L. A. Belady
R. P. Parmelee
C. A. Scalzi

The IBM History of Memory Management Technology

The history of memory management technology in IBM during the period between the 1950s and the early 70s is discussed in this paper. The paper concentrates on the programming and operating system aspects of the problem, rather than the hardware technology involved.

Introduction

The speed at which digital computers operate depends heavily upon how fast the processors can get data and instructions from memory. Over the past quarter century, the internal organization of computers has become increasingly sophisticated as a result of efforts to make data more readily accessible to the central processing unit. In parallel with this engineering activity, systems programmers, installation managers, and computer scientists have been striving toward the same goal. Their effort, to optimize the processor's access to information, has come to be called memory management. In this paper we review the evolution of memory management in IBM, particularly as it concerns and affects programming.

In order to present some ways of managing memory, we first need a model on which to define basic objects and concepts. Data and programs are stored, usually in binary form, in a *memory subsystem*. On early computers, the memory subsystem was a single main memory. Computers became faster and computer problems larger, but a single main memory that was both fast enough and large enough had not really been available. This led to a memory subsystem organization consisting of a set of devices, typically consisting of a small fast main memory for the immediate needs of the processor and some larger, slower devices holding data not expected to be required soon. These devices are usually arranged in a hierarchy and are interconnected so that data can be moved about independent of the processing of other data.

Thus our simple model, or abstraction, consists of a processor and a memory subsystem, with information flowing between them. The processor works cyclically, and at the completion of almost every cycle, a specified piece of information is sent to or requested of the memory subsystem. The memory subsystem then accomplishes the task with some delay. The following questions immediately arise: (1) How is the information piece specified? (2) How large are the pieces? (3) How rapid is the response of the memory subsystem?

Question (1) is that of *addressing*, which can be performed in either of two fundamental ways: by content or by location. In the first, the requested information is found by (partial) matching, as in the process of finding a telephone number in a list in which each number is next to a subscriber's name. (Notice that the name-number pairs do not have to be stored in any order.) An example of the second way of addressing is the looking up of an article in a book by using the table of contents, which gives, in numerical order, the starting page number of each article. Due to its predominance, in our paper the latter, location-based or *coordinate* addressing is assumed, and the extent of addressability is called an *address space*.

Question (2) is not an issue since we introduce a simplifying assumption: An address always refers to a portion of memory whose size is constant in a given context in this paper.

Question (3) relates computer performance to memory management. The time required for the memory subsystem to deliver (or store) a piece of information depends on the arrangement of the devices in the subsystem, as well as the arrangement of the programs and data on those devices. The result is that the rate at which the processor operates is not uniform; periods of full speed execution are interspersed with periods of waiting.

- *Memory management schemes*

One way to reduce if not eliminate idle processor periods is for the programmer to anticipate its demand for new information. Thus, new data are brought into main memory concurrently with, but independently of, processing. A second way, called multiprogramming, is to keep several independent programs in memory concurrently. When one program encounters a delay in the memory subsystem, the processor can be switched to another program which was previously delayed but is now ready to run. We defer till later discussion of multiprogramming. For both schemes, however, the idea is to overlap the time required to move data in the subsystem with some other processing activity.

Until about fifteen years ago, computers only ran one program at a time, and the overlapping of computation with input/output activities was considered a minor task, to be easily included in the program by the programmer. At that time, both the programmer and the machine designer assumed that the computational aspects of computing were predominant, with the programmer concentrating on algorithms and their representations, not on data, and the designer on fast processors, not on memory subsystems. It soon became apparent, however, that the problems of complex program and data structures, often far larger than main memory, were anything but a minor task for the programmer.

In response, system programs handling memory management began to appear. Many of these solutions were also supported by innovation in hardware design. Slowly, a basic dilemma became crystallized: how to make a program containing only functions relevant to the problem it is intended to solve, yet assuring efficient use of machine resources.

One way that this problem was dealt with was to identify the separate memory management functions and to provide system programs to perform them. These must allow the programmer to work only with the logical structure of data, rather than the way the information pieces are arranged in main memory. The decomposition of memory management results in subtasks such as the following:

- Arrange data and program segments within a single linear address space.
- Manage the capacity limitation of this address space.
- Manage the trade-offs introduced by the cost of different devices in the memory hierarchy.
- Manage the capacity variations introduced by the idiosyncrasies of hardware: nonstandard sizes of information pieces (words, records, etc.) and changes in system configurations.

As we see, the situation has become far more complicated than it appeared in our simple model. Not surprisingly, a quite sizable and growing community of researchers and system designers has been involved in solving the problems. IBM has been one of the leaders in its effort to combine the many component solutions in a systematic way and to integrate them into practical systems. However, few of these component solutions have been reported in the literature, and their origins are almost impossible to trace, a situation which is reflected in this paper.

In the next section we discuss how memory management is handled beginning with program preparation and continuing through program execution. Greatest emphasis is on execution-time memory management, that typically done by operating systems. This topic is further elaborated in the following section, which is, in turn, followed by sections on the approaches to memory management taken in several particular IBM operating systems.

The four phases of memory management

The solutions to the problems of memory management are distributed (with some overlap) over several phases: structuring the program *(program design)*, translating the program from a programming language to machine code *(compilation)*, presenting the program material to the machine *(link-editing)*, and executing the program *(dynamic memory management)*.

- *Program design*

In the program design stage decisions must be made about the organization of the program and its data into subroutines, control tables, work areas, and buffers. Key elements of this activity include planning of storage occupancy, storage requirements during execution, linkage among program elements, and availability of data at the proper times. However, programs and their data cannot always be arranged linearly with respect to each other. Also, as execution progresses, programs and data that are no longer required in memory can be removed and the space occupied by them made available for occupation by others.

• *Compilation*

The initial importance of programming languages was to assist the programmer with the expression of algorithms. One aspect of this activity is giving names to and declaring sizes of data elements of a program, which, in turn, means that the compiler must be involved in the clerical task of arranging the program and its data in storage. The first compilers arranged program and data in a single, compact memory space. FORTRAN COMMON was an early compiler extension which allowed the programmer to arrange data elements in a separate memory space. NAMED COMMON was a later elaboration which allowed several separate data spaces. These allowed data to be addressed by separately compiled parts of a program and were particularly valuable when one program was too large to fit into available main memory. These compiler facilities are quite useful when the number and sizes of the data elements are known at compilation time. When they could not be known before run time, however, further compiler extensions were required.

PL/I compilers have perhaps the most detailed facilities for allowing programmers to guide the run-time memory management actions of the operating system. These are based on the idea of "classifying" storage according to how it is to be managed at execution time. Thus, for example, "automatic" storage was provided: Without explicit specification, the main memory requirements of subprograms are honored automatically at invocation time. Upon return of control from the subprogram, the allotted memory is considered surrendered. In providing storage classes, PL/I allows the programmer, the language, the compiler, and the language's run-time environment to cooperate in memory management. Nevertheless, they all assume a storage model which is linear and homogeneous.

