It is probable that for most systems segments will be catalogued in several different directories to aid the
directory search effort and to aid in the collective development of segments. It is not the intent of this thesis to discuss classification problems and information retrieval problems associated with names, directories or catalogues, therefore it is sufficient to assume a directory structure similar in principle but not necessarily in detail to the Multics organization (12). Each directory entry is assumed to contain at least the minimal set of attributes, and will for practical systems need to contain many more. Naturally the directory is itself a segment so that it may be handled like any other segment. The only unique requirement of the directory segment is that the directory root be kept in the lowest level of the hierarchy (wired down) since all directory searches must be recursively limited. It may also be desirable to limit the directory segment to only on-line storage for performance reasons but this is logically not required. The hierarchial structure of the directory provides the necessary framework for people to operate effectively in a computer environment from a logical standpoint. However, it does not provide the organizational capability needed by a data storage system or the addressing efficiency needed by a central processor for computational purposes. There must coexist separate internal name spaces for these purposes.

System Name Space

The physical organization of a collection of segments
in a memory hierarchy should be independent of the logical organization as seen by the users. One method of accomplishing this is to consider the segments to be physically organized into various fixed size data structures known as pages, books, volumes, tomes, etc., such that an active page at one level would be contained in an active book at the next level, that is contained in an active volume at the next level, and so forth. Implicit in this organization is the ability of each level of the memory hierarchy to keep track of its own contents. Each segment may then be assigned one or more system names (unknown to the user) that may be used in the storing and moving of segments between various levels of the memory hierarchy. From time to time the system may reassign system names to segments based on some usage criteria in order to effect a physical reorganization of the collection. Because of this, the pages in any given book may or may not contain logically related information. Systems that physically organize segments only by logical relationships pay the price of storing and moving complete segments even though parts of them have widely different usage.

The set of all system names is defined as the system name space $N_S$. Like the global name space it too must have the ability to grow and evolve so that system names must be variable length names. However, unlike the global name space the system name space has an implied length associated with each part of the system name, thereby making it easier
for referencing and handling by both the hardware and the software. As an example, consider segment ALPHA belonging to user XYZ. The partial directory entry for segment ALPHA might look like this:

Global Name: ROOT.USER.XYZ.ALPHA.$
System Name: BOOK(28).VOL(16).TOME(3).$
Size: BOOKS (2)

Since the size of ALPHA is two books it is implied in this case that the second book of ALPHA may be referred to as BOOK(29).VOL(16).TOME(3).$. The system name can be used by the storage system to determine the location either directly or indirectly. In the proposed logical organization the location is dependent on the state of the memory hierarchy. Segment ALPHA may or may not be on-line. Whether or not it would be on-line is strictly a function of the pattern of references over the memory hierarchy, in particular whether volume 16 was recently referenced. This of course is only one of many possible organizations. For different implementations the system name may have different forms and may even be an absolute device address. Hopefully, if it is a much used segment, it will require less time to locate than one which is not used as frequently.

At this point it should be noted that in any storage organization scheme one cannot forever postpone the allocation of space for objects which must be kept. It is at the moment of assignment of the set of system names to a segment
that logical space in $N_s$ is reserved. If the segment previously existed, as perhaps data in hard copy form, we do not have to assign any physical space because it already exists. The conventional distinctions between the storage of information on-line and off-line have very little significance in this type of organization. Information conventionally thought of as being stored off-line is simply information requiring a greater access time than information that is stored on-line! Because any segment that resides in the master index is capable of being accessed, information in many forms will be available to the computer.

**Process Name Space**

With the proposed ability to index (and thereby access) off-line data bases that may exceed the $10^{13}$ bit (13) capacity of currently available on-line devices, global names may grow in size to be 50 to 100 characters in length. It is presently not attractive, for mainly technological reasons, to have the central processor handle long variable length names for every instruction and data reference. Thus it is customary to use a **fixed size linear address space**, or **process name space** $N_p$, for each process in execution.

The attractiveness of a process name space for referencing information stems from the fact that the amount of information normally referenced during the life of a process is generally a very small portion of the total information.
known to the system or to the user.\footnote{1} However, it is not known a priori which subset of the total information in the system the process is going to reference, and therefore defining the process name space should be deferred until execution time. A user, however, must have the ability to reference all of the global name space $N_g$ if he has that privilege, which points up the obvious face of having access rights as a segment attribute as Graham so clearly details (14).

One solution, which is the approach of most of the currently designed classical and virtual systems, is to fix the size of the process name space at some compromise between the projected requirements of the users and what is technologically feasible, hoping that it is large enough for most applications. If it is not, the user is forced into managing his own process name space using a separate logical file system. When this happens the user might as well be using a conventional preplanned system.

Another solution, which is the subject of the rest of this chapter, is to have the system automatically manage the process name space and reuse process names as the need arises. The rate at which the names in $N_p$ can be reused without intolerable system degradation is however an open

\footnote{1. There exists certain classes of problems in which the information referenced is not a small portion of the total space of information known to a computer. Many of these problems handle large data bases and are economically important. This kind of problem motivated much of this research.}
question and depends on many system hardware and software characteristics, some of them discussed in the next two chapters. Process names may or may not be core addresses. In many instances, and in this chapter, they may be thought of as virtual addresses (virtual memory). It would of course influence system performance in a particular implementation but does not change the proposed method of name management.

Process Name Allocation

A large percentage of the references to segments generated by the processor should be to process name space for reasons of efficiency. Central processor references to a process name will be referred to as a short form reference, in contrast to a long form reference when using the variable length global name for referencing segments. A process name is assumed to have a standard two-dimensional address\(^2\) in the form of:

\[
<\text{Segment Number, Word Offset}>
\]

with additional instructions being provided for referencing characters and bits within the words. Segment numbers are to be allocated as process names to segments referenced in global name space in the order that they are referenced. Although this method is like that of Multics (15), it is

\[2. \text{ For the figures in this chapter the segment number is represented as a decimal number and the word offset as a small alphabetic letter. When the segment number is an * it means this segment.}\]
proposed that an additional mechanism be established for automatically reusing process names (segment numbers) as the process "wanders" through global name space. Thus, the system will "page" back and forth between the process name space $N_p$ which is fixed in size and global name space $N_g$ which is not fixed in size.

In systems that page from process name space to physical memory the concept of a memory map that converts logical names into physical addresses is an important one (16). The usual method of using the page number part of the logical name as an index in a table to obtain a physical address for the beginning of the page cannot be used for paging between global name space and process name space, because the global name space is not an ordered space (i.e. the global name cannot be used as an index into a table). When encountering a global name during the execution of a process the list of global segment names that the process has already activated (and therefore already assigned a process name) would be searched first. If the global name is not present in this list, then the relevant set of directories must be searched for the segment that will be paged into the process name space. Once the process name is assigned to a segment, further references to the segment will automatically use the process name unless the name is reallocated. The strategy for discussing details of the allocation and reallocation of process names is to assume that the size of a segment located
in global name space is less than or equal to the size of a page of process name space and that a page of process name space is available. The additional complexity needed for allocating variable size segments will be explored later.

Since a process may reference a segment by either its global or process name, a "reference" bit will be used to indicate to the processor which name is available. Such reference bits are assumed to be present on all words in core, but are assumed to be transparent to the user in much the same fashion as present day parity bits. A process segment is assumed to make all external references indirectly via a four word reference block located within the executing segment. The first word of this four word reference block is checked for the existence of a short form reference by noting the value of the reference bit. If the reference bit is not 1 (not on) then the next word represents the offset relative to the segment start of a character string that represents the global name as shown in Figure 2. This name is used to search the relevant directories of the global name space. Once the segment is located in the master directory the next available process name is allocated to that segment by making an entry in the Process Name Table (P.N.T.) and assigning the index of that entry as the segment number for the short form reference. The segment number and offset, if

---

3. The analogy with parity bits is intended to make the point that these do not contribute to the basic character or word size.
Figure 2
The Format of an External Reference Block
present, are then placed into the first word of the four word block and the reference bit is turned on as shown in Figure 3. The next time the processor makes a reference to the segment via the four word reference block the process name may be used by the CPU for addressing as noted by the reference bit.

Reallocation of Process Names

If it is assumed that the process name space is large enough to include all of the segments needed by a process during the course of its execution then a process name would always be available for allocation. For reasons already mentioned, many processes will not have enough process name space and therefore it is necessary to include a mechanism to reallocate a process name to a different segment. The mechanism used must insure that all short form references to the segment whose process name is being reallocated will revert back to a long form reference. Unless all occurrences of a short form reference to the particular segment whose process name is being reallocated are eliminated, the danger exists of having the same process name (or segment number) for different segments. This problem is a different form of the dangling reference in high level languages as pointed out by Wegner (2). The solution to the problem lies in the recognition that the lifetime of the process name in the first word of the reference block may be less than the lifetime of the reference block itself. This cannot be true
Figure 3

Allocation of a Process Name to a Global Name
for global names which may always be used for external referencing by turning "off" the associated reference bits.

If a process name is unable to be allocated to a segment because no more names are available in the P.N.T., one must be selected for reallocation from the P.N.T. This selection may be governed by a replacement policy similar to the kinds of page replacement algorithms studied by Belady (17) for use in paged systems. For example, if segment number three in Figure 3 was selected by the replacement policy for reallocation then the reference bits associated with each four word reference block using segment number three as a process name must be turned off. The process name "3" would then be available for use by another segment and any future references to ROOT.USER.XYZ.ALPHA.3 would be trapped as a long form global reference. Thus, during the lifetime of a single process, a segment may take on many different process names in addition to having different process names at the same time for concurrent processes.

In order to turn off the reference bits associated with the segment whose process name is being reallocated, a list is required of the places that are using a process name as an alias for a global name. This list will be called the Substitution List (S.L.) and may be implemented in the form of a singly or doubly linked list (18) for each entry in the P.N.T., the choice depending upon the use of the list. It is desirable, for reasons that will be explained, to use a
doubly linked list for the Substitution List since it has the capability for deleting a single entry without searching the entire list. As shown in Figure 4 the third and fourth words of the reference block are used for the backward and forward links of the Substitution List. Since the links, or pointers in the S.L. would be process names, a question arises as to whether or not a list of the pointers is needed. Fortunately, all process names associated with the particular segment selected for reallocation can be removed by the following two actions:

1) Delete the process names occurring as pointers in a Substitution List by turning off the reference bits of all reference blocks located in the segment.
2) Turn off all the reference bits in the reference blocks that belong to the Substitution List associated with the segment number being reallocated.

In order to accomplish the first action it is proposed that the channel between the lower and higher levels of the memory hierarchy "process" the reference blocks when a segment is physically deallocated from core. This method also has the advantage of avoiding costly accesses to slow devices in the storage hierarchy when processing the Substitution List as required by action two. The processing can be implemented by having the channel detect reference blocks with their reference bits "on", and then implement the necessary node deletion from the Substitution List. Node
Partial Process Name Table Entry for Segment 3 (Segment 0)

<table>
<thead>
<tr>
<th>Segment</th>
<th>ROOT.USER.XYZ.ALPHA.®</th>
<th>BOOK(28), VOL(16), TOME(3),®</th>
<th>PAGE (1)/BOOKS(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Root: USER.XYZ.ALPHA.

Figure 4
An Example of the Linking of Common External References in Different Segments Using a Doubly Linked List
deletion can easily be accomplished by loading the backward and forward list pointers in the reference block into memory address registers and storing the contents of each at the address given by the other, an operation that could be added to present day micro-programmed channels and even overlapped with normal channel operations. It should be noted that the first action required, and the proposed implementation, merely represents the requirement that the physical deallocation of a segment precede the logical reallocation of its name space.

To demonstrate that it is not necessary to keep track of the short form references to a segment that occur as pointers in a Substitution List the situation of two external references to segment ROOT.USER.XYZ.ALPHA.$ is shown in Figure 4. If segment ALPHA is a candidate for process name reallocation its Substitution List would be traced in order to find all the reference books using "3" as an alias for ALPHA. By turning off the reference bits in these blocks, it is possible to prevent the use of "3" as a process name for segment ALPHA. Even though the reference block contained in segment five contains a short form reference to segment ten, it is not necessary to keep track of this reference to segment ten if the segment is reallocated, since segment ten is not eligible for reallocation of its process name unless it is first physically deallocated from the lower level memory. The physical deallocation of segment ten
from lower level memory would cause the channel to execute a node deletion and eliminate all the short form references to ten that occur as pointers in ALPHA's Substitution List. For example, the processing by the channel of the reference block located in segment ten of Figure 4 would replace the forward pointer <10,d> in the reference block of segment five with <0,e> the forward pointer in the reference block of segment ten, and the backward pointer in segment zero with the backward pointer in segment ten. Thus all process name references to segment ten in ALPHA's Substitution List or any other Substitution List would be eliminated.

This dynamic reallocation of process name space is not obtained without incurring some overhead. In addition to the extra channel hardware needed to detect the reference bits and execute the pointer swaps necessary for node deletion the following two actions are required:

1) For each external reference executed in segments that have left and returned to memory the reference block must be added to a Substitution List.

2) Process all segments leaving core through the channel in order to "clear" their reference blocks.

The second action constitutes the physical deallocation that is necessary to make the segment eligible for reallocation. All incoming segments would have the reference bits off. Since most of the returning segments will have a P.N.T. entry and not require a directory search, the first required action
may be aided by either implementing the P.N.T. as a content addressed table using the global name as a key or by hash techniques. The second required action may be aided by classifying the segments in core into the following three categories.

1) Segments that have not modified any words or executed any external references. These do not have to be processed by the channel and can be written over.

2) Segments that have modified only their reference blocks. These can be processed by the channel to execute the necessary node deletions at a rate limited only by the memory cycle speed.

3) All other segments. These have to be written back to the higher levels of storage anyway, and therefore can be processed by the channel during the writing.

Partial Segment Allocation

By their very nature segments will vary in size from a few words to data bases larger than process name space. Therefore it is necessary to provide a mechanism for handling segments of a size larger than the unit allocated to process name space (page in our examples). If the segment size is unduly restricted it can result in an apparent high level language limitation as was experienced by ALGOL users on the Burroughs B5000/B5500 system when they were restricted to a maximum vector size of 1024 words. When the segment
size is unrestricted it may be impossible to allocate it completely to process name space since a segment could be larger than the entire process name space. Such a procedure would be foolish anyway, for it is quite likely that large segments will only have certain portions active at any moment for a given process. The capability for allocating only a page at a time to process name space is inherent in the system name initially assigned to the segment. If, for example, we attempted to reference the second page of segment ROOT.USER.XYZ.ALPHA.$, an address fault handler could check the directory to see whether a second page existed for the segment. Verification of this would allow the allocation of another page of process name space to the segment. The name entered in the P.N.T. would then be the original global name with a special system qualifier and the appropriate system name so that future references could be handled in a similar fashion. The effect of the above method is to allocate process names to pieces of large segments as required during execution, since it is possible to transform a reference to any part of a segment as a page, book, volume, etc., offset that can be validated and allocated as indicated.

An Example of Process Name Reallocation

In order to clarify the automatic mechanism of process name space reallocation, a simplified situation is presented that includes the essential aspects of the method. This
situation is a within segment reference in a segment that is only partially allocated to process name space. It occurs in a full process name space which implies that there are no more process names available for segments. Therefore, it is possible to concentrate on only two entries of the P.N.T. to demonstrate the method. As shown in Figure 5, the first page of segment ALPHA is assumed to have been referenced within segment five and ten. Note also that an additional reference within segment five was made to a page of segment BETA. The process is triggered by a reference occurring in segment ten to the second page of ALPHA. The system qualifier #2 of the triggering reference is next used to verify the existence of a second page for ALPHA via a directory check. If the request is valid, a replacement algorithm of a memory management program will select a process name in the full P.N.T. for reallocation. Assume that the process name "4", which is an alias for segment BETA, is selected for reallocation. The reallocation process must set to zero all reference bits in BETA's Substitution List so that future references will use the global name BETA. Because the page of segment BETA that is about to be logically deallocated from process name space has already been physically deallocated (i.e. does not exist at a storage level below the channel doing the list processing) it is guaranteed that the only outstanding occurrences of the process name "4" may be found by processing the Substitution
Partial P.N.T.

