Virtual Memory

PETER J. DENNING

Princeton University, Princeton, New Jersey*

The need for automatic storage allocation arises from desires for program modularity, machine independence, and resource sharing. Virtual memory is an elegant way of achieving these objectives. In a virtual memory, the addresses a program may use to identify information are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. Two principal methods for implementing virtual memory, segmentation and paging, are compared and contrasted. Many contemporary implementations have experienced one or more of these problems: poor utilization of storage, thrashing, and high costs associated with loading information into memory. These and subsidiary problems are studied from a theoretic view, and are shown to be controllable by a proper combination of hardware and memory management policies.

Key words and phrases: virtual memory, one-level store, memory allocation, storage allocation, dynamic storage allocation, segmentation, paging, replacement algorithms, storage fragmentation, thrashing, working set

CR categories: 4.3, 6.2

INTRODUCTION

From the earliest days of electronic computing it has been recognized that, because fast-access storage is so expensive, computer memories of very large overall capacity must be organized hierarchically, comprising at least two levels, "main memory" and "auxiliary memory." A program's information (i.e. instruction code and data) can be referenced only when it resides in main memory; thus, information having immediate likelihood of being referenced must reside in main memory, and all other information in auxiliary memory. The *storage allocation problem* is that of determining, at each moment of time, how information shall be distributed among the levels of memory.

During the early years of computing, each programmer had to incorporate storage allocation procedures into his program whenever the totality of its information was expected to exceed the size of main memory. These procedures were relatively straightforward, amounting to dividing the program into a sequence of "segments" which would "overlay" (i.e. replace) one another in main memory. Since the programmer was intimately familiar with the details of both the machine and his algorithm, it was possible for him to devise efficient "overlay sequences" with relative ease.

The picture began to change markedly after the introduction of higher level programming languages in the mid-1950s. Programmers were encouraged to be more concerned with problem-solving and less concerned with machine details. As the complexity of their programs grew, so grew the magnitude of the storage overlay problem. Indeed, by the late 1950s it was clear that program operating efficiency could suffer greatly under poor overlay strategies,

* Department of Electrical Engineering. This work was supported in part by National Aeronautics and Space Administration Grant NGR-31-001-170.

CONTENTS

and it was generally agreed that storage allocation had become a problem of central importance. But, since programmers were shielded from machine details by programming languages, it was increasingly difficult to persuade them to expend the now relatively large effort required to devise good overlay sequences. This situation led to the appeal of computers having very large main memories [M5].

Two divergent schools of thought about solutions emerged. These have come to be known as the *static* (preplanned) and *dynamic* approaches to storage allocation. These two approaches differ on their assumptions about the most fundamental aspect of the storage allocation problem, *prediction*, both (1) of the availability of memory resources, and (2) of certain properties of a program's "reference string," i.e. its sequence of references to information.

The static approach assumes that (1) is either given or can be prespecified, and that (2) can be determined either by preprocessing the program and recording its reference string, or by examining the structure of its text during compilation [C5, K1, O1, R1, R4]. The dynamic approach assumes that (1) cannot (or ought not) be prespecified, and that (2) is determinable only by observing the program during execution; the memory space in use by a program should grow and shrink in accordance with the program's needs [S1]. Computer and programming systems during the 1960s have so evolved that, in a great many cases, neither memory availability nor program behavior are sufficiently predictable that the static approach can provide a reasonable solution. The reasons for this can be classed as *programming reasons* and *system reasons*.

To understand the programming reasons, it is useful to distinguish two concepts: *address space*, the set of identifiers that may be used by a program to reference information, and *memory space*, the set of physical main memory locations in which information items may be stored. In early computer systems the address and memory spaces were taken to be identical, but in many contemporary systems these spaces are dis-

tinguished. This distinction has been made to facilitate the eventual achievement of three objectives.

1. *Machine independence.* There is no a priori correspondence between address space and memory space.

The philosophy behind machine independence is: It relieves the programmer of the burden of resource management, allowing him to devote his efforts fully to the solution of his problem; it permits equipment changes in the computer system without forcing reprogramming; and it permits the same program to be run at different installations.

2. *Program modularity.* Programs may be constructed as collections of separately compilable modules which are not linked together to form a complete program until execution time.

The philosophy behind program modularity is: It enables independent compilation, testing, and documentation of the components of a program; it makes it easier for several programmers to work independently on parts of the same job; and it enables the modules constructed for one job to be used in another, i.e. building on the work of others [D4, D5, D10, D11, D12, D13, P2, R3, W4].

3. *List processing.* Languages (e.g. LISP) having capability for handling problems involving structured data are increasingly important.

As we suggested earlier, these three programming objectives invalidate reliable predictability, upon which static storage allocation is predicated. The mechanisms that implement machine independence cannot (by definition) establish a correspondence between addresses and locations until execution time, much too late for a programmer or a compiler to preplan memory use. Program modularity makes it impossible for the compiler of a module to know either what modules will constitute the remainder of a program or (even if it could know) what their resource requirements might be. List processing languages employ data structures whose sizes vary during execution and which, by their very nature, demand dynamic storage allocation.

The major system reasons compelling dynamic storage allocation result from certain objectives arising principally in multiprogramming and time-sharing systems: (1) the ability to load a program into a space of arbitrary size; (2) the ability to run a partially loaded program; (3) the ability to vary the amount of space in use by a given program; (4) the ability to "relocate" a program, i.e. to place it in any available part of memory or to move it around during execution; (5) the ability to begin running a program within certain deadlines; and (6) the ability to change system equipment without having to reprogram or recompile. Program texts prepared under the static approach require that the (rather inflexible) assumptions about memory availability, on which they are predicated, be satisfied before they can be run. Such texts are generally incompatible with these six objectives.

Even within the dynamic storage allocation camp there was disagreement. One group held that the programmer, being best informed about his own algorithm's operation, should be in complete control of storage allocation. He would exercise this control by calling on system routines which would "allocate" and "deallocate" memory regions on his behalf. This thinking is at least partially responsible for the block structure and stack implementation of the ALGOL programming language (1958) and subsequently the ALGOL-oriented Burroughs computers. It has also influenced the implementation of list-processing languages [B8, C4, K4].

The other group in the dynamic storage allocation camp advocated a very different approach: *automatic storage allocation.* Their thinking was influenced by their belief that complicated programs beget storage allocation problems so complicated that most programmers could not afford the time to manage memory well, and most particularly by their belief that multiprogramming would soon be a concept of great importance. Because the availability in main memory of particular parts of address

space may be unpredictable under multiprogramming, a programmer's ability to allocate and deallocate storage regions may be seriously impaired. Realizing that the principal source of difficulty was the small size of programmable main memory, this group advanced the concept of a *one-level store*. In 1961 a group at MIT [M5] proposed the construction of a computer having several million words of main memory (an amount then considered vast) so that the storage allocation problem would vanish. Economic reasons prevented this from actually being realized.

In 1961 the group at Manchester, England, published a proposal for a one-level store on the Atlas computer [F3, K3], a proposal that has had profound influence on computer system architecture. Their idea, known now as *virtual memory*, gives the programmer the *illusion* that he has a very large main memory at his disposal, even though the computer actually has a relatively small main memory. At the heart of their idea is the notion that "address" is a concept distinct from "physical location." It becomes the responsibility of the computer hardware and software automatically and propitiously to move information into main memory when and only when it is required for processing, and to arrange that program-generated addresses be directed to the memory locations that happen to contain the information addressed. The problem of storage allocation (for objects represented in virtual memory) thus vanishes completely from the programmer's purview and appears in that of the computer system. By basing memory use on system-observed *actual* use of space, rather than (poor) programmer estimates of space, virtual memory is potentially more efficient that preplanned memory allocation, for it is a form of adaptive system.

By the mid-1960s the ideas of virtual memory had gained widespread acceptance, and had been applied to the internal design of many large processors—IBM 360/85 and 195, CDC 7600, Burroughs B6500 and later series, and GE 645, to name a few. The fact of its acceptance testifies to its generality and elegance.

The foregoing discussion has summarized the ideas leading to the virtual memory concept. By distinguishing between addresses and locations, and automating storage allocation, virtual memory facilitates certain programming and system design objectives especially important in multiprogramming and time-sharing computers. The discussion in the remainder of this paper divides into two general areas: the *mechanisms* for effecting virtual memory, and the *policies* for using the mechanisms. The principal mechanisms are: *segmentation*, under which the address space is organized into variable size "segments" of contiguous addresses; and *paging*, under which the address space is organized into fixed size "pages" of contiguous addresses. We shall compare and contrast these two mechanisms and show why systems using some form of paging are predominant.

Although it has some very important advantages, virtual memory has not been without its problems. There are four of particular interest. (1) Many programmers, in their illusion that memory is unlimited, are unduly addicted to the old idea that time and space may be traded, in the sense that a program's running time may be reduced if there is more programmable memory space available. But space in a virtual memory may be an illusion; unnecessarily large and carelessly organized programs may generate excessive overhead in the automatic storage allocation mechanism, inevitably detracting from the efficiency of program operation. Nonetheless, as programmers and language designers gain experience with virtual memory, this problem should disappear. (2) Many paged systems suffer severe loss of usable storage—"fragmentation"—because storage requests must be rounded up to the nearest integral number of pages. (3) Many time-sharing systems using "pure demand paging" (a policy under which a page is loaded into main memory only after an attempted reference to it finds it missing) experience severe costs as a program's working pages are loaded singly on demand at the start of each time quantum of execution. (4) Many systems have shown extreme sensi-

tivity to "thrashing," a phenomenon of complete performance collapse that may occur under multiprogramming when memory is overcommitted. We shall demonstrate that these problems may be controlled if virtual memory mechanisms are governed by sound strategies.

The reader should note that these four observed inadequacies of many contemporary systems result not from ill-conceived mechanisms, but from ill-conceived policies. These difficulties have been so publicized that an unsuspecting newcomer may be led erroneously to the conclusion that virtual memory is folly. Quite the contrary; virtual memory is destined to occupy a place of importance in computing for many years to come.

BASIC SYSTEM HARDWARE

As our basic computer system, we take that shown in Figure 1. The memory system consists of two levels, main memory and auxiliary memory. One or more processors have direct access to main memory, but not to auxiliary memory; therefore information may be processed only when in main memory, and information not being processed may reside in auxiliary memory. From now on, the term "memory" specifically means "main memory."

There are two time parameters of interest here. The first, known as "memory reference time," is measured between the moments at which references to items in memory are initiated by a processor; it is composed of delays resulting from memory cycle time, from instruction execution time, from "interference" by other processors attempting to reference the same memory module simultaneously, and possibly also from switching processors among programs. We take the *average memory reference time* to be Δ. The second time parameter, known as "transport time," is the time required to complete a transaction that moves information between the two levels of memory; it consists of delays resulting from waiting in queues, from waiting for the requested information transfer to finish, and possibly

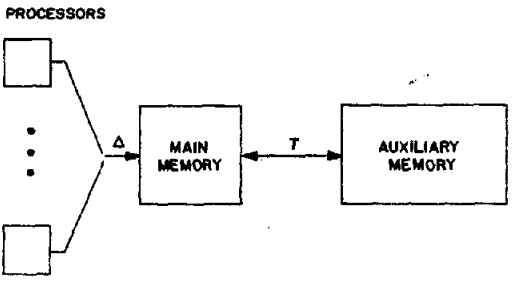


FIG. 1. Basic system hardware

also from waiting for rotating or movable devices to be positioned ("latency time"). We take the *average transport time* to be T. Since main memories are ordinarily electronically accessed and auxiliary memories mechanically accessed, Δ is typically 1 μsec and T is typically at least 10 msec. Thus speed ratios (T/Δ) in the order of 10^4 or more are not uncommon.

Main memory may be regarded as a linear array of "locations," each serving as a storage site for an information item. Each location is identified by a unique "memory address." If the memory contains m locations, the addresses are the integers $0, 1, \cdots, m-1$. If a is an address, the item stored in location a is called the "contents of a," and is denoted $c(a)$. Under program control, a processor generates a sequence of "references" to memory locations, each consisting of an address and a command to "fetch" from or "store" into the designated location.

DEFINITION OF VIRTUAL MEMORY

As mentioned earlier, virtual memory may be used to give the programmer the illusion that memory is much larger than in reality. To do this, it is necessary to allow the programmer to use a set of addresses different from that provided by the memory and to provide a mechanism for translating program-generated addresses into the correct memory location addresses. An address used by the programmer is called a "name" or a "virtual address," and the set of such names is called the *address space*, or *name space*. An address used by the memory is called a "location" or "memory address,"

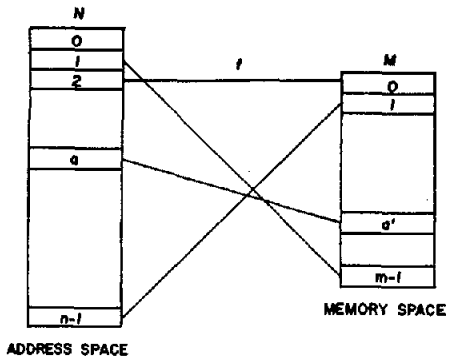


FIG. 2. Mapping from name to memory space

and the set of such locations is called the *memory space*. For future reference we denote the address space by $N = \{0, 1, \cdots, n-1\}$ and the memory space by $M = \{0, 1, \cdots, m-1\}$ and we assume $n > m$ unless we say otherwise.

Since the address space is regarded as a collection of *potentially* usable names for information items, there is no requirement that every virtual address "represent" or "contain" any information.

The price to be paid for there being no a priori correspondence between virtual addresses and memory locations is increased complexity in the addressing mechanism. We must incorporate a way of associating names with locations during execution. To this end we define, for each moment of time, a function $f: N \rightarrow M \cup \{\phi\}$ such that

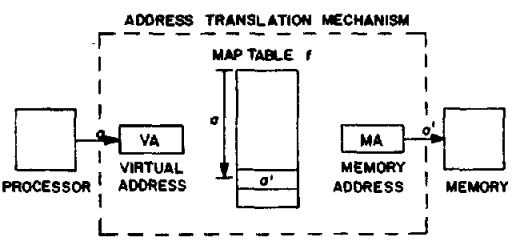
$$f(a) = \begin{cases} a' & \text{if item } a \text{ is in } M \text{ at location } a', \\ \phi & \text{if item } a \text{ is missing from } M. \end{cases}$$

This function f is known as the *address map*, or the *address-translation function*.

For reasons given earlier, it is to our advantage to make n much larger than m, but this is not necessary. Even if $n \leq m$, virtual storage could help with the *relocation problem* [D11], i.e. that of moving information around in memory.

Figure 2 gives an example of a mapping f, where a line (a, a') for a in N and a' in M indicates that item a is stored in location a', and the absence of a line indicates that item a is not present in M. Figure 3 shows how a hardware device implementing f could be interposed between the processor and the memory to handle the mapping automatically. Note that, by virtue of the mapping f, the programmer may be given the illusion that items consecutive in N are stored consecutively in M, even though the items may in fact be stored in arbitrary order. This property of address maps is known as "artificial contiguity" [R3].

The mapping device, when presented with name a, will generate $a' = f(a)$ if item a is present in M, and a *missing-item fault* otherwise. The fault will interrupt the processing of the program until the missing item can be secured from auxiliary memory and placed in M at some location a' (which takes one transport time); the address map f is then redefined so that $f(a) = a'$, and the reference may be completed. If M is full, some item will have to be removed to make way for the item entering, the particular item being chosen at the discretion of the *replacement rule* (if item b is entering and the replacement rule chooses the replace item a, where $a' = f(a)$, then the address map is redefined so that $f(b)$ becomes a' and $f(a)$ becomes ϕ). Contrasted with the replacement rule, which decides which items to remove, are the *fetch rule*, which decides when an item is to be loaded, and the *placement rule*, which decides where to place an item. If no action is taken to load an item into M until a fault for it occurs, the fetch rule is known as a *demand rule;* otherwise, if action is taken to load an item before it is referenced, the fetch rule is known as a *nondemand* or *anticipatory* rule.



