Enterprise Systems Architecture/370: An architecture for multiple virtual space access and authorization

by C. A. Scalzi
A. G. Ganek
R. J. Schmalz

The Enterprise Systems Architecture/370™ provides a significant step in the IBM System/370 evolution by providing new capabilities for virtual addressing and program linkage across multiple address spaces. This paper reviews the evolution that led to this advance and illuminates the goals, such as eliminating growth constraints and improving security, integrity, reliability, and performance, that have guided it. The major architectural capabilities are discussed, along with the system environments in which they are useful. The rationale for design choices is presented and related to issues of performance, access authorization, and constraints relief.

In the late 1970s and early 1980s, attention within IBM focused on extending the System/370-XA architecture to enable continued evolutionary growth of System/370 systems, as required in light of emerging customer requirements and rapidly changing hardware and operating system technologies. The Enterprise Systems Architecture/370™ (ESA/370™) was developed in response to these requirements and trends. The development of more powerful processors was already underway and the continuing evolution of processors of ever-increasing speed was anticipated. As the processor speed grows, larger workloads can be handled and more data must be available to keep the processor highly utilized. In fact, without sufficient data available to the system in high-speed storage, processor power could be under-utilized. The high probability that large, cost-effective electronic storage hierarchies would be generally available on such future processors made the requirement an even stronger one. (The implications of electronic storage hierarchies are discussed further by E. I. Cohen, G. M. King, and J. T. Brady in this issue.[1]) In addition to these hardware trends, new approaches in operating system structure were evolving not only to exploit the advances in hardware, but also to meet customer requirements for data sharing, constraint relief, data integrity, performance, and usability in large systems. The ESA/370 architecture was developed considering all three of these factors—customer requirements, hardware trends, and operating system structural issues.

Evolution

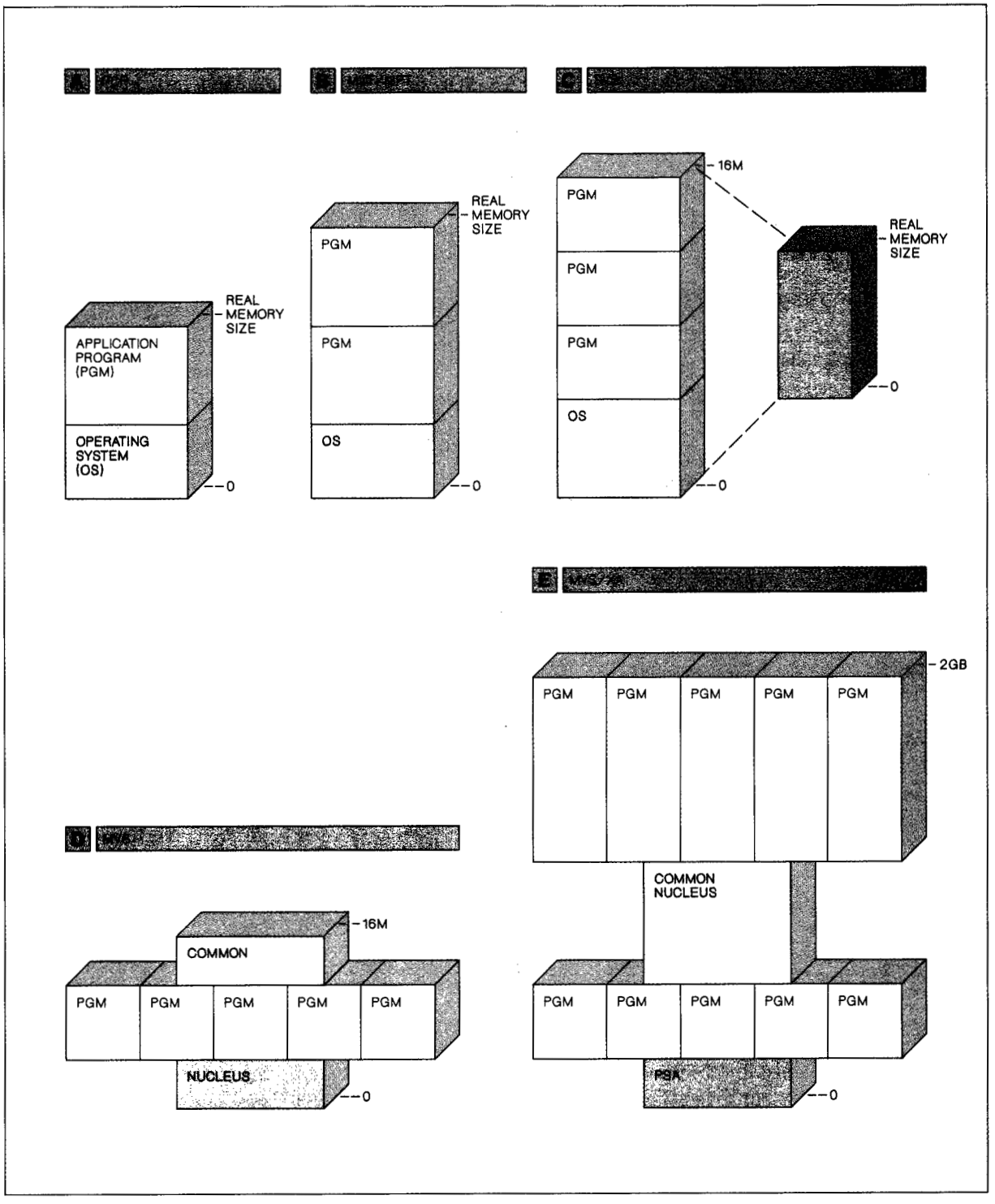
In the development of System/370 architecture, there was a commitment to the idea that the use of virtual addressing in programs would provide a ve-