(Segment 0)

<0, e> ->

<0, f> ->

(Segment 5)

<5, h> ->

<5, c> ->

(Segment 10)

Processor Reference

Figure 5

The Initial Linkage Conditions for an Example of the Reallocation of a Process Name "4"
List for BETA starting at the P.N.T. The global and system names for the second page of ALPHA are next entered in the P.N.T. as shown in Figure 6 and a process name "allocated" for that entry. At this point segment number four is the process name for ALPHA.\#2, the second page of segment ALPHA, and a page call may be initiated using the system name to locate the missing page. When the missing page is moved to low level storage a page presently in low level storage will probably be pushed out to make room for the incoming page. Suppose that segment five is selected to be pushed out. This page migration would cause the channel to set to zero the reference bits in the remaining reference block of segment five and at the same time delete the remaining occurrences of process name five that exist as pointers in the reference blocks within segments ten and zero as shown in Figure 7. Note that if segment five does not get pushed out and it executes a reference to BETA at some future time the reference would be trapped as a global reference and BETA would most likely be allocated a different process name (segment number) than previously allocated.

Relationships Between Name Spaces

The process name allows the convenience of each process to reference common segments by separate process names during the course of a computation so that single copies of segments may be shared without regard to prior agreement on their process names. Since the process name may also be used as
Partial P.N.T.

(Segment 0)

<table>
<thead>
<tr>
<th>3</th>
<th>ROOT.USER.XYZ.ALPHA.$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BOOK(23).VOL(16).TOME(3).$</td>
</tr>
<tr>
<td>PAGE(1)/BOOKS(2)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>d+1</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
</tr>
</tbody>
</table>

(Segment 5)

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>e+1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

(Segment 10)

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>k+1</td>
</tr>
<tr>
<td>1</td>
<td>f+1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>g</td>
</tr>
<tr>
<td>1</td>
<td>c+1</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
</tr>
</tbody>
</table>

Processor Reference

Figure 6

Linkage Changes during the Reallocation of Process Name "k" to the Second Page of Segment ALPHA
Partial P.N.T.

(Segment 0)

<table>
<thead>
<tr>
<th>3</th>
<th>ROOT.USER.XYZ.ALPHA.$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BOOK(28).VOL(16).TOME(3).$</td>
</tr>
<tr>
<td>PAGE(1)/BOOKS(2)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>d+1</td>
</tr>
<tr>
<td>10</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>ROOT.USER.XYZ.ALPHA.#2.$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BOOK(28).VOL(16).TOME(3).$</td>
</tr>
<tr>
<td>PAGE(2)/BOOKS(2)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>m+1</td>
</tr>
<tr>
<td>10</td>
<td>m</td>
</tr>
</tbody>
</table>

(Figure 7)

Final Linkage Conditions after Reallocating Process Name "4" to the Second Page of Segment ALPHA
an index into the P.N.T. that supplies the actual physical address of the page via the system name, the correspondence between process names, system names, and global names enables any name space to represent physical storage and therefore be used as an address.

The system name allows the convenience of binding together different segments into different size data structures (i.e. books, volumes, etc..) so that the storage and handling of information may be based on usage statistics or any other criteria, the advantages having been pointed out previously. The assignment of the various parts of the system name to the different storage devices in a hierarchy has purposely not been discussed. It depends to a large extent on the reference patterns of system names produced by concurrently executing processes, a subject that is treated in the next chapter.

The variable length global name allows the convenience of each user to logically organize and reference his collection of segments as desired. It has a tremendous psychological advantage in the structuring and remembering of logical organizations that should not be overlooked since the user is not forced into generating mononics for segment names that are generally difficult to remember.

Summary

The major characteristic of the proposed organization is the ability to dynamically allocate and reallocate to
processes a fixed length process name space. It has been shown that process names may be considered to be serially reusable resources and therefore must be managed either by the system (automatic allocation) or by the user (preplanned allocation). The completely automatic approach was selected as the most promising for continually evolving memory hierarchies and consequently was developed for application to a slave memory hierarchy.

The concept and implementation of a virtual memory system originated in 1961 with the group at Manchester, England (5). Segmentation appeared in the work of Holt (4), the Burroughs group (7), and at the Rice University (19) also around 1961. The combining of a segmented name space and the dynamic allocation of virtual memory are features of Multics (6,8,12,15,20,21) that originated around 1965. Both the Multics and Burroughs segments are considered as linear arrays with a maximum limitation on segment size, thus forcing the user to partition large files to allow their allocation. In principle both the segmented and virtual memory systems are designed to completely automate storage allocation so that a user does not need a file system. In practice most contemporary systems provide file processing primitives that operate "outside" virtual memory and allow the conversion or allocation of a file to a segment. This is the reason that manipulation of large
files on virtual memory systems in reality must revert back to the preplanned approach.

The physical organization of a large file system based in part on usage statistics has been implemented in the TABLON system (22) and in the TSS/360 environment (23). Both systems provide commands that must be used by the users and/or operators to force migration of the data to and from archival storage. As yet no system has completely automated the migration throughout the entire storage hierarchy. Randell and Kuehner (24), Denning (25,26), Wilkes (27,28) and Katzan (1) place segmentation, virtual memories, slave memories and hierarchial storage in proper perspective among all storage allocation techniques.
CHAPTER III
PROGRAM BEHAVIOR

The behavior of programs in computing systems is perhaps the least understood but the most important single influence on computer architecture. Even though this has long been recognized by the fact that some computers were designed to run specific languages efficiently (7,29) little progress has been made in both the characterization and understanding of program behavior. Yet the requests for allocation of resources in a computing system depends entirely on program behavior.

Most of the effort in the study of program behavior has been concerned with the study of individual program behavior (17,30,31,32,33,34,35) by the use of a "trace" program that provides a list of "references" or addresses during the course of execution. For a majority of programs, references are sensitive to input data and require repeated tracing with different data to obtain an average reference pattern. The repeated tracing, coupled with the fact that one second of program execution results in millions of references, makes such procedures very expensive. In spite of this, tracing has been used to improve programming methods (33,34,35), to
predict program behavior on different hardware (31), and in certain other aspects of system design (11,36).

The current trend (37) of using computers more and more to remember things both off-line and on-line causes the time between references to vary from a few microseconds to a few months or years. Information referenced by a central processor implies its use in a data processing operation and therefore causes data movement within the storage hierarchy. Multiprogramming was introduced to allow the use of the central processor by one program while other programs were waiting for the completion of long duration data movement instructions. Because of this multiprogramming environment, interest is focused not in the individual behavior of programs, but in the global reference patterns produced by a collection of concurrently executing programs in both multiprogrammed and multiprocessing systems.

Although collecting data on global reference patterns in a multiprogrammed system poses formidable problems, and is beyond the scope of this dissertation, it is crucial in guiding the actual development of a memory system. It is proposed, in lieu of such data, that a model of global reference patterns be used to aid in the understanding of the behavior of a hierarchial slave memory system.

**Program Reference Spaces**

Current definitions of a program reference assume a finite address space, virtual or real, so that any reference
generated by the central processor belongs to the defined space. References are also assumed to be generated sequentially in time, even though a single machine operation may require multiple references, thus allowing the resolution of time into discrete instants \( t_1, t_2, \ldots, t_i \). Although the actual elapsed time between the reference instants may in fact be variable, the added complexity is not ordinarily included in models. Given the finite address space \( N \) and the above assumptions, then any particular execution of a program may be viewed as generating a "reference string" \( c \), where:

\[
c = r_1, r_2, \ldots, r_i, \ldots, r_t : r_i \in N
\]

The reference string, \( c \), is considered to be a machine independent representation of program behavior. If virtual addresses are used it is customary to ignore the \( k \) low order bits of the address \( r_t \), where \( 2^k \) is the page size, and use the remaining address as a page reference in order to obtain a "page reference string". References to files outside the finite set of \( N \) addresses for both real and virtual program tracing are usually ignored.

It is possible to consider references over the entire memory hierarchy to both off-line and on-line data. The concept requires one to view the computer as making references to a set of addresses \( N^* = \{ 0, 1, 2, \ldots \} \) having an undefined number of elements. Such sets encounter certain analytical and computational difficulties and would require variable length addressing (e.g. long form referencing).
Nevertheless the concept is valid, and is amenable to certain empirical analyses. To date, reference strings, even to the finite set of all on-line data, have not been collected.

**A Global Reference Model**

The wish to analyze reference strings covering a wide range of behavior, coupled with the fact that no measurements of global reference patterns are available, led to the development of a scheme for modeling global patterns. No claims are made as to how closely the reference patterns modeled approximate the actual patterns in any system; the chief aim of the model is to apply and extend current analysis techniques and to isolate memory system parameters most affected by changes in reference patterns.

The central processing unit is considered to be generating references over the entire memory hierarchy that includes all the instructions and data needed by any executing program. Although the address space may be constantly changing, as allowed by the type of logical organization presented in Chapter II, at any instant in time it is considered to be of finite size $N$. Three basic modes of referencing by the CPU are assumed as follows:

1) Random referencing:

   \[ \Pr (r_t = i) = \frac{1}{N} \quad i \in N, \quad t \geq 1 \quad \text{3-2} \]

2) Sequential referencing:

   \[ r_t = \begin{cases} 0 & t = 1 \\ (r_{t-1} + 1) \mod N & t > 1 \end{cases} \quad \text{3-3} \]
3) Loop referencing where \( k \) is the loop length:

\[
 r_t = r_{t-k} \quad k < t, \ t \geq 1
\]

This model gives the desired flexibility for generating global reference sequences since other modes of addressing may occur, such as looping with only one, two, or several references changing within a loop. The change from one mode of referencing or state of operation to another mode is governed by a Markov process (38), the state transition probability matrix being a model parameter. Markov schemes, using page transition probabilities have been studied by Kral (39) and Pinkerton (40) but such complexity is felt to be unwarranted for this application since specification of page transition probabilities is difficult with the present state of knowledge of global reference patterns over entire memory hierarchies.

Reference Pattern Analysis

Any given reference string may be analyzed in a number of different ways. One of the most straightforward ways is to obtain performance data (i.e. page exceptions) using the reference string to analyze models of proposed system configurations. As useful as this is, it is common for certain reference strings to have acceptable performance on one system and poor performance on another. Therefore, it is desirable to search for system independent measures of software behavior, ones that would allow classifying
reference strings with respect to their rate and amount of storage used as "good" or "poor".

Mattson, et. al. (41) introduce a technique known as "stack processing" that allows various properties relating to storage hierarchy performance to be derived from the reference string in one pass. Denning (42,43) introduces the "working set" measure of program performance with respect to storage usage that may also be derived from reference string processing. It is possible, however, to combine both techniques and extract both of the above measures in one pass of the reference string by a slight modification of the stack processing technique introduced by Mattson, et. al. (41).

Stack processing derives its name from the use of an ordered list of references or "stack" that is updated at each reference instant according to a particular priority scheme. The name, "LRU Stack", is derived from applying the criteria of Least Recent Usage for ordering the references in a stack and is intimately associated with working set statistics. Fortunately, it is easily implemented in either hardware or software due to an iterative technique for maintaining the least recent ordering. Denoting the number of distinct pages referenced since the first reference instant by \( \gamma_t \), then at each reference instant the ordered list or stack \( P_t \) of referenced pages is represented as

\[
P_t = p_t[1], p_t[2], \ldots, p_t[1], \ldots, p_t[\gamma_t]
\]
such that \( p_t[i] \) has been referenced more recently than \( p_t[i+1] \) for all \( i \).

The list is modified by associating with each page \( p_t \) on the list a companion counter \( c_t \) such that \( c_t = c_{t-1} + 1 \), except for the referenced page \( p_t[1] \), its counter being set equal to one. Since the stack must be searched at each reference instant to see if the referenced page is on the stack, the effort required to update the companion counter is negligible. It also provides a convenient method for keeping track of the "reference interval" or the number of time instants that have elapsed since the last reference to a page. This reference interval is the cornerstone for most of the working set statistics (43).

In order to understand the maintenance of the modified stack, it is only necessary to concentrate on two successive reference instants, for example the seventh and eighth reference instants of the page trace shown in Figure 8. After the seventh instant, the list of pages \( P_7 \) is equal to \( b,a,d,c \) with the corresponding counters \( 1,3,4,5 \) providing the elapsed time since the last reference to each page on the list. The eighth reference is to page \( d \), and will cause \( d \) to be placed at the top of the list \( P_8 \) with the pages previously above \( d \) being pushed down one position, and the page or pages previously below \( d \) unchanged in their order. The counters for all the pages on the list except \( d \) are incremented by one, thus \( P_8 = d,b,a,c \) with counters \( 1,2,4,6 \).
### Reference Instant

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

### Reference String

| a | b | c | d | a | b | b | d | c | b |

### Modified LRU Stack

- Stack: \( a(1), b(1), c(1), d(1), a(1), b(1), c(1), b(1) \)
- Distance: \( ^2, ^2, c(2), ^*a(2), a(3), ^*b(3), d(3), ^*c(2), ^*d(3) \)

### Stack Distance

| * | * | * | * | 4 | 4 | 1 | 3 | 4 | 3 |

### Reference Interval

| * | * | * | * | 4 | 4 | 1 | 4 | 6 | 3 |

### Working Set Size

| 1 | 2 | 3 | 3 | 3 | 3 | 2 | 2 | 3 | 3 |

(\( \tau = 3 \))

### Stack Distance

The position from the top of the previous stack at the instant of reference.

### Reference Interval

The number of time instants elapsed since the last reference.

### Working Set Size

The number of distinct references in the preceding \( \tau \) time instants.

---

**Figure 8**

An Example of the Modified LRU Stack Processing Algorithm
The working set at time $t$ is simply the set of distinct pages referenced in the $\tau$ immediately preceding time instants, where $\tau$ is the "window size" of the set. Thus the number of pages that have reference intervals less than or equal to $\tau$ at any reference instant is equal to the working set size for that value of $\tau$. Since $c_t[i] > c_t[i+1]$ for Least Recently Used ordering, the relative position from the top of the stack for which the criteria $c_t \leq \tau$ holds, is an indication of working set size. For example, at the eighth reference instant pages "d" and "b" have reference intervals (counters) less than 3, while pages "a" and "c" have intervals greater than 3, thus indicating that there are only two distinct pages referenced during the last 3 time instants, namely "d" and "b".

An efficient algorithm can be constructed to implement the above modified stack processing. Let the current reference $r_t$ be the integer variable PAGENO, and the reference interval, stack distance, and working set size for $r_t$ be the integer variables INT, DIST, and SIZE, respectively. The window width for the working set is denoted by the integer variable TAU, the current stack length (initially zero) by the integer variable PTR; and the stack of pages and counters for a maximum of $N$ pages by the integer structure STACK declared as

1  STACK,
2  PAGE (0:N),
2  COUNTER (0:N);
The algorithm is:

\[
\begin{align*}
\text{COUNTER}(\text{PTR}) &= \text{INT} = \text{DIST} = \text{SIZE} = 0; \\
\text{PAGE}(\text{PTR}) &= \text{PAGENO}; \\
\text{DO} & \text{ I = PTR - 1 TO 0;} \\
\text{COUNTER}(\text{I}) &= \text{COUNTER}(\text{I}) + 1; \\
\text{IF PAGENO} &= \text{PAGE}(\text{I}) \text{ THEN} \\
\text{DO;} \\
\text{INT} &= \text{COUNTER}(\text{I}); \\
\text{DIST} &= \text{PTR} - \text{I}; \\
\text{DO} & \text{ J = I TO PTR - 1;} \\
\text{PAGE}(\text{J}) &= \text{PAGE}(\text{J} + 1); \\
\text{COUNTER}(\text{J}) &= \text{COUNTER}(\text{J} + 1); \\
\text{END;} \\
\text{END;} \\
\text{IF COUNTER}(\text{I}) < \text{TAU} \text{ THEN SIZE} &= \text{I}; \\
\text{END;} \\
\text{IF INT} &= 0 \text{ THEN } \text{PTR} &= \text{PTR} + 1; \\
\text{SIZE} &= \text{PTR} - \text{SIZE};
\end{align*}
\]

Success Functions and Working Set Statistics

A property of the LRU stack that is detailed by Mattson, et. al. (41) is that for memory hierarchies with the contents managed by a LRU replacement algorithm the pages belonging to the lowest level memory of capacity C pages are always a subset of the pages that would belong to the memory if it had a capacity of C + 1 pages. Again referring to Figure 8, if primary memory had a capacity of two pages, then at reference instant 7 the member pages would be "b" and "a". For a larger capacity primary memory, pages "b" and "a" will always be included. By noting the distance of a referenced page \( r_t \) from the top of the previous stack \( P_{t-1} \) (a measure known as stack distance) it is possible to keep track of the memory size needed to contain the referenced page. If a page has never been previously referenced the convention is to denote the stack distance by a special character (*) in
Figure 8, and 0 in the algorithm) with the meaning that it would not be contained in a memory of any capacity.