OPERATION:

a loaded into VA
if ath entry of f blank, missing-item fault
a' loaded into MA

FIG. 3. Implementation of address map

Consider briefly the implementation of the address map f. The simplest implementation to visualize, called *direct mapping*, is a table containing n entries; the ath entry contains a' whenever $f(a) = a'$, and is blank (i.e. contains the symbol ϕ) otherwise. If, as would normally be the case, n is much greater than m, this table would contain a great many (i.e. $n - m$) blank entries. A much more efficient way to represent f is to create a table containing only the mapped addresses; the table contains exactly the pairs (a, a') for which $f(a) = a'$ and no pair (a, ϕ), and thus contains at most m entries. Such a table is more complicated to use; when presented with name a, we must search until we find (a, a') for some a', or until we have exhausted the table. Hardware associative memories are normally employed for storage of these mapping tables, thereby making the search operation quite efficient. (An associative, or "content-addressable," memory is a memory device which stores in each cell information of the form (k, e),where k is a "key" and e an "entry." The memory is accessed by presenting it with a key k; if some cell contains (k, e) for some e, the memory returns e, otherwise it signals "not found." The search of all the memory cells is done simultaneously so that access is rapid.)

MANUAL VERSUS AUTOMATIC MEMORY MANAGEMENT

The discussion in the Introduction reviewed the motivation for automatic storage allocation from a qualitative view. Before opening the discussion of methods for implementing and regulating virtual memory, we should like to motivate automatic storage allocation from a more quantitative view. The question before us is: How well does automatic storage allocation compete with manual?

Although the literature contains substantial amounts of experimental information about program behavior under automatic storage management [B3, B9, C3, F2, F3, F4, K5, O2, S2], authors have reached conflicting conclusions. Many of these experiments addressed the question "How do programs behave under given automatic storage allocation policies?" but not the question at hand, "How does automatic storage allocation compare with manual?" Experiments for the former question are clearly of a different nature than those for the latter. Therefore, attempts to make inferences about the latter from data gathered about the former are bound to result in conflicting conclusions. The following discussion is based on a paper by Sayre [S2], who has summarized and interpreted the work of Brawn and Gustavson [B9], for these appear to be the only published works addressing the latter question.

If the name space N is larger than the memory space M, it is necessary to "fold" N so that, when folded, N will "fit" into M. Let $g(b, t)$ denote the *inverse* of the address map f:

$$g(b, t) = \begin{cases} a & \text{if } f(a) = b \text{ at time } t, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The address space N is said to be *folded* if, for some b and $t_1 < t_2$, $g(b, t_1) \neq g(b, t_2)$. That is, there is some memory location which has been assigned to more than one address during the course of a program's execution. Between the instants t_1 and t_2, a sequence of commands, **move out** and **move in**, must have been issued, which caused $g(b, t_1)$ to be replaced by $g(b, t_2)$. The name space N is *manually folded* if the programmer has preplanned storage allocation, and has inserted the **move out** and **move in** commands into the program text where needed. The name space is *automatically folded* if the **move out** and **move in** commands are not in the program text, but instead are generated by the replacement and fetch rules, respectively, of the virtual memory mechanism. Note that manually folded text is intended to fit into some *specific* memory space of size s_0, whereas the automatically folded text may fit into *any* nonempty memory space.

The question before us now is: Can automatic folding compete with manual folding? It is reasonably clear that automatic folding should be competitive when the speed ratio

T/Δ between main and auxiliary memory is small; but is it competitive when T/Δ is large (say, 10^4 or greater)? Sayre reports affirmatively.

Brawn and Gustavson, Sayre tells us, considered a number of programs representing a wide range of possible behaviors, and the following experiment in a memory system with T/Δ in excess of 10^4. For a given program, let $T_a(s_0)$ denote the total running time (execution and transport time) when N is folded automatically into a memory of size s_0, when a demand fetch rule and a good replacement rule are in effect. Let $T_m(s_0)$ denote the total running time when N is folded manually for a memory of size s_0. For the programs considered,

$$0.8 \leq T_a(s_0)/T_m(s_0) \leq 1.7, \qquad E[T_a(s_0)/T_m(s_0)] = 1.21, \tag{i}$$

where $E[\ ]$ denotes expected value. In other words, automatic folding was (on the average) no more than 21 percent less efficient than manual folding.

Now, let $K_a(s_0)$ denote the number of transports issued while the program ran under the automatic folding conditions, and $K_m(s_0)$ denote the number of transports under the manual folding conditions. For the programs considered,

$$0.6 \leq K_a(s_0)/K_m(s_0) \leq 1.05, \qquad E[K_a(s_0)/K_m(s_0)] = 0.94.$$

Thus the automatic folder (i.e. the virtual memory) generally produced *fewer moves* than the manual folder (i.e. the programmer). A similar result was observed by the Atlas designers for a more restricted class of programs [K3]. The advantage of manual folding is that, unlike virtual memory with a demand fetch rule, processing may be overlapped with transports This suggests that anticipatory fetch rules might result in ratios $T_a(s_0)/T_m(s_0)$ consistently less than one [P1].

The experiments show also that the automatic folder is *robust*, i.e. it continues to give good performance for memory sizes well below the intended s_0. Specifically, $T_a(s)/T_m(s_0)$ was found essentially constant for a wide range of s, including s much less than s_0. In other words, a given program is compatible with many memory sizes under automatic folding, but only one under manual.

As we shall see in the section on Program Behavior and Memory Management, virtual memory management mechanisms perform most efficiently when programs exhibit good *locality*, i.e. they tend to concentrate their references in small regions of address space. We shall define a measure of locality, the *working set of information*, which will be the smallest set of virtual addresses that must be assigned to memory locations so that the program may operate efficiently. Sayre reports that the running time under automatic folding, $T_a(s_0)$, can be very sensitive to programmers' having paid attention to endowing the programs with small working sets, and relation (i) depends on this having been done. Should programmers not pay attention to this, very large $T_a(s_0)/T_m(s_0)$ can occur. Sayre reports that the costs of producing good manually folded text appear to exceed by 25 to 45 percent the costs for producing nonfolded text with good locality. Thus, one can tolerate as much as 25 percent inefficiency in the automatic folding mechanism before virtual memory begins to be less efficient than manual folding. Relations (i) indicates this generally is the case.

On the basis of the experimental evidence, therefore, we may conclude that the best automatic folding mechanisms compete very well (and may indeed outperform) the best manually folded texts. Virtual memory is thus empirically justifiable.

IMPLEMENTATION OF VIRTUAL MEMORY

The table implementation for the address mapping f described in the section on Definition of Virtual Memory is impractical, because it would require a second memory of size m to store the mapping table. In the following sections we shall examine three methods that result in a considerable reduction in the amount of mapping information that must be stored. Each method

groups information into *blocks*, a block being a set of contiguous addresses in address space. The entries in the mapping table will refer now to blocks, which are far less numerous than individual addresses in address space. The first method—segmentation—organizes *address space* into blocks ("segments") of arbitrary size. The second method—paging—organizes *memory space* into blocks ("pages") of fixed size. The third method combines both segmentation and paging.

Both segments and pages have names, which can be used to locate entries in the map tables. Segment names are usually (but not always) assigned by the programmer and are interpreted by the software, and page names are usually assigned by the system and interpreted by the hardware. Segmentation and paging, when combined, form an addressing system incorporating both levels of names. Otherwise, the only essential difference between the two schemes is paging's fixed block size.

Segmentation

Programmers normally require the ability to group their information into content-related or function-related blocks, and the ability to refer to these blocks by name. Modern computer systems have four objectives, each of which forces the system to provide the programmer with means of handling the named blocks of his address space:

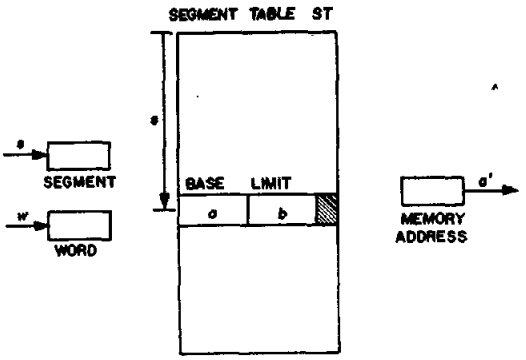
- *Program modularity.* Each program module constitutes a named block which is subject to recompilation and change at any time.
- *Varying data structures.* The size of certain data structures (e.g. stacks) may vary during use, and it may be necessary to assign each such structure to its own, variable size block.
- *Protection.* Program modules must be protected against unauthorized access.
- *Sharing.* Programmer A may wish to borrow module S from programmer B, even though S occupies addresses which A has already reserved for other purposes.

These four objectives, together with machine independence and list processing, are not peculiar to virtual memory systems. They were fought for in physical storage during the late 1950s [W5]. Dynamic storage allocation, linking and relocatable loaders [M3], relocation and base registers [D11], and now virtual memory, all result from the fight's having been won.

The *segmented address space* achieves these objectives. Address space is regarded as a collection of named *segments*, each being a linear array of addresses. In a segmented address space, the programmer references an information item by a *two-component* address (s, w), in which s is a segment name and w a word name within s. (For example, the address (3, 5) refers to the 5th word in in the 3rd segment.) We shall discuss shortly how the address map must be constructed to implement this.

By allocating each program module to its own segment, a module's name and internal addresses are unaffected by changes in other modules; thus the first two objectives may be satisfied. By associating with each segment certain *access privileges* (e.g. read, write, or instruction-fetch), protection may be enforced. By enabling the same segment to be known in different address spaces under different names, the fourth objective may be satisfied.

Figure 4 shows the essentials of an address translation mechanism that implements segmentation. The memory is a linear array of locations, and each segment is loaded in entirety into a contiguous region of memory. The address a at which segment s begins is its *base address*, and the number b of locations occupied by s is its *limit*, or *bound*. Each entry in the segment table is called a *descriptor;* the sth descriptor contains the base-limit information (a, b) for segment s if s is present in memory, and is blank otherwise. The steps performed in forming a location address a' from a name space address (s, w) are shown in Figure 4. Note that a missing-segment fault occurs if s it not present in memory, interrupting program execution until s is placed in memory; and an overflow fault occurs if w falls outside the allowable limit of s. Pro-



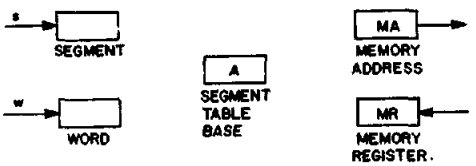
OPERATION:

(s, w) loaded into segment and word registers
if sth entry of ST blank, missing-segment fault
if $w > b$, overflow fault
$(a + w)$ loaded into MA

FIG. 4. Address translation for segmentation

tection bits (the darkened region in the table entry of Figure 4) can be checked against the type of access being attempted (i.e. read, write, or instruction-fetch) and a protection fault generated if a violation is detected.

The segment table can be stored in main memory instead of being a component of the address translation mechanism. Figure 5 shows the operation of the mapping mechanism when the segment table is in memory starting at location A. The segment table is itself a segment, known as the *descriptor segment*, and the segment table base register



OPERATION:

(s, w) loaded into segment and word registers
$(A + s)$ loaded into MA
$c(A + s)$ fetched into MR
if MR blank, missing-segment fault
a := base field of MR
b := limit field of MR
if $w > b$, overflow fault
$(a + w)$ loaded into MA

FIG. 5. Segmentation with mapping table in memory

is known sometimes as the *descriptor base register*.

In this case, each program-generated access would incur two memory references, one to the segment table, and the other to the segment being referenced; segmentation would thus cause the program to run as slow as half speed, a high price to pay. A common solution to this problem incorporates a small high speed associative memory into the address translation hardware. Each associative register contains an entry (s, a, b) and only the most recently used such entries are retained there. If the associative memory contains (s, a, b) at the moment (s, w) is to be referenced, the information (a, b) is immediately available for generating the location address a'; otherwise the additional reference to the segment table is required. It has been found that 8 to 16 associative registers are sufficient to cause programs to run at very nearly full speed [S4]. (The exact number depends of course on which machine is under consideration.)

Historically, the four objectives discussed at the beginning of this section have been provided by "file systems," which permit programmers to manipulate named "files" and to control decisions that move them between main and auxiliary memory. In principle, there is no need for the programmer to use a file system in a virtual memory computer, since auxiliary memory is presumably hidden from him and all his information may be permanently represented in his address space. In practice, most contemporary "virtual memory systems" provide both a virtual memory and a file system, together with "file processing primitives" that operate outside the virtual memory. In these systems, a "segment" is a "file" that has been moved from auxiliary memory into address space. Multics is the only documented exception to this [B7].

Among the earliest proposals for segmentation, though without the use of an address space, was Holt's [H2]. Addressing schemes very similar to that given in Figure 4 were first implemented on the Rice University Computer [I1, I2] and on the Burroughs B5000 computer [B10, M1]. This idea was

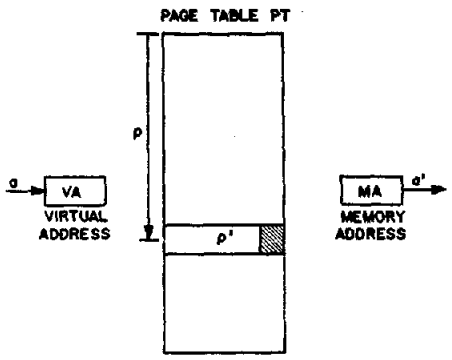
expanded, its implications explored, and a strong case made in its favor by Dennis [D10–D12]. Details of implementing segmentation and of combining segments into programs during execution are given by Arden et al. [A5], and again by Daley and Dennis [D1]. Dennis and Van Horn [D13], Johnston [J1], and also Wilkes [W4], place segmentation in proper perspective among all aspects of multiprocess computer systems. Randell and Kuehner [R3] place segmentation in perspective among dynamic storage allocation techniques, and provide details for its implementation on various machines.

Paging

Paging is another method for reducing the amount of mapping information and making virtual memory practical. Main memory is organized into equal size blocks of locations, known as *page frames*, which serve as sites of residence for matching size blocks of virtual addresses, known as *pages*. The page serves as the unit both of information storage and of transfer between main and auxiliary memory. Each page frame will be identified by its *frame address*, which is the location address of the first word in the page frame.

We suppose that each page consists of z words contiguous in address space, and that the address space N consists of n pages $\{0, 1, 2, \cdots, n - 1\}$ (i.e. nz virtual addresses), and the memory space M consists of m page frames $\{0, z, 2z, \cdots, (m - 1)z\}$ (i.e. mz locations). A virtual address a is equivalent to a pair (p, w), in which p is a page number and w a word number within page p, according to the relation $a = pz + w$, $0 \leq w < z$, where $p = [a/z]$, the integer part of a/z, and $w = R_z(a)$, the remainder obtained in dividing a by z. In machines using binary arithmetic, the computation that generates (p, w) from a is trivial if z is a power of 2 [A5, D11].

Figure 6 shows the essentials of the address translation mechanism that implements paging. The pth entry of the page table contains frame address p' if page p is loaded in frame p', and is blank otherwise. The steps performed in forming location address a' from virtual address a are shown in Figure 6. Note that a missing-page fault occurs if p is not present in memory, interrupting program execution until p has been placed in an available frame of memory. Protection bits (the darkened area in the page table entry) may be compared against the type of reference being attempted, and a protection fault generated if a violation is detected.