hicle for the automatic adaptation of programs to large and multiple-level storage hierarchies. These can be used to provide increased performance without reprogramming, as electronic storage was increased or new physical levels were added. To understand the relationship of storage addressing and access authorization to operating system structure and architectural requirements, it is useful to review the evolution of the addressing structure in System/360 and System/370 systems. In the mid-1960s, System/360 operating environments allowed one program to execute at a time, with an addressing capability that was precisely equal to the available main storage on the machine. The environment then was essentially the same as that of many personal computer systems today, in that job management is left to the human operator, and each job must wait for the prior one to complete. As is indicated by the primary control program (PCP) model in Figure 1A, only the operating system code and one application are loaded into memory at one time. This mode requires only one distinction of authority: either privileged- or problem-program authority. This dedication to a single application is an inefficient use of resource because it leaves the processor idle during input/output processing and between jobs. Figure 1B shows how developments designed to overcome this shortcoming led to the multiprogramming with a fixed number of tasks (MFT) and multiprogramming with a variable number of tasks (MVT) operating systems, by providing multiprogramming environments. Multiple programs are loaded into memory concurrently to share both the processor and memory resources of the system. In this way, when one task requires input/output processing, that task can be suspended until the I/O completion. Meanwhile, another task can be executed. To insure integrity, this arrangement requires a protection mechanism to constrain the addressing of each task to the partition of memory allocated to it. In System/360, protection is accomplished with a storage key architecture, which allows for 16 different protected areas and a control program access key (key zero) that allows addressability to the entire range of memory.

Although multiprogramming has made computer processing more efficient, it has also created much more demand for memory.

Although the introduction of multiprogramming has made computer processing far more efficient and has provided effective utilization of the processor, it has also created much more demand for memory. Thus the available memory had to be partitioned for each of the concurrent tasks. Therefore, the addressability of each task was reduced. The resultant memory constraint limited the size of programs and the amount of data that could be associated with them. This left the users of the system with difficult tradeoffs to manage for balancing the number of concurrent programs and the addressing capacity available to them. This motivated the development of System/370 and the virtual storage system depicted in Figure 1C. In this single virtual storage (SVS) operating system, the memory available to all of the tasks was increased via the use of virtual storage. Virtual storage provides the appearance of more memory by supplementing main memory with high-speed external storage and a paging mechanism. With virtual addressing, a processor performs the instruction if it and its operands are in main storage. If any of the required elements are not in main storage when the instruction is executed, the processor signals the operating system. This process is called a *page fault*. The operating system makes the missing element(s) available in main storage and causes the processor to re-execute the instruction. The program containing the instruction is not involved in the management of the content of main storage and is written as though all instructions and data are resident. The processor and control program, in cooperation, can translate the virtual addresses and find the page frame that contains the desired element in main storage. The processor and the control program also cooperate in keeping track of which elements have been most recently required for processing. This information is used by the operating system to manage the content of the levels of the storage hierarchy for maximum performance benefit in throughput or response time, or to meet installation priorities. This approach in SVS allows the addressing structure to equal the full architectural limit of System/370, which is 16 megabytes for linear addressing.

Figure 1 Evolution of storage maps: (A) primary control program (PCP); (B) multiprogramming with a variable number of tasks/multiprogramming with a fixed number of tasks (MVT/MFT); (C) single virtual storage (SVS); (D) Multiple Virtual Storage (MVS); (E) Multiple Virtual Storage/Extended Architecture (MVS/XA)



SVS provides concurrent addressability to as much as 16 megabytes in a single mapping of all data and programs. Such a single linear mapping of virtual addresses is called an *address space.* In that operating system, the requirement for multiprogramming still requires that the single virtual address space be divided up among the concurrently executing programs, thereby limiting the amount available for any one program. This problem was addressed by the development of operating systems that give each user a virtual address space that is independent of that of other users. Such a structure is depicted in Figure 1D, which shows the Multiple Virtual Storage (MVS) operating system. In this case, the addressing range is divided into segments that are common to all users and contain programs and data relevant to many users. There are also segments called the *private areas* that are relevant only to individual users and therefore need not be commonly addressable. In the VM operating system, users are similarly segregated into the independent memory of their virtual machines. By using virtual addressability, these multiple-address-space structures provide more virtual storage per user and the opportunity to extend protection beyond the limit of storage keys.

By using virtual addressability, multiple address-space structures provide more virtual storage per user.

As multiple-address-space structures were developed, subsystem functions—such as the job entry subsystem and the Information Management System (IMS) database subsystem in MVS—began to exploit the structure to provide function for many users. The MVS subsystems adapted to the multiple-address-space structure of MVS in the System/370 time frame, in some cases by maintaining control information and data buffers in private areas available only to them. Access is generally accomplished by their code operating in the common area. Many operations to be performed require movement of data to or from the subsystem private area and a user's private area. Because of this, mechanisms for data passing and program linkage across address spaces are necessary. The dual address space (DAS) facility was added to the architecture to facilitate such operations. Two virtual address spaces can be made known to a processor simultaneously and authorized programs can easily switch from operating with *primary* space addressability to *secondary* space addressability, thereby making data in either a subsystem's space or its caller's space easily accessible. To facilitate the movement of data from one space to the other, two new MOVE instructions were added to System/370 to move data directly from primary to secondary or from secondary to primary. Otherwise, the data would have had to be moved to the common-addressed area en route between the two private address spaces. In addition, synchronous linkage instructions were provided to enable programs to be located in the private area of an address space and accessed from other address spaces. This allowed service functions to be removed from common storage and implemented in independent address spaces, thus reducing the virtual storage impact of service functions on other address spaces and improving the level of protection of the programs and data associated with them.

While DAS improved the usability of multiple address spaces, a 16-megabyte address space continued to be a constraint in many environments. The System/370-XA architecture was introduced to increase the size of address spaces in this structure. System/370-XA provided a 31-bit virtual address that extended the size of address spaces from 16 megabytes to 2 gigabytes. As depicted in Figure 1E, the MVS/XA™ structure contains larger address spaces than its System/370 forerunner, but the mechanisms to use multiple address spaces concurrently are unchanged. The facilities of DAS were carried forward in System/370-XA to provide dual address-space capabilities for the larger spaces.

