Presented are early developments of storage management techniques, particularly those used in OS/360. Innovations introduced by systems that use dynamic address translation are traced. The impact of these techniques on current IBM System/370 Operating Systems is described.

Functional structure of IBM virtual storage operating systems Part I: Influences of dynamic address translation on operating system technology

by M. A. Auslander and J. F. Jaffe

During the history of System/360, operating system function has continuously increased. This is reflected in the growth of language, device, and data handling facilities, among others, which provide a rich environment for data processing. Such facilities are currently supported by batch-oriented multiprogramming systems. However, demands for interactive facilities, for larger and larger numbers of concurrent users, and for significant sharing of data among users are growing.

The advent of the System/370 virtual storage systems reflects the changing nature of user needs as well as technological innovations, a trend that becomes clearer when one considers the historical development of the general-purpose operating system.[1-3] The first operating systems were used in a single-user, batch mode where the primary purpose was to automate many of the operator functions. In such an environment, all computing resources not used by the operating system (e.g., Central Processing Unit (CPU), storage devices, etc.) were available to the single running job. Although this is a desirable mode of operation for the programmer, sequential operation causes inefficient utilization of resources, since not all jobs use all resources, and, therefore, a substantial portion of the system resources are idle. Significant increases in CPU power and the introduction of autonomous Input/Output (I/O) that could be overlapped with CPU activity reenforced the need for more efficient utilization of

systems resources. Multiprogramming, or the concurrent execution of several programs, was developed to meet this need. In addition, multiprogramming made it possible to service large numbers of users simultaneously through time sharing.

Stated in terms of the impact of these changes on the systems themselves, the movement has been away from the static preplanned environment, to a more dynamic, interactive one. This, in turn, has placed a greater burden on the system, particularly in the area of resource management.

resource management

Main storage and I/O devices were the first resources to be controlled by the system since they had to be shared between the system and the user. This really was a rather static arrangement. The operating system took what it needed and left the remainder to the user program, merely preventing the user program from overstepping its bounds. As more than one user program became active in a concurrent manner, true *resource management* became necessary for the execution-time allocation of main storage, I/O devices, and the CPU.

With the introduction of large direct-access storage devices (DASD), it became important to share devices among users, and this required a distinction to be made between data and devices. Data management was thus created. As users found that data sets (units of allocatable data) were also a subdividable entity, selected portions of which could be shared among users, the concept of *data-base management* was established.

Each of these concepts has been continuously evolving over the years—becoming more and more refined, providing more and more of the necessary function. Although we could reasonably discuss each of these resources, it is the management of the program address space on which we will focus. For, in fact, the evolution of the System/370 virtual storage systems is in large part the story of the evolution of storage management technology.

The storage management problem

Storage management is a crucial aspect of operating system design because storage is both a scarce resource and a resource that is not easily shared. Storage is scarce for economic reasons, and is difficult to share if the program address space and the actual physical address space in which it resides for execution are considered equivalent. By the use of relocatable loading techniques, initial assignment need not be at a fixed, preplanned location. However, once space is assigned, all program address references are made in terms of the actual physical locations, and these locations must remain available throughout the entire

execution. (References 4, 5, and 6 are given to clarify the distinction being made between "relocation" and "dynamic address translation.")

main storage pre-emptibility

It follows that attempts to treat main storage as a pre-emptible resource cause difficulties. To clarify this point, let us contrast the management of CPU and storage. The sharing of the CPU in multiprogramming is a good example of the pre-emptive allocation of a resource. At any instant, the CPU is serving some program. If that program cannot immediately proceed, or, if a more important program must be executed, the CPU can be pre-empted from its first task and assigned to another task. Thus the CPU can always be assigned to work that is, momentarily, the most important.

Consider the analogous situation for main storage allocation. As a running program needs main storage, it can be assigned from a pool of free space. Assume now that a program needs more space than is free and that another, less important program, has space. The operating system, to let the more important program continue, should allow it to use the space occupied by the less important program. However, the operating system must first save the contents of the main storage occupied by the pre-empted program (much as the CPU scheduler first saves the CPU registers before reassigning the CPU). Moreover, since the two programs now share some common storage, they can never both be available for CPU assignment simultaneously.