• *Link editing and loading*

After the programmer has expressed the computational and memory management aspects of the separate pieces of the total program, they must next be combined into a single unit residing in memory. Thus, each piece is *linked* together, and the composite is *loaded* into memory. The need for computer tools to help the programmer with this linking/loading process was recognized from the very onset of computer programming (even before compilers and assemblers). One of the first computer aids to programming was a form of relocating loader, the one provided before 1950 in EDSAC [1].

A direct extension of linking/loading is the technique called *overlay*. Here the entire program is not in memory at one time; rather, as the name implies, parts (sometimes called phases or segments) of the program are brought in, when needed, overlaying those that are no longer needed. The separate modules of the program are organized into segments, and an overlay program structure is planned in terms of which program segments are required in memory at the same time and which segments can overlay other segments.

Generally, the overlay structure of the program is not part of the code modules themselves. This means that the structure can be changed without requiring a recompilation of the program modules. Early implementations of the concept provided a basic phase-to-phase transition within a program and read the phases from a tape on program demand, with the relative origin for loading the next phase specified on the tape as part of the overlay program format.

• *Execution-time memory management*

Several factors influence memory management during execution. First, for all but the earliest computers, the amount of memory available to the system has been a variable. This is because most computers can operate in a variety of configurations, allowing a choice of memory devices of different speed, capacity, and cost. It also allows periodic upgrading, and it permits system elements to be unavailable while they are serviced. A second, and somewhat stronger, influence has been that the memory requirements of some programs are difficult to predict. The memory requirements of programs that serve users at keyboard terminals often only become known at execution time.

The final, and perhaps strongest, influence on execution-time memory management has been that computer systems are often operated in a multiprogramming mode. The result is that any one program is only given some fraction of main memory, an amount which is unpredictable at the time the program is written or compiled.

Memory management and operating system design

The evolution of the techniques of memory management followed, roughly, the four phases, namely, design, compile, link/load, and execute. In its first decade, programming moved from specifying programs and data as numeric sequences (*i.e.*, machine order codes) to expressing them as fairly abstract units (such as separately compiled FORTRAN programs, common areas, etc.). This evolution did more than just simplify, or clarify, the algorithmic aspects of a program; it led to the abstraction of the memory allocation aspects as well. The result was that at the end of the first decade memory management was done at each of the four phases. However, only at execution time are physical addresses assigned, for only then are total memory size, the space currently remaining, the size of the various routines from libraries, etc., known.

The collection of programs that control the execution of the processor and to a large degree effect memory management became a part of what was first called a "monitor" and later an "operating system." The functions of an operating system can usually be separated into two parts: (1) those providing services to individual user programs; (2) those applying some particular policy (*e.g.*, job priority) to each program with respect to its use of shared or limited resources. Since a principal objective of an operating system—keeping the processor busy running programs—rests on having the programs and their data in memory, memory management is central to its design. However, since there are memory management aspects to each of the three other phases of program preparation, and since these three phases span the time between creating and running the program, the following questions arose: Who is responsible for memory management? How can memory management requirements be expressed so they are consistently stated at each phase? Can the whole problem be handled automatically? By the end of the 1950s, this last question had become the key issue in design of operating systems.

By 1961 [2] this issue came to be characterized as "preplanning *versus* dynamic storage allocation." Pre-planning aimed at maximum run-time efficiency; it entailed greater human effort and required more sophisticated programmer tools. Dynamic storage allocation aimed at increasing programmer productivity even at the possible expense of run-time efficiency. Both required innovative techniques for execution-time memory management, for with either the system designer had to confront the limited size of memory and the contiguity of its addresses.

In June 1961, at the ACM Dynamic Storage Allocation Symposium [3], the developers of the ATLAS computer from Manchester, England, described a hardware technique termed the one-level storage system. This hardware allowed the address of a data item to be treated, not as its coordinate (*i.e.*, location), but as its *identifier*. Only when the processor requested the data at an "address" was it necessary to determine the memory coordinates of that data. Termed "dynamic address translation," this conceptually simple extension of computer addressing gave the operating system designer a means to attack all three parts of the memory management problem: Program addresses could be larger than real memory addresses; they could remain consecutive, even though the associated main memory addresses were not; and allocation of programs to main memory could be changed dynamically—even during execution of the program. To some extent offsetting these advantages was the fact that the translation hardware added complexity (and cost) to the processor and, in general, caused it to run slower.

Until 1972, IBM supported two relatively independent approaches to operating system organization. The main line of support managed program execution in the real memory of the system. DOS/360 and OS/360 were the two IBM operating system families in this line. Examples are drawn from OS/360 to demonstrate the progress made in memory management. Designed for good run time speed of production (*i.e.*, polished, frequently run) programs, OS/360 provided both the programmer and the system operator with a large selection of memory management capabilities that permitted maximizing both program and system efficiency. The other approach to operating system organization exploited dynamic address translation and aimed at providing an execution-time environment for programs with highly dynamic memory requirements. Often characterized as "time-sharing" systems, these latter systems allowed the determination of memory requirements to be deferred until a user entered a command.

In the following sections we describe first a few representative stages in the development of the "real memory" based operating systems and then some developments in virtual storage based systems. We finally consider the merging of these two independent lines into the most recently available IBM product—MVS.

Conventional memory management—Stretch

An early, rudimentary attempt at the management of real storage among independent users is found in project STEM (STretch Experiment in Multiprogramming) in the late 1950s [4]. This was a prototype batch-job multiprogramming system for the IBM 7030 computer. Its design emphasized strategies for scheduling (*i.e.*, selecting jobs to be run, placing them in memory, and getting them "started") and dispatching (*i.e.*, selecting which job in memory is to be assigned next to the processor) that provide for the efficient use of the central processor. The goal was to be achieved by attaining the maximum multiprogramming level. The memory requirement of a job was a key parameter in its scheduling, and each job included a declaration of its maximum memory requirement, which was allocated to it by the system. Whatever memory management functions were required within this allocated space had to be incorporated into the application program itself.

A second key input parameter to the job scheduler was the expected running time of the job; jobs were scheduled based on this estimate. The job expected to run the longest was placed at one extreme of memory; the next longest running job was placed at the other extreme. Scheduling proceeded in this manner, with short running jobs in the middle and long running jobs at the extremes.

The dispatching strategy was to attempt to complete the center, shorter running jobs first. Thus, on two successive job completions the two blocks of storage that were relinquished would yield a single block. Ideally, there would be no storage fragmentation. (See Fig. 1.)

Dynamic real memory management—OS/360

The several versions of OS/360 provide many good examples of the progress made in providing generalized aids for the memory management problem in the real memory operating systems. Prior to DOS/360 and OS/360 magnetic tape was the principal medium for secondary storage, but its sequential nature limited its possibilities for memory management. The arrival of disk storage as an economical second-level storage with good random access capabilities was the catalyst for a new approach to memory management. Central to this was the concept of a program as a set of separate pieces each of which could be brought, when needed, from secondary to main storage.