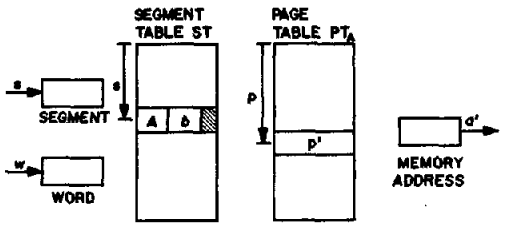


OPERATION:
a loaded into VA
$p := [a/z]$
$w := R_z(a)$
if pth entry of PT blank, missing-page fault
$(p' + w)$ loaded into MA

FIG. 6. Address translation for paging

As in the implementation of segmentation, the page table can be stored in memory. The modification of the address translation mechanism follows the same lines as Figure 5, and is not shown here. As before, program operation may be speeded up by incorporating an associative memory into the address translation mechanism to retain the most recently used page table entries.

Paging was first used in the Atlas computer [F3, K3], and is presently used by almost every manufacturer in at least one of his products [R3]. As with any virtual memory system, it shields the programmer from storage allocation problems, and is therefore susceptible to misuse; its performance has generally been encouraging [A4, O2, P1, S2], but occasionally discouraging [K6]. Because paging has received a great deal of attention in the literature, and its behavior nonetheless tends not to be widely understood, we shall



OPERATION:
(s, w) loaded into segment and word registers
if sth entry of ST blank, missing-segment fault
if $w > b$, overflow fault
$p := [w/z]$
$w' := R_z(w)$
if pth entry of PT_A blank, missing-page fault
$(p' + w')$ loaded into MA

FIG. 7. Address translation for segmentation and paging

devote most of the later sections of this paper to it.

Segmentation and Paging

Because paging by itself does not alter the linearity of address space, it does not achieve the objectives that motivate segmentation. Because segmentation by itself requires that contiguous regions of various sizes be found in memory to store segments, it does not result in the simple uniform treatment of main memory afforded by paging. To understand what is meant by "uniform treatment" of memory, compare the problem of loading a new segment into memory with that of loading a new page into memory. Loading a segment requires finding an unallocated region large enough to contain the new segment, whereas loading a page requires finding an unallocated page frame. The latter problem is much less difficult than the former: whereas every unallocated page frame is exactly the right size, not every unallocated region may be large enough, even though the sum of several such regions may well be enough. (The question of finding or creating unallocated regions will be considered later.)

It is possible to combine segmentation and paging into one implementation, thereby accruing the advantages of both. Figure 7 shows the essentials of such an addressing mechanism. Each segment, being a small linear name space in its own right, may be described by its own page table. The sth entry of the segment table contains a pair (A, b) where A designates which page table describes segment s and b is the limit for segment s. The word address w is converted to a pair (p, w') as in paging, and p is used to index page table A to find the frame address p' containing page p. As before, protection bits may be included in the segment table entry. As before, the segment and page tables may be stored in memory, the addressing mechanism being appropriately modified. As before, associative memory may be used to speed up address formation; indeed, the associative memory is essential here, since each program-generated memory reference address incurs two table references, and the program could run at one-third speed without the associative memory. (If the processor has a sufficiently rich repertoire of register-to-register operations, speed degradation would not be as bad as one-third.)

We mentioned earlier that segmentation and paging combined serve to achieve the objective of sharing or borrowing programs (see the section on Segmentation above). Programmer X, who owns segment s, may allow programmer Y to borrow s, and Y may choose to call s by another name s'. Then programmer X's segment table will contain (A, b) at entry s, and programmer Y's segment table will contain (A, b) at entry s', where A designates a *single* (shared) page table describing the segment in question. The details of implementation, as well as a description of advantages and difficulties of sharing segments, are adequately described in [A5, B7].

Most addressing mechanisms use a single register to implement the segment and word registers shown separately in Figure 7. Typically the leftmost q bits of this register contain the segment name, and the rightmost r bits contain the word name; thus there may be as many as 2^q segments and 2^r words per segment. In these implementations the r word-bits serve as the program counter (PC). Now suppose the program attempts to increment the program counter (i.e. PC : =

PC + 1) when its contents are $c(PC) = 2^r - 1$; the result will be $c(PC) = 0$ and a carry from the leftmost program counter position. Some implementations require that a segment's size limit b satisfy $0 \leq b < 2^r$, whereupon this carry would trigger an overflow fault. Other implementations allow the carry to propagate into the segment field; thus if $c(PC) = 2^r - 1$ in segment s and the operation PC := PC + 1 is performed, the result is $c(PC) = 0$ in segment $s + 1$ [R3].

STORAGE UTILIZATION

Our previous discussion has directed attention to the *mechanisms* of implementing segmentation, paging, or both. A virtual memory system, however, is more than mere mechanism; it necessarily includes the *policies* whereby the mechanisms are used. We mentioned earlier that policies fall into three classes:

1. *Replacement policies.* Determine which information is to be removed from memory; i.e. create unallocated regions of memory.

2. *Fetch policies.* Determine when information is to be loaded; i.e. on demand or in advance thereof.

3. *Placement policies.* Determine where information is to be placed; i.e. choose a subset of some unallocated region.

Replacement and fetch policies use essentially the same principles in both paged and nonpaged systems, and present the same degree of difficulty in either case; we therefore defer discussion of these topics until later. The placement policy for placing k pages in a paging system is in principle quite elementary; use the replacement policy to free k pages. Placement policies for nonpaging systems are, however, considerably more involved. To investigate why this is so, we consider a very elementary model for the behavior of a nonpaged memory system.

Placement Policies

We suppose that a linear m-word memory is to be used to store each segment contiguously (in the manner of the section on Segmentation). At certain moments in time *transactions* occur, which change the configuration of the memory. A transaction is either a request to *insert* a new segment of given size, or to *delete* some segment already present. We assume that the system is in equilibrium; i.e. that, over a long period of time, the number of insertions is the same as the number of deletions for segments of each size. (For our purposes, the fetch policy is the source of insertion requests and the replacement policy the source of deletion requests.) After a long time, the memory will consist of segments interspaced with *holes* (unallocated regions); as suggested by Figure 8, the memory has the appearance of being "checkerboarded."

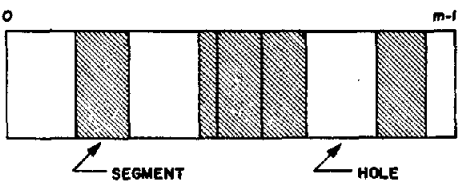


FIG. 8. Checkerboarding of memory

The placement algorithm, which implements the placement policy, makes use of two tables: the "hole table," which lists all the holes, and the "segment table," which already exists for use by the addressing mechanism. An insertion request for segment s, which always adds entry s to the segment table, may increase, leave unchanged, or decrease the number of holes depending respectively on whether s is inserted so as to be surrounded by two holes, a hole and a segment, or two segments. The last possibility occurs with very low probability and may be ignored; and the first possibility is usually precluded because placement policies make insertions beginning at a boundary of the hole. A deletion request for segment s, which always removes entry s from the segment table, may decrease, leave unchanged, or increase the number of holes, depending respectively on whether s is surrounded by two holes, by a hole and a segment, or by two segments. Both the hole table and the segment table must be modified appropriately at each transaction.

We shall derive now two simple but important relationships for placement policies having the properties described above. The

first is the "fifty percent rule" (due to Knuth [K4]), which states that the average number of holes is half the average number of segments. The other is the "unused memory rule," which establishes a relation between the difficulty of placing a segment and the amount of unused memory.

FIFTY PERCENT RULE [K4]. *Suppose the memory system described above is in equilibrium, having an average of n segments and h holes, where n and h are large. Then h is approximately n/2.*

To establish this, we find the probability p that an arbitrarily chosen segment has a hole as right neighbor ("right" has meaning according to Figure 8). Over a segment's lifetime in memory, half the transactions applying to the memory region on its immediate right are insertions, half are deletions; thus $p = \frac{1}{2}$. Therefore, the number of segments with holes as right neighbors is $np = n/2$, i.e. the number of holes is approximately $n/2$.

UNUSED MEMORY RULE. *Suppose the memory system described above is in equilibrium, and let f be the fraction of memory occupied by holes. Suppose further that the average segment size is s_0 and that the average hole size is at least ks_0 for some $k > 0$. Then $f \geq k/(k + 2)$.*

To establish this result for an m-word memory we note that, by the fifty percent rule, there are $n/2$ holes in memory; since each segment occupies an average space of size s_0, the amount of space occupied by holes is $m - ns_0$, and the average space per hole (hole size) is $x = (m - ns_0)/h = 2(m - ns_0)/n$. But we assume $x \geq ks_0$, which implies

$$(n/m)s_0 \leq 2/(k + 2).$$

Then

$$f = (m - ns_0)/m = 1 - (n/m)s_0$$
$$\geq 1 - 2/(k + 2) = k/(k + 2).$$

In other words, if we wish to limit placement algorithm overhead by maintaining large holes, we must be prepared to "pay" for this limitation by "wasting" a fraction f of memory. This is not quite as serious as it might seem, for simulation experiments [K4] show that there is a large variance in hole sizes, and it is often possible to make f as small as 10 percent (i.e. k approximately $\frac{1}{4}$). Even so, it is not possible to reduce f to zero.

Of the many placement algorithms having the properties described above, there are two of special interest. The first is appealing because it makes best use of holes, and the second is appealing because it is simple to implement. Assume there are h holes of sizes $x_1, x_2, \cdots, x_h$, and an insertion request of size s arrives.

1. *Best fit.* The hole table lists holes in order of increasing size (i.e. $x_1 \leq x_2 \leq \cdots \leq x_h$). Find the smallest i such that $s \leq x_i$.

2. *First fit.* The hole table lists holes in order of increasing initial address. Find the smallest i such that $s \leq x_i$. (After a long time, small holes would tend to accumulate at the head of the hole list, thereby increasing the search time. To prevent this, the hole table is implemented as a circular list with a "start pointer"; each search advances the pointer and begins searching with the designated hole.)

Knuth [K4] reports detailed simulation experiments on these and other placement policies. He finds that the first-fit algorithm is the most efficient of a large class of algorithms, including the best-fit. He finds also that the memory size must be at least ten times the average segment size for efficient operation. Similar conclusions are also reported by Collins [C6].

Knuth reports also on another algorithm which he found slightly better than first-fit but which, being not in the class of placement policies described above, does not follow the fifty percent rule and the unused memory rule. This policy is called the "buddy system." Its dynamic properties have not yet been completely deduced [K4].

3. *Buddy system.* Assume that the request size is $s = 2^i$ for some $i \leq k$. This policy maintains k hole-lists, one for each size hole, $2^1, 2^2, \cdots, 2^k$. A hole may be removed from the $(i + 1)$-list by splitting it in half, thereby creating a pair of "buddies" of

sizes 2^i, which are entered in the i-list; conversely, a pair of buddies may be removed from the i-list, coalesced, and the new hole entered in the $(i + 1)$-list. To find a hole of size 2^i, we apply this procedure recursively:

```
procedure gethole(i)
begin if i = k + 1 then report failure;
  if i-list empty then
    begin hole := gethole(i + 1);
        split hole into buddies;
        place buddies in i-list;
    end
  gethole := first hole in i-list;
end
```

Overflow and Compaction

The unused-memory rule tells us that, in equilibrium, we must tolerate a significant loss of memory. In terms of Figure 8, the memory has become so checkerboarded that there are many small holes, collectively representing a substantial space. Indeed, it is possible that, when we scan the hole sizes $x_1, x_2, \cdots, x_h$ for a request of size s, we find $s > x_i$, $1 \leq i \leq h$ (i.e. the request cannot be satisfied) even though $s < \sum_{i=1}^{h} x_i$ (i.e. there is enough space distributed among the holes). What can be done about this?

The solution usually proposed calls for "compacting memory," i.e. moving segments around until several holes have been coalesced into a single hole large enough to accommodate the given request. Knuth [K4] reports that simulation experiments showed that, when the first-fit algorithm began to encounter overflow, memory was nearly full anyway; thus compacting it would provide at best marginal benefit. In other words, a good placement policy tends to obviate the need for a compacting policy.

A somewhat different point of view can be adopted regarding the role of memory compaction. Instead of using a sophisticated hole selection policy and no compaction, we may use a sophisticated compaction policy and no hole selection. Just as overhead in maintaining the hole list previously limited our ability to use memory fully, so the overhead in running a compaction policy limits our ability to use memory fully. To show this, we consider the compaction scheme suggested in Figure 9. At certain moments in time—"compaction initiations"—computer operation is suspended and all segments are moved together at the low end of memory, creating one large hole at the high end of memory. Each insertion request is placed at the low end of the hole, thereby moving the boundary rightward; when the boundary reaches the high end of memory, the next compaction initiation occurs.

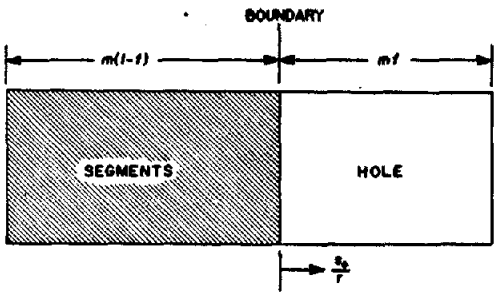


FIG. 9. Configuration of memory after compaction

COMPACTION RESULT. *Suppose the memory system described above is in equilibrium, a fraction f of the memory being unused; suppose that each segment is referenced an average r times before being deleted, and that the average segment size is s_0. Then the fraction F of the time system expends on compaction satisfies* $F \geq (1 - f)/[1 - f + (f/2)(r/s_0)]$. To establish this result, observe that a reference occurs to some segment in memory each time unit, and that one segment is deleted every r references. Because the system is in equilibrium, a new segment must be inserted every r references; therefore the rate of the boundary's movement is s_0/r words per unit time. The system's operation time t_0 is then the time required for the boundary to cross the hole, i.e. $t_0 = fmr/s_0$. The compaction operation requires two memory references—a fetch and a store—plus overhead for each of the $(1 - f)m$ words to be moved, i.e. the compaction time t_c is at least $2(1 - f)m$. The fraction F of the time spent compacting is $F = 1 - t_0/(t_0 + t_c)$, which reduces to the expression given.

Figure 10 shows a plot of F versus f, from which it is evident that, if we are to avoid expending significant amounts of time com-

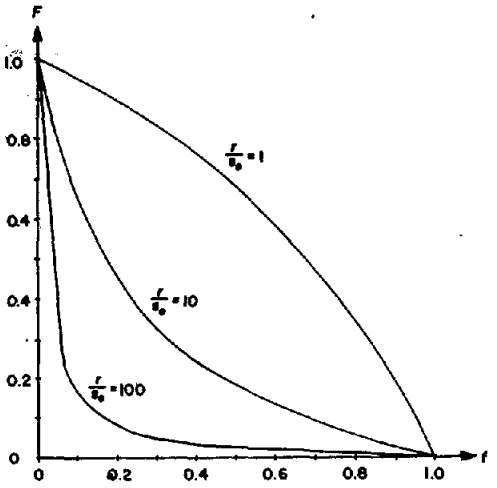


FIG. 10. Inefficiency of compaction

pacting, we must tolerate a significant waste of memory. Because of the relative slowness of compaction compared to searching a well-organized hole list, the former tends to be less efficient than the latter, and compaction is not often used.

In summary, nonpaged memory requires an "investment," i.e. a certain amount of unused memory and overhead in placement policies, for efficient operation. Some systems, notably the Burroughs B5000 series [R3] and certain CDC 6600 installations [B1], have chosen to make this investment; but most have elected to use paged memory, which can be fully utilized by pages at all times. Many of the techniques discussed in this section have been used with great success in applications of a less general purpose nature, particularly in list-processing systems [B8, C4, K4].