If a scheme like this were to be implemented, the interprogram dependencies could lead to situations in which only one program would be available for CPU assignment at a time. Since throughput, system performance, and responsiveness are all dependent upon the system's ability to select the right job (not just a job) from a set of ongoing jobs, the impact of such a limitation is extremely significant.

There are two practical ways out of this situation. The first is to give up pre-emptibility of main storage. However, without pre-emptibility, it is not possible to allow each program to request more storage as needed, due to the danger of deadlock. For example, if two programs are started whose aggregate main storage needs exceed the available space, they may reach a point of conflict in which each is requesting storage, but in which the remaining available space cannot satisfy either request. Without pre-emptibility, since neither program can complete until one has obtained additional main storage, a *deadlock* situation arises. To avoid deadlock, each program must be restricted, beforehand, to a maximum storage size. This solution, in one guise or another, is reflected in the various System/360 Operating Systems (OS/360).

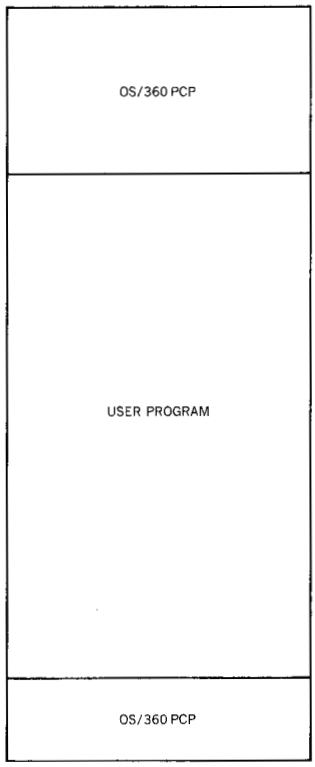
The other way out is to eliminate the need to return exactly the same main storage locations after a pre-emption. This approach—effectively distinguishing between program address space and physical address space reminiscent of the earlier distinction between data and devices as supported by the technique of dynamic address translation—is the very essence of the *virtual storage* systems.

The road to virtual storage systems has been a long one, that has been marked by significant advances. To place these advances into perspective, we proceed by examining the development of storage management techniques in OS/360, and then turn to the path followed by systems utilizing dynamic address translation.

OS/360

OS/360 was originally conceived as a complete multiuser system. The designers recognized the need for dynamic allocation of the computer main storage amongst the system users. As we now understand, it is extremely difficult to attain this goal with the hardware that was then available. Thus, three versions of OS/360 were finally developed: primary control program (PCP), multiprogramming with a fixed number of tasks (MFT), and multiprogramming with a variable number of tasks (MVT). The major difference follows from the storage management schemes that they employ. In all cases, however, the physical address space and the program address space are considered equivalent, and the total usable program address space is limited to the size of main storage remaining after system requirements have been satisfied.

Figure 1 OS/360 PCP main storage map



OS/360 PCP

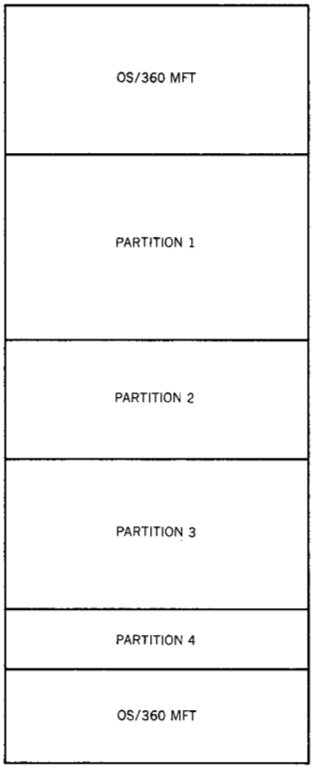
PCP reflected the most drastic simplification in that only one job could be run at a time. Thus neither main storage nor the central processor need be dynamically allocated. The operating system itself occupies a certain portion of main storage, and the remainder is available for the currently running job. No multiprogramming is allowed, and there are no contenders for main storage—therefore, no consideration is needed for allocating the available user space. Main storage allocation in OS/360 PCP is shown in Figure 1. Here is a concrete case of total static allocation. As has previously been pointed out, since most jobs cannot use all of the available resources, significant portions of the system can be underutilized.