Requirements

As we have seen, the evolution of System/360 has continually been guided by the system goals of eliminating growth constraints; improving security, integrity, and reliability; and providing improved performance. The objective of Enterprise Systems Architecture/370 (ESA/370) is to continue this process by enhancing system capabilities in the areas of virtual addressing, isolation and protection of code

and data, and program linkage. To understand the ESA/370 solution, we examine each of these capabilities in light of the current function, the motivation for change, and the nature of the change or new function required.

Extended addressability. Current System/370-XA architecture allows a virtual addressing capability of up to 2 gigabytes (2 billion bytes) derived from a 31-bit virtual address that maps storage in a single MVS address space. Although there can exist many such address spaces, only one is generally addressable at any point in time. The dual address space (DAS) architecture inherent in System/370-XA provides a limited capability to access concurrently an additional address space. However, this access applies only to data movement from one address space to another. The balance of the System/370-XA instruction set applies only to a single address space, thereby limiting a program's addressability to 2 gigabytes.

From the application programmer's perspective, it is often highly desirable to map an entire data structure into virtual storage at one time.

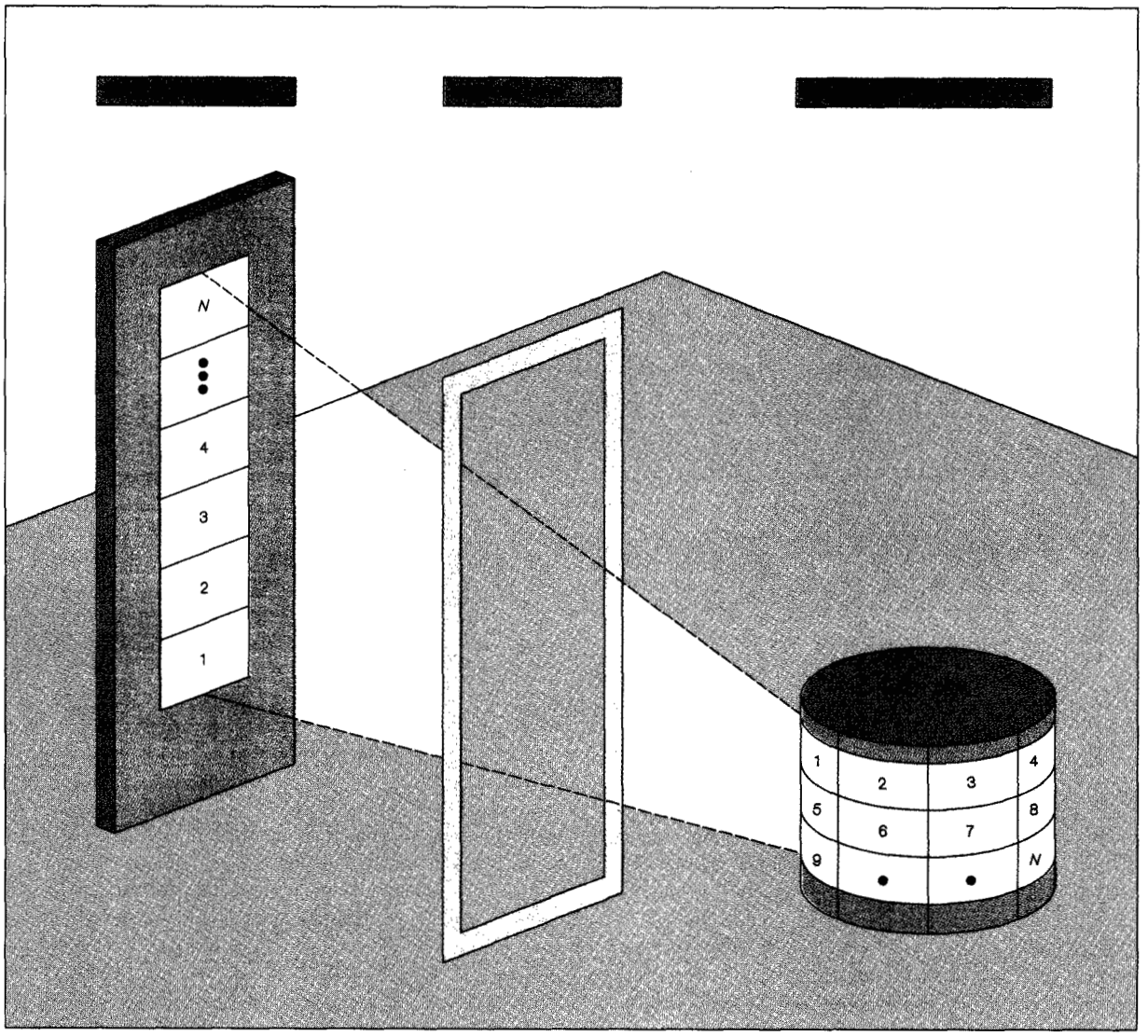
To understand the motivation to increase virtual addressing capacity, we must first examine the kinds of things that consume virtual storage. One of the most important of these is control blocks. System and subsystem control blocks consume large amounts of virtual storage. Control block storage generally increases proportionally with entities such as the following that correlate to workload per: terminal, user, transaction, database, device, and application. Consequently, virtual storage limitations that restrict the amount of control-block consumption can place a ceiling on workload growth regardless of processing power or direct-access storage device (DASD) configuration. This phenomenon of virtual storage constraint can in turn severely limit the usefulness of high-performance processors and improved storage devices.

In the System/370 systems, permanent data normally reside in datasets on external storage devices, such as disks and tape devices. In the context of permanent data, virtual storage is primarily used as a buffer area to make permanent data addressable to the processor. As applications increase the size and complexity of the data structures they deal with, the required buffer areas grow commensurately, requiring larger virtual buffers to contain the data. Graphic representation of three-dimensional objects, image processing, and large arrays related to numeric computation are examples of the many types of applications requiring large virtual storage buffers. From the application programmer's perspective, it is often highly desirable to map an entire data structure into virtual storage at one time, rather than deal with the complexity of overlay data or complex access methods for explicitly managed I/O. However, as a program obtains virtual storage for the variety of entities it requires within its address space, the available space can easily become fragmented, constraining further the ability to take advantage of large contiguous areas of virtual storage.