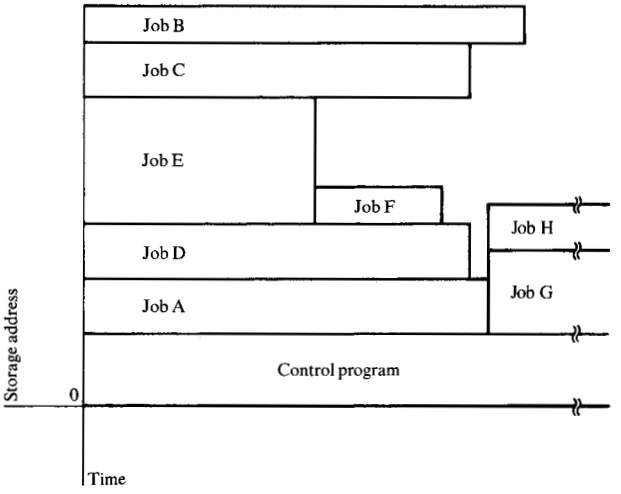


Figure 1 A scheduled job period in project STEM; jobs were scheduled in their alphabetical order.

Some of the earliest programming aids were based on the concept of forming a program by binding together separately named pieces of source code and data areas which were drawn from libraries. These libraries were later pre-compiled, which not only saved the compilation step for standard modules but also allowed deferring until load time the selection of specific standard modules to use for a particular execution. In the design of OS/360 it was decided not only to support compile-time and load-time binding but, by taking advantage of random access disk storage, to provide execution-time binding as well. In OS/360 a set of services provide the link between the application program and the program modules on direct access storage. For example, during execution, a program can request the appropriate subroutine; only then will it be brought from secondary storage. This defers main memory occupancy until it is actually needed and also allows the selection of the subroutine to be based on the data at hand. Execution-time binding is a particularly important feature of OS/360 for two reasons: first, because this capability offers the potential of reducing the maximum amount of storage required by a program during its execution without the preplanning becoming hopelessly complex, and second, because the operating system itself can use it to great advantage. This is because, just as with the application program, the operating system requires subroutines and data areas.

It is fundamental to the design of OS/360 that the management of the main memory requirements of the system, from the application program right down to the innermost services of the operating system itself, had to be done consistently. To achieve this, OS/360 was designed to include a single set of services which allocate main memory to a program on demand, take it back when no longer required, and do all the required housekeeping. These services had to maintain lists of free and occupied space and to associate all occupied space with the appropriate program. Efficient algorithms were devised to make free space available in response to specific requests, including the fitting and grouping necessary to minimize fragmentation. These routines, called GETMAIN and FREEMAIN, are available to all programs within the operating system.

Before proceeding further with the description of memory management in OS/360, let us refine slightly this concept of a program as a set of pieces of code/data/memory. The application program is made up of a "mainline" program and subroutines, some of which are specific to the application and some of which come from system-wide libraries. In addition, the program calls upon system services (such as access to the printer). This is shown schematically in Fig. 2. The important points to be made are that these pieces are not necessarily in main memory at the same time, that they may be shared by several programs, and that some of them were not called for directly by the program but brought in on behalf of it by a system service. The possibilities are almost endless, particularly if several programs are sharing subroutines.

Thus the program can be thought of as a network of named pieces. At any one time some of these pieces are in main memory, forming, in effect, a subnet of the program network. The memory management part of the operating system must keep track of this subnet and provide enough

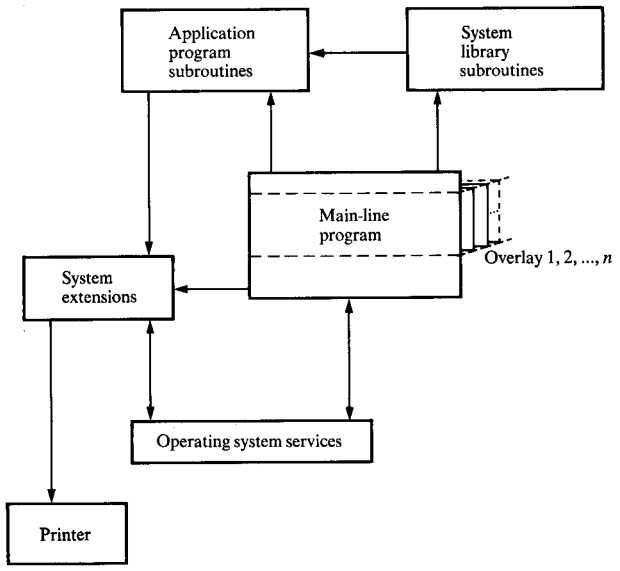


Figure 2 Example of an application program composed of a main-line program, subroutines from various libraries, and overlays.

information so that as program execution progresses the proper program subnet is always in memory. Other memory management functions include such things as making sure that, when a job is completed, all memory pieces have been returned and as identifying portions to reclaim if the program is abruptly stopped.

Before disk storage was introduced and the overlay structure of a program was stored on tape, program execution tended to follow the sequence of segments as they appeared on the tape. OS/360 took advantage of disk storage devices to provide a random access program library and, thereby, to provide a very general implementation of the overlay technique. Segments could be called and recalled in pretty much any sequence.

As discussed above, an important design objective of a memory overlay system is that the program modules need not be recompiled to reflect the overlay segment structure. That is, we write and compile modules—which call other modules as necessary. Next we form the compiled modules into segments. In OS/360, the "program" which manipulated program modules to form segments was called the linkage editor. It resolved the intra-segment calls between modules; the inter-segment calls were converted into calls to the overlay supervisor to bring another segment into memory. Several segments could be combined to form a larger segment, or a segment could be broken into several smaller segments (provided, of course, that it was composed of several modules). In addition to providing these means of manipulating segments and modules, the linkage editor also provided aid in forming a correct structure by automatically positioning COMMON data areas at the proper place in the memory structure so as to be addressable by all modules requiring access to them.

As flexible and general as this implementation is, however, an overlay structure is not sufficient to support the execution dynamics of some programs. The overlay is based on a preplanned arrangement of programs into segments, and some programs, particularly those that serve terminal users, do not admit to such preplanning.

OS/360 also provided facilities for more dynamic assignment of programs to memory by supporting three types of supervisor-assisted linkages. Each is based on a different assumption about the memory residence of the calling and called program. One service, LINK, is a dynamic subroutine call for which both the called and calling programs are to be in memory at the same time. A second service, called Transfer Control (XCTL), allows a program to call another while relinquishing the storage held by the caller. A third service, ATTACH, allows a program to establish the execution of another program concurrently with its own execution. This permits the application program to establish its own multitasking environment. All of these facilities eased the management of the linear nature of memory and scheduling its contents, and they took over much of the housekeeping that would otherwise have to be part of the application program.