OS/360 was never intended to operate this way. Even the design of the job input and output facilities assumed concurrency of spooling and job execution. Concurrency was realized by the introduction of multiprogramming of a restricted kind.

OS/360 MFT

To support multiprogramming, main storage must be shared among several programs. The simplest solution to the storage allocation problem (which could not be handled dynamically) was the permanent partitioning of available main storage into a number of pieces, each of which could support sequential processing of the PCP variety. MFT permits an installation to allocate a fixed number of partitions, each of a fixed size. An individual job is then assigned to one of these partitions for execution. Main storage allocation in OS/360 MFT is illustrated in Figure 2. If a job requires less than the predetermined main storage allocated to its partition, that storage remains unused for the duration of the job. Multiprogramming then takes place over the partitions in use. Thus MFT provides dynamic allocation of CPU and I/O resources among the jobs running in its several partitions. In an environment where the main storage requirements of the installation's jobs are known in advance, are reasonably homogeneous, and remain relatively unchanged, MFT is a reasonable vehicle.

Figure 2 OS/360 MFT main storage map



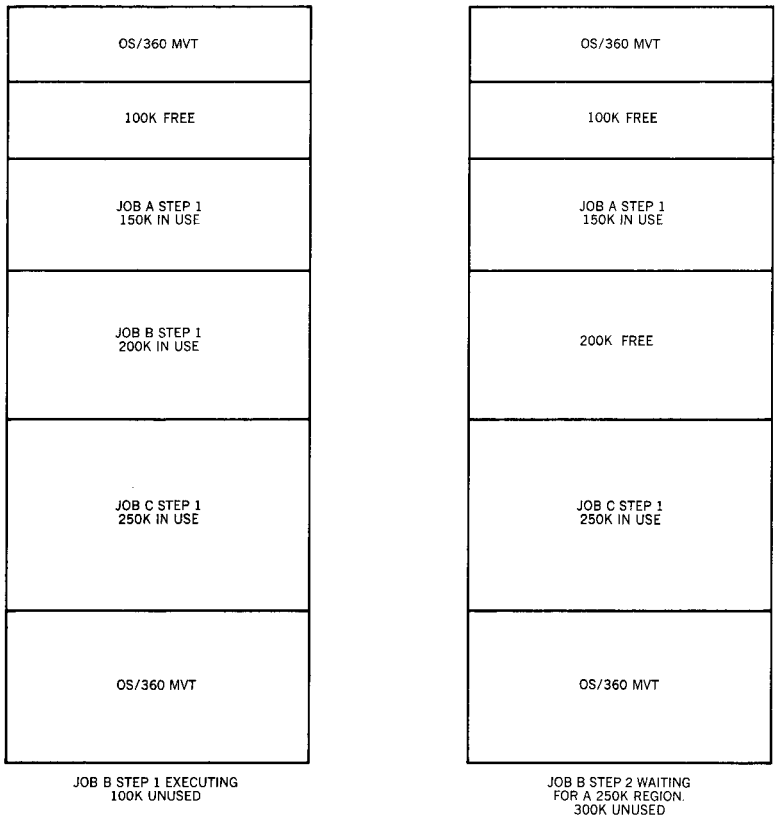
This solution has several important defects. Each job must be designed to limit its peak main storage requirement to the size of its partition. Thus if its requirements vary, it will of necessity underutilize the main storage allocated to it. Since each partition is used serially, a job that must spend a long time idle (e.g., waiting for a volume to be mounted) continues to hold its main storage resource during that idle period. Finally, the available main storage must be quasi-permanently divided into partitions of fixed size. This leads to a compromise between a few large partitions and many small partitions. Large partitions leave the system underutilized when many small jobs are present, and small partitions leave the system unable to service large jobs.