To obtain the frequency of access to memories of various capacities from a sequence of stack distances, the frequency of occurrence of each stack distance \( x \) must be calculated. If \( n(x) \) is the number of times stack distance \( x \) occurred during the processing of a string of length \( T \), the stack distance density function \( f(x) \) is given by

\[
f(x) = \frac{n(x)}{T} \quad x = 1, 2, \ldots, N \quad 3-6
\]

The stack distance cumulative distribution function \( F(s) \), known as the "success function," gives the percentage of time that a referenced page \( r_t \) would be found in a memory of capacity \( s \) when managed by the LRU replacement rule. This is simply

\[
F(s) = \sum_{x=1}^{s} f(x) \quad s = 1, 2, \ldots, N \quad 3-7
\]

Figure 9 shows the stack distance density and distribution functions for the reference string presented in Figure 8. Note that if primary memory had a capacity of three pages, then the percent of time the referenced page is found in primary memory is 30%, the three successes occurring at reference instants 7, 8, and 10.

Given the sequence of reference intervals, it is possible to obtain in a similar fashion the missing page rate \( m(z) \), which measures the number of pages returning per unit of time to the working set, and the average working set
Reference
Instant 1 2 3 4 5 6 7 8 9 10
Stack Distance * * * * 4 4 1 3 4 3

\( f(x) \)

\[
\begin{array}{c|c}
\hline
x & f(x) \\
\hline
0 & 0.1 \\
1 & 0.2 \\
2 & 0.3 \\
3 & 0.4 \\
4 & 0.4 \\
\hline
\end{array}
\]

Stack distance density function.

\( F(s) \)

\[
\begin{array}{c|c}
\hline
s & F(s) \\
\hline
0 & 0.2 \\
1 & 0.4 \\
2 & 0.6 \\
3 & 0.8 \\
4 & 1.0 \\
\hline
\end{array}
\]

\( F(3) = 0.3 \)

Stack distance distribution function.

(Success Function)

Figure 9

Stack Distance Functions for the Page Trace of Figure 8
size \( s(\tau) \). Letting \( n(y) \) be the number of times that reference interval \( y \) occurred in processing a reference string of length \( T \), the reference density function \( f(y) \) is

\[
f(y) = \frac{n(y)}{T} \quad y = 1, 2, \ldots, T
\]

and the cumulative reference distribution \( F(z) \) is

\[
F(z) = \sum_{y=1}^{z} f(y) \quad z = 1, 2, \ldots, T.
\]

Since this gives the percent of pages referenced at intervals less than \( z \), the missing page rate \( m(z) \) is given by

\[
m(z) = 1 - F(z) \quad z = 1, 2, \ldots, T
\]

and the average working set size \( s(\tau) \) by

\[
s(\tau) = \sum_{z=0}^{\tau-1} m(z) \quad \tau = 1, 2, \ldots, T.
\]

In order to take into account initial conditions the inter-reference interval for pages not previously referenced must be interpreted as though they had been referenced at time \( t = 0 \). In practice, the reference string length \( T \) is many times larger than the number of times any one reference interval occurs, so that the effect of the initial conditions is negligible. This, of course, is not true for the simple example in Figure 8. Figure 10 shows the reference density and distribution functions for the example of Figure 8 while Figure 11 shows the corresponding missing reference rate and average working set size functions.

In order to demonstrate the calculation of the average
Reference Instant 1 2 3 4 5 6 7 8 9 10
*Reference Interval 1 2 3 4 4 4 1 4 6 3

\[ f(y) \]

0.4
0.3
0.2
0.1
0.1
0 1 2 3 4 5 6

Reference interval density function.

\[ F(z) \]

1.0
0.8
0.6
0.4
0.2
0
0 1 2 3 4 5 6

Reference interval distribution function.

*Initial conditions included.

Figure 10
Reference Interval Functions for the Page Trace of Figure 8
Reference Instant 1 2 3 4 5 6 7 8 9 10

*Reference Interval 1 2 3 4 4 4 1 4 6 3

\[ m(z) \]

0.0 0.2 0.4 0.6 0.8 1.0

0 1 2 3 4 5 6

Missing reference rate.

\[ s(\tau) \]

0.0 1.0 2.0 3.0 4.0

0 1 2 3 4 5 6

Average working set size.

*Initial conditions included.

Figure 11

Working Set Functions for the Page Trace of Figure 8
working set size for the reference string of Figure 8, using the equations 3-8 through 3-11, it is necessary to start with the reference interval distribution of Figure 10. The density \( f(3) \), for reference interval 3, equals \( 2/10 \) since \( n(3) = 2 \) and \( T = 10 \). The cumulative distribution \( F(3) \), for the same value is merely the summation of the densities \( f(1) \) through \( f(3) \), or \( 5/10 \) by equation 3-9. This indicates that 50% of the pages were referenced at intervals less than or equal to 3 time units. Now the missing page rate \( m(3) \) is \( 1/2 \) by equation 3-10, thus for a window width of 3, \( 1/2 \) of the references were to pages not belonging to the working set. Equation 3-11 for a window width of 3 merely sums \( m(0) \) through \( m(z) \) giving an average working set size of 2.5, as could be obtained from averaging the working set sizes for the ten reference instants of Figure 8.

It should be evident from the brief example presented that the stack must be searched at each reference instant, and that its length may eventually be equal to the number of elements in the set of references. Because of this, the processing of the reference string in the manner described can be expensive for even moderately large reference sets. In fact, with current page sizes of \( 10^2 \) or \( 10^3 \) words and memory sizes of \( 10^7 \) words, a stack of \( 10^4 \) or \( 10^5 \) items would have to be continually searched. As simple as the LRU stack algorithm is, it still is not feasible for memory systems when long stacks occur, as would be the case for a large
capacity device and a small page size. For this reason, and others to be detailed in the next two chapters, it is not desirable to handle data in only one size unit throughout an entire storage hierarchy. Rather, the size of the data units handled should vary with their physical location in the hierarchy. The logical structure necessary for this already exists in the form of the system name outlined in Chapter II.

**Referencing Slave Memory Hierarchies**

The performance of a slave memory system, or any other memory hierarchy, hinges on the data traffic between the various levels of the hierarchy, a subject to be discussed in the next chapter. By studying the mechanisms at work between the \( j, j+1, \) and \( j+2 \) levels of a larger \( m \)-level hierarchy, it is possible to gain insight into the operation of the \( m \)-level hierarchy. It is therefore assumed in the following analysis that the specific levels studied are only a subset of a much larger memory hierarchy.

In the memory system studied the central processor is considered to be generating all references in the form of a finite length system name, thus implying a finite capacity storage hierarchy of \( n \) pages.\(^1\) The particular system name used is such that at each reference instant \( r_t \) a word, page, book, and volume are implied by the name, thus producing four

---

\(^1\) From the standpoint of the logical organization presented in Chapter II the variable size name spaces may be considered to be fixed in size during the short time that the address traces are being made. This assumption is made only for analytical convenience.
related reference strings from a single central processor reference string. The average working set size, missing reference rate, and success function are all machine independent aspects of a reference string that relate directly to the software demands on a storage system. It is useful to examine these measures in light of the multiple reference structure produced by references to system names. Since a single underlying process, the reference of the central processor, is responsible for all four reference strings, the various measures derived from the strings are also related, though in a subtle way. The average working set size is chosen for analysis first since the missing reference rate is simply the first order difference of this measure.

In reference (43) it is shown that the working set size \( s(\tau) \) is bounded above and below. More specifically, for a finite set of \( n \) pages

\[
1 = s(1) \leq s(\tau) \leq s(\tau+1) \leq \min(n, \tau+1).
\]

Thus the average working set size must contain more than one page, but equal to or less than the total of \( n \) pages, or one plus the window width, whichever is smaller. The lower bound is achieved whenever the same page is continually referenced, while the upper bound is achieved whenever the page reference string is cyclic (i.e. \( 1, 2, 3, \ldots, n, 1, \ldots \)). The same reasoning can be applied to a set of \( v \) books to show that its working set size is also bounded.

In order to show the relationship between the working
set sizes for books and pages in a slave memory hierarchy, let the storage system contain a set \( B \) of \( v \) books and a set \( P \) of \( n \) pages so that the \( j^{th} \) book contains \( w = \lceil n/v \rceil \) pages. Denoting the working set of pages by \( W_p(t,T) \), and the working set of books by \( W_b(t,T) \) and their average sizes by \( p(t) \) and \( b(t) \) respectively it is evident that \( W_p(t,T) \geq W_b(t,T) \) since a book is referenced because a page in it has been referenced. The equality holds when exactly one page out of each book is referenced. Because the working set size is bounded, the largest difference between the size of the working set of pages and the size of the working set of books occurs when all \( w \) pages of a book are referenced. In this case \( W_p(t,T) = w \cdot W_b(t,T) \). Thus

\[
w \cdot W_b(t,T) \geq W_p(t,T) \geq W_b(t,T)
\]

To relate this to a page trace \( \rho = x_1, x_2, \ldots, x_i, \ldots, x_t \) where \( x_t \in P \), and the corresponding book trace \( \gamma = y_1, y_2, \ldots, y_j, \ldots, y_t \) where \( y_t \in B \), two binary random variables \( \alpha \) and \( \beta \) are defined as follows:

\[
\alpha_i(t,T) = \begin{cases} 1 & \text{if } x_i \in W_p(t,T) \\ 0 & \text{otherwise} \end{cases}, \quad i=0,1,\ldots,n-1 \quad 3-14
\]

\[
\beta_j(t,T) = \begin{cases} 1 & \text{if } y_j \in W_b(t,T) \\ 0 & \text{otherwise} \end{cases}, \quad j=0,1,\ldots,v-1 \quad 3-15
\]

Because the \( j^{th} \) book is referenced whenever the \( i^{th} \) page within it is referenced equations 3-14 and 3-15 are related. In particular if a page is in the working set the
corresponding book is in the working set. Specifically:

\[ w(j+1)-1 \]

\[ \bigcup_{i=wj}^{w(j+1)-1} \alpha_i(t, \tau) = 1 \rightarrow \beta_j(t, \tau) = 1 \]  \hspace{0.5cm} 3-16

The average size of the working set of pages is

\[ p(\tau) = \bar{w}_p(t, \tau) = \sum_{i=0}^{n-1} \frac{\alpha_i(t, \tau)}{n-1} \]  \hspace{0.5cm} 3-17

and the average size of the working set of books is

\[ b(\tau) = \bar{w}_b(t, \tau) = \sum_{j=0}^{v-1} \beta_j(t, \tau) \]  \hspace{0.5cm} 3-18

Since \( \alpha \) and \( \beta \) are related through equation 3-16 the average size of the working set of books may be determined from the page reference string statistics. More precisely

\[ b(\tau) = \sum_{j=0}^{v-1} \frac{w(j+1)-1}{w(j+1)-1} \alpha_i(t, \tau) \]  \hspace{0.5cm} 3-19

The evaluation of equation 3-19 will involve knowledge of the \( w \)th order moments (e.g. \( E[\alpha_1(t, \tau)\alpha_2(t, \tau)\ldots\alpha_w(t, \tau)] \)) of the page reference string. This requires assumptions about the statistical structure of the page reference string which will not be made here.

It is useful to estimate the software measures for each of the three possible modes of addressing since these represent extremes in program behavior. The random addressing
mode, mode 1, which is also known as the independent reference model (44) is chosen first. This mode provides references such that \( \Pr [x_t = i] = 1/n \) and \( \Pr [y_t = j] = 1/v \). For this model Denning and Schwartz (43) have shown that

\[
s(\tau) = n - \sum_{i=1}^{n} (1-\lambda_i)^\tau
\]

where \( s(\tau) \) is the average working set size, \( n \) is the number of elements in the reference set, and \( \lambda_i = \Pr [r_t = i] \).

Noting that \((1-1/n)^\tau \approx 1-\tau/n-\tau(\tau-1)/2n^2 \) for \( n \gg \tau \) it follows that

\[
p(\tau) \approx n \quad \text{for} \ \tau \gg n \quad 3-21
\]
\[
b(\tau) \approx v \quad \text{for} \ \tau \gg v \quad 3-22
\]

and

\[
p(\tau) \approx \tau - \tau(\tau-1)/2n \quad \text{for} \ \tau \ll n \quad 3-23
\]
\[
b(\tau) \approx \tau - \tau(\tau-1)/2v \quad \text{for} \ \tau \ll v \quad 3-24
\]

These equations indicate that if the window width \( \tau \) is much larger than the number of pages in the storage system, the working set will contain most of the pages in the system. However, if the window width is much smaller than the number of pages, as would be the case for large storage systems, then the working set size as given by 3-23 would be about equal to the window width \( \tau \).

The relationship of the success function for pages and books may be analyzed in much the same fashion. If the stack distance for the reference \( r_t \) is denoted as \( d(r_t) \) then two binary random variables \( \psi \) and \( \delta \) may be defined as:
\[
\psi_t(s) = \begin{cases} 
1 & \text{if } d(x_t) \leq s \\
0 & \text{otherwise}
\end{cases} \quad 3-25
\]

\[
\delta_t(s) = \begin{cases} 
1 & \text{if } d(y_t) \leq s \\
0 & \text{otherwise}
\end{cases} \quad 3-26
\]

The success functions for pages, \(F_p(s)\), and for books, \(F_b(s)\), are related to \(\psi\) and \(\delta\) by

\[
F_p(s) = \frac{\psi_t(s)}{s=1,2,...,n} \quad 3-27
\]

\[
F_b(s) = \frac{\delta_t(s)}{s=1,2,...,n} \quad 3-28
\]

where the expectation is with respect to time. Since a successful page reference implies a successful book reference the stack distance for the book cannot be greater than that of the page. Specifically,

\[
\psi_t(s) = 1 \quad \delta_t(s) = 1 \quad s=1,2,...,n \quad 3-28
\]

For a hierarchy with \(w\) pages per book it follows that if the stack distance for a page is greater than \(x\), then the stack distance for a corresponding book is strictly greater than \([x/w]\). More precisely

\[
\psi_t(x) = 0 \quad \delta_t([x/w]) = 0 \quad 3-29
\]

By using equations 3-27 through 3-29 it is easy to show the relationship between the success functions as:

\[
0 \leq F_p(x) \leq F_b(x) \leq \min \{w \cdot F_p(x), 1\} \quad x=1,2,...,\gamma \quad 3-30
\]

Equation 3-30 indicates that the chances of a successful book reference are at worst equal to those of a successful
page reference and at best equal to 1 or \( w \) times those of a page, whichever is smaller.

The success functions for pages and books with independent referencing are

\[
F_p(s) = \frac{s}{n} \quad 1 \leq s \leq n \tag{3-31}
\]

\[
F_b(s) = \frac{s}{v} \quad 1 \leq s \leq v \tag{3-32}
\]

For the sequential addressing mode, mode 2, the success function for pages is constant at \((z-1)/z\) where \(z\) is the number of words per page. Likewise, for books it is constant at \((zw-1)/zw\) where \(zw\) is the number of words per book. These relationships hold because in a full storage system every \(z\)th reference will push out one page regardless of the capacity and bring in a new one. The average working set size for pages and books in this mode is:

\[
p(t) = \left\lfloor \frac{t}{z} \right\rfloor + \frac{(t-1) \mod z}{z} \tag{3-33}
\]

\[
b(t) = \left\lfloor \frac{t}{zw} \right\rfloor + \frac{(t-1) \mod (zw)}{zw}
\]

For the case where the addressing mode is looping, mode 3, both the success function and the working set curves depend on the particular pattern of references that is repeated within the loop, as well as the length of the loop in reference intervals. For example, if the loop includes only two pages, then a buffer of a capacity of only two pages is needed in order to contain them. The working set size for a window width sufficient to contain both pages would also be two, but the working set size for books would
either be one or two, depending on whether or not both pages belong to the same book.