A final aspect of memory management in OS/360 is the device independence of the program libraries. Subprograms could reside permanently on any device in the memory subsystem—including main memory. This allowed improved system performance by placing frequently used routines in main memory: No input-output activity would be necessary to make these subprograms available to the caller. This area of memory is called the Link Pack Area.

The evolution through the various versions of OS/360 and OS/370 can be viewed as an evolution in memory management strategies. The principal design objective of OS/360 was, from the first, that it be a generalized multiprogramming operating system. Early developments in its evolution were aimed at meeting this objective in the best ways possible consistent with schedule and resource constraints. Later, on-line environments, such as teleprocessing applications and timesharing, were added to the

design objectives. These new objectives required extension to the memory management services of the system.

The first version of OS/360 was made available in 1966. Called the Primary Control Program (PCP), it was a single job-at-a-time operating system. MFT (Multiprogramming with a Fixed number of Tasks) was released the following year in order to provide a minimum multiprogramming facility. The limited capability of these early versions of OS was to a large extent the result of limited memory management. The next major evolutionary step—MVT (Multiprogramming with a Variable number of Tasks)—was to provide for variably sized job regions. In MVT, the memory requirements for the job or job step were specified in job control statements. The scheduler requested the necessary memory to schedule a job and, if not enough were available, the job was enqueued, waiting for the resource. When memory became available, the scheduler would be started in this dynamically assigned space. On an installation level, care had to be exercised not only over the region size requested but also over whether these requests were made for an entire job or a job step in order to avoid serious fragmentation effects. And the application programmer still had to be generally aware of the maximum addressable storage required by his program and the system services that were used by it.

MFT-II was introduced in 1968 to provide generalized multiprogramming on computers that were too small to run MVT efficiently. MFT-II provided several fixed partitions into which jobs could be scheduled dynamically. It allowed for small partitions which were not large enough to contain the scheduler code itself. These were scheduled from larger partitions when one of them was free (between job steps). This could result in some inefficiency but did provide a smaller system than the more elaborate MVT system. By way of contrast, to schedule a small job in MVT, the scheduler obtained enough space for itself and, when it transferred control to the job it had scheduled, the extra memory it had required for itself was released for re-use by the system.

The next major development was in support of the time-shared use of memory for the Time Sharing Option (TSO) of OS/360. Since it was impossibly inefficient to dedicate physical storage to a program interacting with a user at a terminal, the function of swapping was added to the system. This allowed OS/360 to time-share a portion of addressable memory among multiple users by keeping on external storage the contents of that portion of memory assigned for each user and bringing it back in only when that user was allowed use of the time-shared region. It was also possible to have more than one such region. However, as a consequence of the binding of programs to real addresses, each terminal user was locked into the region to which he was initially assigned, regardless of how the time-sharing user load shifted among regions over time.

Virtual storage memory management

Recall from the previous discussion the notion of address-as-location and address-as-identifier. This distinction, and the hardware that supports it, has given rise to address relocation, dynamic address translation, paging, and virtual storage [5, 6]. In all of these a distinction is made between what we call a "logical" (or "virtual") and a "real" address. Logical addresses are those defined by (and in some sense defining) the program to be run, and real addresses are those defined by (and defining) physical storage. The translation of logical addresses into real addresses ("dynamic address translation" or "relocation") is performed by the CPU as data in memory are needed. This translation (or mapping) can take any of several forms.

The simplest mapping function (called single register relocate) adds a constant to the logical address and compares the sum to some limit. This map allows several logical address spaces to exist in main memory at once, but it has the limitation that the maximum logical address can be no larger than the maximum physical address.

This simple address translation technique was part of the special modifications made to the IBM 7094 for MIT's Compatible Time Sharing System [7]. (See subsequent section "Early time-sharing and the MIT-RPQ.") It was also incorporated into both the Models 135 and 145 of System/370 [8] to support a logical address space for DOS/360 to co-exist with another system.

The ATLAS project provided a more elaborate mapping system. In ATLAS, the main memory of 16K words was divided into 64 blocks of 256 words each. Associated with each block was a "Page Address Register," which contained the virtual address of the 256-word portion of virtual memory that was in that block of main memory. Virtual addresses were translated to real addresses by taking the page address part of the address (that is, all but the least significant 8 bits) and comparing it with the contents of all 64 Page Address Registers. If a match was made, say in register no. 9, then the ninth page of main memory must have contained the data being requested. If no match was found, then that virtual address was not in main memory and had to be brought in from secondary storage.

ATLAS allowed several logical address spaces to reside concurrently in main memory. It also allowed por-

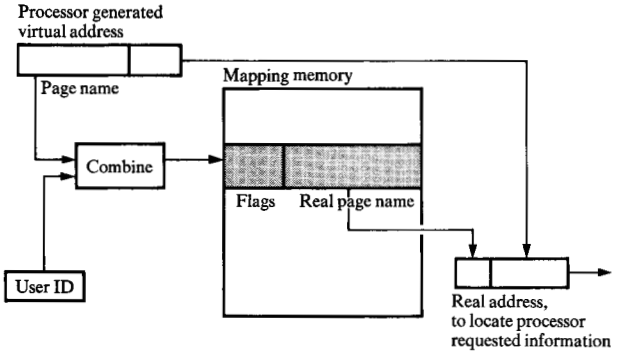


Figure 3 The virtual-to-real address translation used in the M44.

tions of virtual memory to be absent from main memory. This aspect of dynamic address translation gives rise to "demand-paging." When an absent page of virtual memory is referred to, a "page fault" occurs and the memory management part of the system must move the requested page of virtual memory to main memory and update the mapping hardware to reflect this change. Address translation hardware can be used to support paging only if a page fault is detected early enough in the instruction execution cycle, as illustrated later in the case of the Virtual Machine Control Program, CP/40. If the page fault occurs after instruction execution has progressed too far to "back up" (or if the machine is not organized so that instruction execution can be stopped and restarted in mid-cycle), then demand-paging is not possible. This is not to say that address translation cannot be employed, only that the memory management part of the operating system must be capable of enough "pre-planning" to ensure that page faults will not happen.

In the following sections we discuss how this dynamic address translation became generalized into paging systems, and many other memory management tools. The central idea is that address translation allows system cost/performance design problems to be compartmentalized. Thus, broadly viewed, the problem programmer manages things (program and data) in a space of logical addresses. The operating system manages the real storage of the computer by maintaining the mapping function. A characteristic of a computer system, including its operating system, is the extent to which these separate problem areas can be handled separately and concurrently.