OS/360 MVT

OS/360 MVT, by performing a more dynamic allocation, removes the necessity for making the compromises implied by OS/360 MFT. Using the MVT option, the available main storage (i.e., that which is not used by the operating system) is divided into contiguous storage locations called regions whose size is determined by job requirements. Originally, allocation was done at job initiation time for the largest amount of main storage needed by a job at any point in its execution. Further refinements now permit allocation on a step basis. Therefore, only the amount of main storage needed by the executing step is reserved. Main storage allocation as performed under OS/360 MVT is shown in Figure 3.

The OS/360 MVT technique allows the number of running jobs to vary according to their peak step main storage needs and thus increases hardware utilization. Unfortunately, even this more flexible form of dynamic allocation causes trouble. The problem is that—as mentioned previously—once main storage is assigned

Figure 3 OS/360 MVT main storage map at two instants of time



to a job, not only the space but the actual locations must be reserved to that job. Thus a situation known as *fragmentation* arises in which free main storage consists of small pieces that could, in aggregate, support another job, but which are each too small to be used. Fragmentation is illustrated in Figure 3. Two successive time periods are shown; the left period precedes the right period. During the earlier time period, step 1 of jobs A, B, and C are executing and job B, step 1 completes. Next, on the right, job B, step 2, which requires a 250K-byte region, is ready to execute, but it cannot do so because of storage fragmentation. Observe that, although there are 300K bytes of unused storage, 250K bytes of contiguous storage are not available for the execution of job B, step 2.

In addition, MVT is still plagued with the two other faults of main storage preallocation. The maximum possible instantaneous requirement must be allocated for the duration of each step, and this space remains in use even if a step must wait an inordinate amount of time for some event. (Rollout-rollin, a fea-

ture of MVT that attempts to address this question, is severely limited by the requirement of returning a pre-empted job to its original main storage locations).

OS/360 MVT time sharing option

The introduction of timesharing to OS/360 MVT took place in the form of the Time Sharing Option. In the case of timesharing, responsiveness to user requests and support of multiple users becomes exceedingly important. In cases where users have frequent interactions with the system or effectively use little CPU, the main storage reserved is unused for significant periods of time (while the user is typing in information, receiving information, thinking, etc.). Therefore, the system must share main storage among several ongoing programs. TSO uses a *swapping* technique to achieve this objective. That is, a user's program space is written out to auxiliary storage (drum or disk) when it is inactive, and is brought back into main storage upon his next interaction. (This can also be done when time slicing with several CPU-bound users). However, a user's program—due to the problems discussed earlier in this paper—must be reloaded into the same storage locations from which it was removed. Therefore, even though there are several TSO regions operating, the region to which a user is initially loaded is the one in which he must continue to run.

Dynamic address translation

During the period of increasing function in general-purpose operating systems, significant work was going on in the area of dynamic address translation. Before proceeding into a discussion of these systems and their innovations, let us define *dynamic address translation.*[7] The concept is simply stated as follows:

1. An address space(s) (virtual storage) is defined that may exceed the size of main storage.
2. All main storage references are made in terms of virtual storage addresses.
3. A hardware mechanism is employed to perform a mapping between the virtual storage address and its current physical location.
4. When a requested address is not resident in main storage, an interruption is signaled, and the required data can be brought into main storage.

virtual storage

The Atlas system is generally accepted as the originator of the one-level storage system (virtual storage).[8-10] Thus, Atlas was the first system to establish the distinction between *program address space* and *physical address space.* This concept enables an address to be " . . . an identifier of a required piece of infor-

mation but not a description of where in main memory that piece of information is."[9] By so doing, a natural outgrowth was the ability to make the program address space larger than the physical address space. Thus, programs were no longer constrained to the size of main storage.

Dynamic address translation and the notion of paging were used to implement this facility. *Paging* is the technique of dividing both main storage and program address space into fixed-size blocks in a manner that is transparent to the user. By dividing the program address space into blocks, a reference to any address within a block causes that entire piece to be brought into main storage. This technique dramatically reduces the size of the tables used for mapping virtual addresses to physical addresses. In addition, since programs tend to access many contiguous or neighboring addresses during their execution, the number of *page faults*—or times when an address is not found resident in main storage—is diminished. This, in turn, reduces paging activity and improves performance. By establishing a convention that divides main storage and virtual storage into equal-size fixed blocks, the fragmentation problem is solved. That is, when a page is no longer needed and a new page is required, the new page can replace the old one without leaving sections of unused main storage.