As the function of data processing applications evolves to meet ever-broadening requirements, the complexity and sophistication of the underlying programs expand. This results in the necessity for very large programs and libraries of many programs to be available to provide the needed function. As these programs are executed, they are brought into virtual storage for access by the processor. To achieve high performance and to avoid complex overlay structures, it is highly desirable to keep these programs in virtual storage. The accumulation of programs can consume substantial amounts of virtual storage. Because many programs are of value to multiple users, it is desirable to keep them in commonly addressable ranges of virtual storage, so as to minimize the usage of real storage. In the MVS operating system, this mechanism is the common area, and in VM it may be implemented with shared segments. In either case, the accumulation of shared programs can reduce the amount of privately addressable virtual storage available.

The concept of mapping permanent or temporary data in virtual storage is called *data in virtual* (DIV) and allows the application developer an interface to data without coding any record-oriented I/O protocols, but instead to use virtual addresses as an access

Figure 2 Data-in-virtual usage



method. This capability increases the demand for virtual addressability by augmenting its functional utility. The key aspects of this concept, as depicted in Figure 2, are as follows. First, data are stored on external devices in a format that corresponds to page size. When the dataset is *opened* or identified to the system, it is assigned a virtual storage range to map the dataset, although no data are brought into processor storage immediately. The application program can then reference the data by virtual address, which, in turn, causes page faults for any addresses not previously primed with data. The page fault process handles all I/O transparently to the user, brings the page into real storage so as to represent the user's specified virtual address and then re-executes the user's instruction to operate on the virtual storage. When the dataset is *closed* by the user, all changed pages in the virtual storage range may be written out to the permanent DASD locations, if the user so chooses.

There are many advantages of the data-in-virtual data management approach. In this issue, K. Rubsam (Reference 2), describes the implementation and

benefits more completely than we do here. Briefly, a key value is that the user's view of data is greatly simplified. The normal instruction set is available to manipulate any data items of interest. From the user's perspective, all required data are immediately addressable at one time, and program logic need not be encumbered by considerations of data buffering and I/O protocols. This view of data eliminates dependencies on the specific characteristics of DASD hardware that can allow both easier adaptation of data residence to new technologies, and better usability for end users.

Also, from the standpoint of performance, there is substantial potential for reduced I/O overhead because, with DIV, only referenced data are read into main storage. Many applications need reference only small portions of an entire data structure, but those portions are determined dynamically during computation. Without DIV, the entire structure must be read into main storage to establish addressability to the data in virtual storage. This results in needless I/O operations for those pages of the file that contain records that are not referenced. In addition, the integration of data management services and processor storage management can enable more efficient utilization of the electronic storage hierarchy. The attractiveness of the data-in-virtual concept can, of course, be limited by constraints on the amount of virtual storage available. As large databases and other large data structures become mapped virtually, the requirement for extended virtual addressability increases.

Since all of these factors combine to consume virtual storage, resultant virtual storage constraints can adversely affect systems by precluding the addition of new applications and databases. Also, limited virtual storage can restrict the amount of virtual space for control blocks and tables, a fact that constrains the size of system configurations that can be handled. It is evident that virtual storage is a precious resource that is limited architecturally in the System/370-XA environment and therefore is a system constraint. Such potential for constraint generates the requirement for an open-ended addressing capability that allows increased exploitation of virtual storage. If a program is to utilize virtual storage effectively, the entire instruction set must be available to manipulate all storage available to the program. The objective of ESA/370 is to extend the addressing environment to allow concurrent access to multiple virtual spaces with the full richness of the instruction set. The intention is to provide an environment in which the logical entities of interest to a program can be segmented into multiple virtual spaces, each of which can be up to 2 gigabytes in size, and to allow concurrent access to this entire set of virtual spaces. Here we call attention to the fact that the virtual spaces in this type of structure need not all represent as much as 2 gigabytes, but in many instances they could be smaller.

Isolation and protection. The basic System/370-XA protection mechanism is the storage key by which each block of storage (4096 bytes) has a four-bit key value of 0–15 associated with it. On any storage update, this key must match with the associated access key in the program status word (PSW), with the exception of PSW key zero, which allows the program to access storage in any key. This architecture, therefore, provides for a storage protection granularity of just 16 levels.

Storage in the private areas is protected from other users by not being addressable through dynamic address translation.

The structures of most System/370 operating systems are based on multiple distinct address spaces. Storage in the various private areas is protected from other users by virtue of not being addressable through dynamic address translation (DAT). Conversely, any virtual storage that is made virtually addressable to multiple users is limited in granularity to 16 keys for protection.

With respect to isolation, the separation of users into distinct address spaces enhances protection and integrity by making them mutually exclusive of one another during execution. The unit of protection becomes the address space instead of storage keys. Within any one address space, the storage keys are used to protect data areas. System and subsystem functions use storage keys to provide integrity for their data areas in both the commonly and privately addressable address spaces. This enables the control

program to place various control information that is relevant only to a particular user in that user's address space and at the same time prevent the user from modifying it. Localizing control information in this way further isolates errors at the address-space level. This approach has been found to increase the probability that errors can be contained to single users, thus reducing the probability of impact to other users.

Localizing control information in this way further isolates errors at the address-space level.

Using System/370-XA architecture, systems can do a good job of isolating users from one another by way of multiple address spaces. Because each user's code and data reside in a distinct virtual address space, addressability does not allow interference with other users. If the isolation of users in today's structure can be effectively achieved, one may well ask why enhancements in this area are required. The answer is resource sharing. Individual users must share certain physical resources such as CPU, channels, real storage, direct-access storage devices (DASD), printers, etc., as well as certain logical resources like programs, data, system functions, and subsystem functions. Sharing system and subsystem resources is implemented by system components and subsystems whose code and data areas exist in virtual storage. Such programs and data areas must either reside in commonly addressable storage available to all users, or they must reside in separate address spaces and execute in a multiple-space environment. These alternatives apply equally well to subsystem-oriented functions that may provide services for a subset of the system's user population.