Early time-sharing and the MIT-RPQ

Starting in the 1940s, with its development of the WHIRLWIND computer, MIT was involved in the development of computers and significant extensions to computer system organization. In the late 1950s the notion of interactive access (or time-sharing) was being discussed [9], and by the early 1960s MIT had evolved a design for a Compatible Time-Sharing System (CTSS). The goal set for CTSS [7] was: ". . . drastically to increase the rate of interaction between the programmer and the computer without large economic loss and also to make each interaction more meaningful by extensive and complex system programming to assist in the man-computer communication." Given the great discrepancy between human and machine processing speeds, it was felt that the program material for a given on-line user need appear in the execution memory only burst-wise, and between such bursts human thinking would take place. Memory management was central to realizing this goal.

IBM worked with the MIT staff to modify their 7090/94 processor. The 7090/94 was limited to 32K 36-bit words of memory, which, though adequate for many user problems, was not enough for several active user programs and the sort of system support that the staff at MIT envisioned. Additional memory as well as a means of swapping the active user's program into main memory for its "burst" of execution were required. IBM also worked with MIT to devise several extensions to the 7090 to allow it to support interactive access and swapping of programs into (and out of) main memory. Thus, in addition to a second main memory unit (32K words), address relocation, checking, and protection hardware were developed and delivery to MIT was completed by 1963.

The additional memory unit, termed "core-B," was added to the 7090/94 in such a way that the processor executed out of either core-A or core-B. The CTSS operating system resided in core-A, and user programs were moved into and out of core-B. In order to allow several users to co-reside without interfering with each other, memory was "protected" in units of 256-word blocks. CTSS services allowed the user programs to set their address space "size" of up to 32K in units of 256 words. During the bursts of execution, user address space from location 0 to "size" (N*256)-1 was in real memory at locations 0 to (N*256)-1 in core-B. Locations at and greater than (N*256) were protected by the hardware and thus could hold the "upper" portions of the address spaces of other users. This came to be called an "onion-skin" algorithm.

The M44 system

Between 1961 and 1964 the IBM Thomas J. Watson Research Center at Yorktown Heights, NY, undertook a systematic study of the memory usage patterns of 7090 programs [10]. The intent of this work was to learn about the behavior of these programs in a paging environment.

In order to study a program one must first interpretively execute the program and then store on magnetic tape the sequence of memory addresses it just generated. The resulting "address traces" can subsequently be used as input to a paging simulator program. To test program behavior in a paging environment, the simulator in fact allowed page size, real memory size, and page replacement algorithm to be varied. As an example, for different replacement algorithms, the number of page replacements which had to take place in main memory to run a program were counted and displayed as a function of page size and (restricted) memory size supplied. The details of these studies can be found in [11].

These studies created a wealth of insight and, incidentally, led to discovery of the principle of *locality* [11]. Nevertheless, simulation could not be the whole story, and by 1964 it became clear that full-scale experimentation was necessary to test the relation of memory management to some of the ideas which were emerging at that time: time sharing, multiprogramming, operating system structures, simulated or "virtual" machines. An IBM 7044 was chosen as the experimental vehicle. It was modified to support not one but many alternative designs by allowing variable parameter values for page size, memory size, level of multiprogramming, etc.

In order to modify the 7044 into what later became known as the M44, the address field had to be extended, in all instructions with such a field, from 15 bits to 21 bits in length. This made two million words addressable. Of course, the index registers and other hardware involved in addressing were also enlarged. Dynamic address translation was hardware-aided, using a dedicated "mapping memory" of 32K words. The high order position of each processor-generated "virtual" address was used to locate a word in the mapping memory, where either the "real" address of the page containing the desired memory cell could be found or else a "page exception" was generated, *i.e.*, an entry to system programs which then acted to bring in the missing page from some back-up memory (see Fig. 3). Actually, the mapping was somewhat more complex, due to the experimental nature of the machine. For example, a mechanism was included to permit the machine to perform with any page size between 256 and 4096 words that was an integer power of two.

But what about multiprogramming using multiple address spaces? With large pages, smaller mapping tables were needed, and thus only a fraction of the mapping memory was utilized. This made possible the storing of mapping information about several address spaces or virtual machines. Correspondingly, an additional register for "user ID" was installed; it was set whenever a new user's program was dispatched. Its contents were used as an offset in accessing the mapping memory which contained address translation data for several virtual machines, as indicated in Fig. 3. With a 2048-word page size, for example, sixteen independent so-called "44X virtual machines" could share the physical resources.

Another useful thing about the M44 hardware was that a single 36-bit word in the mapping memory could hold much more information than a real page address. Correspondingly, several bits called *status flags* were used to designate certain pages as read-only, privileged, transfer-protected, etc., thus packaging information on capabilities, protection, and address transformation in a single structure and making the operating system simpler.

The operating system, of course, had to perform many other functions, including the management of secondary storage devices, such as disks. All these functions were to achieve *device independence* for programmers such that they need not manage the details and limitations of physical devices. Programmers, for example, could be given only the description of a machine with *virtual memory* and other abstract resources. As the programs in execution issued requests for the resources, the operating system translated these into requests for *real devices*. In experimentation, the M44/44X system verified the usefulness of these ideas [12].

But experimentation helped in many other ways. It was observed, for example, that under certain circumstances a program will run faster with fewer pages assigned to it [13]. Another discovery was the nonlinear effect of program localities: The decrease in processor idleness is generally more abrupt than the increase of assigned memory space which caused it. It was indeed experimentally verified that dynamically varying memory space could be superior to static allocation.

Significant progress in system performance methods was also facilitated by experimentation. At one time a programmed control loop was added to the operating system; it sensed the page traffic intensity between main and back-up memories and the fraction of time the processor was idle. With both variables high, the number of multiprogrammed tasks was reduced by one until congestion eased. The rationale was to allow more average main memory to accommodate the natural locality of each participating program. This dynamic memory-space sharing was then monitored graphically to further the insights gained by the system designers [14].

Another performance oriented experiment was the "loose coupling" of program control and system control

of memory. The programmer (or the compiler) could, at any point in the program, insert one or the other of two special instructions (actually supervisory calls): "named page soon needed" or "named page not needed." While this violated device independence, the supplied information was potentially very useful to the operating system in its effort to maximize throughput. The instructions were, incidentally, *not* taken as mandatory by the operating system; hence the loose coupling.

After having served as an experimental vehicle for more than three years, the M44 system was dismantled in 1968, following the original plan.

Virtual machine systems—CP/40, CP/67, and VM/370

In 1964 the staff at IBM's Cambridge Scientific Center undertook a project intended partly to investigate program and machine structures for an interactive system [15]: "Central to the idea of this system is the concept of the 'virtual machine' and, in our case, the 'virtual 360.' " This work led first, in 1966, to a virtual Machine Control Program (CP/40). This served as a prototype, demonstrating feasibility, and provided the basis for CP/67, an operating system for the System/360 Model 67, and in 1972, VM/370, an operating system for the System/370 machines. The history of the development of this family of operating systems is treated separately in this issue in the paper by R. S. Creasy [16].