Thus, the Atlas system provided dynamic storage allocation. Dynamic address translation made it unnecessary for pages to be returned to any specific locations, thereby making pre-emption safe, and making possible the support of address spaces larger than main storage. The concept of dynamic address translation solves many of the problems previously discussed.

multiple address spaces

Another major innovation occurred in the IBM M44/44X system.[11] This system introduced the concepts of multiple address spaces and—a natural outgrowth—the "virtual machine." These concepts are at the heart of CP/67 and VM/370. The basic idea here is to allow each user to have a virtual storage space of the maximum possible size. This was done by establishing a unique identifier for each active user that could be associated with a set of *page tables* (maps for address translation). If each user has a separate set of page tables then, in fact, each user has a unique virtual address space. The concept of the virtual machine is really a restatement of this concept in terms of what the user sees—a total set of system resources seemingly unshared with other users. Such a machine looks the same to the user each time he runs a job. The environment is clear and constant.

segmentation

The concept of segmentation was a third innovation in the evolution of dynamic address translation systems. The Burroughs B5000 used a variable size entity—a segment—to contain logical

portions of programs and data and used the segment as the basic unit of allocation.[12-14] Segmentation has two major distinctions from the Atlas type system previously described. The most obvious difference is the fixed-versus variable-size unit of storage allocation. The second distinction is that segmentation introduces a less arbitrary subdivision of a program. That is, instead of dividing a program into fixed-size blocks that are not logically distinguished from each other, programs are divided into pieces that correlate with the user's view of his program's logical construction. He can meaningfully refer to segments by name.

TSS/360 used the notion of segments in quite a different way. In that system, a segment is a collection of one or more fixed-length pages where a page is the basic unit of storage allocation. Whereas segments can be shared and protected as entities (although not directly by name), the primary rationale is an implementation issue, i.e., reduction of the space required for page tables, since only page tables for active segments need be kept in main storage. In addition, although a portion of the file system is also included in the user address space, data and programs are, in the main, handled differently.

MULTICS[15-16] combines the concepts previously described. On the one hand, the entire address space—including data—is contained in named segments, each of which can be protected and shared. On the other hand, a segment is made up of one or more fixed-size pages that are used for storage allocation. Thus both the user concern for dealing with logical named entities and the system concern for minimizing main storage fragmentation are dealt with.

System/370 virtual storage systems

In the System/370 virtual storage systems, we see the convergence of two major trends: the multifunction capability for which the System/360 operating systems have been noted, and a significant portion of the technological advances demonstrated in the various systems that have used dynamic address translation.

The System/370 virtual storage systems are the result of applying the technique of dynamic address translation to the storage allocation problems of OS/360. As we have seen, OS/360 MFT and OS/360 MVT reflect two possible techniques for main storage allocation at the job-step level. Both are ultimately limited by the need to allocate to each job step an amount of real storage sufficient for its most extreme need.

Although the System/370 virtual storage systems provide virtual storage for computation, they have retained the OS/360 file sys-

tem. Thus the persistent storage of the system continues in terms of data sets, volumes, and devices. The retention of the OS/360 data set system, and the ability to do standard I/O, have been realized by implementing a simulation of the dynamic address translation mechanism for the data channels. This channel program translation approach leads to a high degree of compatibility between the nonvirtual storage OS/360 and the virtual storage operating system. Thus many programs written for OS/360 can be carried over with little or no change. The need for this compatibility is the best justification for choosing this approach rather than introducing a segment oriented file system.

OS/VS1 and OS/VS2 Release 1

OS/VS1 and OS/VS2 Release 1 are exploitations, in the OS/360 MFT and OS/360 MVT architectures, respectively, of dynamic address translation.[17,18] By applying these techniques, a single main storage of maximum possible size (i.e., 16 million bytes) is simulated, and job step oriented allocation is performed within this virtual storage. Because the main storage is so large, this fixed allocation is expected to be able to serve the needs of most installations.