System components and subsystems that reside in commonly addressable storage not only aggravate the virtual-storage constraint problems, discussed previously, but also limit the granularity of protection to key protection. A single addressing range contains different entities in virtual storage, all of which represent some form of the code and data of the operating system, the subsystems, or the users. Because there are many more such resource categories residing in a given address space than 16 keys, the storage key mechanism provides an inadequate level of protection granularity, even for the environment of a single address space with current System/370 structures. Errors in programs that either execute in key zero or in a key shared by other programs in the same addressing range can potentially result in damage to data or code associated with other functions or the currently addressable private area.

The alternative of implementing shared functions in a multiple-space environment is hampered by architectural limitations inherent in System/370-XA with respect to concurrent use of multiple address spaces. Such limitations not only affect the existing multiple-space subsystems and system components, but also discourage other components and subsystems from new designs that might achieve greater isolation and protection in a multiple-address-space environment.

The concept of functional isolation is to limit as much as possible the errors of any function to its own data or to data passed to it by its caller. This technique is called encapsulation or error confinement. DAS enables this by allowing code and data related to a function to be placed in a unique address space and by providing the program call instruction in another address space to invoke the code. Using DAS, the caller's parameters can be copied from the caller to target address spaces, and the output of the function can be copied back from the functional space to the caller.

One major difficulty with using cross-memory services is the limited instruction set that the called function has available to manipulate the caller's parameters and data. A short parameter list can be completely copied across spaces, but it can be very clumsy if, for example, the parameter list is of varying length. If the caller needs to supply large data areas that are impractical to copy, the called routine must execute in commonly addressable storage and switch addressability back and forth between the calling and called spaces. This is also true if the called program must manipulate data objects that are imbedded in a complex data structure, making them difficult to locate and move.

Another alternative is to place the data in commonly addressable storage, but the utilization of code and

data in common storage in this manner undermines the benefit of a multiple-space structure for enhanced protection and virtual-storage constraint relief. It also introduces the need to fragment code artificially into commonly and privately addressable segments, thus increasing the complexity of programs.

Another limitation of the DAS architecture relates to the authorization mechanisms. The DAS mechanisms in MVS require the invoked function to have the ability to address the caller's address space to access parameters. Thus the DAS architecture requires that the called function be given access authority to all potential callers' address spaces. Because DAS authorization is on the address-space level, any code that executes in the invoked function's address space can establish addressability to any of its potential callers' address spaces and can reference or update data in that space. This means that an error in a shared program residing in its own address space can corrupt data in any potential caller's address space at any time. The possibility of such errors precludes this type of implementation for shared programs that are not authorized and trusted.

System reliability can be improved if error confinement can be achieved by limiting addressability to those required objects only.

Whereas authorized addressability is warranted in this DAS environment, it is really required only for the duration of the function being performed and for the data specific to that caller's request. In essence, the caller should pass addressability to the function and withdraw this capability when control is returned. In this way, addressing control is associated with the executing process that is making the functional request, rather than the function being invoked. DAS does not allow this capability.

The passing of the authorization to update a data area to a shared program that is invoked in a multiple-space environment, instead of each shared function possessing only a static set of capabilities—regardless of the caller—is an objective of ESA/370. This approach is conducive to an environment in which problem-state unauthorized programs may be safely shared.

Another restriction imposed by the address-space-level authorization architecture of DAS is the inability to support multiple programs in one address space, with distinct addressing capabilities to access other address spaces. This precludes programs that require different levels of authorization from coexisting in one address space. An objective of ESA/370 is to allow authorization granularity at the functional level appropriate to the application. To meet this requirement, architecture and operating-system support must be flexible enough to allow data isolation and protection to be provided at different levels within an address space.

The basic notion of storage isolation of code and data suggests that all addressable entities in a computer system be divided into clearly defined units called *objects*, and the authorization mechanisms of the system be such that any program executing in this environment can access only those objects appropriately authorized to that program. The basic premise of this approach is that system reliability can be improved if error confinement can be achieved by limiting addressability to those required objects only.

The previously described requirement for concurrent addressability to multiple address spaces to extend the amount of addressability, can also be used to achieve enhanced isolation and protection, if it can be combined with authorization mechanisms that control addressing capability. In describing authorization capabilities, it is useful to introduce the term *domain*, which is an isolated unit consisting of code and data relating to a specific function or related group of functions. A domain must be protected from unauthorized access from outside the domain, but yet must be capable of providing services to callers outside the domain and must be capable of invoking function provided by other domains. In the context of ESA/370, a domain is implemented as an independent *functional address space* that provides services that can be invoked by work units in other address spaces. The domain achieves isolation by the unique addressability associated with its address space and its authorization to other address spaces.

Authorization capabilities for domains can be accomplished by extending and refining the capabilities introduced with DAS for invoking shared programs residing in an independent address space. Depending on the size or logical structure of the function, a program might require additional address spaces to contain objects relevant to the function. Authority becomes activated on entry to the address space via a program call instruction. This technique allows access privilege to be provided on a module or functional basis.

Another authorization requirement calls for the ability to provide access privileges to objects at the granularity of the work unit or process. The access privileges associated with such processes are not associated with domains, but rather can be passed to domains or can be restricted from access by domains, as required. A simple example of an object that would be process-related is a virtual buffer containing information entered at a terminal by a user in making a database retrieval request. What the terminal operator enters on the screen might be relevant only to that user. By contrast, access to the database itself might be associated with the domain of the database management function and must be denied to programs not executing in the database management domain. Conversely, the database manager does not require access to the user's virtual buffer unless invoked by the user and passed parameters identifying it.