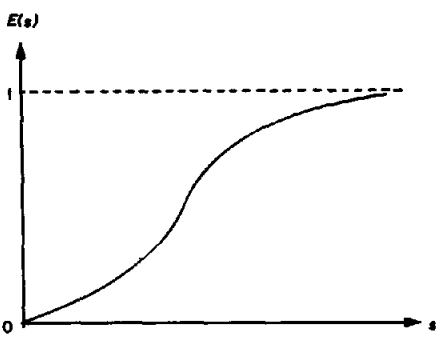


FIG. 11. Probability of external fragmentation

Fragmentation

Our discussion in the previous section unveiled a problem of some importance in virtual memory systems; *storage fragmentation*, the inability to assign physical locations to virtual addresses that contain information.

There are three major types of storage fragmentation. The first is *external fragmentation* [R2], which occurs in nonpaged memories when checkerboarding becomes so pronounced that every hole is too small to be used. (More precisely, external fragmentation occurs for segments of size s with probability $E(s)$, the probability that $s > \max\{x_i\}$, where $\{x_i\}$ are the hole sizes. $E(s)$ follows the curve suggested in Figure 11.) The second is *internal fragmentation* [R2], which results in paged memories because storage requests must be rounded up to an integral number of pages, the last part of the last page being wasted (Figure 12). (More precisely, if z is the page size and s a segment size, then s is assigned to k pages, where $(k-1)z < s \leq kz$; then $kz - s$ words are wasted inside the last page.) The third is *table fragmentation*, which occurs in both paged and nonpaged memories because physical locations are occupied by mapping tables and are therefore unavailable for assignment to virtual addresses.

Randell [R2] reports simulation experiments showing that fragmentation may be serious, and that internal fragmentation is more troublesome than external. His experiments rely on three assumptions: (1) each segment is entirely present or entirely missing from memory, (2) each segment begins at a new page boundary, and (3) segments are inserted or deleted one at a time. Many systems violate (1), there being some nonzero probability that a segment's final page is missing. Many systems violate (2) and (3) by providing facilities that allow

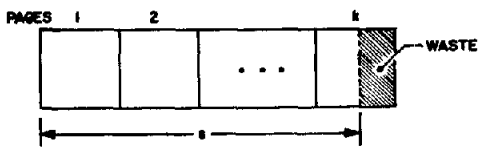


FIG. 12. Internal fragmentation

many small segments to be combined into one large contiguous region of address space (e.g. a "relocatable loader" for virtual memory, or a file system separate from virtual memory). Thus fragmentation is not as serious in practice as it could be, but then again it cannot be ignored.

Page Size

Two factors primarily influence the choice of page size: fragmentation, and efficiency of page-transport operations.

There is a page size optimal in the sense that storage losses are minimized. As the page size increases, so increases the likelihood of waste within a segment's last page. As the page size decreases, so increases the size of a segment's page table. Somewhere between the extremes of too large and too small is a page size that minimizes the total space lost both to internal fragmentation and to table fragmentation.

OPTIMAL PAGE SIZE RESULT. *Let z be the page size and s_0 the average segment size; suppose c_1 is the cost of losing a memory word to table fragmentation and c_2 the cost of losing a memory word to internal fragmentation, and let $c = c_1/c_2$. If $z \ll s_0$, the optimal page size z_0 is approximately $(2cs_0)^{\frac{1}{2}}$.*

To establish this result, suppose segment size s is a random variable with expectation $E[s] = s_0$. A segment may be expected to occupy approximately s_0/z pages, each being described by one page table word; the page table cost for this segment is therefore approximately $c_1 s_0/z$. If $z \ll s_0$, the expected loss inside the last page is approximately $z/2$; the internal fragmentation cost for this segment is therefore approximately $c_2 z/2$. The total expected cost for fragmentation is then

$$E[C \mid z] = (s_0/z)c_1 + (z/2)c_2 .$$

If we set $dE[C \mid z]/dz = 0$ and solve for z, we obtain the expression given for z_0.

These results presume that each segment begins on a page boundary (as suggested by Figure 12), and that both the segment and its page table are entirely present in memory. Many virtual memory computers provide mechanisms for loading or relocating a collection of segments contiguously in address space, in which the internal fragmentation will occur only in the last page of the last segment in a such collection. If there are k segments in such a collection on the average, then the foregoing results are modified by replacing s_0 by ks_0, whence $z_0 = (2cks_0)^{\frac{1}{2}}$.

These results are by no means new. In fact, the problem of choosing page size to minimize fragmentation is identical to that of choosing block size in variable length buffers to minimize space lost to internal fragmentation and to chaining information. Wolman [W7] has studied this issue in some detail; he gives a detailed account of the accuracy of the approximation $z_0 = (2s_0)^{\frac{1}{2}}$.

What might be a typical value for z_0? The available data on segment size [B2] suggests that $s_0 \leq 1000$ words in most cases; taking this and $c = 1$, we find $z_0 \leq 45$ words. This is rather startling when we consider that pages of 500–1000 words are commonly used.

When we consider the other factor—efficiency of page-transport operations—we discover the motivation for using a large page size. Each page-transport operation takes one transport time T (see the section on Basic System Hardware above) to be completed. The following expressions for T on typical devices are lower bounds because in deriving them, we have ignored queueing delays and processor overhead expended on name conversion and auxiliary memory control.

1. *Drums.* To obtain a page from a drum, one must wait an average of half a drum revolution time t_r for the initial word of the desired page to rotate into position. If there are w words on the circumference of the drum, the page transfer time t_t is $t_r z/w$. Therefore

$$T = t_r/2 + t_t = t_r(1/2 + z/w).$$

Typically, $t_r = 16$ msec and $w = 4000$ words.

2. *Disks (moving arm).* A disk access is just like a drum access except there is an additional "seek time" t_s required to move the arms into position. Therefore

$$T = t_s + t_r/2 + t_t = t_s + t_r(1/2 + z/w).$$

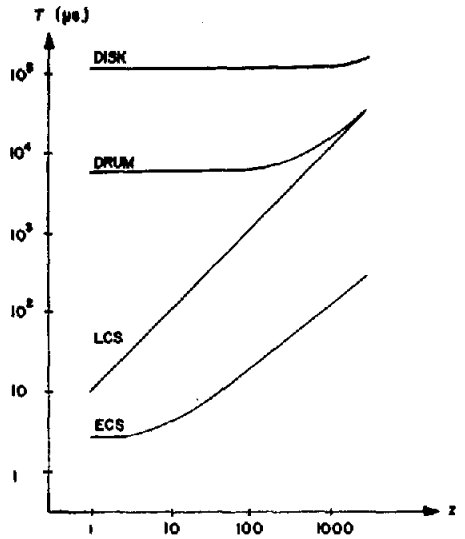


FIG. 13. Lower bound transport times

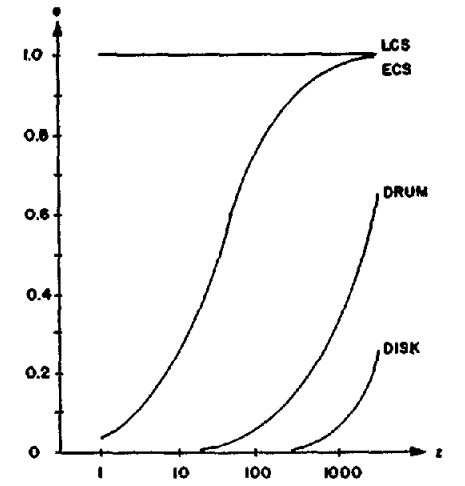


FIG. 14. Upper bound transport efficiencies

Typically, t_s = 100 msec, t_r = 30 msec, and w = 4000 words.

3. *Large capacity storage (LCS).* This is nothing more than a slow-speed core memory. If its cycle time is t_c, then

$$T = t_t = t_c z.$$

Typically, t_c = 10 μsec.

4. *Extended core storage (ECS).* This is a form of core memory with special transmission facilities; after an initial "access time" t_a, it delivers v words per main memory cycle. Therefore

$$T = t_a + t_t = t_a + (z/v)t_c.$$

Typically, t_a = 3 μsec, t_c = 1 μsec, and v = 10 words.

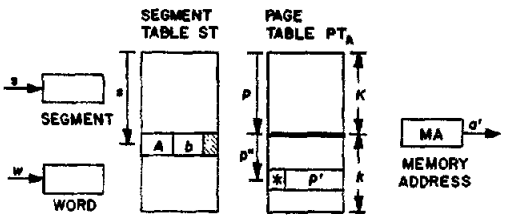
Figure 13 shows these four lower bound transport time expressions plotted for various values of z. Note the several orders of magnitude differences at small page sizes. Figure 14 shows the corresponding upper bound efficiencies $e = t_t/T$ plotted for various values of z. It is immediately apparent from these figures that moving-arm disks should never be used, neither for paging applications nor for any other heavy-traffic auxiliary memory applications [D3]. It is also apparent that drums should be used with care [C2, D3]; and that if drums are used, a page size of at least 500 words is desirable. This is why most paging systems use drums instead of moving-arm disks for auxiliary storage, why page sizes of 500–1000 words are common in these systems, and why some systems have been experimenting with LCS [F1], ECS [F5], and other [L2] auxiliary stores.

It is equally apparent that there is a great discrepancy between the page size for maximizing storage utilization and the page size for maximizing page-transport efficiency—about two orders of magnitude discrepancy. It is easy to see that the poor performance of some of these systems [K6] is at least partially attributable to this factor.

It is sometimes argued that another factor inhibiting small page sizes is the additional hardware cost to accommodate the larger number of pages. Whereas this hardware cost is an initial one-shot investment, the increased storage utilization provides a continuing long-term payoff, and the extra hardware is probably worthwhile. The cache store on the IBM 360/85 is an example of a system where this investment has been made, with apparently good effect.

One approach to constructing a system in which a page size z_0 is feasible would be to use a much faster device, such as LCS or ECS, to handle the traffic of pages in and out of main memory. Some systems have adopted this approach [F1, F5, L2].

Another approach—"partitioned segmentation"—has been suggested by Randell [R2]. It effects a compromise between the large page size required for transport efficiency from rotating auxiliary devices and the small page size required for good storage utilization. We shall describe a slight variant to Randell's scheme. The memory system uses two page sizes: a "major page" whose size is chosen to make transports efficient, and a "minor page" whose size is chosen close to z_0. Suppose the major page size is Z and the minor page size z, where Z is a multiple of z. A segment of size s is assigned a "head" consisting of K major pages such that $ZK \leq s < Z(K + 1)$, and a "tail" consisting of k minor pages such that $zk < s - ZK \leq z(k + 1)$, and $K + k \geq 1$. Internal fragmentation thus occurs only within the last minor page. An address translation mechanism that implements partitioned segmentation is shown in Figure 15. A major drawback to this scheme is that, to operate effectively, segments must be large enough so that they consist mostly of major pages. Available data [B2] suggests that this need not be the case.



OPERATION:

(s, w) loaded into segment and word registers
if sth entry of ST blank, missing-segment fault
if $w > b$, overflow fault
$p := [w/Z]$
$p'' := 0$
if pth entry of PT_A marked by *,
  $p'' := [(w - ZK)/z]$
if $(p + p'')$-th entry of PT_A blank, missing-page fault
$w' := R_z(w - ZK)$
$(p' + w')$ loaded into MA

FIG. 15. Partitioned segmentation

Compression Factor

During any given run, certain sections of a program's code will never be referenced because conditional branch instructions will have unfavorable outcomes. In other words, an n-word program will, on a given run, have occasion to reference only $n' \leq n$ of its words, and $n' - n$ addresses will have been unreferenced. These $n' - n$ unreferenced words are said to be *superfluous* [K5]. Storage losses due to loading superfluous words into main memory are less serious in paged memories using small page sizes because, for small page size, unreferenced blocks of code will tend to be isolated on their own pages, which need never be brought into memory. Belady's simulations [B3] and O'Neill's data [O2] confirm this.

The more are superfluous words isolated on their own pages, the less space will a program require, and the more "compressible" will it be. For page size z and a given run of the program, define the *compression factor* $c(z)$ to be the ratio of the number of referenced pages to the total number of pages. That $c(z) = x$ implies that at least a fraction $1 - x$ of a program's words are superfluous, or conversely that x is the maximum relative amount of memory space a program needs on a given run. Note that $c(n) = 1$ and $c(1) = n'/n$. According to the data presented by Belady [B3] and O'Neill [O2], the compression factor is approximated by the expression

$$c(z) = a + b \log_2 z, \qquad 2^5 \leq z \leq 2^{11},$$

where $a \geq 0$ and $b \geq 0$. The data suggests the following properties of $c(z)$:

1. Halving the page size tends to decrease the compression factor by 10 to 15 percent; thus $0.10 \leq b \leq 0.15$ [B3].

2. For small z, $1 \leq z < 2^5$, the expression $a + b \log_2 z$ is a lower bound on $c(z)$, and in particular $c(1) = n'/n \geq a$. Extrapolating the data, a in the range $0.1 \leq a \leq 0.4$ appear typical.

3. For page sizes $z \geq 2^9$, $c(z) > 0.8$ appear typical.

These results are significant. They reveal a frequently overlooked potential advantage of virtual memory: *small page sizes permit a great deal of compression without loss of efficiency*. Small page sizes will yield significant improvements in storage utilization,

TABLE I. COMPARISON OF PAGED AND NONPAGED MEMORY

Factor	*Paged*	*Nonpaged*
Segmented name space	Feasible	Feasible
Number of memory accesses per program reference:		
1. With paging	2	—
2. With segmentation	—	2
3. With both	3	—
4. With associative memory mapping	≈1	≈1
Replacement policy	Required	Required
Fetch policy	Usually demand	Usually demand
Placement policy	Required, but simple	Required, but complicated
Memory compaction	Not required	Optional; of marginal value
External fragmentation	None	Yes; controlled by placement policy and memory size at least ten times average segment size
Internal fragmentation	Yes, but can be controlled by proper choice of page size	None
Table fragmentation	Yes	Yes
Compression factor	Can be much less than 1 with small page sizes	Usually 1

over and above those gained by minimizing fragmentation. Nonpaged memory systems (or paged systems with large page sizes) cannot enjoy this benefit.

COMPARISON OF PAGED AND NONPAGED MEMORIES

As we have discussed, the various implementations of virtual memory fall into two classes: paged and nonpaged. We have discussed a great number of facts pertaining to each. Table I summarizes these facts and compares the two methods.

According to Table I, paging is superior to nonpaging in all respects save susceptibility to internal fragmentation; but internal fragmentation can be controlled by proper choice of page size. Not listed in the table is an aspect of paged memory that makes its implementation more elegant and much "cleaner" than implementations of nonpaged memory: its "uniform" treatment of memory. Whereas paging regards main memory simply as a pool of anonymous blocks of storage, segmentation regards it as a patchwork of segments and holes of various sizes. The same statement holds for auxiliary memory. Therefore (fixed length) page transports are much simpler to manage than (variable length) segment transports. The difficulty of transporting variable length segments is compounded by overhead in watching out for the specific segment length in order not to overrun buffers. It is no surprise that some form of paging is used in almost all virtual memories.

DEMAND PAGING

Because paging is so commonly used and so frequently discussed in the literature, the remainder of our discussions center around this topic. Demand paging, the simplest form, is the most widely used. Demand paging has—unfairly—been subjected to widely publicized criticism [F2, F4, K6, R3], before anyone has had enough experience to evaluate it properly.