Of interest here is the dynamic address translation technique implemented experimentally in 1965 on a System/360 Model 40 [17]. Resembling closely that developed by the ATLAS group, the mapping system was quite simple. For each 4K-byte page frame of real main memory, there was a 16-bit register which contained the identifier of the page of virtual memory that was, at that moment, in that page frame. Thus, for the 256K-byte memory of the Model 40, there were 64 registers. Each page of virtual memory was "identified" by a 4 bit *user id* and the 6 bits which identified the page of the virtual address. Only 6 bits were required because, although System/360 memory addresses are, architecturally, 24 bits, just 18 bits were implemented on the Model 40. That is to say, the Model 40 provided a maximum address space of 256K bytes. During execution, CP/40 loaded the user-id register with the id of the current user and thus specified which of 16 virtual address spaces was to be active. Each time the processor requested data, the 6-bit page number and the user id were combined to form a 10-bit identifier of that page of virtual address space.

To translate an address, the Model 40 compared the page identifier part of the virtual address with the contents of the first 10 bits of each of the 64 identifier registers. (The remaining 6 bits were used to reflect such things as whether the page had been referenced or changed.) If a match was found, then the requested data were located in the corresponding page of main memory. If no match was found, then the requested page was not in main memory. The address translation hardware interrupted the processor so that the data could be transferred from disk to some page of main memory.

To support a demand paging system, the Model 40 had to be further modified to detect page faults before instruction execution was started, since many System/360 instructions cannot be interrupted (or backed-up). The instruction and address structure of System/360 leads to a situation in which, as a maximum case, data from eight pages are required for execution of a single instruction.

The simultaneous comparison of the virtual address (identifier) with that in the 64 identifier registers was performed by an associative memory system [18], which was fast enough so that the execution speed of the Model 40 was not reduced due to address translation.

In addition to its importance as a prototype for two IBM products (CP/67 and VM/370), CP/40 served as a vehicle for experimentation on the performance aspects of memory management and paging systems. One topic of study was the impact on paging performance of program structure [19, 20]. This work led to techniques for arranging code modules in virtual memory so as to minimize the number of page faults during execution. Other results were obtained, *e.g.*, those from a multifactor paging experiment (in which replacement algorithms, load sequence of subroutines, set of problem programs, and main memory were investigated) and from analysis of free storage algorithms. These are reported in [21].

Time sharing system—TSS/360

The TSS/360 operating system [22, 23] was IBM's first offering with virtual memory, aided by the hardware address transformation scheme of the IBM 360 Model 67 computer. TSS/360 was developed in the 1965–67 period, and many of its memory management features became similar to those of the M44. We therefore restrict ourselves here to outlining the differences only.

First, the dynamic address transformation was done in two stages, since the 24-bit (virtual) address field was subdivided into three subfields, containing segment, page, and line numbers. The 4-bit segment field was first used as an offset in the so-called segment table, where the address of the particular page table describing the pages of the addressed segment was found. The (middle) page subfield was then used to locate the corresponding real

page, within which the last 12 bits of the original (virtual) address were used for locating the desired byte. (See Fig. 4.)

This hierarchical address organization made the total amount of mapping information to be stored smaller than the M44's one-step map; this was necessary because segment and page tables, without a dedicated memory, had to share the main memory with all other processor-accessible information. On the other hand, address mapping became slower since two, instead of one, additional memory cycles were needed to find the real address.

To improve the situation, G. A. Blauuw, at that time with IBM, invented a "black box" that could store in its content-addressable memory the mapping information (virtual-real page association) of several most recently used address associations. (M44 experimental data were used in the design and feasibility studies of the Blauuw box.) Each time a virtual address was generated, this box was interrogated and, more often than not, the associated real page number found, resulting in a tenfold speed-up with respect to in-memory two-stage transformation.

A further complication was introduced by a fast drum device which, as back-up, was too small to support the many virtual address spaces, each 16 megabytes long. The result was a three-level memory hierarchy, consisting of main memory, drum, and disk of essentially unlimited capacity.

The memory management functions of TSS thus became quite elaborate. Many innovative algorithms were designed and implemented to cope with the competition between page tables and pages containing programs and data. Sharing read-only information was implemented without duplicating pages, by mapping segments of different address spaces into the same pages in memory. (Each address space was subdivided into 16 equal-sized segments.) Also, adaptive algorithms decided about the "page-out" target area—drum or disk—depending upon scheduling status, memory demand, current space occupied, and other factors. And for cases when available drum space became short, decision-making rules about replacement, quite similar to those between main and simple back-up memory, were incorporated into the operating system. The subsequent migration process was tied to the time slice allotments controlled by the scheduler.

The particular arrangement of records on drum was such that high transfer rates could be achieved only by combining eight (or nine) pages on a single track. This made necessary the introduction of "prepaging," *i.e.*, the loading of several pages into main memory prior to the beginning of a time slice—a slight departure from pure "demand" paging. This made memory management even more closely coupled to the scheduling of the time-shared and multiprogrammed tasks.

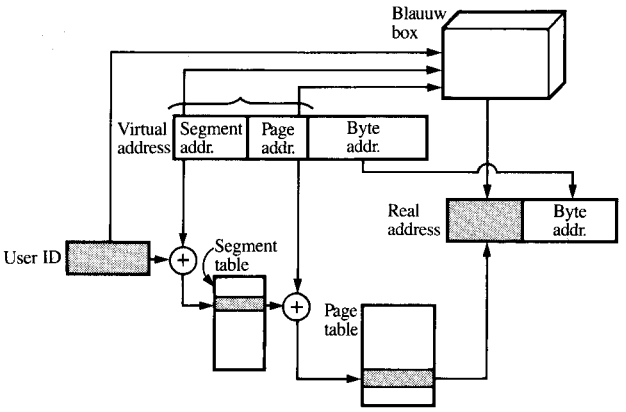


Figure 4 The address translation technique used by the System/360 Model 67.

Multiprogramming demanded the sharing of main memory. This was done by statically assigning areas of different sizes for each time slice of a task, estimated from the demand which had been recorded during previous time slices. Later this scheme was somewhat relaxed by "page stealing," *i.e.*, the reassignment of a page from one task to another. And if there was in memory some residual page left from a previous time slice, the system was programmed to reclaim it.

In summary, TSS was the first comprehensive integrated operating system built for the computers in the 360 and 370 line which had dynamic address translation. It helped gather valuable experience for follow-on operating systems with virtual memories in the early 70s.