An objective of ESA/370 is to provide an authority mechanism with the flexibility to allow distinct access privileges to be associated with both processes and functions, and yet allow the intersection of these privileges to be dynamically provided where appropriate. This allows objects to be isolated by a controlled authorization technique, yet made available when appropriate to various functions.

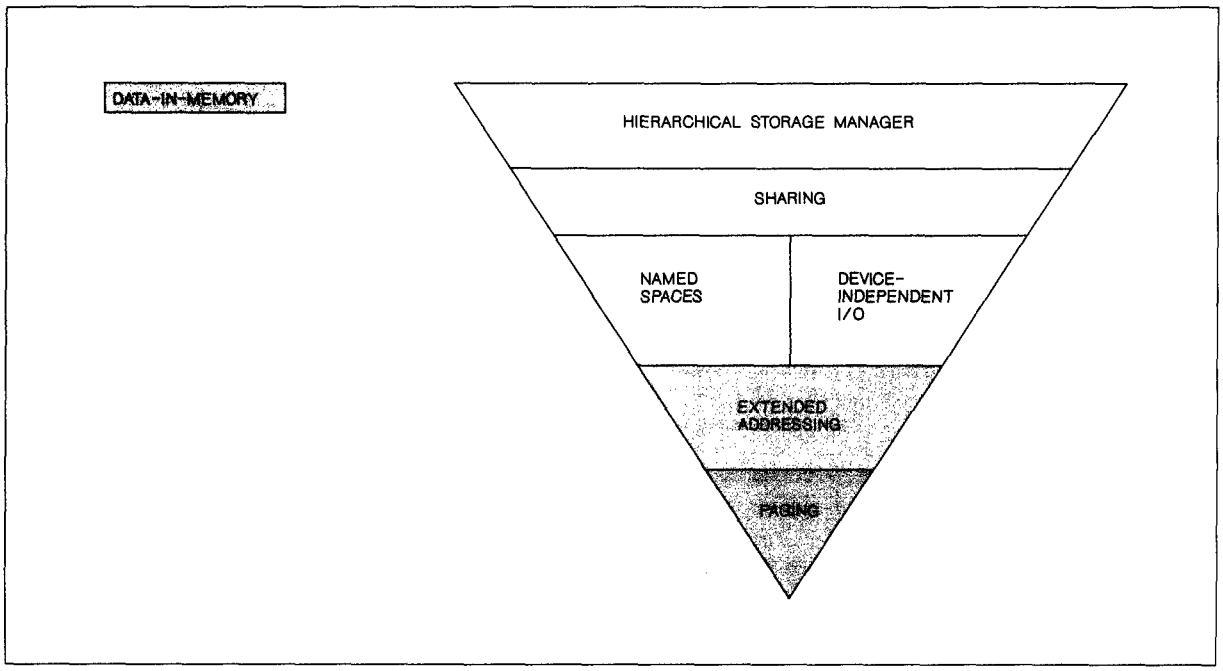
The ability to share objects in virtual storage is provided by the current System/370-XA architecture. DAS authorization mechanisms allow multiple users to share code or data in the private area, and common area objects have always been entities that could be shared in System/370-XA. The objective of ESA/370 is to maintain the ability to share in this manner, while at the same time enhancing the facilities for addressing and controlling access to objects in virtual storage. The concept of sharing virtual objects is particularly well-suited to the data-in-virtual idea discussed earlier in this paper. The sharing of access to permanent data on direct-access storage devices (DASD) can be implemented with good performance characteristics and reduced real-storage requirements, if asynchronous processes can share the same virtual mappings of data instead of using distinct virtual mappings for each user that must be synchronized when updates are made.

Linkage requirements. The requirement for register and status handling in the context of interaddress space linkage is to remove the job of register and status saving from the user and to have the system provide this function. Any such system function, of course, must have good performance characteristics. Ideally, from the user perspective, the calling mechanism would pass the caller's registers to the target program unchanged, yet not require the target program to save register values or state information. Obviously, to achieve this, some enhancement to the calling mechanism is required when control is returned. The technique must also allow the user to return parameter values across the interface.

The state information associated with a process defines its environment within the system at any point in time.

The state information associated with a process defines its environment within the system at any point in time. Users require the ability to invoke functions that execute in different environments. In general, any program should be concerned only with the capabilities of its own environment and the interfaces required to invoke other functions. The nature of the state transitions that can occur between functions should not create a burden to be placed on the functions themselves. Thus the fundamental requirements for managing state transitions are the following: the ability to define an interface to a function, the ability to determine the environment in which the function will run, and the ability to restore the environment of the caller when the function has completed execution.

Figure 3 Data-in-memory based on Large Systems Requirements for Application Development report



Authorization considerations encompass a number of items in System/370-XA, including problem/supervisor state, PSW key, PSW key mask, and address-space authorization. With respect to these aspects of authorization, as well as to new ones that govern address-space authorization, linkage transitions must be allowed that increase authority, decrease authority, leave authority unchanged, and change authority in a nonhierarchical way.

Customer requirements. An informed customer view of emerging requirements was documented in Reference 3. The Large Systems Requirements for Application Development (LSRAD) Task Force was organized to "propose evolutionary operating system enhancements to improve the usability of both MVS and VM/370." A major set of the proposals were categorized as "data-in-memory" and included the ideas of named spaces, convenient dynamic sharing of programs and data, 31-bit addressing, device-independent I/O, and a hierarchical storage manager. Figure 3, which has been adapted from Reference 3, illustrates the task force view of requirements in this area. The findings of this study with regard to data-in-memory supported the technical objectives that were being established for the ESA/370 architecture extensions, and they provided a customer requirement statement that clearly supported the developing architecture, hardware, and software facilities.

ESA/370 addressing architecture

Storage addresses in the System/360 architecture are specified in instructions as displacements from base addresses in programmable general-purpose registers. In the most general case, an effective storage address is the sum of the displacement, the contents of the base register specified as holding the operand base address, and the contents of the index register specified in the instruction. Of course, not all these elements are present in every instruction's format. In System/370, these same effective storage addresses are interpreted as virtual, with system-managed translation tables specifying the actual physical residence of the data being addressed in virtual storage. In System/370-XA, the length of addresses was extended from 24 to 31 bits, providing concurrent addressability to 2 gigabytes of programs and data.