In order to avoid maintaining a large number of lightly used resources, time-sharing and multiprogramming systems attempt to increase the load factors on resources by sharing them. To do this, time is partitioned into disjoint intervals, each program being allocated resources during certain intervals but not during others. (This is sometimes called *resource multiplexing*.) These intervals are defined either naturally, by the alternation between running states and input-output waiting states of processing, or artificially, by *time quanta* and *preemption*. The latter method is used primarily in time-sharing systems, where response-time deadlines must be satisfied. We restrict attention to this case throughout this section.

At the beginning of its allotted time quanta, a program's working information must be loaded into main memory. Older time-sharing systems employed *swapping* to do this, i.e. they would transport a program's working information as a contiguous unit into memory just before each time quantum began, and out of memory just after each time quantum ended. Demand paging systems transport just one page (that containing the next instruction to be executed) into memory just before a program's time quantum begins, and "page in" additional pages as the program demands them; at time quantum end, no immediate action will be taken to remove a program's pages from memory, that being left up to the replacement policy.

One occasionally hears proposals to the effect that paging systems could be improved markedly if swapping were used to load (unload) a program's working information at the beginning (end) of a time quantum, and demand paging were used within a time quantum. We shall show that swapping is at best of marginal value in systems using either a nonmoving auxiliary store or a specially organized drum, the paging drum. Prepaging, however, may have some value when properly managed from a paging drum.

Paging Drum

We pointed out in the section on Page Size above that among all rotating or moving auxiliary stores, only drums (or drumlike stores [A1]) may be suitable for handling page traffic through main memory. Even then, a particular drum organization is required for efficient operation. A *paging drum* [A1, C2, D3, W1] consists of a drum memory together with hardware (or software) implementing an optimal scheduling policy. As shown in Figure 16, the drum surface is laid out into equal areas, each capable of storing one page; each such "drum page" is identified by its "sector address" i and its "field address" j. Each field is equipped with a set of read-write heads. As shown in Figure 17, the scheduler sorts incoming requests into s separate "sector queues" according as which sectors are requested. Within a given sector queue, service is in order of arrival (i.e. "first-come-first-served"). The rotary switch arm revolves synchronously with the drum, pointing to queue i whenever sector i is under the read-write heads. Suppose a read (write) request for drum page (i, j) is at the head of sector queue i. Just as the switch arm

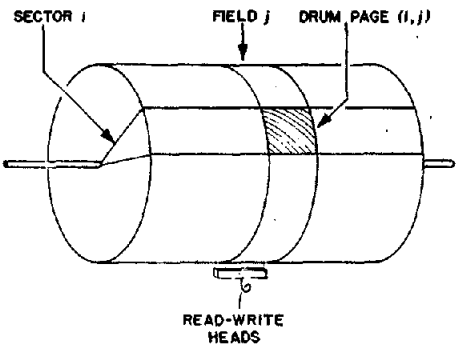


FIG. 16. Layout of paging drum

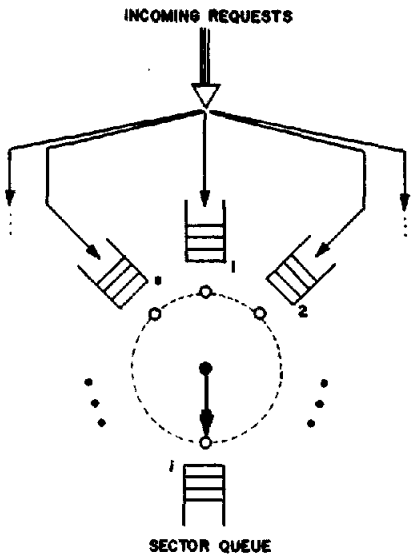


FIG. 17. Paging drum queue organization

reaches sector queue i, the heads for field j are set to read (write) status and connected to the drum channel. Then transmission begins.

Paging drums are sometimes known as "slotted drums" or "shortest-access-time-first" drums. Some manufacturers market drumlike "paging disks," which are fixed-head disks with one head per track. These are equivalent to paging drums.

The paging drum stands in contrast to its historical predecessor, the "first-come-first-serve" (FCFS) drum, which collects all incoming requests into a single, order-of-arrival queue. To compare these, we imagine two systems: System P is a paging drum, and System F an FCFS drum. In both systems, the drum revolution time is t_r and the number of sectors is s. Since most drum allocation policies do not attempt to group contiguous pages of a given program on contiguous sectors, we may assume that each request selects a sector at random [A1, C2, D3, W1]. The "drum load" L is the number of requests waiting in the drum queue(s).

DRUM EFFICIENCY RESULT. *Let $e_P(L)$ denote the expected efficiency of System P and $e_F(L)$ that of System F, when the drum load is held fixed at L. Then*

$$e_P(L) = (L + 1)/(s + L + 1),$$
$$e_F(L) = 2/(s + 2), \qquad L \geq 1.$$

Consider System P. The expression for $e_P(L)$ is an approximation derived as follows. Let $t_0, t_1, \cdots, t_k, \cdots$ be a sequence of time instants at which requests complete service and depart from the drum system. (Since L is assumed fixed, a new request is added to the drum system at each time t_k.) Then $x_k = t_k - t_{k-1}$ denotes the service time of the kth request. Since the requested sector positions are statistically independent and L is fixed, the service times x_k have a common distribution with expectation $E[x]$. Now, $E[x]$ can be written $E[x] = t + E[r]$, where $t = t_r/s$ is a transmission time and $E[r]$ an expected rotational delay. To approximate $E[r]$, we imagine a circle with circumference t_r having $L + 1$ points distributed randomly about its perimeter; one of these points represents the drum position at the moment a request departs, and the remaining L points represent the positions of the requested sectors. The expected distance between two of these points is $t_r/(L + 1) = E[r]$. The efficiency is $e_P(L) = t/E[x] = t/(t + E[r])$, which reduces to the expression given. In System F, each request in the queue must complete its service before the next may begin, and each selects its sector randomly. Therefore $e_F(L)$ is independent of L, and indeed $e_F(L) = e_F(1) = e_P(1) = 2/(s + 2)$.

Several facts follow from this result. (1) For small page sizes (large s) the efficiency e_F is always small. (2) For any page size there are always values of L that make e_P close to 1. (3) Whereas e_F is constant, $e_P(L + 1) > e_P(L)$; in other words, the paging drum is "self-regulating," becoming more efficient under heavier loads. (4) For $L > 1$ and $s \geq 1$, $e_P(L) > e_F(L)$.

As one would suspect, the paging drum (System P) gives smaller transport times than the less efficient FCFS drum (System F).

DRUM TRANSPORT TIME RESULT. *Sup-*

pose a page request arrives when the drum load is L. The time each system delays this request is

$$T_P = t_r(L/s + (s + 2)/2s), \qquad L \geq 0.$$
$$T_F = t_r(L + 1)(s + 2)/2s,$$

The incoming request will be known as the "tagged" request. In System P, the tagged request enters a sector queue whose expected length is $L' = L/s$. Before completing service the tagged request experiences the following additive delays: $t_r/2$ for the drum to begin serving the first request in the queue; $L't_r$ for the drum to begin serving the tagged request; and t_r/s for its own transmission. Thus $T_P = t_r(L' + \frac{1}{2} + 1/s)$. In System F, the tagged request joins the single queue with L requests ahead of it. There are now $L + 1$ requests in the queue, each requiring time $t_r(\frac{1}{2} + 1/s)$ to complete.

From these two results we see that, under normal drum loads ($L > 0$), $e_P > e_F$ and $T_P < T_F$, with the greatest differences occurring at heavy loads. For these reasons, paging systems using FCFS drums may experience severe loss of efficiency.

Cost

To evaluate the "cost" of demand paging, two concepts are useful: "space-time product" and "working set." Suppose a program occupies $m(t)$ pages of memory at time t; the *space-time product* of memory usage across an interval (t_1, t_2) is defined to be

$$C(t_1,t_2) = \int_{t_1}^{t_2} m(t)\, dt.$$

Since memory usage charges are usually based both on the extent and duration of memory usage, $C(t_1, t_2)$ relates to the actual dollar cost of using memory, and is often termed "cost." Space-time cost has become an important aid in determining the efficacy of memory allocation strategies [B5, B6, D5, D9, F1, L1, P1, R3]. The *working set* of a program at a given time is the smallest collection of its pages that must reside in memory to assure some level of efficiency (a more precise definition will be given later) [D4, D5].

Let $C_d(A)$ denote the space-time cost of loading a working set into memory under demand paging from auxiliary memory A, and $C_s(A)$ the cost of loading a working set into memory under swapping from auxiliary memory A. We shall establish four assertions:

- Under demand paging, the paging drum costs significantly less than the FCFS drum (i.e. $C_d(F) - C_d(P)$ is large).
- With nonmoving auxiliary storage (e.g. A is LCS or ECS), demand paging never costs more than swapping (i.e. $C_d(A) \leq C_s(A)$).
- The combined swapping and demand paging strategy is at best of questionable value when compared to "pure" demand paging with a paging drum.
- Unless predictions can be made with little error, prepaging, even from a paging drum, may not be advantageous.

These assertions are considered in the following paragraphs. Assume that a working set of size w is to be loaded, that a single transport operation requires processor time t_0, and that the drum has revolution time t_r and s sectors.

The cost $C_d(A)$ is determined as follows. Suppose $k - 1$ of the w pages have already been loaded and a fault for the kth page occurs; we must reserve one more page of memory and stop the program for the kth transport time T_k. Since there is no correlation between the order of page calls and their order of storage on the drum, $T_k = T$ for $1 \leq k \leq w$. Thus

$$C_d(A) = \sum_{k=1}^{w} kT_k = \sum_{k=1}^{w} kT \qquad \text{(i)}$$
$$= T(w(w + 1)/2).$$

Now if A is the paging drum system P (see the section on Paging Drum above), then $T = t_0 + T_P$. Similarly, $T = t_0 + T_F$ for the FCFS drum system F. Applying the Drum Transport Time Result for load L,

$$C_d(F) - C_d(P) = (w(w + 1)/2)\, t_r(L/2).$$

As long as $L \geq 1$ (the usual case) the cost difference grows as the square of the working set size. This establishes the first assertion.

The cost $C_s(A)$ is determined as follows. We reserve w pages of memory, then transport the entire working set as a unit in a transport time T'. Thus

$$C_s(A) = wT'. \qquad \text{(ii)}$$

If A is ECS with access time t_a (see the section on Page Size above) and page transmission time t_t, then

$$T = t_0 + t_a + t_t,$$

$$T' = t_0 + t_a + wt_t.$$

Substituting these values into (i) and (ii) respectively, we find

$$C_s(A) - C_d(A) = (w(w-1)/2)(t_t - t_0 - t_a).$$

This expression is positive if $t_t \geq t_0 + t_a$, which normally is the case. If A is LCS, then $t_a = 0$, and the same conclusion follows. This establishes the second assertion.

A "swapping drum" is an FCFS drum F for which the system guarantees that each working set's pages are stored on contiguous sectors. Suppose a request for a working set of w pages arrives when L other requests of sizes $v_1, \cdots, v_L$ are in the drum queue; the swapping drum transport time is given by

$$T' = t_0 + t_r\left(\sum_{i=1}^{L}\left(\frac{1}{2} + \frac{v_i}{s}\right) + \frac{1}{2} + \frac{w}{s}\right).$$

(The argument to derive T' is analogous to that for deriving T_F.) We are interested in comparing

$$C_d(P) = (w(w+1)/2)(t_0 + T_P),$$

$$C_s(F) = wT'.$$

We shall ignore t_0 since $t_0 \ll t_r$. Consider two extremes of the drum load $v_1, \cdots, v_L$. At the one, each v_i is a request to swap in a full working set; taking w as the average working set size and each $v_i = w$, we find (after some algebra) that for all $w > 0$, $C_s(F) > C_d(P)$. At the other extreme, each v_i is a request for a single page; taking each $v_i = 1$, we find (after some algebra) that

$$w > w_0 = 1 + 2Ls/(2L + s - 2)$$

is necessary for $C_s(F) < C_d(P)$. For the normally heavy drum loads (L large) found in paging systems, $w_0 \cong s + 1$ is slightly more than a full drum circumference. If we repeat the analysis to include the cost of swapping w pages *out* again at time quantum end, we find $w_0 \cong 2s$; for typical drums $2s$ is approximately 8000 words, a substantial working set. To sum up: as the drum load varies from the former extreme to the latter, the system enters and exits states unfavorable to swapping; even in favorable states, swapping is cheaper only when working sets of substantial size are moved. Our analysis does not account for two other factors: it may be expensive to find or maintain a supply of contiguous sectors into which working sets may be swapped, and it may be expensive to implement both a swapping policy and a demand paging policy in the same system. Swapping thus appears at best to be of marginal value in a demand paging system. This establishes the third assertion.

Now, let $C_p(P)$ denote the cost of prepaging from drum system P, and suppose $\epsilon \geq 0$ is the probability that a prepaged page is not used. To prepage from drum P, we would specify the w pages as a group and add them to the drum load L. Ignoring t_0, this costs approximately wT_P', where T_P' is T_P evaluated at load $L + w$. Of these w pages, ϵw were preloaded erroneously, so there will be ϵw additional page faults; assuming each of these replaces an erroneous page with a correct one, the cost for each is wT_P. Thus,

$$C_p(P) = wT_P' + \epsilon w(wT_P).$$

After some algebra, we find

$$w \geq w_0 = (2L + s + 2)/((1 - 2\epsilon)(2L + s + 2) - 4)$$

is sufficient for $C_p(P) \leq C_d(P)$ to hold. This has two consequences. First, if ϵ is small and L large, then $w_0 \cong 1$, and prepaging would almost always be advantageous. Second, in order that the denominator of the expression for w_0 be positive, we require

$$\epsilon < \tfrac{1}{2}(2L + s - 2)/(2L + s + 2).$$

If ϵ is not small and L is small, then w_0 would be large, and prepaging would not be advantageous. Since the foregoing argument is very qualitative and based on average-value arguments, we must be careful not to attach too much significance to the particular expressions given. Our intention is showing that the advantage of prepaging may be very sensitive to the relations among ϵ, L, and s, and that careful analysis would be required to assess its value in a given system. (See [P1].) This establishes the fourth assertion.

The foregoing discussion establishes also that the performance of virtual memory may depend strongly on the capacity of the channel carrying the traffic of pages through main memory. Although we have not studied it, the reader should realize that several parallel channels between main and auxiliary memory (contrasted with the single channel presumed above) would provide further increases in capacity.

In general, the smaller the ratio of paging traffic through memory to the system's capacity for handling it, the better the performance of the virtual memory. To minimize this ratio, we must (1) choose a *memory management* policy to minimize the rate at which a given program load generates page faults, (2) modify *program structure* to reduce the rate at which a given program generates new page faults, and (3) provide *hardware support* to increase the system's capacity for handling page traffic. These three aspects are examined in detail in the following sections.

PROGRAM BEHAVIOR AND MEMORY MANAGEMENT

Program behavior is among the least understood aspects of computer system design and analysis. And yet we need to model program behavior if we are to have a sound basis on which to predict a program's future memory needs or if we are to understand how close resource allocation policies are to being optimal.

Replacement Algorithms

From now on we shall use $N = \{1, 2, \cdots, n\}$ to denote the pages of a given program. A program's dynamic behavior may be described in machine independent terms by its *reference string*

$$\omega = r_1 r_2 \cdots r_k \cdots, \qquad r_k \in N,\ k \geq 1,$$

which is a sequence of those pages from N which are referenced by the program (not necessarily distinct). We suppose this program has been allocated a memory space of size m, where $1 \leq m \leq n$, and is to operate in that space under paging. If $t(r_k)$ denotes the time instant at which page r_k is referenced, then the expected time $E[t(r_{k+1}) - t(r_k)]$ is Δ if r_k is present in memory and $\Delta + T$ otherwise (see the section on Basic System Hardware). Therefore the expected increment in space-time cost is

$$C(t(r_k), t(r_{k+1})) = \begin{cases} m\Delta & \text{if } r_k \text{ in memory,} \\ m(\Delta + T) & \text{otherwise.} \end{cases}$$

When the page size is fixed and $T > \Delta$ (typically, in fact, $T \gg \Delta$), minimizing the total cost of running a program under paging requires minimizing the number of page faults. To understand what this entails, we need a precise definition of replacement algorithm.