Experimental Results from the Global Reference Model

The Global Reference Model was initially run for each of the three different modes of addressing in order to verify the correctness of the stack management algorithm given on page 54, and the correctness of the analytical results obtained in the preceding section. The particular parameters chosen for the runs were for a slave memory hierarchy containing 256 pages in 16 books, and a window width of 40 for the working set statistics.

The results of the independent referencing runs gave average working set sizes of 37 for pages and 14.8 for books. These values compare with \( p(\tau) = 37 \) and \( b(\tau) = 16 \), that were calculated using equations 3-23 and 3-22 respectively. However the condition that \( \tau \gg v \) is not met for equation 3-22 so that its estimate of \( b(\tau) \) is poor.

A typical success function for pages is shown in Figure 12, that was the result of a run where all modes of addressing were possible. This curve exhibits a sharp break around 15 pages, indicating that the program is page avid for primary memory capacities less than 15 pages. This type of behavior is known to exist for real programs (34). Figure 13 shows the average working set size as a function of window width for both books and pages during the same run. For the particular window width of 40, the distribution of
Figure 12. A Typical Success Function for Pages using the Global Reference Model
Figure 13
Average Page and Book Working Set Sizes vs. Window Width for the Global Reference Model
Figure 14. Typical Working Set Distribution for Pages using the Global Reference Model
of the working set size for pages $W_p(t,40)$ is shown in Figure 14 and is quasi-normal, although no tests of normality were applied. Although not shown, the distribution of the working set of books for the same run was also quasi-normal. Normal distributions for working sets have theoretical justification elsewhere (43) and have been used by Coffman and Ryan (45) for studying memory utilization strategies.

The primary use of the model in this chapter was to discover software influences on a slave memory hierarchy, and to obtain qualitative results that could be used in the study of data movement through the hierarchy, a subject treated in the next chapter.

Figures of Merit

One of the aims of this chapter was to search for a measure that would allow the classification of a reference string as either "good" or "poor", with respect to the demands that a program or collection of programs might make on a storage hierarchy. Presently there is no way to tell whether a process will require a different page, book, etc., for each instruction executed, or whether the process will require only one page out of one book during its entire lifetime. There is certainly no way of discovering the behavior of programs that have never been executed, although compilers might be able to estimate the behavior of the section of code compiled by them. However, once a program's reference pattern has been evaluated there ought to be a
value, or "figure of merit" that suggests qualitatively the programs behavior independent of the computer used for its execution. Such a figure of merit could be used in comparing different programming techniques as to the economy of data movement, or in estimating the demands on a storage hierarchy.

The working set measure may be viewed as estimating dynamically the storage behavior of a program, thus suggesting a dynamic figure of merit in that a smaller working value implies the program makes less demands on storage than a larger one for a particular window width. Equation 3-12 establishes that the measure is bounded, with the extremes epitomizing good and poor behavior. However, unless a standard window width is chosen comparisons of working set sizes are difficult.

Another measure is suggested by the success function. If at one extreme all of a programs references are made to only one page, then the corresponding success function, \( F_p(s) \), is equal to unity everywhere. At the other extreme, if each program reference is made to a new page, then \( F_p(s) \) is zero everywhere. Thus a reasonable figure of merit for program behavior might be:

\[
FOM = \sum_{s=1}^{N} 1 - F_p(s) \quad 3-34
\]

Like the working set measure, a smaller value implies less demand on a storage system. This kind of figure of merit
has strong intuitive appeal since it is related to the number of page faults produced if memory is managed by the LRU replacement rule. For an N page hierarchy, \(0 \leq \text{FOM} \leq N\), the proposed figure of merit is bounded since \(0 \leq F_p(s) \leq 1\) for all \(s\). The chief disadvantage of the proposed figure of merit, as with all figures of merit, is that many peculiarities of individual cases are disguised by using a single value for an overall performance measure. Table 1 gives the figure of merit of equation 3-3\(^4\) for five different addressing patterns. Included for comparison are the average working set sizes for a window width of 40, and the average reference interval. In actual practice a more direct measure of program performance on a particular system would be to simply record the total number of page faults each time a program executes and maintain a running average that could be used for scheduling purposes.

Summary

It is apparent that the behavior of programs from the viewpoint of memory demand covers a wide range of demands. The possibility exists in a slave memory hierarchy system handling fixed blocks of information at each level that a program might reference one item in each block of the hierarchy during each instruction cycle. On the other hand a program may be able to execute in its entirety using only the information contained in one block. This variation of software behavior in turn will critically effect the performance of any storage system. How much the performance varies
Table 1
Comparison of Program Behavior Measures

<table>
<thead>
<tr>
<th>% State Occupancy</th>
<th>Time in Reference Mode</th>
<th>Page Figure of Merit (POM)</th>
<th>Average Page Working Set Size for $\tau = 40$</th>
<th>Average Reference Interval</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td>146</td>
<td>37</td>
<td></td>
<td>Independent</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>67</td>
<td>37</td>
<td></td>
<td>Looping*</td>
</tr>
<tr>
<td>-</td>
<td>100</td>
<td>16</td>
<td>3.4</td>
<td>221</td>
<td>Sequential</td>
</tr>
<tr>
<td>7</td>
<td>31</td>
<td>14.5</td>
<td>9.8</td>
<td>208</td>
<td>Mixed Mode</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>11.7</td>
<td>8.8</td>
<td>195</td>
<td>Mixed Mode</td>
</tr>
</tbody>
</table>

* Loop length of 76 over an independent sequence of 71 pages.
depends on the particular capacities and access times of the storage hierarchy in question. This is the subject of the next chapter.

Between the two extremes of programs lies the typical program, the one that systems must be designed around. To discover the typical program will require extensive measurements on present systems subjected to the kind of analysis presented in this chapter.
The use of memory hierarchies in computer architecture is motivated by a fundamental desire to achieve reasonable system performance at a reasonable cost. In order to achieve a specific objective a system designer has many alternatives available. It is possible, for example, to trade primary memory capacity for channel bandwidth because increased memory capacity allows larger portions of a program and its data to be resident, thus requiring fewer accesses to secondary memory. This kind of relationship between primary memory allocated, program behavior, and channel traffic (page fault rate) has been extensively studied in a two level hierarchy (17,32,33,34,35,40,46,47,48,49,50,51). Limited studies have been made of multi-level hierarchies (30,41,54,55) although some multi-level hierarchies are already in existence (22,23). A few of the concepts used in studying two level memory hierarchies are applicable to the study of multi-level memory hierarchies as will be pointed out, while other concepts need further research.

The intent of this chapter is not to detail a specific design, although only one type of memory hierarchy is studied, but is instead to present a methodology for exposing
the effects of program behavior on the utilization and control of system resources in a multi-level slave memory hierarchy. Although the software load is a major unknown in computer system design, it is the thesis of this chapter that it represents the primary factor in determining system performance.

**Fixed and Variable Block Size Hierarchies**

Many schemes are possible for hierarchial memory organizations. One type of slave memory organization that uses a fixed block size throughout the entire hierarchy is detailed by Mattson, et. al. (41) and is shown in Figure 15. This type of organization constrains data moving up the hierarchy (i.e. from level \(i\) to \(i+1\)) to pass through each level, but data moving down the hierarchy is unconstrained and may travel directly from any level \(i\) to level 1, the primary memory. The unit of information transferred is always a page, and pages are moved on demand from any level \(i\) to the 1st level until it is full, at which time the least recently used page is pushed up to the next higher level. This suggests the name "trickle through" for this hierarchy since pages seem to trickle up the hierarchy when they fall into disuse. Although this leads to convenient analytical determinations of the hierarchy performance for a given address trace, there are serious problems in practice. First, the overhead necessary to access the higher and slower levels of the hierarchy may not warrant frequent fetches for small
Figure 15

Data Paths for a "Trickle Through" Storage Hierarchy
units (pages) of information. Second, the handling of a fixed page size throughout the hierarchy requires long page lists for large capacity memories.

Another type of slave memory hierarchy, the type shown in Figure 1, handles different size blocks of information at each level. This approach has the following rules of operation.

1) If a unit of information is stored in memory \( M_i \), there is always a copy of it in each of the higher levels \( M_{i+1}, \ldots, M_n \). If a unit of information is modified in memory \( M_i \), it must be copied into memory \( M_{i+1} \) when the replacement algorithm operating at level \( i \) selects the modified unit for replacement.

2) Whenever a unit of information not in \( M_j \) is referenced, the request for it is sent to successively higher levels of storage until it is satisfied. The time to service a request therefore depends on the level at which the desired information resides.

3) If memory \( M_j \) is full, then a replacement algorithm operating at level \( i \) must select the unit of information to be replaced, first initiating the copying of the unit to memory \( M_{i+1} \) if it has been modified.

The action of successively staging and handling smaller units of information in faster access storage devices suggests the name "telescope" for this type of hierarchial memory.
organization because of the analogy of stages of the memory hierarchy with the stages of a telescope.

The "telescope hierarchy" is chosen for detailed study for the following reasons:

- The maximization of transport efficiency demands the handling of larger blocks of information as the average access time to a storage device increases.
- The desire to handle content directories of reasonable size at all levels of the hierarchy demands the use of larger block sizes as the storage capacity increases.
- An attractive logical organization, in particular the use of a variable size system name space as presented in Chapter II, is possible for the telescope hierarchy approach.
- The organization allows the staging of working sets of information at the various levels.

Replacement policies operating at each level must all observe the rule that a block is not eligible for replacement if any part of that block is active at a lower level, since the ability to copy back modified information into the next higher level must be kept. The rule demands that there is at least one more block staged at level i+1 than at level i, a relationship that must hold throughout the hierarchy.

For memory $M_i$ let $c_i$ be the capacity in words, $b_i$ the block size in words and $n_i$ the total number of blocks contained in
the memory so that
\[ c_i = n_i b_i. \]

Since the number of blocks in memory \( M_{i+1} \) must be at least one more than the number in memory \( M_i \) in order to guarantee at least one block eligible for replacement, the capacity of memory \( M_{i+1} \) must be
\[ c_{i+1} = (n_i + 1)b_{i+1}. \]

If \( R_{i+1} \) is the ratio of the block size in memory \( M_{i+1} \) to the block size in memory \( M_i \) (i.e. \( b_{i+1}/b_i \)) then the ratio of the storage capacities between any two adjacent levels for this type of hierarchy is
\[ \frac{c_{i+1}}{c_i} = R_{i+1} \left(1 + 1/n_i\right). \]

Equation 4-3 shows that the capacity at the \( i+1 \)st level must always be greater than at the \( i \)th level by approximately the ratio of block sizes handled by the two memories. For example, if primary memory \( M_1 \) has a capacity of 128K words, or 128 pages at 1024 words per page, and a blocksize or book of 8K (8 pages) is chosen for the next higher level memory \( M_2 \) then the capacity of \( M_2 \) must be at least 1025K words.

A Four Level Variable Block Size Memory Model

The particular implementation of the telescope slave memory configuration shown in Figure 16 is chosen for study to explore system performance as a function of certain software parameters and to expose the issues present in any \( n \)-level hierarchy of this type. In the hierarchy
Figure 16

A Four Level "Telescope" Slave Memory
configuration studied each level is connected to the next level by a single channel operating in a half duplex mode. The logical organization of the referencing system presented in Chapter II is assumed, thus precluding any I/O activity (i.e. card input, print output) at the lower levels since the organization treats all information as segments with print and input segments handled at the top level by a special controller. The off-line unit record activity is not explicitly included in the model since the major emphasis is on the on-line activity of the memory system. Unit record segments are the same as any other segments at the lower levels of the hierarchy and thus constitute part of the overall data traffic between the various levels of the hierarchy.

The first level, primary memory, is assumed to be a conventional random access magnetic core with a read/write cycle time of 2 microseconds, with the attached CPU generating references to data or instructions every $\frac{1}{4}$ microsecond (four way interleaving assumed). Primary storage, the only one that is "word addressable" by the CPU, is allocated for processes in page units, the size of the page unit being a model parameter. Particular memory allocation strategies used in the model are discussed later.

The second level is assumed to be Extended Core Storage (E.C.S.) that is not addressable by word, but only by page number, so that no programs may be executed from the second
level. Storage for processes is allocated in fixed size blocks, known as books, from which missing page requests originating from primary memory are satisfied. The transfer rate for pages between E.C.S. and primary memory is at primary memory speed, ½ microsecond per word, after an initial connect time of 10 microseconds to the E.C.S.

The third level in the memory hierarchy is assumed to be a large capacity random access disk system with an average access time of 30 milliseconds and a data transfer rate of 400,000 words per second. Fixed size blocks, known as volumes, are allocated and staged for each active process in the system. Missing book requests originating at the E.C.S. level are handled by the third level controller and transferred to E.C.S. from the disk on demand.

The fourth, and last level of the hierarchy, is assumed to be a large capacity archival storage (write once, read many) having a very slow write speed of 10,000 words per second and a read speed of 400,000 words per second thus requiring no buffering for data transfers to the third level disk. Because of the slow write speed buffering is provided in the fourth level controller for up to 8 volumes that are being written to the archival storage. An average access time for information read from this storage is assumed to be ½ second, part of which is the necessary connect time necessary to establish control. This is the final repository
of all information from which missing volume requests originating at the third level are satisfied.

Memory Management and Control

If a program is under execution in primary memory it is assumed to run for a certain length of time before referencing a page not contained in primary memory. Executing programs, hereafter known as processes, that are allocated more pages in primary memory will usually run longer before referencing a missing page than a similar process that has been allocated a smaller number of pages. The length of time between these "page faults" is defined as the execution burst time \( \alpha \), measured in the number of instructions executed between page faults. Real execution time between page faults of course depends on the number of instructions per second that can be executed by the CPU (2,000,000 in the model simulated). The relationship between the storage allocated to a process and the execution burst time has been studied elsewhere (47). This "lifetime function" (46) is influenced by factors such as programming style, page size, problem type, and replacement algorithm. However, for this study a representative lifetime function as shown in Figure 17 resembling the S.D.C. data of Fine, et. al. (47) is assumed to be the same for all the active processes in the system. The effect of the lifetime function on system performance is pointed out in the results section.

Primary memory \( M_1 \) contains \( S \) page frames in the model
Figure 17
Assumed Process Lifetime Function
and it is assumed that \( N : (1 \leq N \leq S) \) processes \( P_1, P_2, \ldots, P_N \)
are active, (i.e. the system is multiprogrammed at a level of \( N \)). In the initial model studied each process \( P_i \) is
allocated an equal number of pages \( s_i \) such that \( s_i = \lfloor S/N \rfloor \).
The pages allocated to each process \( P_i \) are assumed to be some
fraction of the total page requirements for the process.
Demand paging is assumed to be in effect so that whenever
process \( P_i \) references a missing page, a page \( s_i \) belonging to
the process \( P_i \) is selected to be replaced by the newly
referenced page in accordance with an established replace-
ment algorithm.\(^1\) Such a policy is said to be a "local"
paging policy, as opposed to a "global" policy in which the
replaced page may be selected from any \( s_j, j = 1, \ldots, N \). The
policy described thus constitutes a fixed partition policy
with local replacement in the sense of Coffman and Ryan (45).
Other schemes, such as a variable partition policy are
discussed later.

Process \( P_i \) operating with a local paging policy with
memory partition \( s_i \) will give rise to a sequence of execution
intervals independent of the other processes. Even though
the length of the execution burst \( \alpha_i \) for process \( P_i \) is
independent of that for process \( P_j \) (\( j \neq i \)) it may not be
independent for successive execution intervals of itself.
However, it is assumed that for a given degree of multi-

\(^1\) The replacement algorithm used is not explicitly
specified, its effect being incorporated in the lifetime
function of the process.
programming $N$, and a fixed replacement algorithm operating locally, successive execution intervals are independent and have the same negative exponential distribution. Thus:

$$\alpha_i(t) = \lambda(s_i) e^{-\lambda(s_i)t} \quad i = 1, 2, \ldots, N \quad 4-4$$

where $\lambda(s_i)$ is the mean execution time given by the lifetime function of Figure 17. The appropriateness of the exponential and independent assumptions have been discussed elsewhere (50,53,56).