The addressing structure of MVS/XA is shown in Figure 1E. There is a single mapping of the virtual range in that the nucleus and common areas, which are occupied and used by the control program and the subsystems, occupy the same virtual address

range in every user's virtual mapping. Each user has a separate private storage area, and all private areas are in the same numerical address range. Such a structure provides access by system and subsystem services to their callers in private areas with good performance. A disadvantage is that any new or increased requirement for common storage must, of necessity, be taken from all users.

The basic architecture extension is the association of an identification of a virtual space with each storage reference.

Enhancements to the multiple-space facilities of System/370 beyond DAS were called for in order to support the evolution toward more multiple-address-space use in MVS and its subsystems. Better performance and more granular authorization were clearly understood objectives.

An evolutionary addressing extension to System/370-XA that meets the requirements is an extension of the base address of an operand to include the specification of a virtual space identifier. This extended definition of a base address is in harmony with System/360 and System/370 base addressing architecture. In ESA/370, the virtual space is the fundamental unit of protection and isolation. The objective of ESA/370 is to extend the use of virtual spaces to correlate to objects as defined earlier. Programs and data can reside in multiple spaces, and multiple-space addressing environments can allow concurrent access to all required objects.

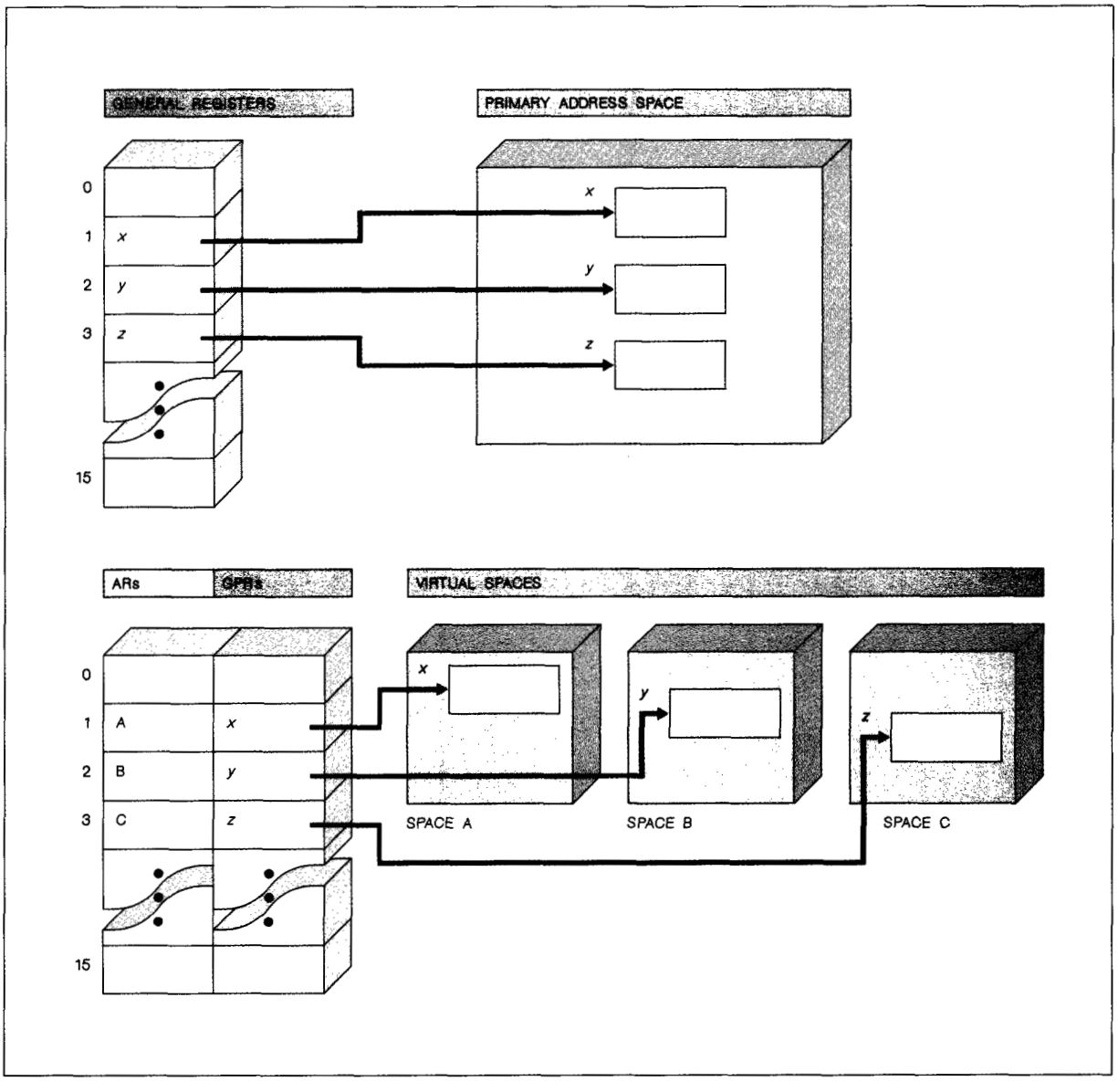
The basic architecture extension is the association of an identification of a virtual space with each storage reference. A space identification register with its associated general-purpose register is used for addressing operands in such a way that a processor can find the addressed data in real storage by the use of the correct address translation tables. The use of multiple space identification registers—each of which can potentially cause a different virtual space to be accessed—greatly expands the concurrent virtual addressability of a processor. These registers are named *access registers* (AR) and allow each virtual space specified by them to be up to 2 gigabytes in size architecturally. An access register is associated with each general-purpose register (GPR) in such a way that when the GPR is used as a base register for an operand location, the corresponding AR specifies the space in which the operand is located. Thus it is possible for a different virtual space to be associated with each general-purpose register so that concurrent virtual addressability is greatly expanded beyond the capability of System/370-XA architecture with DAS.