Multiple Virtual Storage (MVS)

This then was the situation in the early 70s. The real memory-based systems were in extensive use and offered a great range of memory management tools which, in the main, met the objective of high processor efficiency. However, the human effort—of both programmer and system support staff—needed to manage memory was very high and represented a barrier to adding new applications to the computer. In short, programmer productivity had to be improved. Some years earlier, the direction

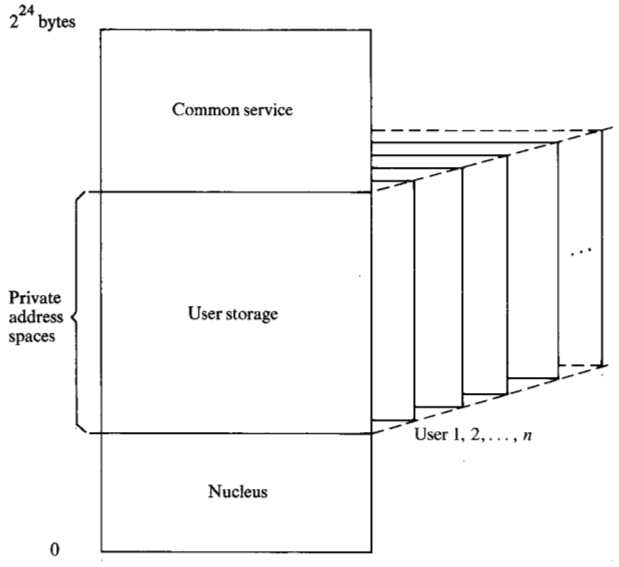


Figure 5 MVS storage map.

set for the virtual storage-based systems had been programmer productivity, and the success of this approach demonstrated by the M/44, CP/40, TSS/360, and CP/67. Thus, in 1972 IBM introduced virtual storage on all of its processors and their operating systems. An overriding consideration in doing this was to minimize disruption to customers: Their old code must continue to run without change (to preserve their investment), and they must not have to accommodate a new operating system structure. The virtual storage design approach taken was substantially that taken by the System/360 Model 67: an address translation extension to the System/360 architecture, rather than a change to that architecture. It did not affect any of the instructions used by application programs, and most of the instructions used by the operating systems were unchanged. This allowed the operating systems for System/360 to be adapted to virtual storage by adding the memory management functions necessary to create and manage a single virtual addressable space (of up to 16 million bytes), running the operating system in that space, and, by demand paging, bringing only the necessary portions of that space into real memory. Though an oversimplification, the initial versions of these operating systems appeared to provide a 16-megabyte main memory, when in fact the real memory was, say, only one megabyte.

The problem that this approach presented was that acceptable performance made it mandatory that portions of the operating system code and data not be demand-paged. It was necessary to locate these "unpageable" portions and organize them into pages; the virtual storage in which the operating system ran could then be categorized as either nonpageable (*i.e.*, fixed in real memory) or pageable.

In this straightforward manner, the real memory-based operating systems were adapted to virtual storage. DOS became DOS/VS, MFT became VS-1, and MVT became VS-2. Each achieved its objective of improving application programmer productivity by allowing much larger regions in which to place programs, with the result that overlay structures became simpler or, quite often, unnecessary. In addition, each supported greater levels of multiprogramming, and many system functions otherwise available only on large machines were available on smaller ones as well. Still the maximum addressable space was limited to 16 megabytes and had to be shared among all jobs. The greater exploitation of virtual storage and the restructuring of the operating systems that this entailed were left to subsequent versions.

As they were the most extensive, we touch briefly on the exploitations of virtual storage made in the subsequent versions of VS-2. The major restructuring of VS-2, called Multiple Virtual Storage (MVS) [24], provided each user with a 16-megabyte address space. (A batch job or someone doing time-sharing is regarded in this context to be a user.) Recall the previous discussion of a program as a network of named pieces and of memory management keeping track of the subnet residing in main memory. In the exploitation of virtual storage in MVS, the System/370 hardware took over the major part of keeping track of the resident memory subnet. When a user or system component is to be given control of the processor, the MVS memory map is established so as to include the virtual pages assigned to that user component.

Another important aspect of the management of multiple address spaces by MVS is that user programs were effectively isolated from each other. All user address spaces share, at one end, the MVS supervisor and, at the other, a Common Service Area (See Fig. 5). Between these lie the portions that contain programs and data private to each user. Real memory assignment of these pieces is by demand paging; therefore, not only can it be deferred until needed by the program, it can also be treated independently of other users. Importantly, much of this burden is handled by the System/370 hardware. During execution no user can refer to (or store into) any other user's private area. In order for one user address space to communicate with another, it had to use the supervisor services that reside in the areas common to both. In later versions of MVS a new hardware facility allowing authorized direct communication between user address spaces was supported.

Two more aspects of the exploitation of virtual storage by MVS should be mentioned. Some system services (*e.g.*, telecommunications access methods) were assigned virtual address spaces. They could then be treated much like problem programs, with the programmer productivity benefits accrued to IBM's own development process. Finally, substantial enhancements were made to the resource scheduling and dispatching functions of the system. It was here, particularly, that the experience with CP/40, M44, TSS/360, and CP/67 was applied.

Conclusions

We conclude this review by noting two trends. First we note the widespread acceptance both inside and outside of IBM of virtual memory. One of two trends now apparent is extension of the System/370 architecture so that it takes over more of the memory management functions. An example of this is the recently announced IBM 4300 processor family, which performs much of the paging management for DOS/VSE.

The second trend is to remove from the programmer the constraints of the linear nature of the address space. This is exemplified by IBM's System 38 [25], in which the addressing structure allows not just an enormous address space (48-bit addresses yielding a 281-trillion-byte address space) but is also designed to directly name the separate pieces of the program. No longer needed are the programs that assign named pieces to their relative positions in a linear space. Rather, the name given by the programmer to each part of a program serves as its address. In this manner, much of the memory management burden shifts to the hardware.