Each time process $P_i$ incurs a page fault the process is suspended from execution, and the page request is transferred to the next level for processing. A queue of processes waiting for the CPU, the READY queue, is maintained in first in, first out order so that CPU control is transferred to process $P_j$ when $P_i$ incurs the page fault if the queue is not empty. If the queue is empty the CPU remains idle, thus intuitively it appears that for effective CPU utilization the page faults should be serviced and returned to the CPU queue at least as fast as the CPU is servicing the processes.

It may happen that the page selected for replacement in the faulting process $P_i$ has been modified during its residency in primary core and will have to be written to E.C.S. storage before the referenced page is read into primary core. This write probability is denoted by $p_{12}(w)$, the subscripts indicating a data transfer from level 1 to level 2. It is an example of a software variable that is not very often measured but has a definite effect on system
performance. The referenced page that initiated the page fault may not be present in E.C.S. thus initiating a request to the third level disk system for the book that contains the missing page. The probability of this book fault occurring depends on three factors.

- How much information has been staged for the faulting process at the second level.
- The particular characteristics of the reference patterns of all the active processes.
- The particular replacement algorithm used at the second level.

All of the above factors are characterized in the model by a success function similar to the ones presented in Chapter III.

If the missing book is located at the third level a book must be selected for replacement at the second level. A local replacement policy operating in a fixed partition is assumed so that each process is guaranteed a fixed number of books, to contain its working set of books. If the selected book has been modified during its residency in E.C.S., a write of the book to the third level disk is assumed to precede a read of the referenced book. This write probability is denoted by $p_{23}(w)$. The whole line of reasoning that has been applied to the first two levels may be applied to the higher levels, and is in fact done for the model studied.

In using a fixed partition rule for allocating memory
at all levels of the hierarchy, there is no way of preventing eventual collapse in system performance because of excessive data movement for all the active processes. The system is virtually running uncontrolled without feedback, because there is no measure of a process's demand on memory. The collapse in performance introduced by the overcommitment of memory is known as "thrashing", its causes and prevention detailed in an article by Denning (57). The prevention of thrashing lies in allowing a process to become active only if there is enough memory available to contain its working set, the window size for measuring the working set being chosen to achieve a desired page fault rate. Therefore, the basic aim of the working set policy is to schedule processes for execution only if the estimated storage resources are available. By preventing the overcommitment of a system resource the working set policy is actually assuming a degree of control over program and system behavior.

Simulation Description and Results

The desire to model a slave memory system that is possible to develop with current technology led to a decision to use discrete simulation rather than analytical techniques. Although the theory of cyclic queues (58) is applicable, because transactions are serviced in successive stages and then returned to the CPU, only certain service time distributions at the various stages are analytically tractable (40,56,59). A simulation provides the basic flexibility to
investigate arbitrary distributions but leads to making a host of other decisions that influence system performance as the model is detailed. It should be pointed out that the designers of real systems are faced with the same kinds of decisions, but on a much larger scale. Many of the seemingly insignificant design assumptions and decisions made may effect system performance so much that they overshadow the theoretical performance.

A complete listing of the GPSS/360 programs used, as well as a description of the simulation models used, appears in Appendix A. Briefly, the main characteristics are these:

• The basic unit of time is 10 microseconds.

• A 128K word primary memory is allocated to the active processes by one of two different primary memory management rules and all processes are assumed to have the lifetime function of Figure 17 in determining the time between page faults.

• A queue of processes ready to use the CPU is maintained in FIFO order so that while a page fault is being serviced for one or more processes another process may use the CPU.

• Each page fault has a finite probability of being satisfied at any level of the hierarchy, the servicing of the fault requiring staging of the associated blocks of information from that level forward to the primary memory.
Of major interest in a multiprogrammed computer system is the effect on the utilization of the CPU and channels as the degree of multiprogramming increases. Contention for the CPU, channels, and storage devices results in changing resource utilization as the number of active processes increases; thus the degree of multiprogramming is used as the major independent variable in the performance curves presented. In all cases the simulation was run with one through ten processes active until steady state conditions were in effect. Channel, CPU, page fault service time, and various queue statistics were obtained for the steady state conditions.

The first experiment was run to obtain a baseline condition from which the effect of changing certain model parameters could be studied. For this experiment a fixed partition rule was used for allocating storage at all levels of the hierarchy so that each of the N processes had available $\frac{1}{N}$th of the storage at each level. The resource utilization curves for the baseline configuration are shown in Figure 18. These indicate that between nine and ten processes active create enough paging traffic so that congestion of the channel between the third and fourth memory levels limits CPU utilization.

For the second experiment the probability of a volume in the third level being modified, $p_{34}(w)$ was changed from .05 to .10, therefore demanding a write to the fourth level
Figure 18
Baseline Configuration for Memory Managed with a Fixed Partition Rule
archival storage twice as often. The lifetime function of
Figure 17 together with the success functions for the first
two levels resulted in infrequent accesses to the archival
storage until seven or more processes are active, so the
effect of the changed write probability as seen in Figure 19
is most noticeable with ten active processes. At this point
most of the processes are waiting for a volume on the disk
to be written to archival storage because of the unavaila-
bility of a free buffer at the controller for archival
storage.

The third experiment lowered the probability of a page
request being satisfied at the second level memory by
changing the success function for that level between one and
ten percent. As seen in Figure 20 the CPU utilization with
more than six active processes is degraded due to a larger
percentage of the requests being referred to the higher
levels of the hierarchy than with the baseline configuration
and therefore taking longer to service.

The simulation model was then restructured in order to
implement a working set policy for allocating primary memory.
Working set sizes were assumed to be normally distributed,
a fact verified by the results of the model of program
behavior studied in Chapter III (see Figure 14) as well as
eversewhere (43,45). The program and model described in
Appendix A assumes that each process has a mean working set
size of twenty pages with a standard deviation of five pages
Figure 19

System Sensitivity to a Decreased Read/Write Ratio Between Levels 3 and 4
Figure 20
System Sensitivity to a Decrease in the Level 2 Success Function
and that each process is assigned a CPU time quantum of \( \frac{1}{4} \) second if enough primary memory is available to contain its working set. The time between page faults is still given by Figure 17 based on the number of active pages for the executing process. After expiration of the time quantum a process frees all of its pages allocated to primary memory, remeasures the working set size, and joins the queue of processes waiting for primary memory. This allows another processes working set of pages to be swapped into memory if core is available. The queue of processes waiting for primary memory is maintained in FIFO order and is searched on a first fit basis when scheduling a process to run. Information for all of the processes is allocated at the higher levels of the hierarchy using the previous fixed partition rules regardless of their eligibility for execution.

The result of the restructured simulation is shown in Figure 21, and points up a serious problem in the management and control of multilevel hierarchies. Performance collapsed, although at a slower rate, due to thrashing at the second and higher level memories. Although the page fault rate was held approximately constant by the working set policy, as evidenced by the constant utilization of channel 12, this was not true of the book and volume fault rates. In principle, the working set policy should be applied to the higher levels of memory as well as primary memory in order to prevent a collapse in performance. Such a scheme may
Figure 21
Resource Utilization for Primary Memory Managed by a Working Set Policy
introduce severe control problems in the scheduling of processes for the CPU because each level of the hierarchy would have to be interrogated to see if it contained a working set of information for the candidate processes. The actual implementation of a working set policy in a two-level hierarchy is difficult enough due to a lack of information on what constitutes reasonable time and space quantums to be allocated to a process, let alone what would be reasonable quantums in a multilevel hierarchy. Nevertheless similar chemes are needed to guarantee controlled and equitable allocation of resources in a multiprogramming environment. It is an area that needs much research.

Page Service Time Analysis

It is apparent, as previously mentioned, that for effective CPU utilization the page faults should be serviced and returned to the ready queue of the CPU as fast as the CPU is servicing the processes on the queue. As long as there is relatively little waiting for any of the memory hierarchy resources an expected value analysis may be used to approximate the average page fault service time. The total average time for a page request to be satisfied, \( E_{T[I/O]} \), is the weighted sum of the average request completion times at the various memory levels. More precisely:

\[
E_T[I/O] = \sum_{j=2}^{4} a_j E_j[I/O]
\]
where $E_j[I/0]$ is the average time for a block request to be satisfied at the $j^{th}$ memory, and $a_j$ is the probability of the block request accessing the $j^{th}$ memory. All of the page requests are referred to the second level memory so $a_2 = 1$. However, $a_3$ and $a_4$, the probabilities of the request being referred to the third and fourth levels, are dependent upon the success functions $F_2(s)$ and $F_3(s)$. In particular:

$$a_3 = 1 - F_2(s)$$
$$a_4 = a_3 \cdot [1 - F_3(s)]$$

If $E[x_{ij}(t)]$ is the average time for a request to move data from the $i^{th}$ level to the $j^{th}$ level memory, and $p_{ij}(w)$ is the probability of the request first having to write back information from the $i^{th}$ to the $j^{th}$ level, then for the telescope hierarchy with equal read and write times the average request completion time for the $j^{th}$ level is

$$E_j[I/0] = E[x_{j-1,j}(t)] \cdot (1 + p_{j-1,j}(w)) \quad j=2,3,\ldots,N$$

The derivations of $E[x_{ij}(t)]$ are given in Appendix A and are equal to $E[x_{ji}(t)]$ for equal read/write speeds. Although the read and write speeds at the fourth level are not equal, the assumption is made of a buffer always being free for a lightly loaded system thus negating the read/write speed difference.

The total average time for a page request to be satisfied was calculated using equations 4-5 through 4-7 for the configuration of Figure 20 with five active processes. The
result of 5.46 milliseconds is about equal to the average CPU service time of 5.35 milliseconds that is obtained from the lifetime function of Figure 17 with \([128/5]\), or 25 pages allocated to each active process. The simulation results give an average CPU service time of 5.16 milliseconds and an average page fault service time of 7.76 milliseconds, the discrepancy in the \(E_{t}[I/0]\) time is due to queuing delays throughout the memory hierarchy. This crude analysis indicates that if the time required to locate a missing page in the hierarchy increases, as would be the case with more active processes, then further degradation of CPU utilization may be expected. Unfortunately the analysis of expected page fault service times does not indicate how rapidly resource utilization will deteriorate, but it does indicate that the success function represents a crucial parameter in determining system performance.

The extreme cases of software loading, those in which reference patterns are either sequential, independent or looping, were not run for the model since the results are fairly predictable. For processes looping through a set of pages contained entirely within E.C.S. the system needs only to service page faults using channel 12. This channel can sustain about 1200 page requests per second, enough to support 15 active processes with full CPU utilization assuming the lifetime function of Figure 17. For processes that make independent references over the first 3 levels of the
hierarchy, which contains about 210,000 pages, the page fault service time is about 54 milliseconds because 90% of the pages reside at the third and slowest level. Average CPU utilization would be about 10% with one process active, and even less for more processes active. For sequential referencing the lifetime function would be 0.5 milliseconds/page and the $E_T[I/O]$ would be about 6.5 milliseconds so that the system could support 13 pages/process or about 10 active processes with full CPU utilization.

The spectrum of system performance is very wide. With well behaved programs the system can support about fifteen processes at close to full CPU utilization, while for independent referencing it cannot even support one process at better than 10% utilization. The performance is totally dictated by the behavior of the software.

The effect of changes in the lifetime function appear as changes in the rate at which page faults occur, thus either enhancing or degrading resource utilization. The system modeled can obviously be balanced (e.g. full utilization of CPU and channels) with the correct software load. The procedure of adjusting the program mix and operating system software, often called "tuning a system" represents an attempt to balance the given hardware system via adjusting the software load.

Summary

The study of data movement in a multilevel memory
hierarchy has many dimensions, only a few being singled out in this chapter. The reference patterns generated by the central processor certainly influence system performance and are in turn affected by many seemingly unrelated decisions. The decision of what programming languages to support will influence methods of algorithm development and coding, which by design (i.e. block structured languages) may restrict certain reference patterns while favoring others. The choice of an overall logical organization like the one described in Chapter II heavily influences reference patterns and read/write ratios by virtue of the methods used to share and organize different segments. In addition the multitude of software decisions made in implementing and coding an operating system result in their own unique reference patterns that may occur very frequently. The decision of what particular storage devices should be used and their technological implementation will influence software decisions such as block sizes and access strategies. These decisions in turn will affect the global reference patterns and page fault rates. What is needed in order to gain insight into the "art" of system design are extensive measurements of present day computer systems and an analysis of the effect of specific design decisions on system and user behavior. One type of system measurement is presented in the next chapter.
CHAPTER V

FILE TRAFFIC IN SEVERAL CURRENT SYSTEMS

A key concept in the organization of a slave memory hierarchy is the use of fixed size blocks for handling information at the various levels of the hierarchy. The use of fixed size blocks simplifies storage allocation, content management and the moving of information between the various hierarchy levels. These reasons are already acknowledged in present systems by the fact that current operating systems such as CDC's "SCOPE" and extensions to operating systems like IBM's "HASP" use fixed block sizes in handling certain file I/O.

Block size happens to be a parameter that directly affects system performance and is influenced by both technological factors and user behavior. As already mentioned in Chapter I, decisions concerning block sizes compare the cost in time of moving wanted and unwanted information together at a lower cost per word against only moving wanted information at a higher cost per word. Technological factors will affect the access time cost and transfer rate while user behavior will influence the ratio of wanted to unwanted information.

In order to understand the behavior of file traffic in
current systems that allow specification of block sizes by the user, a series of measurements were conducted on four IBM 360 systems used for the following applications:

1) IBM 360/75 - university instruction and research environment

2) IBM 360/50 - library circulation system and computer assisted instruction system

3) IBM 360/50 - business and scientific production environment

4) IBM 360/65 - information abstracting and processing service

Control of input and output in a multiprogramming system is briefly sketched to provide the background for the measurement technique, as well as the measurements themselves.

I/O Control in Multiprogrammed Systems

Present multiprogrammed general purpose computers need to handle a wide variety of input and output devices having a wide range of data transmission speeds. All of the devices may be thought of as constituting the second level of a two level hierarchy because data is transferred between the devices and primary memory in all cases.

One of the traditional reasons for handling all of the devices at the second level has been the desire to share much of the logic and control circuitry for device and channel control with that of the central processing circuitry for reasons of economy. In fact, for some of the smaller and
less expensive computers such sharing causes a cessation of all central processing activity during data transfer operations. In the larger general purpose systems the wish to increase the utilization of all the system resources has led to a separation of the logic and control functions of the channel from those of the central processor in order to allow true simultaneous operation of both the central processor and the I/O operations.

Competing processes in a multiprogrammed environment will often require access to the same device for the same or different data. Unless the processes are mutually cooperative it is imperative that all the requests for I/O be channeled through a common supervisory program, the program assuming the responsibility for the scheduling and resolution of all requests. Requests that cannot be handled because a device or channel is busy must be queued for the channel or device, while requests that are refused because of device or program malfunctions must be returned to the requesting process with proper notification as to the reasons for refusal. The control program therefore keeps track of the status of all known devices and the paths available for data flow to and from them. Most all multiprogrammed systems exhibit an input/output supervisory program in one form or another.

In the IBM operating system this program is known as the "I/O Supervisor", or simply IOS, and constitutes part of
the nucleus of the operating system. The program contains a table for each device attached to the system that is capable of doing I/O. These tables are called Unit Control Blocks, UCB's for short, and contain information concerning the status and type of device for all on-line units. All UCB's remain permanently in core and are maintained by IOS in order to keep track of the progress of an I/O request. For each incoming I/O request the supervisor associates a Request Queue Element, or RQE, which represents a unit of work for it to do. The RQE will exist until the I/O has been finished for the particular request and is linked via a pointer to the UCB for which the I/O is requested. If the unit is free, and a data path to it available, the I/O supervisor will service the request by starting the associated channel program. Channel programs are a feature of IBM/360 I/O that will be discussed in more detail in the next section. The RQE is said to be active once the I/O is started. Requests that cannot be immediately serviced are queued for future servicing in any one of several different queue disciplines.

The RQE's reside permanently in core and are either threaded on a free list or to a particular UCB. Because of this it is possible to discover at any instant in time what I/O is pending or in progress in the system.