Of course, the System/370 configurations do not contain 16 million bytes of main storage. Rather, the physical storage is dynamically (on an instant by instant basis) assigned to support the actual computing needs of the job in progress. A job that requests 200,000 bytes and uses only 100,000 bytes, reserves 200,000 bytes of virtual storage but uses (at most) 100,000 bytes of real main storage. If several job steps are running concurrently, real main storage is assigned first to one, then another, as the real computing demands require. Thus truly dynamic main storage allocation, never practical on System/360 hardware, is now functioning.

As we have seen, the single virtual storage provides true dynamic main storage allocation. For job mixes previously run in smaller stores, the sharing of 16 million bytes normally causes no problem. However, 16 million bytes can be a limitation. This is particularly true if some of the workload is interactive. In such a case, a number of jobs—all using parts of virtual storage, but normally dormant—can consume the 16 million byte space. (The problem is rare without some form of normally dormant job, since a collection of active jobs that fill the virtual storage usually overloads other system resources).

A good example of the interactive use of OS/VS2 Release 1 is TSO. If each TSO user were given a private section of the address space, the initial space might be consumed. Thus TSO is realized through the use of time-sharing regions, much as it was in OS/360 MVT. These regions are shared through a form of virtual swapping.

Another problem with sharing a single virtual storage is that programmers must still act to conserve the addressability used by their programs. If they do not, several programs that should be able to run concurrently in the single virtual storage may not fit.

The solution to these weaknesses, and ultimately the most natural solution to the storage allocation problem, is the multiple virtual storage approach of OS/VS2 Release 2. As we have previously described, computer architects have expanded on the Atlas idea by designing systems in which each user has his own virtual main storage. Again, the hardware/software techniques of dynamic address translation and paging provide real main storage for just those sections of virtual storage that need to be accessable.

In OS/VS2 Release 2, this approach is used to provide each job step with its own virtual PCP environment, wherein PCP exists in an apparent 16 million bytes. All the storage space not allocated to control program use is available to user programs and there is minimal cost for the possibility of using it. Significant cost comes only for its actual use. Thus the programmer does not have to limit his use of addresses to make multiprogramming possible. Beyond this, he sees the same sized address space, no matter what the current resources and work load of the installation.

With this advance, TSO can be realized by providing a separate virtual storage for each user, just as for any other job. When the TSO user is thinking, the control program releases real main storage allocated to his job for use by other, more active jobs.

OS/VS2 Release 2

In this manner, OS/VS2 Release 2 supports the shared use of a System/370 computer. Each sharer can request CPU, main storage, and I/O paths as needed, leaving the operating system to allocate the available hardware in support of these requests as required.

Paging implementation considerations

The idea of paging is straightforward and solves many problems. It makes dynamic storage allocation safe, and allows the programmer to ignore the constraint of available physical storage size. If the technique is to be useful, however, it must not lead to unacceptable cost.

When a program is paged, the operating system controls the movement of pieces of the program and its working data between main storage and secondary storage. This mechanism

serves to replace the strategies of overlay and spill files that are used to fit programs into conventional systems. The standard approach to realizing paging decisions is to collect information about the characteristics of programs as they run. Broadly speaking, the system attempts to keep available those pages of the program and its data that are used frequently, and to send unused sections to secondary storage. The system also attempts to control the level of multiprogramming so that each active program has enough pages in main storage to run efficiently.

The details of page replacement algorithms have been studied[19] during the development of paging systems, and techniques evolved for making these decisions reasonably well. The success of a paging algorithm, however, still depends on the programs it is dealing with. As an example, it is possible to write a program in a way that makes the amount of main storage needed for efficient execution greater than the main storage available. When this happens, that program usually performs poorly. The best a good paging algorithm can do is to prevent that poorly written program from affecting the other users of the system.