A subset S of N such that S contains m or fewer pages (written $|S| \leq m$) is a possible *memory state*, and $\mathfrak{M}_m$ is the set of all such S. A replacement algorithm generally keeps records about the program's behavior; the status of its records will be called a *control state* q, and Q is the set of all such q. A replacement algorithm *configuration* is a pair (S, q). If the configuration is (S, q) and page i is referenced, a new configuration (S', q') is entered. We describe this behavior by the *allocation mapping*

$$g: \mathfrak{M}_m \times Q \times N \rightarrow \mathfrak{M}_m \times Q,$$

where

$$g(S, q, i) = (S', q')$$

and i is in S'. Starting from an initial configuration (S_0, q_0), a replacement algorithm

processes the references $r_1 r_2 \cdots r_k$ by generating a sequence of configurations

$$(S_0, q_0), (S_1, q_1), \cdots, (S_k, q_k),$$

where

$$(S_k, q_k) = g(S_{k-1}, q_{k-1}, r_k), \qquad k \geq 1.$$

Thus a *replacement algorithm* A may be described by specifying the 3-tuple $A = (Q, q_0, g)$.

Now if A is a demand paging replacement algorithm, then whenever $(S', q') = g(S, q, i)$, the memory state S' must satisfy these properties:

- If $i \in S$ then $S' = S$ (no page fault).
- If $i \notin S$ and $|S| < m$, then $S' = S \cup \{i\}$ (page i added to memory).
- If $i \notin S$ and $|S| = m$, then A selects some $j \in S$ and $S' = (S - \{j\}) \cup \{i\}$ (page i replaces j).

It can be shown that, for any nondemand paging algorithm A, one may construct a demand paging algorithm A' that produces no more faults than A on every reference string [A2, M2]. We are therefore justified in restricting attention to demand paging algorithms. From now on, the term "algorithm" specifically means "demand paging replacement algorithm."

Optimal Paging Algorithms

Suppose $r_1 \cdots r_k \cdots r_K$ is the reference string generated by a given run of a program, and the reference moment $t(r_k)$ is that of a page fault. If algorithm A requires precise knowledge of the future $(r_{k+1} \cdots r_K)$ to make its replacement decision at $t(r_k)$, A is an "unrealizable" algorithm. Otherwise, if A bases its decision at $t(r_k)$ only on assumptions about the future (e.g. probabilities), A is a "realizable" algorithm. In most practical applications, we must be content with realizable algorithms; unrealizable ones would require "preprocessing" the program and recording its reference string. Not only is this operation costly, but the record so obtained may well be invalid, due to conditional branching.

As discussed in the previous section, we take as our optimality criterion the minimization of the number of faults generated. Since the days of the earliest paging machine, people have reasoned that, to minimize the number of faults, it is necessary to maximize the times between faults [K3]. Therefore the following has been the accepted

PRINCIPLE OF OPTIMALITY. *Let* $S = \{1', 2', \cdots, m'\}$ *be the memory state at time* t, *the moment of a page fault, and let* $t(i') \geq t$ *be the earliest moment at which page* i' *is next referenced. Define* $\tau(i') = t(i') - t$. *Replace that page* i' *for which* $\tau(i')$ *is maximum. If the future is not precisely known, replace that page* i' *for which the expected time* $E[\tau(i')]$ *is maximum.*

In the case that we maximize $E[\tau(i')]$—the case of realizable algorithms—we are attempting only to minimize the expected number of faults, rather than the actual number of faults. Thus an optimal unrealizable algorithm would produce fewer faults than an optimal realizable algorithm.

The principle of optimality has great intuitive appeal. Belady [B3] has used it to develop an optimal unrealizable algorithm. Many other authors have developed various optimal realizable algorithms, each depending on the particular assumptions used to determine $E[\tau(i')]$; for example, the Atlas machine's algorithm assumed most programs were looping and therefore generating periodic reference strings [K3], and several systems used an algorithm that supposes $E[\tau(i')] = t - t'(i')$ where $t'(i') < t$ is the time i' was most recently referenced (this rule is called "least recently used"). We shall not attempt to survey the multitude of paging algorithms that have been proposed and studied, these being amply treated in the literature [B3, B4, B6, C3, C8, D4, D5, D9, H1, K5, J2, K3, O2, S2, S3, S5].

Despite its intuitive simplicity, the Principle of Optimality is known not to hold for arbitrary assumptions about reference string structure and statistics. Even when it does hold, proofs of this are difficult, and are known only in simple cases [A2, M2].

Even though the Principle of Optimality may not in fact be always optimal, it is a good heuristic, and experience and experi-

mental evidence indicate that algorithms based on this principle give nearly optimal performance. This evidence, suggested in Figure 18, is abstracted from the work of Belady [B3], and of Coffman and Varian [C3]. Let $F(A, m, \omega)$ denote the number of faults generated as algorithm A processes the reference string ω under demand paging in an initially empty memory of size m, and define the *fault probability*

$$f(A, m) = \sum_{\text{all } \omega} \Pr[\omega](F(A, m, \omega)/|\omega|),$$

where $\Pr[\omega]$ denotes the probability of occurrence of ω, and $|\omega|$ denotes the length of ω. The curves $f(A, m)$ for "reasonable" algorithms A lie in the shaded region of Figure 18 (by "reasonable" we mean that the assumptions used to determine $E[\tau(i')]$ in the Principle of Optimality are reasonable). For comparison we have shown the relative position of $f(A, m)$ for Belady's optimal unrealizable algorithm [B3]. The point is: *for reasonable* A, $f(A, m)$ *is much more sensitive to* m *than to* A. Therefore, although the choice of paging algorithm is important, the choice of memory size is critical.

Figure 18 brings out one other point. Occasionally in the literature one finds analyses of program behavior based on the assumption of randomness, i.e. that each page of a given program is equally likely to be referenced at any given reference. This is equivalent to the assumption that $E[\tau(i')] = E[\tau(j')]$ in the Principle of Optimality. If this were so, the fault probability for every realizable algorithm A would have to be $f(A, m) = (n - m)/n$. This simply is not the case. Programs tend to reference certain pages heavily, others lightly, still others rarely.

Contrary to intuition, increasing the memory size m may not always result in a corresponding decrease in $f(A, m)$; that is, $f(A, m)$ may not be decreasing in m, as suggested by Figure 18. The FIFO (first-in-first-out) replacement algorithm, for example, is known to exhibit an increasing section in its fault probability curve, for certain reference strings [B6]. Mattson et al. [M2] have discovered a very interesting class of replacement algorithms, called *stack algorithms*, whose f-curves are always decreasing in m. These algorithms are defined as follows. Let ω be a reference string, and let $S(A, m, \omega)$ denote the memory state after A has processed ω under demand paging in an initially empty memory of size m. Algorithm A is a stack algorithm if

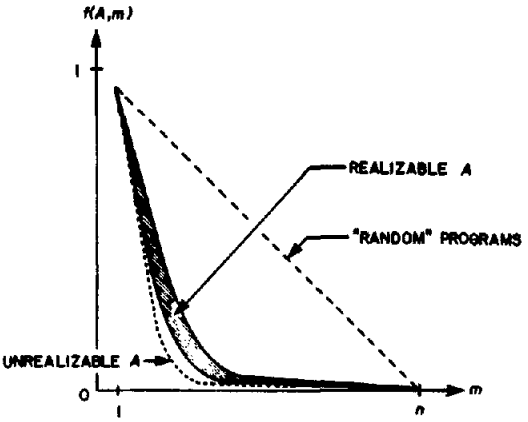


FIG. 18. Fault probability

$$S(A, m, \omega) \subseteq S(A, m + 1, \omega), \quad 1 \leq m < n, \tag{i}$$

for every reference string ω. That is, the contents of the m-page memory are always contained in the $(m + 1)$-page memory, so that the memory states are "stacked up" on one another. The LRU (least-recently-used) replacement algorithm, for example, is a stack algorithm (to see this, note that $S(\text{LRU}, m, \omega)$ always contains the m most recently used pages). Consider a stack algorithm A and a reference string ωx. If x is in $S(A, m, \omega)$—there is no fault when x is referenced—then by (i) x is also in $S(A, m + 1, \omega)$; thus increasing the memory size can never result in more page faults, and $f(A, m)$ must be decreasing in m for every stack algorithm A. The class of stack algorithms contains all the "reasonable" algorithms, and two algorithms known to be optimal [A2, M2]. They are particularly easy to analyze [M2].

The Principle of Locality and the Working Set Model

An important program property, alluded to in previous sections, is *locality*. Informally, locality means that during any interval of execution, a program favors a subset of its pages, and this set of favored pages changes membership slowly. Locality is an experimentally observed phenomenon manifesting itself partly as a tendency for references to a given page to cluster, partly in the shape of the $f(A, m)$ curve in Figure 18 [B3, B4, D4, D5, D9], and partly in the rapidity with which a program acquires certain pages on demand at the beginning of a time quantum [C3, F2]. Locality is not unexpected, by the very nature of the way programs are constructed:

—*Context.* At any given time a program is operating in one of its modules, which causes a concentration of references in certain "regions" or "localities" of address space. For example, its instructions are being fetched from within the pages of some subroutine, or its data are being fetched from the content of some specific data segment.

—*Looping.* Programs tend often to loop for a long time within a small set of pages.

In order to render the statement of locality more precise, we introduce the notion of the "reference density" for page i:

$$a_i(k) = \Pr[\text{reference } r_k = i], \quad i \in N.$$

Thus $0 \leq a_i(k) \leq 1$ and $\sum_{i=1}^{n} a_i(k) = 1$. Although a program's reference densities are unknown (and perhaps unknowable), the definition of "working set" given below obviates the need for attempting to measure them. By a "ranking" of a program's pages we mean a permutation $R(k) = (1', 2', \cdots, n')$ such that $a_{1'}(k) \geq \cdots \geq a_{n'}(k)$; a ranking $R(k)$ is "strict" if $a_{1'}(k) > \cdots > a_{n'}(k)$. A "ranking change" occurs at reference k if $R(k-1) \neq R(k)$; a "ranking lifetime" is the number of references between ranking changes. Ranking lifetimes will tend to be long if the $a_i(k)$ are slowly varying functions of k.

PRINCIPLE OF LOCALITY. *The rankings $R(k)$ are strict and the expected ranking lifetimes long.*

From the principle of locality comes the notion of "working set." A program's *working set* at the kth reference is defined to be

$$W(k, h) = \{i \in N \mid \text{page } i \text{ appears among } r_{k-h+1} \cdots r_k\}, \quad h \geq 1.$$

In other words, $W(k, h)$ is the "contents" of a "window" of size h looking backwards at the reference string from reference r_k. The working set at time t is $W(t, h) = W(k, h)$ where $t(r_k) \leq t < t(r_{k+1})$. Page i is expected to be a member of the working set if it is referenced in the window, i.e. if

$$\sum_{j=k-h+1}^{k} a_i(j) \geq 1.$$

(This equation, together with assumptions about the $a_i(k)$, could be used to determine a value for h. For example, if it were assumed that $a_i(k) = a_i$ and it were declared that pages with $a_i < a_0$ for some given a_0 ought not be expected as members of the working set, then $h = 1/a_0$.) Therefore, a working set is expected to contain the "most useful" pages; by the principle of locality it changes membership slowly.

Now suppose locality holds and $R(k) = (1', 2', \cdots, n')$. If i' is ranked higher than j' (i.e. $a_{i'}(k) > a_{j'}(k)$) then $E[\tau(i')] < E[\tau(j')]$, and because ranking lifetimes are long, this relation is expected not to change. Since i' is more likely than j' to be in $W(k, h)$, there follows:

WORKING SET PRINCIPLE. *Suppose memory management operates according to the following rule: A program may run if and only if its working set is in memory, and a page may not be removed if it is the member of a working set of a running program. Then, according to the principle of locality, this rule is an implementation of the principle of optimality.*

The working set principle is more than a memory management policy, for it implies a strong correlation between processor and memory allocation. Its implementation does not depend on measurement of reference

densities. This principle is used explicitly in at least one computer system, the RCA Spectra 70/46 [D2, O3, W1].

Working sets exhibit a number of important properties. Let $w(h)$ denote the *expected working set size*, i.e. $w(h) = E[\,|\,W(t, h)\,|\,]$. It is shown in [D5] that, for $h \geq 1$,

(1) $1 \leq w(h) \leq \min\{n, h\}$,
(2) $w(h) \leq w(h+1)$ (nondecreasing),
(3) $w(h+1) + w(h-1) \leq 2w(h)$ (concave down),

which give $w(h)$ the general character of Figure 19. The following is also shown in [D5]. Let $g(h)$ denote the probability that a page, when referenced, is not in $W(t, h)$. Suppose h is increased by 1, so that a new reference (r_{t-h}) is included in the window; the resulting change in the working set size is

$$\Delta W = \begin{cases} 1 & \text{if } r_{t-h} \text{ is not in } W(t, h), \\ 0 & \text{otherwise.} \end{cases}$$

But then $E[\Delta W] = g(h)$, and we have the important result that

$$g(h) = w(h+1) - w(h).$$

This suggests that measurements of a program's working set size function can be used to obtain approximations to $f(A, m)$, for $m = w(h)$ and working set strategy A. It is possible to relate $w(h)$ to certain properties of reference strings [D5], and to use $w(h)$ in determining how much memory is required in a given computer system [D7]. Finally, let $w(h, z)$ denote the expected working set size (in pages) when the page size is z, and apply the compression results of the section on Compression Factor:

$$z_1 w(h, z_1) \leq z_2 w(h, z_2) \qquad \text{if } z_1 < z_2 .$$

That is, a working set will comprise fewer words for smaller page sizes.

The definition given above is not, of course, the only possible definition for working set. As specified, the method for measuring a working set is after the fact and its reliability depends on the slowly varying assumption about reference densities. The method will fail to predict the imminent

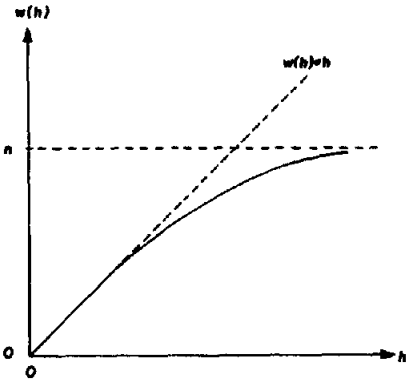


FIG. 19. Expected working set size

presence in the working set of a page which was not referenced in the window. This definition is designed for systems where the future is unknown, where the principle of locality holds most of the time, and where a "maximum likelihood" estimate of the future is sufficient. A still open question concerns how to use "context" and "looping" properties, together with knowledge of program structure, to predict *before it is referenced* that a page will shortly become a member of the working set.

Multiprogramming and Thrashing

Paging algorithms for multiprogrammed memories normally lie at or between two extremes:

1. *Locally.* The memory is partitioned into "work spaces," one for each program. The paging algorithm is applied independently in each work space. In particular, a page fault in a given program can cause a replacement only from its own work space. The size of a work space remains fixed until allowed to change by the system.