Channel Programs and Device Characteristics

As previously mentioned there is a channel program
associated with each request for I/O. Channel programs consist of one or more channel commands which represent instructions to the channel and an associated I/O device. These channel commands are always 64 bits in length and have the following format:

```
Bits 0-7   Command Code  
Bits 8-31  Data Address  
Bits 32-36 Flags        
Bits 37-39 Zero Bits    
Bits 40-47 Ignored      
Bits 48-63 Byte Count
```

Command codes may be classified into the following five basic types:

1) Read forward
2) Read backward
3) Write forward
4) Control
5) Branch unconditional

The type of command code can be distinguished by the four low order bits of the command code regardless of the particular I/O device to which they are addressed. The execution of each command by the channel initiates a corresponding operation between primary memory and a particular I/O device. If the command is "chained" to the next one as noted by a command chain flag bit in the flag field then the next sequential instruction is executed by the program unless the
instruction is a branch or control type. Certain of the
control commands may receive a status indication from a
device that will allow skipping of the next sequential
instruction and fetching of the one beyond it. By using an
unconditional branch, and a control command, looping in the
channel program is possible until a device condition is
satisfied. Thus a series of I/O operations may be fashioned
together in a program for a particular device, and executed
independently of the central processor unless the same
control logic is shared. The amount of data transmitted by
each command is termed a physical block unless the command
is chained with a data chain flag to succeeding commands in
which case the total amount of data so chained is termed a
physical block.

In order to collect data on the physical blocks
transmitted between primary memory and the secondary memory
devices one of two methods may be used. In the software
method, the one actually used, the channel program for each
request is "traced" or followed and the byte count field is
extracted from each channel command in the program. The
hardware method consists of attaching electronic instruments
to each channel and is the more accurate method if the means
for its use are available. However, the hardware method
cannot discover which program initiated the request. Each
method has its own advantage, so that the two should actually
be used together. The particular systems tested had two or
three types of devices supporting file I/O. They were fixed head drums, movable head disks, and magnetic tape. For these devices many of the channel programs are precompiled and the various addresses and byte counts are filled in by the access method interface programs between the user and IOS. The two random access devices represent the fastest access secondary storage. The fixed head drums had an average access time of 8.6 milliseconds while the movable head disk system had an average access time of about 60 milliseconds. The only sequential access device was a magnetic tape system with an average access time of 20 milliseconds if the next record was in position. However, the magnetic tape could only transfer data at 1/10 the rate of the drum and 1/3 the rate of the disk. The magnetic tape could be considered to be a type of archival storage, although it was not always used that way.

Each of the random access storage devices had the capability of performing an associative search by proper channel programming. Therefore certain channel programs had a program "fork" or "branch" such that either one of two different programs may be executed depending on the outcome of the associative search. This capability makes program tracing difficult via software since the conditions affecting

---

1. Since the main emphasis was to discover the behavior of file traffic in the system, I/O requests to telecommunications and unit record devices (card readers and line printers) were ignored.
the branch are unknown unless the search is actually executed. The trace program used assumed the philosophy of tracing only one side of the fork. The bias introduced into the data by this kind of tracing is unknown although it should be negligible if there is an equal chance of executing either side of the fork. The flow chart and code responsible for the tracing of a channel program is given in Appendix B.

**Sampling Method and Data Collection**

The most desirable technique for collecting data in operating systems by software methods is either to modify or build into the operating system the necessary code for passing control to the data collection routines at the appropriate points of interest. This insures that each event is monitored and is absolutely mandatory for compiling certain types of statistics. It has been used successfully by Pinkerton (40) and Saltzer and Gintell (60) in studying the performance of virtual memory operating systems, and by Stanley and Hertel (61) in studying the IBM 360 operating system. The technique usually imposes the requirement that data collection routines be debugged in a supervisory mode which may require a stand alone machine during program development.

For some measurements it is possible to sample the "state" of the system according to a periodic or random schedule that is statistically independent of any natural pattern in the measured event. If this premise is met then
the frequency with which events are observed will be proportional to the true distribution of events had all of them been recorded. This was the technique selected for use in collecting data on file traffic.

The maximum number of RQE's kept by the supervisor is a parameter selected at the time an operating system is assembled for a particular installation. If, during the operation of the computer, more requests for I/O occur than the number of available RQE's, then the I/O supervisor will maintain control by locking out all other processes until an active request is cleared, thus forcing an RQE free. Fortunately, all of the RQE's are stored contiguously in core so that they may be sequentially scanned to determine if they are "free" or "allocated", a condition signified by a flag byte in each RQE. Upon finding an allocated RQE, selected information concerning the associated I/O device and the channel program was saved.

Because the data collection program was operating as a user problem program it was possible that control could be taken away from it during the tracing of a channel program with the possibility of the channel program under trace being destroyed. In this case all the data collected for the particular I/O request under scan was purged. When control was returned the scan of allocated requests was resumed with the next RQE. Besides the core used, which represented between .1 and .5% of the total available
depending on the installation, the CPU time required for each scan was estimated to be under 100 milliseconds depending on the number of resident RQE's.

Data Analysis

Some general observations about the data collected are in order before presenting the results of the measurements. The data collected was intended to reflect as nearly as possible normal user oriented file traffic for the environment in which it was collected. In order to accomplish this, an attempt was made to remove obvious biases by analyzing the raw data for inconsistencies. An example of an inconsistency occurs when the operating system purposely executes a channel program containing a direct access erase command with a byte count of 32,767 seeking the remaining space on a track. Efforts were also made to remove the effect of HASP'S spooling operations on the overall class statistics since it used fixed length records of 688 bytes. This was done by not allowing I/O requests to the dedicated HASP disk packs to be summarized in the final data. In the university environment the SPOOL traffic and traffic related to the JOB queue was removed even though it accounted for over 75% of the file traffic.

Frequency plots, or histograms, were constructed for each device as well as for each class of devices for various combinations of channel commands. The program for analyzing the data and plotting the histograms included the capacity
to select certain commands or combinations of commands to aid in determining the detailed use of the devices.

Of the four installations surveyed, two were found to be heavy tape users. Both installations had about the same average byte count and the same percentage of read commands as is shown in Table 2, although the actual distributions were quite different as shown in Figures 22 and 23. The data shown in Figure 23 was from an installation that had standardized its tape operation to the use of variable blocked records with a maximum block size of either 3600 or 7200 bytes. The other installation using tape had no particular standards thereby leaving the choice of block size completely to the user as evidenced by Figure 22.

All four installations used the same model movable head disk system, an IBM 2314, for the on-line storage of both user and system data. Only the first installation had both a movable head disk system and a fixed head drum. The format of the physical records on the disk and drum consisted of a fixed length count field and two variable length fields, the key and data field. Channel commands used for reading and writing the different fields are presented in Table 3 along with a summary of the disk activity for all four installations.

A distribution of the byte counts for all read and write operations on the IBM 2314 disk is shown in Figure 24 for the fourth installation. The figure indicates a heavy
Table 2

Tape Block Size Summary
For Sites 3 and 4

<table>
<thead>
<tr>
<th></th>
<th>Total Commands Sampled</th>
<th>Average Byte Count</th>
<th>Total Read Commands</th>
<th>Average Byte Count</th>
<th>Percent Read Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>762</td>
<td>3480</td>
<td>406</td>
<td>4195</td>
<td>53.2</td>
</tr>
<tr>
<td>Site</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. 4</td>
<td></td>
<td>3915</td>
<td>679</td>
<td>4082</td>
<td>46.6</td>
</tr>
</tbody>
</table>
Table 3

Direct Access Block Size Summary

Disk System Summary

<table>
<thead>
<tr>
<th>Site No.</th>
<th>Sample Total</th>
<th>Average Size</th>
<th>Read Total</th>
<th>Average Size</th>
<th>Percent Read</th>
<th>Data Total</th>
<th>Average Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>416*</td>
<td>442</td>
<td>269</td>
<td>491</td>
<td>64.5</td>
<td>322</td>
<td>569</td>
</tr>
<tr>
<td>2</td>
<td>917</td>
<td>985</td>
<td>819</td>
<td>1005</td>
<td>89.3</td>
<td>749</td>
<td>1205</td>
</tr>
<tr>
<td>3</td>
<td>1463</td>
<td>380</td>
<td>1287</td>
<td>368</td>
<td>87.9</td>
<td>811</td>
<td>641</td>
</tr>
<tr>
<td>4</td>
<td>489</td>
<td>633</td>
<td>389</td>
<td>457</td>
<td>79.6</td>
<td>376</td>
<td>647</td>
</tr>
</tbody>
</table>

Drum System Summary

<table>
<thead>
<tr>
<th>Site No.</th>
<th>Sample Total</th>
<th>Average Size</th>
<th>Read Total</th>
<th>Average Size</th>
<th>Percent Read</th>
<th>Data Total</th>
<th>Average Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>319</td>
<td>849</td>
<td>319</td>
<td>849</td>
<td>100.</td>
<td>164</td>
<td>1645</td>
</tr>
</tbody>
</table>

* Excluding HASP units and SYSJOBQ unit.
Table 3
Direct Access Block Size Summary
- Continued -

Read and Write Commands

<table>
<thead>
<tr>
<th>Command Name</th>
<th>Binary Operation Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M/T</td>
</tr>
<tr>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>Home Address</td>
<td>X 0</td>
</tr>
<tr>
<td>Count</td>
<td>X 0</td>
</tr>
<tr>
<td>Record R-0</td>
<td>X 0</td>
</tr>
<tr>
<td>Data</td>
<td>X 0</td>
</tr>
<tr>
<td>Key &amp; Data</td>
<td>X 0</td>
</tr>
<tr>
<td>Count, Key &amp; Data</td>
<td>X 0</td>
</tr>
<tr>
<td>IPL</td>
<td>0</td>
</tr>
</tbody>
</table>

Write            |        |      |       |       |     |      |      |       |
<p>| Home Address     | 0 0    | 0    | 0     | 1     | 1   | 0    | 0    | 1     |
| Record R-0       | 0 0    | 0    | 0     | 1     | 1   | 0    | 0    | 1     |
| Count, Key &amp; Data| 0 0   | 0    | 0     | 1     | 1   | 1    | 0    | 1     |
| Spec. Count/Key/Data | 0 0 | 0    | 0     | 0     | 0   | 0    | 0    | 1     |
| Data             | 0      | 0    | 0     | 0     | 0   | 1    | 0    | 1     |
| Key &amp; Data       | 0      | 0    | 0     | 1     | 1   | 0    | 0    | 1     |
| Erase            | 0      | 0    | 0     | 1     | 0   | 0    | 0    | 1     |</p>
<table>
<thead>
<tr>
<th>BYTF-COUNT</th>
<th>DISTRIBUTION OF BYTE COUNTS FOR UNIT SUM</th>
<th>UNIT TYPE IS TAPE</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 199</td>
<td>*****************************************</td>
<td>PERCENT</td>
<td>15</td>
</tr>
<tr>
<td>200 - 399</td>
<td>**</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>400 - 599</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>600 - 799</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>800 - 999</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1000 - 1199</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1200 - 1399</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1400 - 1599</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1600 - 1799</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1800 - 1999</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2000 - 2199</td>
<td>*****************************************</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>2200 - 2399</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2400 - 2599</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2600 - 2799</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2800 - 2999</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3000 - 3199</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3200 - 3399</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3400 - 3599</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3600 - 3799</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3800 - 3999</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4000 - 4199</td>
<td>*****************************************</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>4200 - 4399</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4400 - 4599</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4600 - 4799</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4800 - 4999</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5000 - 5199</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5200 - 5399</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5400 - 5599</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5600 - 5799</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5800 - 5999</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6000 - 6199</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6200 - 6399</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6400 - 6599</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6600 - 6799</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6800 - 6999</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>7000 - 7199</td>
<td>*****************************************</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>7200 - 7399</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>7400 - 7599</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>7600 - 7799</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>7800 - 7999</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>8000 - 32768</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 22. The Distribution of Tape Block Size for Site #3**
<table>
<thead>
<tr>
<th>BYTE-COUNT</th>
<th>DISTRIBUTION OF BYTE-COUNTS FOR UNIT SUM</th>
<th>UNIT TYPE IS TAPE</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>600</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>800</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1200</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 23.** The Distribution of Tape Block Size for Site #4
<table>
<thead>
<tr>
<th>BYTE-COUNT</th>
<th>DISTRIBUTION OF BYTE-COUNTS FOR UNIT SUM* UNIT TYPE IS DISK</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>128</td>
<td>22</td>
</tr>
<tr>
<td>200</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>400</td>
<td>3</td>
<td>0.5</td>
</tr>
<tr>
<td>600</td>
<td>7</td>
<td>0.5</td>
</tr>
<tr>
<td>800</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>0.5</td>
</tr>
</tbody>
</table>

NO. OF SAMPLES = 487

AVG. BYTE COUNT = 673

Figure 24. The Distribution of Disk Block Size for Site #4
concentration of small byte count commands and was typical of all the installations surveyed. An explanation for this effect is found in the fact that most channel programs include a command for reading the 8 byte count field for record length and address verification. The read count operation is essentially a control function that could be incorporated with other control data in a revised channel architecture and was possibly masking other types of small byte count commands. It was therefore decided to analyze only those operations involving the data field by only selecting for analysis channel commands with the "data" bit on (see Table 3). This data is summarized by the cumulative distributions presented in Figure 25 for all four installations. The figure indicates that more than ¾ of the direct access data commands involve byte counts of less than 500 bytes. Even though efforts were made to eliminate the obvious system influence on file traffic as previously mentioned, Figure 25 still indicates a surprising amount of file traffic whose block size is influenced by the system. Specifically, the influence of the various control blocks listed in the bottom of the figure and described in reference 62 contribute heavily to the small block sizes. Most of these control blocks are used by the system for direct access data management.

The distribution of byte counts for the single installation using a fixed head drum is shown in Figure 26. As with
Figure 25
Cumulative Block Size Distribution for Only Disk Data Commands
Figure 26. The Distribution of Drum Block Size for Site #1
the disk system, the occurrence of the count field commands with their small block size accounts for a large percentage of the channel commands. The drums were primarily used as the main program library for the system at the time the data was collected thus accounting for its use as a read only storage device. The large average byte count for this device might be contributed to the fact that the tract capacity is three times that of the 2314 disk. Note the heavy concentration of 260 byte load module control records and 256 byte data set directory blocks for this device.

Summary

Of the three different classes of storage devices for which data was collected the two direct access devices were evidently being used as an integral extension of the operating system, or as a subsystem as in the case of the library circulation system. It is clear that the majority of the file traffic on the direct access devices is influenced by and under control of the operating system conventions for data management. Therefore the primary conclusion of this chapter is that, for systems that allow user control of block size; the major influence of file traffic and block sizes for direct access devices lies in established system conventions and software. The situation with respect to the magnetic tape devices is entirely different. Most of the file traffic between primary memory and tape is completely specified and under control of the user. However, the user
may be influenced by local installation standards or customs when using these devices. Because of the above conclusions it seems reasonable that the specification of block sizes could and should be taken over by the system.

The capability of a system to adapt to both technological and behavioral patterns without fundamental changes in the logical organization is very important in both the automatic and preplanned approach. This adaptation should be a system responsibility for those cases where device and system complexity make it difficult to understand the strategy necessary for good resource utilization. However, the assuming of the responsibility by the system for controlling block size should not prevent knowledgeable users from setting their own system standards of block size and block content for unusual data processing environments, a fact not overlooked in the logical organization presented in Chapter II. The system name space provides to some extent the ability to adjust to both conditions. The size of a block may be adjusted in powers of two by the different partitioning of the variable length system name, while the reassignment of system names provides a degree of control over the contents of a block.
CHAPTER VI
CONCLUSIONS

In the majority of computer applications much of the programming and execution effort is still directed towards the efficient moving and storing of information. This study has concentrated on identifying the logical and physical system requirements needed to shift the primary responsibility for information storage and movement from the programmer to the computer system. It describes a computer organization in which information known to and needed by the computer resides in an open ended N-level slave memory hierarchy. Each level of the hierarchy is responsible for the management of its own contents although the information transfer between levels is the result of activity (either directly or indirectly) in the primary memory. Requests for information not contained in the primary memory are passed to the successively slower speed memories in the hierarchy for resolution, with a successful request causing the movement of the desired information into the primary memory. Information associated with the requested information is moved to the various levels of the hierarchy in anticipation of its future use.