In addition to that of page replacement technique, an issue that has been long discussed is the optimal page size. As page size increases, more data are transferred with each page, and fewer page faults occur. However, larger pages lead to increased traffic between main and secondary storage and an increased amount of wasted main storage. Studying these issues is made more difficult because the page size of a machine is usually fixed by its design, making comparison experiments difficult.

The practical aspects of paging do lead the programmer to consider the fact that his program address space is being paged. This is particularly true if his program strains the capacity of the system. In this case, the programmer must remember that only a certain portion of his program address space is in main storage at any instant and design his program to work under that constraint.

Concluding remarks

The operating system started as a means for controlling the sequential use of a computer more efficiently than a machine operator could do himself. As computers have grown larger, faster, and more able to support independent concurrent operations, operating systems have grown to provide for the simultaneous shared use of such systems. Thus multiprogramming, originally developed to make still more efficient use of the hardware, has become essential in its own right with the development of interactive computing.

IBM operating systems have followed this evolution with a series of increasingly sophisticated multiprogramming mechanisms. We have shown that these mechanisms can be characterized as resource managers, and that the management of main storage was a perennial problem for OS/360 caused by its inability to distinguish program address space from physical address space. The introduction of dynamic address translation in System/370 has provided a solution to the storage allocation problems of OS/360. Thus the initial effect of this feature is an operational improvement for current workload. Virtual storage, however—just as multiprogramming—can also be a virtue in itself. As programmers use the increased capabilities of the virtual environment, they should find it easier to accomplish their goals. This effect derives from their freedom to ignore many of the space limitations that complicate programming in limited storage. The consequences of this change may turn out to be the real story of virtual systems.

CITED REFERENCES

1. S. Rosen, "Electronic computer: a historical survey," *Computing Surveys* **1,** 1, 7–36 (March 1969).
2. S. Rosen, "Programming systems and languages 1965–1975," *Communications of the ACM* **15,** 7, 591–600 (July 1972).
3. R. F. Rosin, "Supervisory and monitor systems," *Computing Surveys* **1,** 1, 37–54 (March 1969).
4. B. W. Arden, B. A. Galler, T. T. O'Brien, and F. H. Westervelt, "Program and addressing structure in a time-sharing environment," *Journal of the ACM* **13,** 1, 1–16 (January 1966).
5. G. O. Collins, "Experience in automatic storage allocation," *Communications of the ACM* **4,** 10, 436–440 (November 1961).
6. W. C. McGee, "On dynamic program relocation," *IBM Systems Journal* **4,** 3, 184–199 (1965).
7. P. J. Denning, "Virtual memories," *Computing Surveys* **2,** 3, 153–189 (September 1970).
8. C. H. Devonald and J. A. Fotheringham, "The Atlas computer," *Datamation* **7,** 5, 23–27 (May 1961).
9. J. Fotheringham, "Dynamic storage allocation in the Atlas computer, including the use of a backing store," *Communications of the ACM* **4,** 10, 435–436 (November 1961).
10. T. Kilburn, D.B.G. Edwards, M. J. Summer, and F. H. Summer, "One-level storage system," *IRE Transactions on Electronic Computer* **EC-11,** 2, 223–235 (April 1962).
11. R. W. O'Neill, "Experience using a time sharing multiprogramming systems with dynamic address relocation hardware," *AFIPS Conference Proceedings, Spring Joint Computer* Conference **30,** 611–621 (1967).
12. Burroughs Corporation, "The descriptor—a definition of the B500 information processing system," Detroit, Michigan (1961).
13. W. Lonegram and P. King, "Design of the B500 system," *Datamation* **11,** 11, 24–28 (November 1965).
14. F. B. MacKenzie, "Automated secondary storage management," *Datamation* **11,** 11, 24–28 (November 1965).

15. A. Bensoussan, C. T. Clingen, and R. C. Daley, "The MULTICS virtual memory, concepts and design," *Communications of the ACM* **15,** 5, 308–318 (May 1972).
16. J. Dennis, "Segmentations and the design of multiprogrammed computer systems," *Journal of the ACM* **12,** 4, 589–602 (October 1965).
17. See the article by A. L. Scherr in this issue.
18. See the article by J. P. Birch in this issue.
19. L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal* **5,** 2, (1966).