2. *Globally.* The paging algorithm is applied to the entire collection of running programs, as if that collection were one large program, without regard for which pages belong to which programs. In particular, a page fault in a given program may cause a replacement from any program in memory. The size of a program's work space is therefore randomly variable.

The working set principle, so formulated

that it tells how memory is to be managed under multiprogramming, is a form of a local policy. Indeed, global policies are in general suboptimal, partly because there is no way to determine when memory is "overcrowded," and partly because there is no way to guarantee that a program's work space is large enough to contain its working set even if memory is not "overcrowded."

Multiprogramming under a global policy is susceptible to *thrashing*, a collapse of performance that may occur when memory (or parts of memory) is overcommitted [D6]. Thrashing is a complicated phenomenon. At the risk of oversimplification, we shall derive a condition that estimates when it will occur. We assume (1) that the ith program in memory has average work space m_i and fault probability $f_i(m_i)$ under the given global policy, where f_i is stationary over the time interval under consideration; and (2) for each i, $f_i(m') \geq f_i(m)$ whenever $m' \leq m$.

A "duty factor" $d(m)$ for a program occupying a work space of average size m may be defined as follows: if $f(m)$ is the program's fault probability, then the expected number of references between faults is $1/f(m)$; if each memory reference takes expected time Δ (see the section on Basic System Hardware) and each page transport takes expected time T, then the expected fraction of time this program spends in execution is

$$d(m) = [\Delta/f(m)]/[\Delta/f(m) + T], \qquad \alpha = T/\Delta.$$
$$= 1/[1 + \alpha f(m)],$$

Using condition (2) above, it is not difficult to show that, if $m' \leq m$,

$$0 \leq d(m) - d(m') \leq \alpha(f(m') - f(m)). \quad \text{(i)}$$

If $d(m) - d(m')$ is near its upper bound and α is large, a relatively small change in work space size will be reflected as a large change in d. This is necessary to induce thrashing.

Now imagine the following conditions holding for an M-page multiprogrammed memory using a global policy. Initially there are $k - 1$ programs in memory, the ith program occupies a work space of average size $m_i \geq 1$, and $m_1 + \cdots + m_{k-1} = M$. When the kth program is introduced, it is granted m_k' pages and the global policy changes the remaining m_i to $m_i' \leq m_i$. Letting D_j denote the total expected processing efficiency when j programs are in memory, we have

$$D_{k-1} = \sum_{i=1}^{k-1} d_i(m_i),$$

$$D_k = \sum_{i=1}^{k} d_i(m_i').$$

Thrashing occurs if $D_k \ll D_{k-1}$,[1] i.e. the addition of one more program triggers a collapse of processing efficiency. Using (i) we find

$$D_{k-1} - D_k \leq \alpha \sum_{i=1}^{k-1} (f_i(m_i') - f_i(m_i)) - d_k(m_k') \quad \text{(ii)}$$
$$\triangleq \alpha F_0 - d_k(m_k').$$

Now if the quantity $D_{k-1} - D_k$ is near its upper bound and αF_0 is not small, then it is possible to obtain $D_{k-1} \ll D_k$. Experiments on the RCA Spectra 70/46 computer system, for which $\alpha > 10^4$ (a drum auxiliary memory), show that this condition is easy to induce [D2]. Conversely, we can prevent thrashing if we can guarantee that αF_0 is small, which may be done by using faster auxiliary memory or by operating programs with space allocations which vary only in ranges where F_0 is small.

Now suppose a working set policy is in effect. Let the random variable $\omega_i(h_i)$ denote the working set size of program i for window size h_i, and let $g_i(h_i)$ denote the probability that a page is not in the working set. Because the pages with highest reference densities are most likely to be members of the working set, g_i is decreasing, i.e. $g_i(h_i) > g_i(h_i + 1)$. The duty factor $d_i(h_i)$ for program i under a working set policy satisfies

$$d_i(h_i) \geq 1/[1 + \alpha g_i(h_i)],$$

where the inequality holds because a page not in the working set may still be in the memory, so that $g_i(h_i)$ is at least as large as the fault probability. Since g_i is decreas-

[1] Notation $x << y$ means "x is much less than y."

ing, we may always choose h_i large enough so that $g_i(h_i) \leq g_0$ for some given g_0, $0 \leq g_0 \leq 1$; therefore we may guarantee that

$$d_0 \triangleq 1/(1 + \alpha g_0) \leq d_i(h_i) \leq 1.$$

In other words, we may always choose h_i large enough that program i operates at or above the desired level d_0 of efficiency. (Normally, we would choose d_0 so that the relation $d_0 \ll 1$ is false.) This implies that

$$kd_0 \leq D_k \leq k. \qquad \text{(iii)}$$

If we are considering adding the kth program to memory, we may do so if and only if

$$\omega_k(h_k) \leq M - \sum_{i=1}^{k-1} \omega_i(h_i),$$

i.e. there is space in memory for its working set. Assuming that $d_0 \ll 1$ is false, the addition of the kth program cannot cause thrashing. Suppose it does, i.e. suppose $D_k \ll D_{k-1}$; by (iii) we have

$$kd_0 \leq D_k \ll D_{k-1} < k,$$

which yields the contradiction $d_0 \ll 1$. Thus working set policies may be used to prevent thrashing. Experiments on the RCA Spectra 70/46 computer system appear to verify this [D2].

PROGRAM STRUCTURE

Careful attention to algorithm organization and program structure can improve the performance of virtual memory systems. There are two ways in which this can be accomplished: distributing program code properly into pages, and improving programming style.

Program code is normally assigned to pages simply by assigning the first z words to page 1, the next z words to page 2, and so on. There is considerable evidence that this may be far from satisfactory. Comeau [C7] describes an experiment in which a program consisting of many subroutines was paged, first with the subroutines in alphabetic order, then with the subroutines grouped together according as they were likely to call one another; there was a remarkable reduction in the number of page faults using the latter method. McKellar and Coffman [M4] have studied how matrix elements should be assigned to pages and how standard matrix operations could be organized to give better performance under paging; they too report a rather remarkable improvement in certain cases.

Informally, the code distribution problem is: How can the compiler (or the subroutine linker) be employed to distribute program code and data into pages in order to improve locality and obtain small, stable working sets? Formally, the code distribution problem may be stated in the following way. A program is regarded as a directed graph G whose nodes represent instructions or data and whose edges represent possible single-step control transfers. With edge (i, j) is associated a cost $c_{ij} \geq 0$ of traversing that edge (c_{ij} might, for example, represent the probability that (i, j) will be used). Given a page size $z \geq 1$, a *pagination* of the program is a partition of the nodes of G into disjoint sets $X_1, \cdots, X_r$ such that X_k contains at most z nodes, $1 \leq k \leq r$. Each X_k will be placed on its own page. For a given pair of pages (X, X'), let

$$V(X, X') = \sum_{i \in X} \sum_{j \in X'} c_{ij}$$

denote the total cost of all edges passing between X and X'. The cost of the pagination $X_1, \cdots, X_r$ is then

$$C(X_1, \cdots, X_r) = \sum_{1 \leq i \leq j \leq r} V(X_i, X_j).$$

A pagination is *optimal* if it achieves minimal cost. Calculating an optimal pagination for a given program is in general a hopelessly complex computation, and relatively simple algorithms are known only in special cases [K2, R1]. Even then, the prospective user of such a scheme would be faced with the problem of deciding whether he would be executing the optimized code sufficiently often that the long-term savings would balance the initial high cost of obtaining the optimized code.

One must be careful with this sort of approach. However attractive the mathe-

matics involved, the results may not be particularly useful except in certain obvious cases such as those mentioned above. If the trend toward increased use of modular programming continues, the value of using a compiler to determine an optimal pagination is questionable: (1) program modules tend to be small, and very often fit on their own pages; and (2) in contradiction to the assumption that the code optimizer must know the connectivity structure of the entire program, the compiler of a module may not know the internal structure of any other module. (If it did, the very purpose of modular programming would be defeated.) The optimization process cannot, therefore, be invoked prior to loading time; and if the trend toward data dependent program structures continues, there is some question whether even the loader can perform meaningful optimization.

Improving programming style to improve locality is an almost intangible objective and is something about which little is known or can be said [K6]. A few experiments show that locality (and therefore paging behavior) is strongly a function of a programmer's style, and it is possible to improve many programs significantly by relatively minor alterations in strategy, alterations based on only a slight knowledge of the paging environment [B9, S2]. It is not known, however, whether programmers can be properly educated and inculcated with the "right" rules of thumb so that they habitually produce programs with "good" locality. If any such education is to be fruitful for a large class of programmers, it must teach techniques that may be applied without knowledge of machine details (page size, memory size, and the like). Highly structured programming languages, where the "context" (see the section on The Principle of Locality and the Working Set Model) is readily detectable at the machine level, may be the answer; in other words, the programming language would "force" the programmer into the "correct" style. The programming language ALGOL, which makes heavy use of a stack during execution, is an example of this; the working set will surely contain the information near the top of the stack, and is therefore easily measured. Much more sophisticated approaches have been conceived [D14].

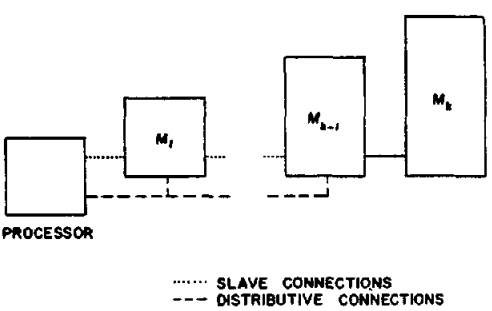


FIG. 20. Memory hierarchy structure

HARDWARE SUPPORT

We have seen that the three principal potential difficulties with multiprogrammed, paged memory systems are fragmentation, thrashing, and the high space-time cost of loading working sets into memory under demand paging. These three problems are partially attributable to the large speed ratio T/Δ between the main and auxiliary memory; if this ratio is large, it forces large page sizes in order to make page transport operations efficient, it makes processing efficiency very sensitive to fluctuations in fault probability, and it causes the space-time cost of a single page-transport operation to be very high. Therefore, one aspect of improving hardware for virtual memory concerns the reduction of this ratio.

The literature reports two directions in which approaches to reducing the ratio T/Δ have proceeded, to which we shall refer as *slave memory* ("cache" memory) [F5, L2, W3, W4] and *distributive memory* [A3, D8, F1, L1, V1]. Both approaches employ a memory hierarchy (Figure 20) consisting of k "levels"; levels $M_1, \cdots, M_{k-1}$ are electronically accessed (e.g. core memory, thin film memory, or silicon-register memory), and level M_k is mechanically accessed (e.g. drum or disk). The electronic levels may be accessed without

latency time. Generally, the lower the number of the level, the faster its speed, the higher its cost, and the lower its capacity. The distinguishing feature is that slave memory permits processing only from level M_1, whereas distributive memory allows processing from any of the electronic levels $M_1, \cdots, M_{k-1}$.

Typically, the combined capacity of the electronic levels in these approaches is large enough to hold all the information of all active programs. Therefore, the transport time for a page among the electronic levels is small, because the speed ratios between adjacent levels can be made small. Accordingly, a hierarchical memory organization of this kind can achieve the objectives required to make paged virtual memory perform well.

The slave memory approach [W3] was first implemented as the "cache store" on the IBM 360/85 [L2]. This approach is so named because information transfers among levels are entirely controlled by activity in the ("master") level M_1. The rules of operation are:

1. Whenever a page is stored in M_i, there is a copy of it in each of $M_{i+1}, \cdots, M_{k-1}$. Whenever a page in M_1 is modified, all copies of it in the lower levels must be modified likewise.

2. Whenever a page not in M_1 is referenced, a request for it is sent to the lower levels; the retrieval time depends on the "distance" to the "nearest" level containing a copy of the required page.

3. Whenever M_i is full and a new page is brought in from M_{i+1}, a replacement policy, usually least recently used, is invoked to select a page to be deleted (since there is already a copy in M_{i+1}, there is no need to move the displaced page).

The principal advantage of this organization is that a program's working set will rapidly accumulate in M_1 and be retained there; accesses will thus be completed at nearly the speed of M_1. A second advantage is that, because transport times are small, pages may be small, and all the advantages of small pages are accrued. A third advantage is that the mechanism is simple enough to be implemented almost entirely in hardware [W3]. A fourth advantage is the possibility of implementing certain associative processing operations in the main level [S6].

Many modern processors employ an "instruction stack," which is a small number of registers (usually no more than 32) that store the most recently referenced instructions of a program. Not only does this stack permit "lookahead," it acts as a small slave memory that allows processing to proceed at nearly register speed for loops that are contained in the stack [W3]. The most notable examples of slave memory implemented as discussed above are the cache memory [L2] on the IBM 360/85, IBM 360/195, and CDC 7600. These systems use $k = 3$, M_1 being a silicon-register memory with cycle time about 0.1 μsec and M_2 a core memory with cycle time about 1 μsec. The level M_1 is about 32K bytes capacity, and has been found substantial enough to accumulate the working sets of all but the largest programs. Even if the working set cannot be contained in M_1, performance is not appreciably degraded because the speed ratio between M_1 and M_2 is small.

In the distributive memory approach, the processor may access information stored in any of the electronic levels. Thus the pages of a given program may be distributed among the various levels while being processed. Generally, the more frequently a page is referenced, the higher should be the level in which it is stored. The most notable example of such a system is that at Carnegie-Mellon University [F1, L1, V1], which uses $k = 3$; M_1 is a standard core memory with cycle time about 1 μsec and M_2 a large capacity store (LCS) with cycle time about 8 μsec.

The distributive memory system presents certain sticky implementation problems not found in the slave memory system. The worst is a requirement that there be a policy to determine when a page should be moved to a higher (or lower) level. These policies are generally based on a tradeoff between

the cost of not moving the page and running at slower speed, and the cost of moving the page; they generally require some estimate of each page's reference density for these decisions, the estimates being obtained by preprocessing [C1], by measurements taken in a previous time quantum [F1], or dynamically [D8]. Systems using dynamic measurement techniques require additional mechanism to avoid instability [D8].

Which of the two approaches—slave or distributive memory—is superior is an unsettled question. That the implementation problems of distributive memory seem more severe leads one to suspect that perhaps the slave memory approach may be the better way to use the hardware.

Reducing the ratio T/Δ is not alone sufficient to improve performance of virtual memory systems. A second aspect of improving hardware for these systems concerns mechanisms for obtaining measurements useful in memory allocation. Most systems implement page table entries with one or more of these extra bits present:

1. *Modified bit.* Set to 1 if and only if the page was modified since being placed in memory. If this bit is 0, the page may be deleted rather than replaced, assuming there is a copy in a lower level of memory.

2. *Use bit.* Set to 1 whenever the page is referenced, and to 0 by a usage metering routine. The metering routine can compile statistics on page use by reading these bits.

3. *Unused bit.* Set to 1 when a page is placed in memory and to 0 the first time it is referenced. This bit signifies that the page has not yet been referenced by the program that demanded it, and should not be removed from memory at least until that time.

The use bits may serve to determine a working set or to calculate reference densities. Counters can also be used for this purpose [D8]. If the addressing mechanism contains a large enough associative memory that its contents remain stable, then the pages entered there may be regarded as the program's working set; similarly, the pages which accumulate in the level M_1 of the slave memory may be regarded as the program's working set.

A third aspect of improving virtual memory hardware concerns the nature of the addressing mechanisms. Difficulties have occurred in virtual memories where information is potentially sharable among distinct address spaces [B7, D13]. Here each segment may have two names: a "local" name which serves to identify it within a given address space, and a "global" name which serves to identify it systemwide. Local names are interpreted in the usual way by hardware (see the section on Implementation of Virtual Memory), and global names are interpreted by software (e.g. "file directories"). The mechanism for converting global names to local names is quite involved and time consuming [B7, D1]. The solution appears to require that every segment have one, system-wide name which may be interpreted by hardware at every level of memory [D14].