The major conclusion of the study is that the necessary logical and physical requirements for this type of computer
organization can be met with current knowledge and technology. However, the system performance, and hence the economic feasibility of such a system depends primarily on global information usage patterns by a community of users, a subject that is not well understood.

Summary of Ideas

The major viewpoint pervading the entire study is one of information movement. That is, computational processes are primarily viewed as generating a sequence of requests for moving varying amounts of information. All requests for the movement of information refer to the information by name. Therefore a study was made of the way that names are used by people and computers in solving problems with computers. The result was the development of a logical organization for the management of three name spaces.

The first name space, an open ended global name space, was introduced in order to avoid the necessity for renaming information as the space of information known to a computer system expands. It affords the user the important psychological advantage of always referring to information by the same name regardless of the physical implementation of the system.

However, the small space of information ordinarily referenced during the lifetime of a single computational process, or computer job, favors the use of a second fixed length name space for the efficient addressing of information
during execution. By viewing the fixed length name space as a serially reusable resource, a modification to the Multics virtual memory scheme was made to allow the continuous reallocation of a fixed length name space to portions of the variable length global name space. This provides the capability of a user to reference all the information known to the system without renaming any of the information.

The third name space introduced is a variable length system name space that provides the key to the automatic management of the two previous name spaces and to the actual physical handling and storing of the information. This name space has the ability to physically relate logically unrelated information needed for the efficient storing and moving of information as well as the ability to relate the two previous logical name spaces during execution. Therefore, the capability of a user to reference information without regard to its location rests directly on this name space.

It is well known that the sequences of requests for information generated by executing processes are directly related to the demands for information movement in a storage system. To aid in studying the full range of these storage system demands a generalized Markov model was developed for generating process request sequences. The model was subsequently used with a modified stack algorithm to simultaneously collect data on two machine independent measures of the storage demand of a program. These measures were then
investigated experimentally and analytically with respect to the previously proposed system name space in order to determine the amount of information moved in a slave memory hierarchy for various request sequences. Upper and lower bounds on the amount of information moved between adjacent levels of a hierarchy are obtained from the model and an overall figure of merit for the entire storage demand of a process is suggested.

The measures of program behavior studied were then used in a simulation to determine the performance of one possible implementation of a slave memory hierarchy. The results indicate that performance measured in terms of resource utilization for a slave memory hierarchy depends to a large degree on program behavior, a conclusion also supported by an analysis of average I/O service times throughout the hierarchy. An existing method for controlling the storage demands of programs in a two level hierarchy was investigated for applicability to the four level model used in the simulation. The results indicate a need for research into the whole area of process control throughout a distributed computer system.

An investigation into information movement between primary and secondary storage on several existing computer systems revealed a surprising uniformity of information movement patterns and sizes irrespective of the system application. This uniformity is attributable to the influence of
the operating system on the amount and kinds of information moved since all systems measured were using the same operating system.

Technological Assessment

Technological forecasting is risky business at its best, nevertheless a brief look is taken at the prospects of implementing the proposed slave memory organization. Clearly a system of the type presented in the fourth chapter could be developed with current technology. The question that administrators and manufacturers of computer systems always face is one of performance. The particular question of interest is whether the overall cost in human and hardware resources justifies a certain machine organization. It is possible, by enforcing suitable restrictions on programming practices and by careful attention to the software implementation of the operating system, to achieve almost full utilization of the hardware available. However, if the efforts needed to achieve full utilization cost more in terms of time and money than that of an alternative machine organization then the design is not acceptable. The intriguing part of designing any system is to leave the organization flexible enough to capitalize on technological improvements and user behavior. As long as storage technology follows the rule that increases in storage capacity occur at a faster rate than increases in access speed, memory hierarchies will be around in one form or another.
Directions for Further Research

One of the more important research areas concerning the design and organization of memories is the study of reference patterns by concurrently executing processes. Although the behavior of single programs is beginning to be generally understood there is a paucity of data on the behavior of collections of programs executing in a multiprogramming or multiprocessing environment. Reference patterns should be studied not only for addressable storage but also for secondary storage since data on secondary storage reference patterns is virtually nonexistent.

If a manufacturer sets out to design a specific memory hierarchy it is clear that considerable effort must be expended in obtaining data on program behavior in order to obtain some confidence in the expected performance of any design. The utility and generality of the proposed Markov model of global reference patterns that was presented in Chapter II should be investigated when such data becomes available.

In addition to studying the demands that programs make on storage systems some research should also be directed towards quantifying the demands that are made by programs for other services and resources. If these other demands of programs can be more precisely quantified then research in the scheduling and control of programs will be greatly enhanced. The quantification of program behavior, together
with the body of knowledge concerning feedback theory, might also be used to explore adaptive control schemes for maintaining a desired level of resource utilization.

Finally there is a whole area of research concerned with distributing the cost of a computer system equitably among a community of users. It seems clear, at least on the surface, that user behavior and hence resource utilization can be controlled to some degree by pricing policies. Whether or not the problem is a microcosm of the national economic problem is a matter for conjecture.
APPENDIX A

THE GPSS/360 SIMULATION MODEL

The General Purpose Simulation System is a discrete simulation program that is a program product of the IBM Corporation. The primary use of the program is in the flow analysis of manufacturing systems and job shop scheduling, although it is also useful for the modeling of computer systems at the level of detail used in Chapter IV. The disadvantages of its slow execution speed and its difficulty in learning were considered to be outweighed by the advantages of its generality and flexibility for gaining insight into the operation and performance of a slave memory hierarchy.

Hardware Parameters

The basic hardware of the memory system modeled is described in GPSS by the organization of the program statements which govern the flow of transactions through the model as well as by the use of various functions, variables and constants. The hardware parameters easiest to model and understand are those defined by constants. Among these are storage sizes and the number of available write buffers for
the photostore device. They are commented on and are self-explanatory in the attached listing.

The main hardware feature defined by the organization of the program statements is the half duplex channel that was used between the various levels of the slave memory hierarchy. This is accomplished in GPSS by organizing the program statements so that requests for information passing between adjacent levels of the memory hierarchy must use a single facility or channel. Another hardware characteristic that is defined by statement organization is the disk storage control system. The disk control system modeled has the capability for simultaneously positioning the movable read/write heads associated with each disk sub-unit. Therefore, within the coding that represented the disk storage system are eight sub-units that are assigned to servicing requests for books. Once the request for a book is assigned to a disk sub-unit the head movement needed for requested book is initiated. This head movement or "seek" may take place at the same time other requests for books are being serviced in the other sub-units.

Many of the hardware features and specifications were combined with software assumptions in the coding of the GPSS statements. One example is the inclusion of the uniform variate RN3 in the variable that determines disk read or write time. This incorporates the hardware assumption of a 6000 rpm disk rotation speed with the software assumption of
independent referencing to the disk storage. In deriving the
distribution of the read/write head seek time shown in
Figure 27 the same software assumption is combined with the
assumption of a maximum seek time of 55 milliseconds and a
minimum seek time of 5.5 milliseconds for any one of 200
possible head positions. The last example of combined
software and hardware assumptions in a single statements
occurs in modeling the instruction execution speed. The
hardware assumption of a basic clock cycle of \( \frac{1}{2} \) microsecond
for each instruction is combined with the software assumption
of each program executing for about 10,000 instructions
before incurring a page fault. These assumptions are
combined in the lifetime function of Figure 17 in Chapter IV
and are represented by the SPACE function in the GPSS
program.

Additional hardware characteristics are included by
adding an overhead time to each request to account for the
synchronization and control between storage devices.

**Software Parameters**

Since software parameters influence the flow of data
throughout the hierarchy these too are modeled by the
arrangement of GPSS statements and by the use of various
functions and constants. The major GPSS programming effort
was in modeling the two different software algorithms for
memory management and CPU scheduling. The fixed partition
rule was the simplest to implement and is briefly described
The Distribution of Direct Access Head Seek Time used in the GPSS Simulation Model
because a listing is not provided. Tasks were simply queued in FIFO order and ran with mean execution times based on the number of pages in a fixed partition and the accompanying lifetime function. Thus a task cycled through CPU execution and page fault servicing and was therefore either waiting for the CPU, executing, or waiting for a missing page.

The algorithm used in modeling the working set strategy for process control was much more complicated than the fixed partition strategy and therefore is the program listed in this appendix. To each task the number of pages in its working set was assigned by drawing a sample from a normal distribution. Tasks were then queued for CPU execution and assigned a time quantum of \( \frac{1}{4} \) second if enough core was available to contain their working set, the selection being done on a first fit basis. If not, the tasks were held in an idle queue until another task finished its time slice. At that time the first task whose working set would fit in core was selected for the execution queue. Thus a task could be in any one of four states; waiting for core, waiting for the CPU, executing, or waiting for a page.

Other software parameters that were modeled included the probability of a read only request at each level, the specification of page, book and volume sizes and the success function for each level of the hierarchy. All of these are summarized in Table 4 for the baseline configuration. The two different level 2 success functions used in studying
Table 4
Software Parameters for the Baseline Configuration of the GPSS Simulation

Page Size . . . . 1K words
Book Size . . . . 8K words
Volume Size . . . 128K words

Probability of a read only request.

<table>
<thead>
<tr>
<th></th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words</td>
<td>128K</td>
<td>8192K</td>
<td>200,000K</td>
<td>*</td>
</tr>
<tr>
<td>Pages</td>
<td>128</td>
<td>8192</td>
<td>200,000</td>
<td>*</td>
</tr>
<tr>
<td>Books</td>
<td>16</td>
<td>1024</td>
<td>24,320</td>
<td>*</td>
</tr>
<tr>
<td>Volumes</td>
<td>1</td>
<td>64</td>
<td>1,520</td>
<td>*</td>
</tr>
</tbody>
</table>

* Considered to be unlimited capacity since all requests were satisfied.
system sensitivity to changes in success functions are shown in Figure 28.

**Derivation of Expected Page Service Times**

This section is devoted to calculating the expected I/O service times referenced in Chapter IV based on the software and hardware parameters previously explained. Equation 4-5 indicates that the total average time to service an I/O request, $E_T[I/O]$, is the sum of the average time to service a request at each level weighted by the probability of accessing that level. As already mentioned in Chapter IV the probability of accessing the jth level (the $a_j$ in equation 4-5) is related to the success function for the particular level as given by equation 4-6 in Chapter IV. With five active processes, and the configuration parameters used in obtaining the results shown in Figure 20, the values of the level 2 and level 3 success functions are 0.95 and 0.96, respectively. Since all requests are serviced by the second level memory (E.C.S.) the access probabilities are

\[
\begin{align*}
  a_2 &= 1.0 \\
  a_3 &= 0.05 \\
  a_4 &= 0.002
\end{align*}
\]

for the three levels.

The average time to service a request by E.C.S. is denoted by $E[x_{21}(t)]$, the subscripts indicating a transaction with data flow from level 2 to level 1. The value is dependent on the number of words per page, the channel transfer
Figure 28. Level 2 Success Functions used in the GPSS Simulation Model.
rate, and the connect time. Thus

\[ E[x_{21}(t)] = t_{2c} + n_2 \cdot r_2 \]  \hspace{1cm} A-2

where

\[ t_{2c} = \text{connect time} = 10 \text{ microseconds} \]
\[ n_2 = \text{page size} = 1024 \text{ words} \]
\[ r_2 = \text{transfer rate} = 0.5 \text{ microseconds/word}. \]

Evaluation of equation A-2 yields a service time of 522 microseconds for each page transfer. If a page must be written back to E.C.S. the time is doubled since \( E[x_{21}(t)] \) is equal to \( E[x_{12}(t)] \). The probability of this happening is \( 1 - p_{21}(r) \), or 0.60 for the model, so the average request completion time for level 2, \( E_2[I/O] \) is given by equation 4-7 which evaluates to 835 microseconds.

For the third level disk system the average time to service a read request is given as:

\[ E[x_{32}(t)] = t_{3c} + n_3 \cdot r_3 + \frac{t_{rd}}{10} + \frac{t_{st}}{10} \]  \hspace{1cm} A-3

where

\[ t_{3c} = \text{connect time} = 100 \text{ microseconds} \]
\[ n_3 = \text{book size} = 8192 \text{ words} \]
\[ r_3 = \text{transfer rate} = 2.5 \text{ microseconds/word} \]
\[ t_{rd} = \text{average rotational delay} = 5 \text{ milliseconds} \]
\[ t_{st} = \text{average seek time} \]

The average seek time, \( t_{st} \), may be estimated from the density function used to model seek time. For independent referenc-
ing of any of the $N$ possible head positions the density function $p(d)$ of the head distance moved is

$$p(d) = \begin{cases} 
\frac{1}{N} & d = 0 \\
\frac{2}{N^2(N-d)} & 1 \leq d \leq N 
\end{cases} \quad A-4$$

where

$$N = \text{number of head positions} = 200$$
$$d = \text{distance head is moved}.$$

The cumulative distribution of this density is given by

$$F(x) = \sum_{d=0}^{x} p(d) \quad A-5$$

If the time to move the head $d$ positions is equal to $d$ times a constant for all $d$ then this would simply translate into the head seek time distribution shown in Figure 27 of this appendix. However, the function was modified as shown in Figure 27 in order to give more realistic movement times for short distance head moves. For the purpose of estimating average seek time the distribution of equation A-4 will be assumed. The average distance moved, $\bar{d}$, is computed from $p(d)$ as:

$$\bar{d} = \sum_{d=0}^{N} d \cdot p(d) \quad A-6$$

Substituting A-4 for $p(d)$ in A-6 yields

$$\bar{d} = \sum_{d=1}^{N} \frac{2d(n-d)}{N^2} \quad A-7$$

which reduces to

$$\bar{d} = \frac{(N-1)/3}{N}.$$

$A-8$
For $N = 200$ equation A-8 evaluates to 67. Therefore, assuming that it takes 55 milliseconds to move the maximum distance of 200 positions and that the time to move other distances are proportional to this then $t_{st} = (55/200)67 = 18.3$ milliseconds. When this value is used in equation A-3, along with the other values noted below A-3, the average time to service a read request, $E[x_{32}(t)]$, evaluates to 43.8 milliseconds. The average completion time for all requests to the third level takes into account the probability of a book first being written back to the disk. Thus $E_{3}[I/O]$ evaluates to 55 milliseconds.

A free buffer for the photostore is always assumed in this lightly loaded analysis so that the read and writes to the photostore may proceed at disk speed. As a result the average time to service a read request by the photostore is given by

$$E[x_{43}(t)] = \bar{t}_{4c} + n_{4}r_{4} + \bar{t}_{rd} + \bar{t}_{st}$$

A-9

where

$\bar{t}_{4c}$ = average connect time

$n_{4}$ = volume size = 131,072 words

$r_{4}$ = transfer rate = 2.5 microseconds/word

$\bar{t}_{rd}$ = rotational delay = 5 milliseconds

$\bar{t}_{st}$ = average seek time = 18.3 milliseconds

The connect time $t_{4c}$ was assumed to be uniformly distributed between 0.4 and 0.6 seconds so the average connect time, $\bar{t}_{4c}$, was 500 milliseconds. The average service time given by equation A-9 evaluates to 852 milliseconds using the above
values. Because 5 percent of the requests to the photostore first required writing a volume back to the photostore the average request service time for the fourth level evaluates to 940 milliseconds, or almost a full second.

The average page service time of the hierarchy may now be evaluated using equation 4-5 since the average service time at each level and the probability of access to each level is known. Thus:

\[
E_T[I/O] = 1(.835) + .05(55) + .002(940) \\
E_T[I/O] = 5.46 \text{ milliseconds}
\]

This is the value referenced in Chapter IV and compared to the actual page service time in the simulation. The MARK and TABULATE blocks were used in the program to time the total request service times. It is worthy to note the effect the access probability has on the total service time for the model. A slight change in the level 3 success function will contribute heavily to the total I/O time, indicating the importance of collecting information having the same usage into volumes and books.

The Program

This next section consists of the program listing for the configuration used in obtaining the results shown in Figure 21 on page 99 of Chapter IV.
SIMULATE

* 10 MICRO-SEC CLOCK.
* 1/2 MICRO-SECOND PER INSTRUCTION.
* PAGE SIZE CANNOT BE CHANGED UNLESS SPACE FUNCTIONS ARE ALSO
  CHANGED.
* WORKING SET STRATEGY.
* DEFINE MEMONICS FOR FACILITIES, QUEUES, ETC.