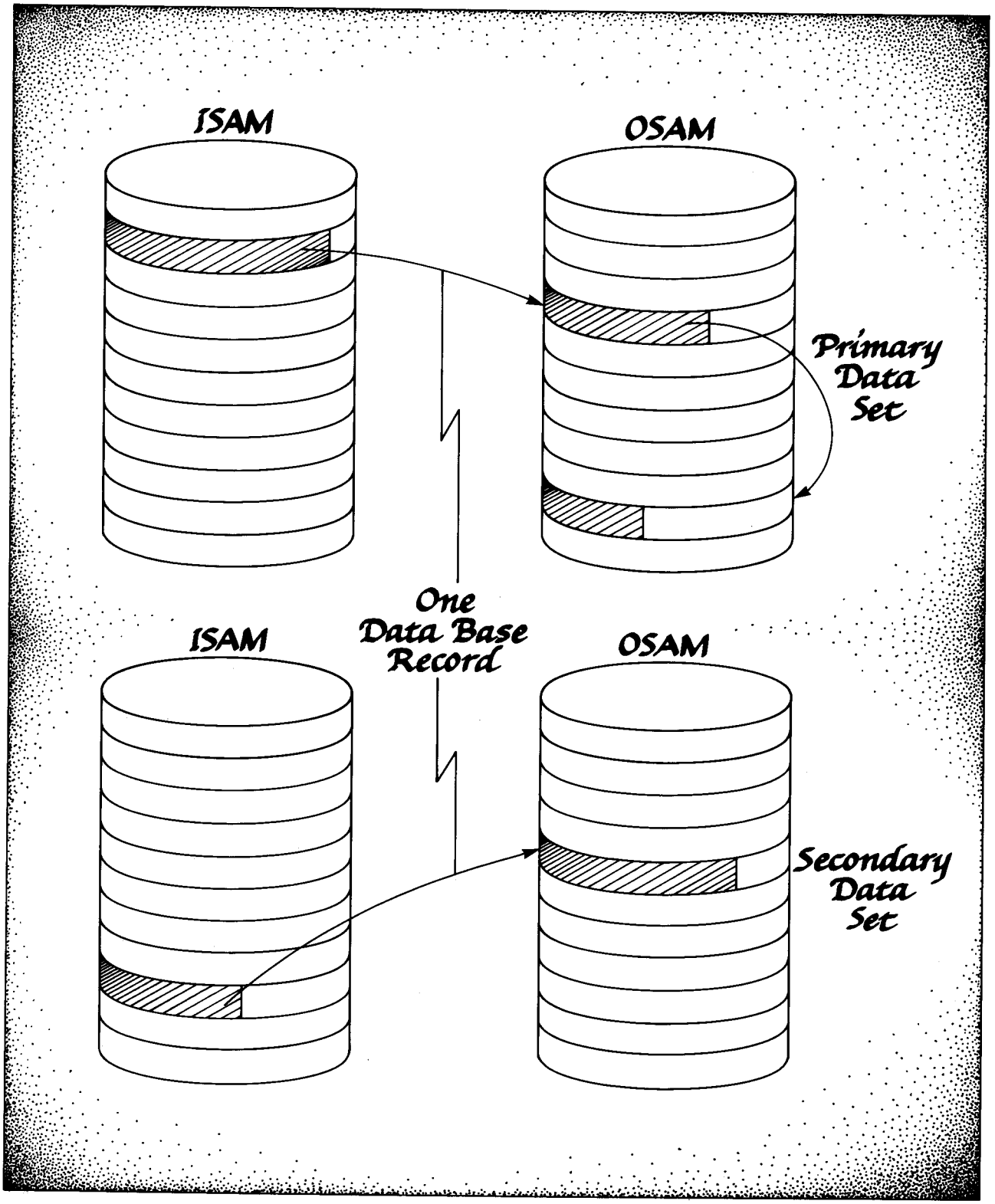
How does one assess these trends? What can be said is that the forces that drive the change will be the shifting costs within the total computing system—where "system" spans not just the processors and storage devices but includes the people who own, program, and operate them. It is a recognition of this total system cost which has governed IBM's past memory management approaches and will continue to do so into the future.

References and notes

1. M. Campbell-Kelly, "Programming the EDSAC," *Annals of the History of Computing* **2,** 7-36 (1980).
2. "Discussion Two: Preplanned vs. Dynamic Storage Allocation Techniques," *Commun. ACM* **4,** 416 (1961). This entire issue is devoted to the discussion and papers presented at the ACM Storage Allocation Symposium, June 23-24, 1961.
3. J. Fotheringham, "Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store," *Commun. ACM* **4,** 435-436 (1961).
4. E. F. Codd, E. S. Lowry, E. McDonough, and C. A. Scalzi, "Multiprogramming STRETCH: Feasibility Considerations," *Commun. ACM* **2,** 13-17 (1959).
5. P. J. Denning, "Virtual Memory," *Computing Surv.* **2,** 153-189 (1970).
6. R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual Storage and Virtual Machine Concepts," *IBM Syst. J.* **11,** 99-130 (1972). In addition to a review of virtual storage technology and developments, an annotated bibliography is included.
7. F. J. Corbató *et al.*, *The Compatible Time-Sharing System, a Programmer's Guide*, MIT Press, Cambridge, MA, 1963.
8. *Emulating DOS under OS for IBM System/360*, Order No. GC26-377, available through IBM branch offices.
9. C. Stratchey, "Time Sharing in Large Fast Computers," *Proceedings of the International Conference on Information Processing*, Paper B.2.19, UNESCO, New York, June 1959.
10. R. A. Nelson, "Problems in Automatic Storage Allocation," *Research Report RC601*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1961.
11. L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Syst. J.* **5,** 78-101 (1966).
12. R. W. O'Neill, "Experience Using a Time-Shared Multiprogramming System with Dynamic Addressing Relocation Hardware," *Proc. Spring Joint Computer Conference* (AFIPS) **30,** AFIPS Press, Montvale, NJ, 1967, pp. 611-621.
13. L. A. Belady, R. A. Nelson, and G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," *Commun. ACM* **12,** 349-353 (1969).
14. L. A. Belady and C. J. Kuehner, "Dynamic Space-Sharing in Computer Systems," *Commun. ACM* **12,** 282-288 (1969).
15. R. J. Adair, R. V. Bayles, L. W. Comeau, and R. J. Creasy, "A Virtual Machine System for the 360/40," *IBM Scientific Center Report 320-2007*, Cambridge, MA, 1966.
16. R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM J. Res. Develop.* **25,** 483-490 (1981, this issue).
17. G. E. Hoernes and L. Hellerman, "An Experimental 360/40 for Time-Sharing," *Datamation* **1,** 39-42 (1968).
18. A. B. Lindquist, R. R. Seeder, and L. W. Comeau, "A Time-Sharing System Using an Associative Memory," *Proc. IEEE* **54,** 1774-1779 (1966).
19. D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Syst. J.* **10,** 168-192 (1971).
20. D. J. Hatfield, "Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance," *IBM J. Res. Develop.* **15,** 58-66 (1972).
21. *Statistical Computer Performance Evaluation*, W. Freiberger, Ed., Academic Press, Inc., New York, 1972.
22. A. S. Lett and W. L. Konigsford, "TSS/360: A Time-Shared Operating System," *Proc. Fall Joint Computer Conference* (AFIPS) **33,** Part I, AFIPS Press, Montvale, NJ, 1968, pp. 15-28.
23. O. W. Johnson and J. R. Martinson, *Virtual Memory in Time-Sharing System/360, TSS/360 Compendium*, available through IBM branch offices.
24. A. L. Scherr, "Functional Structure of IBM Virtual Storage Operating Systems Part II: OS/VS2-2 Concepts and Philosophies," *IBM Syst. J.* **12,** 382-400 (1973).
25. *IBM System/38 Technical Developments*, Order No. G580-0237, available through IBM branch offices.

Received August 2, 1980; revised March 10, 1981

L. A. Belady is located at IBM Corporate Headquarters, Old Orchard Road, Armonk, New York 10504; R. P. Parmelee is located at the IBM Scientific Center, 545 Technology Square, Cambridge, Massachusetts 02139; and C. A. Scalzi is located at the IBM Data Systems Division laboratory, Poughkeepsie, New York 12602.


ISAM
OSAM
Primary
Data
Set
One
Data Base
Record
ISAM
OSAM
Secondary
Data
Set