CONCLUSIONS

We began this survey of virtual memory system principles by tracing the history and evolution of the forces that compelled dynamic storage allocation, i.e. desires for program modularity, machine independence, dynamic data structures, eliminating manual overlays, multiprogramming, and time-sharing. Among the most elegant solutions to the dynamic storage allocation problem is virtual memory, wherein a programmer is given the illusion that his address space is the memory space. There are two basic approaches to implementing the automatic translation of addresses from address to memory space, these being segmentation and paging; since segmentation is desired by programmers and paging by system implementers, the best implementation combines the two. We compared "pure" segmentation with paging, and found paged memory systems generally superior except for three potential difficulties: (1) susceptibility to low storage utilization for large page sizes, (2) propensity toward thrashing under multiprogramming, and (3) the high cost of loading working sets under demand paging at the start of a time quantum. One

problem with all implementations of virtual memory in which the address space is much larger than the memory space is potential misuse by programmers clinging unduly to the idea that space and time may be traded. This last statement must, however, be interpreted carefully. Programmers who have been warned that the space-time tradeoff does not hold, and have gone to the extra work of reducing the total amount of address space employed, have often *increased* the size of the working set. The objective is to have a small, stable, slowly changing working set. If this is achieved, the amount of address space employed is immaterial.

These problems can be controlled, but require hardware support above and beyond that offered by many current systems. Since a memory system is more than mere implementation of an address map, we included a study of the principles of optimal replacement policies, and found that the working set principle, together with the principle of locality, is an implementation of the Principle of Optimality. By stating a method whereby one may determine each program's working set, this principle implies that one may take steps to avoid overcommitment of memory, and thrashing.

ACKNOWLEDGMENTS

I am deeply grateful to Jack B. Dennis (of MIT), to Bernard A. Galler and Bruce W. Arden (both of the University of Michigan), and to David Sayre (of IBM T. J. Watson Research Center), whose penetrating comments proved invaluable in improving the manuscript. I should also like to thank John E. Pomeranz (of the University of Chicago) for suggesting some refinements in the section on Demand Paging.

REFERENCES

A1. Abate, J., and Dubner, H. Optimizing the performance of a drum-like storage. *IEEE Trans. C-18*, 11 (Nov. 1969), 992–997.

A2. Aho, A. V., Denning, P. J., and Ullman, J. D. Principles of optimal page replacement. Computer Science Tech. Rep. No. 82, Princeton U., Princeton, N. J., Jan. 1970.

A3. Anacker, W., and Wang, C. P. Performance evaluation of computing systems with memory hierarchies. *IEEE Trans. EC-16* (Dec. 1967), 764–772.

A4. Arden, B. W., and Boettner, D. Measurement and performance of a multiprogramming system. Proc. Second ACM Symp. on Operating Systems Principles, Princeton, N. J., Oct. 20–22, 1969, pp. 130–146.

A5. ——, Galler, B. A., O'Brien, T. C., and Westervelt, F. H. Program and addressing structure in a time-sharing environment. *J. ACM 13*, 1 (Jan. 1966), 1–16.

B1. Baskett, F., Browne, J. C., and Raike, W. M. The management of a multi-level non-paged memory system. Proc. AFIPS 1970 Spring Joint Comput. Conf., Vol. 36, pp. 459–465.

B2. Batson, A., Ju, S., and Wood, D. Measurements of segment size. Proc. Second ACM Symp. on Operating Systems Principles, Princeton, N. J., Oct. 20–22, 1969, pp. 25–29. Also, *Comm. ACM 13*, 3 (March 1970), 155–159.

B3. Belady, L. A. A study of replacement algorithms for virtual storage computers. *IBM Syst. J. 5*, 2 (1966), 78–101.

B4. ——. Biased replacement algorithms for multiprogramming. Rep. NC697, IBM T. J. Watson Res. Center, Yorktown Heights, N. Y., March 1967.

B5. —— and Kuehner, C. J. Dynamic space sharing in computer systems. *Comm. ACM 12*, 5 (May 1969), 282–288.

B6. ——, Nelson, R. A., and Shedler, G. S. An anomaly in the space-time characteristics of certain programs running in paging machines. *Comm. ACM 12*, 6 (June 1969), 349–353.

B7. Bensoussan, A., Clingen, C. T., and Daley, R. C. The Multics virtual memory. Proc. Second ACM Symp. on Operating Systems Principles, Princeton, N. J., Oct. 20–22, 1969, pp. 30–42.

B8. Bobrow, D. G., and Murphy, D. L. Structure of a LISP system using two-level storage. *Comm. ACM 10*, 3 (March 1967), 155–159.

B9. Brawn, B., and Gustavson, F. Program behavior in a paging environment. Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33, pp. 1019–1032.

B10. Burroughs Corporation. The descriptor —A definition of the B5000 information processing system. Burroughs Corp., 1961.

C1. Chen, Y. C. Selective transfer analysis. Rep. RC-1926, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., 1968.

C2. Coffman, E. G., Jr. Analysis of a drum input/output queue under scheduled operation in a paged computer system. *J. ACM 16*, 1 (Jan. 1969), 73–90.

C3. ——, and Varian, L. C. Further experimental data on the behavior of programs in a paging environment. *Comm. ACM 11*, 7 (July 1968), 471–474.

C4. Cohen, J. A. Use of fast and slow memories in list processing languages. *Comm. ACM 10*, 2 (Feb. 1967), 82–86.

C5. COHEN, L. J. Stochastic evaluation of a static storage allocation. *Comm. ACM 4*, 10 (Oct. 1961), 460–464.

C6. COLLINS, G. O., JR. Experience in automatic storage allocation. *Comm. ACM 4*, 10 (Oct. 1961), 436–440.

C7. COMEAU, L. A study of the effect of user program optimization in a paging system. ACM Symp. on Operating System Principles, Gatlinburg, Tenn., Oct. 1–4, 1967 (7 pp.).

C8. CORBATÓ, F. J. A paging experiment with the Multics system. Rep. MAC-M-384, MIT Project MAC, Cambridge, Mass., May 1968.

D1. DALEY, R., AND DENNIS, J. B. Virtual memory, processes, and sharing in multics. *Comm. ACM 11*, 5 (May 1968), 306–312.

D2. DEMEIS, W. M., AND WEIZER, N. Measurement and analysis of a demand paging time sharing system. Proc. 24th Nat. Conf. ACM, ACM Pub. P-69, 1969, pp. 201–216.

D3. DENNING, P. J. Effects of scheduling on file memory operations. Proc. AFIPS 1967 Spring Joint Comput. Conf., Vol. 30, pp. 9–21.

D4. ——. The working set model for program behavior. *Comm. ACM 11*, 5 (May 1968), 323–333.

D5. ——. Resource allocation in multiprocess computer systems. Tech. Rep. MAC-TR-50, MIT Project MAC, Cambridge, Mass., 1968 (Ph. D. thesis).

D6. ——. Thrashing: Its causes and prevention. Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33, pp. 915–922.

D7. ——. Equipment configuration in balanced computer systems. *IEEE Trans. C-18* (Nov. 1969), 1008–1012.

D8. —— AND BRUNO, J. L. On the management of multilevel memories. Computer Science Tech. Rep. 76, Princeton U., Princeton, N. J., April 1969.

D9. ——, CHEN, Y. C., AND SHEDLER, G. S. A model for program behavior under demand paging. Rep. RC-2301, IBM T. J. Watson Res. Center, Yorktown Heights, N. Y., Sept. 1968.

D10. DENNIS, J. B. Program structure in a multi-access computer. Tech. Rep. MAC-TR-11, MIT Project MAC, Cambridge, Mass.

D11. ——. Segmentation and the design of multiprogrammed computer systems. *J. ACM 12*, 4 (Oct. 1965), 589–602.

D12. —— AND GLASER, E. L. The structure of on-line information processing systems. Proc. Second Congress on Information Syst. Sci., Spartan Books, Washington, D. C., 1965, pp. 5–14.

D13. —— AND VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (March 1966), 143–155.

D14. ——. Programming generality, parallelism and computer architecture. Proc. IFIP Congr. 1968, Vol. 1, North-Holland, Amsterdam, 1969, pp. 484–492 (Computation Structures Group Memo 32, MIT Project MAC, Cambridge, Mass., Aug. 1968).

F1. FIKES, R. E., LAUER, H. C., AND VAREHA, A. L., JR. Steps toward a general-purpose time-sharing system using large capacity core storage and TSS/360. Proc. 23rd Nat. Conf. ACM, ACM Pub. P-68, 1968, pp. 7–18.

F2. FINE, G. H., JACKSON, C. W., AND MCISAAC, P. V. Dynamic program behavior under paging. Proc. 21st Nat. Conf. ACM, ACM Pub. P-66, 1966, pp. 223–228.

F3. FOTHERINGHAM, J. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Comm. ACM 4*, 10 (Oct. 1961), 435–436.

F4. FREIBERGS, I. F. The dynamic behavior of programs. Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33, pp. 1163–1168.

F5. FUCHEL, K., AND HELLER, S. Considerations in the design of a multiple computer system with extended core storage. *Comm. ACM 11*, 5 (May 1968), 334–340.

H1. HELLERMAN, H. Complementary replacement—A meta scheduling principle. Proc. Second ACM Symp. on Operating Systems Principles, Princeton, N. J., Oct. 20–22, 1969, pp. 43–46.

H2. HOLT, A. W. Program organization and record keeping for dynamic storage allocation. *Comm. ACM 4*, 10 (Oct. 1961), 422–431.

I1. ILIFFE, J. K. *Basic Machine Principles*. American Elsevier, New York, 1968.

I2. —— AND JODEIT, J. G. A dynamic storage allocation scheme. *Comput. J. 5* (Oct. 1962), 200–209.

J1. JOHNSTON, J. B. The structure of multiple activity algorithms. Proc. Third Annual Princeton Conf., Princeton, N. J., March 1969.

J2. JONES, R. M. Factors affecting the efficiency of a virtual memory. *IEEE Trans. C-18*, 11 (Nov. 1969), 1004–1008.

K1. KELLEY, J. E., JR. Techniques for storage allocation algorithms. *Comm. ACM 4*, 10 (Oct. 1961), 449–454.

K2. KERNIGHAN, B. W. Optimal segmentation points for programs. Proc. Second ACM Symp. on Operating Systems Principles, Princeton, N. J., Oct. 20–22, 1969, pp. 47–53.

K3. KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., AND SUMNER, F. H. One-level storage system. *IRE Trans. EC-11*, 2 (April 1962), 223–235.

K4. KNUTH, D. E. *The Art of Computer Programming, Vol. 1*. Addison-Wesley, Reading, Mass., 1968, pp. 435–455.

K5. KUCK, D. J., AND LAWRIE, D. H. The use and performance of memory hierarchies: A survey. Tech. Rep. No. 363, Dep. of Computer Sci., U. of Illinois, Urbana, Ill., Dec. 1969.

K6. KUEHNER, C. J., AND RANDELL, B. Demand paging in perspective. Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33, pp. 1011–1018.

L1. LAUER, H. Bulk core in a 360/67 time sharing system. Proc. AFIPS 1967 Fall Joint Comput. Conf., Vol. 31, pp. 601–609.

L2. LIPTAY, J. S. The cache. *IBM Syst. J. 7*, 1 (1968), 15–21.

M1. MACKENZIE, F. B. Automated secondary storage management. *Datamation 11*, 11 (1965), 24–28.

M2. MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. W. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J. 9*, 2 (1970), 78–117.

M3. MCCARTHY, J., CORBATÓ, F. J., AND DAGGETT, M. M. The Linking Segment Subprogram Language and Linking Loader. *Comm. ACM 6*, 7 (July 1963) 391–395.

M4. MCKELLAR, A., AND COFFMAN, E. G. The organization of matrices and matrix operations in a paged multiprogramming environment. *Comm. ACM 12*, 3 (March 1969), 153–165.

M5. MIT. Report of the long range computation study group, April 1961.

O1. O'NEILL, R. W. A preplanned approach to a storage allocating computer. *Comm. ACM 4*, 10 (Oct. 1961), 417.

O2. ——. Experience using a time sharing multiprogramming system with dynamic address relocation hardware. Proc. AFIPS 1967 Spring Joint Comput. Conf., Vol. 30, pp. 611–621.

O3. OPPENHEIMER, G., AND WEIZER, N. Resource management for a medium scale time sharing operating system. *Comm. ACM 11*, 5 (May 1968), 313–322.

P1. PINKERTON, T. Program behavior and control in virtual storage computer systems. CONCOMP Project Rep. No. 4, U. of Mich., April 1968 (Ph.D. thesis).

P2. POOLE, P. C., AND WAITE, W. Machine-independent software. Proc. Second ACM Symposium on Operating Systems Principles, Princeton, N. J., Oct. 20–22, 1969, pp. 19–24.

R1. RAMAMOORTHY, C. V. The analytic design of a dynamic look ahead and program segmenting system for multiprogrammed computers. Proc. 21st Nat. Conf. ACM, ACM Pub. P-66, 1966, pp. 229–239.

R2. RANDELL, B. A note on storage fragmentation and program segmentation. *Comm. ACM 12*, 7 (July 1969), 365–369.

R3. —— AND KUEHNER, C. J. Dynamic storage allocation systems. *Comm. ACM 11* (May 1968), 297–305.

R4. RISKIN, B. N. Core allocation based on probability. *Comm. ACM 4*, 10 (Oct. 1961), 454–459.

S1. SAMS, B. H. The case for dynamic storage allocation. *Comm. ACM 4*, 10 (Oct. 1961), 417–418.

S2. SAYRE, D. Is automatic folding of programs efficient enough to displace manual? *Comm. ACM 12*, 12 (Dec. 1969), 656–660.

S3. SHEMER, J. E., AND GUPTA, S. C. On the design of Bayesian storage allocation algorithms for paging and segmentation. *IEEE Trans. C-18*, 7 (July 1969), 644–651.

S4. —— AND SHIPPEY, B. Statistical analysis of paged and segmented computer systems. *IEEE Trans. EC-15*, 6 (Dec. 1966), 855–863.

S5. SMITH, J. L. Multiprogramming under a page on demand strategy. *Comm. ACM 10*, 10 (Oct. 1967), 636–646.

S6. STONE, H. S. A logic-in-memory computer. *IEEE Trans. C-19*, 1 (Jan. 1970), 73–78.

V1. VAREHA, A. L., RUTLEDGE, R. M., AND GOLD, M. M. Strategies for structuring two-level memories in a paging environment. Proc. Second ACM Symp. on Operating Systems Principles, Princeton, N. J., Oct. 20–22, 1969, pp. 54–59.

W1. WEINGARTEN, A. The Eschenbach drum scheme. *Comm. ACM 9*, 7 (July 1966), 509–512.

W2. WEIZER, N., AND OPPENHEIMER, G. Virtual memory management in a paging environment. Proc. AFIPS 1969 Spring Joint Comput. Conf., Vol. 34, p. 234.

W3. WILKES, M. V. Slave memories and dynamic storage allocation. *IEEE Trans. EC-14* (April 1965), 270–271.

W4. ——. *Time Sharing Computer Systems*. American Elsevier, New York, 1958.

W5. ——. Computers then and now. *J. ACM 15*, 1 (Jan. 1968), 1–7.

W6. ——. A model for core space allocation in a time sharing system. Proc. AFIPS 1969 Spring Joint Comput. Conf., Vol. 34, pp. 265–271.

W7. WOLMAN, E. A fixed optimum cell-size for records of various lengths. *J. ACM 12*, 1 (Jan 1965), 53–70.