READY EQU 1, C
CPU EQU 1, F
CHA10 EQU 2, F, Q
CHA20 EQU 3, F, Q
CHA30 EQU 4, F, Q
IDLE EQU 5, Q
PHTW EQU 6, F, Q
TAB1 EQU 1, T

* DEFINE CORE STORAGE AVAILABLE.
* AND NUMBER OF PHOTO-STORE WRITE BUFFERS.

CORE STORAGE 128
BUFF STORAGE 8

* INITIALIZATION.

INITIAL X1, 131072 WORDS OF USER CORE.
INITIAL X2, 1 ACTIVE TASKS.
INITIAL X3, 1024 WORDS/PAGE.
INITIAL X4, 8 PAGES/BOOK.
INITIAL X5, 16 BOOKS/VOLUME.
INITIAL X6, 8388608 E.C.S. SIZE WORDS
INITIAL X7, 8 NO. OF 25MW0 PACKS.
INITIAL X12, 25000 1/4 SECOND MACRO-SLICE.

* WORKING SET BOUNDS FOR NORMAL DISTRIBUTION. MEAN & STD.
* CORE > X14 + OR - 3*X13 > 0

INITIAL X13, 5 STD. DEV. OF WORKING SET.
INITIAL X14, 20 AVG. WORKING SET SIZE.
INITIAL X15, 25000000 WORDS PER DISK PACK.

* HALF-WORD INITIALIZATION.

INITIAL XH1, 400 40 PER CENT READ ONLY (E.C.S.)
INITIAL XH2, 750 75 PER CENT READ ONLY (DISK)
INITIAL XH3, 950 95 PER CENT READ ONLY (PHOTOSTORE)

* COMPUTED BY VARIABLE AND SAVEVALUE.

INITIAL X8, 0 PAGES PER TASK.
INITIAL X9, 0 BOOKS/E.C.S.
INITIAL X10, 0 VOLUMES/DISK.
INITIAL X11, 0 PAGE READ WRITE TIME.
INITIAL X16, 0 VOLS. PER PACK.
**DEFINE TABLE FOR I/O TRANSIT TIMES DISTRIBUTION.**

**FUNCTION DEFINITIONS.**

<table>
<thead>
<tr>
<th>TABLE</th>
<th>MP6,0,1000,20</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>EXPONENTIAL FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN3,C24</td>
<td>EXPONENTIAL FUNCTION</td>
</tr>
<tr>
<td>0</td>
<td>0.1 1.04 2.22 3.35 5.5 9.69</td>
</tr>
<tr>
<td>0.6</td>
<td>0.915 1.2 1.75 1.38 1.8 1.83 2.65 2.12</td>
</tr>
<tr>
<td>0.9</td>
<td>2.3 2.52 2.94 2.81 2.95 2.99 2.96 3.2</td>
</tr>
<tr>
<td>0.98</td>
<td>3.9 4.6 4.95 5.3 9.98 6.2 9.99 7.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPACE FUNCTION</th>
<th>P9,C12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 3 5.3 5 18.0 7 40.0 10 95.0 15 247.</td>
</tr>
<tr>
<td>0.2</td>
<td>409.5 25 530 30 560. 35 590. 40 600. 600 600.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SEEK FUNCTION</th>
<th>RN4,C11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0.195 0.3 0.2 0.20 0.515 0.3 0.64 0.40 0.750 0.5</td>
</tr>
<tr>
<td>0.835</td>
<td>0.70 0.955 0.80 0.990 0.90 1.0 1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUCC1 FUNCTION</th>
<th>X9,C10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>500 25 750 50 840 75 910 100 950 150 980</td>
</tr>
<tr>
<td>200</td>
<td>985 250 990 500 1000 1100 1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUCC2 FUNCTION</th>
<th>X10,C12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>750 50 810 100 860 150 900 200 925 300 960</td>
</tr>
<tr>
<td>400</td>
<td>970 500 975 750 985 1000 990 2000 1000 5000 1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NORM FUNCTION</th>
<th>RN5,C25</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.0 0.006 2.5 0.023 2.0 0.05 1.64 1.0 1.28 0.15 1.04</td>
</tr>
<tr>
<td>0.20</td>
<td>0.84 0.25 0.67 0.30 0.52 0.35 0.39 0.40 0.25 0.45 0.13</td>
</tr>
<tr>
<td>0.50</td>
<td>0.0 0.55 -0.13 0.60 -0.25 0.65 -0.39 0.70 -0.52 0.75 -0.67</td>
</tr>
<tr>
<td>0.80</td>
<td>-0.04 0.85 -1.04 0.90 -1.28 0.95 -1.64 0.977 -2.0 0.994 -2.3</td>
</tr>
<tr>
<td>1.0</td>
<td>-3.0</td>
</tr>
</tbody>
</table>

**IT TIMES DISTRIBUTION.**

**EXPONENTIAL FUNCTION**

| .9  | .915 1.2 1.75 1.38 1.8 1.83 2.65 2.12 |
| .98 | 3.9 4.6 4.95 5.3 9.98 6.2 9.99 7.9 |

**MEAN EXECUTION TIME VS. PAGES**

| 0  | 3 5.3 5 18.0 7 40.0 10 95.0 15 247. |
| 20 | 409.5 25 530 30 560. 35 590. 40 600. 600 600. |

**SEEK DISTRIBUTION.**

| 0  | 0.195 0.3 0.2 0.20 0.515 0.3 0.64 0.40 0.750 0.5 |
| 0.835 | 0.70 0.955 0.80 0.990 0.90 1.0 1.0 |

**E.C.S. BOOKS/TASK**

| 0  | 500 25 750 50 840 75 910 100 950 150 980 |
| 200 | 985 250 990 500 1000 1100 1000 |

**DISK VOLS/TASK**

| 0  | 750 50 810 100 860 150 900 200 925 300 960 |
| 400 | 970 500 975 750 985 1000 990 2000 1000 5000 1000 |

**NORMAL DISTRIBUTION.**

| 0  | 3.0 0.006 2.5 0.023 2.0 0.05 1.64 1.0 1.28 0.15 1.04 |
| 0.20 | 0.84 0.25 0.67 0.30 0.52 0.35 0.39 0.40 0.25 0.45 0.13 |
| 0.50 | 0.0 0.55 -0.13 0.60 -0.25 0.65 -0.39 0.70 -0.52 0.75 -0.67 |
| 0.80 | -0.04 0.85 -1.04 0.90 -1.28 0.95 -1.64 0.977 -2.0 0.994 -2.3 |
| 1.0 | -3.0 |

**IT TIMES DISTRIBUTION.**

**EXPONENTIAL FUNCTION**

| .9  | .915 1.2 1.75 1.38 1.8 1.83 2.65 2.12 |
| .98 | 3.9 4.6 4.95 5.3 9.98 6.2 9.99 7.9 |

**MEAN EXECUTION TIME VS. PAGES**

| 0  | 3 5.3 5 18.0 7 40.0 10 95.0 15 247. |
| 20 | 409.5 25 530 30 560. 35 590. 40 600. 600 600. |

**SEEK DISTRIBUTION.**

| 0  | 0.195 0.3 0.2 0.20 0.515 0.3 0.64 0.40 0.750 0.5 |
| 0.835 | 0.70 0.955 0.80 0.990 0.90 1.0 1.0 |

**E.C.S. BOOKS/TASK**

| 0  | 500 25 750 50 840 75 910 100 950 150 980 |
| 200 | 985 250 990 500 1000 1100 1000 |

**DISK VOLS/TASK**

| 0  | 750 50 810 100 860 150 900 200 925 300 960 |
| 400 | 970 500 975 750 985 1000 990 2000 1000 5000 1000 |

**NORMAL DISTRIBUTION.**

| 0  | 3.0 0.006 2.5 0.023 2.0 0.05 1.64 1.0 1.28 0.15 1.04 |
| 0.20 | 0.84 0.25 0.67 0.30 0.52 0.35 0.39 0.40 0.25 0.45 0.13 |
| 0.50 | 0.0 0.55 -0.13 0.60 -0.25 0.65 -0.39 0.70 -0.52 0.75 -0.67 |
| 0.80 | -0.04 0.85 -1.04 0.90 -1.28 0.95 -1.64 0.977 -2.0 0.994 -2.3 |
| 1.0 | -3.0 |
* VARIABLE DEFINITION.

* TSKPG VARIABLE X1/(X2*X3) PAGES PER TASK.
* ECSBK VARIABLE X6/(X3*X4*X2) BOOKS/ECS/TASK
* VOLP VARIABLE X15/(X3*X4*X5) VOLUMES PER PACK.
* VOLPD VARIABLE (X7*X16)/X2 VOLS/TASK

* E.C.S. TRANSFER RATE. 1/2 MICROSEC. PER WORD (4 BYTES)
* PGRW VARIABLE X3/20+1

* DISK TRANSFER RATE. 4 MICRO-SECOND PER WORD (4 BYTES)
* BKRW VARIABLE (X3*X4*X5)/20+RN3+10

* SWAP VARIABLE (X3*P9)/20+1 SWAP IN TIME FROM ECS

* WORKING SET ASSIGNMENT RULES.
* WSET VARIABLE X14*X13*NORM

* PHOTO STORE WRITE TIME. 100 MICRO-SECONDS PER WORD.
* PHW VARIABLE X3*X4*X5*10

* PHOTO STORE READ TIME. READ AT DISK SPEED.
* PHR VARIABLE (X3*X4*X5*X5)/20+RN3

* PACK SELECTION VARIABLE.
* PACK VARIABLE (RN5*X7)/1000+10 PACK FACILITIES 10 ON UP
**MAIN PROGRAM**

*GENERATE 1,2,9F*  
*SAVEVALUE 8,V$TSKPG*  
*SAVEVALUE 9,V$ECSBK*  
*SAVEVALUE 10,V$VOLP*  
*SAVEVALUE 10,V$VOLPD*  
*SAVEVALUE 11,V$SPGRW*  

**INACT ASSIGN 7,X12**  
**ASSIGN 9,V$VHS**  
**QUEUE IDLE**  
**ENTER CORE,**  
**DEPART IDLE**  
**QUEUE CHA10**  
**SEIZE CHA10**  
**DEPART CHA10**  
**ADVANCE V$SWAP**  
**EXEC ASSIGN 8,FNSSPACE,1**  
**ASSIGN 8,P7**  
**QCPU LINK READY,FIFO,NEXT**  
**NEXT SEIZE CPU**  
**ADVANCE #8**  
**RELEASE CPU**  
**UNLINK READY,NEXT,**  
**ASSIGN 7,P8**  
**TEST E 0,P7,IOLV1**  
**LEAVE CORE,**  
**TRANSFER INACT**

**E.C.S. I/O LEVEL 1**

**IOLV1 MARK**  
**ASSIGN 1,2**  
**TRANSFER XH1,BKCK**  
**READP QUEUE CHA10**  
**SEIZE CHA10**  
**DEPART CHA10**  
**ADVANCE CHA10**  
**RELEASE CHA10**  
**TEST E #1,2,ACTIV**  
**BKCK TRANSFER FNS$SUCC1,IOLV2,PLOOP**  
**PLOOP LOOP 1,READP**  
**ACTIV TABULATE TAB1**  
**TRANSFER EXEC**
IOLV2 ASSIGN 2,2 ASSIGN BOOK CONTROL
ASSIGN 4,VSACK AND DISK PACK
TRANSFER *XH2,VOLCK BOOK WRITE NEEDED?
READB QUEUE *4 YES.. QUEUE FOR DISK PACK
SEIZE *4 OK.. TO USE IT
DEPART *4 LEAVE QUEUE
ADVANCE 5500,FN$SEEK DISK SEEK TIME
QUEUE CHA20 WAIT FOR CHANNEL TO BE FREE
SEIZE CHA20 GET CHANNEL IF FREE
DEPART CHA20 LEAVE CHANNEL QUEUE
ADVANCE V$BKRW DISK READ WRITE TIME
RELEASE CHA20 FREE CHANNEL
RELEASE *4 RELEASE DISK PACK
TEST E *4,VLOOP HAS BOOK READ FINISHED?
VOLCK TRANSFER *FN$SUCC2,IOLV3,BLOOP NO. VOL. MISSING?
BLOOP LOOP 2,READB NO. TRANSFER BOOK

PHOTO-STORE I/O LEVEL 3
SINGLE PORT ASSUMED
BUFFERING INCLUDED FOR WRITES

IOLV3 ASSIGN 3,2 ASSIGN VOLUME CONTROL
TRANSFER *XH3,VLOOP VOLUME WRITE NEEDED?
ENTER BUFF,1 YES.. BUFFER AVAILABLE?
READB ADVANCE 50000,10000 CONNECT TIME
QUEUE *4 QUEUE FOR A PACK
SEIZE *4 SEIZE IT IF AVAILABLE
DEPART *4 LEAVE QUEUE FOR PACK
ADVANCE 5500,FN$SEEK DISK SEEK TIME
QUEUE CHA30 YES.. GET CHANNEL IF FREE
SEIZE CHA30 LEAVE THE LINE
DEPART CHA30 LEAVE CHANNEL
ADVANCE V$PHR READ INTO BUFFER OR TO DISK
RELEASE CHA30 FREE CHANNEL
RELEASE *4 RELEASE PACK
TEST E *3,2,BLOOP WAS THIS A READ?
SPLIT 1,OPW NO.. SET UP WRITE
VLOOP LOOP 3,READV READ NEEDED
QPW QUEUE PHTW WRITE STATION FREE?
SEIZE PHTW YES
DEPART PHTW LEAVE LINE
ADVANCE V$PHW WRITE A CHIP
RELEASE PHTW RELEASE WRITE STATION
LEAVE BUFF,1 AND FREE BUFFER
TERMINATE DESTROY XACT

TIMER FOR RUN
GENERATE 10000
TERMINATE 1
APPENDIX B
THE CHANNEL TRACE PROGRAM

The complete program used to collect the blocksize data presented in Chapter V is not included in this appendix for two main reasons.

- Operating systems are continually undergoing change so that the location and meanings of addresses, flags, and status bits associated with the operating system are extremely time dependent.

- Assembler languages are continually evolving and are liable to change with associated hardware changes so that any algorithm coded in assembler is not easy to convert or use in other environments.

The section of the data collection program that is presented is included for the sake of clarifying the particular data shown in Chapter V by explaining the exact method used to collect it. It is assumed that the reader is familiar with the IBM System 360 Assembler Language, although a flow chart is presented in Figure 29 for the section of code presented so that it may be used in lieu of reading the listing of the code.

When the section of code presented in this appendix
Figure 29
Program Flow Chart of the Channel Trace Logic
receives control two registers are assumed to contain particular addresses. One register contains the address of the start of the channel program and is named CCWREG. Throughout the trace routine it is updated to point to the current channel command under consideration. The other register, TBLREG, is assumed to contain the address of an internal buffer where the desired command codes, flags, and byte counts are saved. This register is incremented and maintained during the course of the trace by a call to an internal subroutine EOTBLE at instruction 3750 in the listing. If a buffer is completely filled during the trace of a long channel program then control is not returned by EOTBLE and the particular channel program under trace is aborted. Other mnemonics peculiar to the section of code listed deal with the byte offsets to the various fields within a channel command and within the internal buffer.

It should be mentioned that not all channel programs can be traced with this type of routine. Certain channel programs are dynamically constructed during their execution depending on the outcome of the earlier part of the channel program. Therefore this kind of trace does not represent the actual execution sequence of some channel programs. The Indexed Sequential Access Methods are most noticeable in this regard. However, because many access methods used precompiled routines this method was found in practice to be fairly reliable.
The Program Listing

The next section contains a listing of the section of the program used to collect the data for Chapter V that is concerned with tracing channel programs.
LIST OF REFERENCES
(In order cited)

These abbreviations are used in the reference list:

CACM  --  Communications of the ACM
JACM  --  Journal of the ACM
SJCC  --  Spring Joint Computer Conference
FJCC  --  Fall Joint Computer Conference
IEEETC --  IEEE Transactions on Computers
AFIPS  --  American Federation of Information Processing Societies


37. Saltzer, J. H., Some Observations about Decentralization