The role of access registers in ESA virtual addressing is illustrated in Figure 4. The provision of a means for an executing program to change the space specification in one of these registers dynamically—while at the same time remaining in problem state—greatly expands the amount of data that the program may virtually address efficiently during execution. In this environment, virtual addressing is transformed to an *address-space.offset* style. Addressing consists of space selection, via a new hardware mechanism called *access register translation* (ART), offset calculation within an address space, and dynamic address translation. This process is depicted in Figure 5.

Figure 6 illustrates many of the possibilities that are available with the new addressing capabilities in access register mode. Three virtual spaces are shown: the primary address space P, which contains an executing program; and two other spaces named Q and R, in this example. The sample instructions address operands in all three spaces.

- Instruction 1 moves characters from operand *a* to operand *b*. The instruction and both operands are in the primary space P. The associated access registers indicate the primary space with an architecturally defined zero value.
- Instruction 2 does a decimal Add of operand *a* in space P to operand *x* in space Q. The example indicates that the access register (AR) corresponding to the general register containing the base address of *x* within space Q specifies space Q. Operand *a* is in the primary space P as indicated by the associated access register.
- Instruction 3 illustrates an instruction in one space operating on two operands in a second space. A logical And is performed of operand *x* to operand *y*, both in space Q. The access registers are shown

Figure 4 ESA/370 addressing architecture

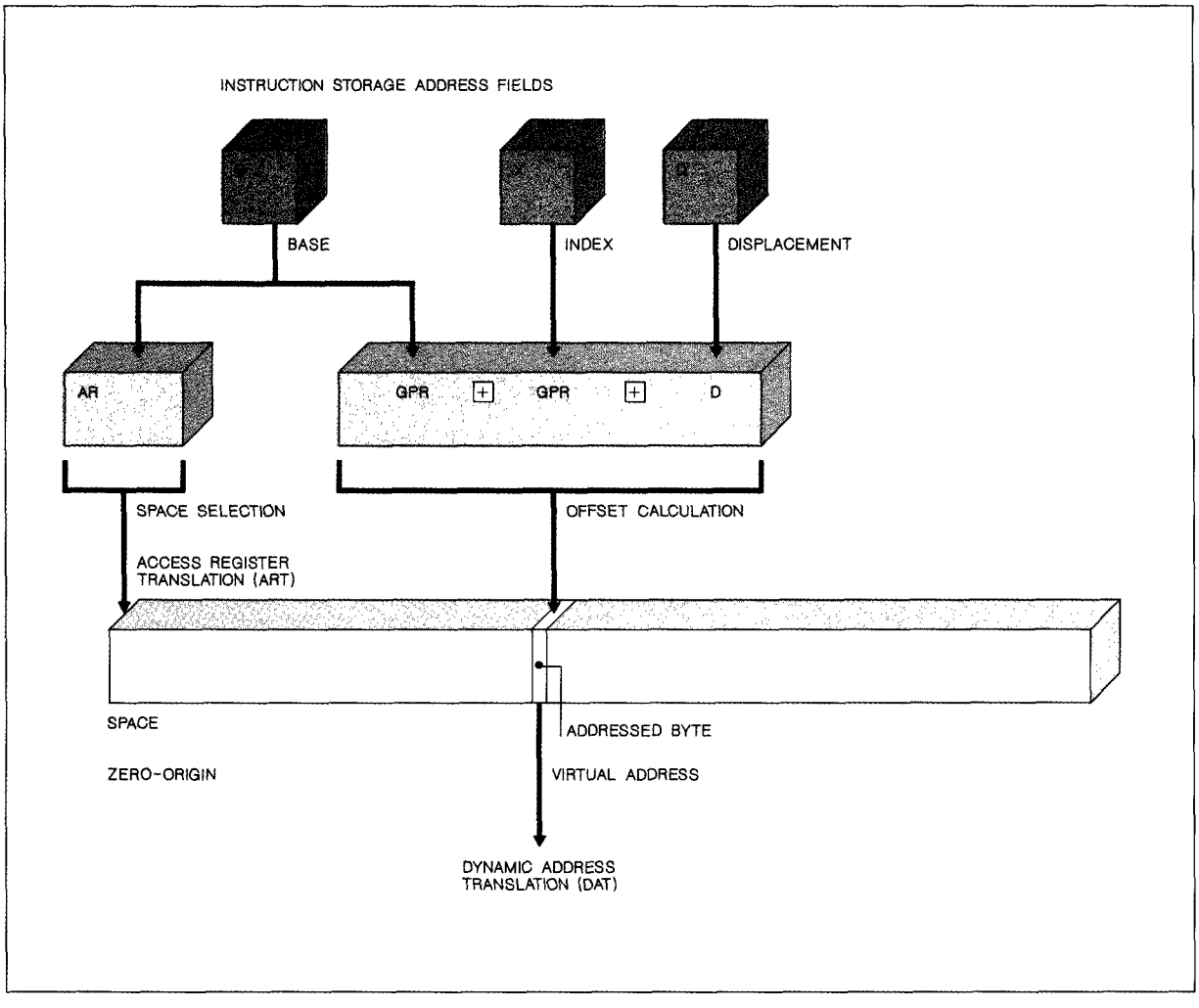


qualifying the base addresses of the operands to the space that contains them.

- Instruction 4 illustrates the operation of an instruction in one space with two operands, each in a different space. The instruction in space P causes operand z in space R to replace operand y in space Q.
- Instruction 5 does a compare of operand z in space R to operand b in the primary space P.

ESA/370 architecture provides the authorized, system-controlled sharing of access to virtual spaces. Each space is independent of any other and access may be allowed by specific users or by specific programs. In the architecture a token, called an access list entry token (ALET), represents a space. The ALET is an indirect representation of the *name* of the space in that the token is local to a dispatchable unit or to all programs executing in a particular address space.

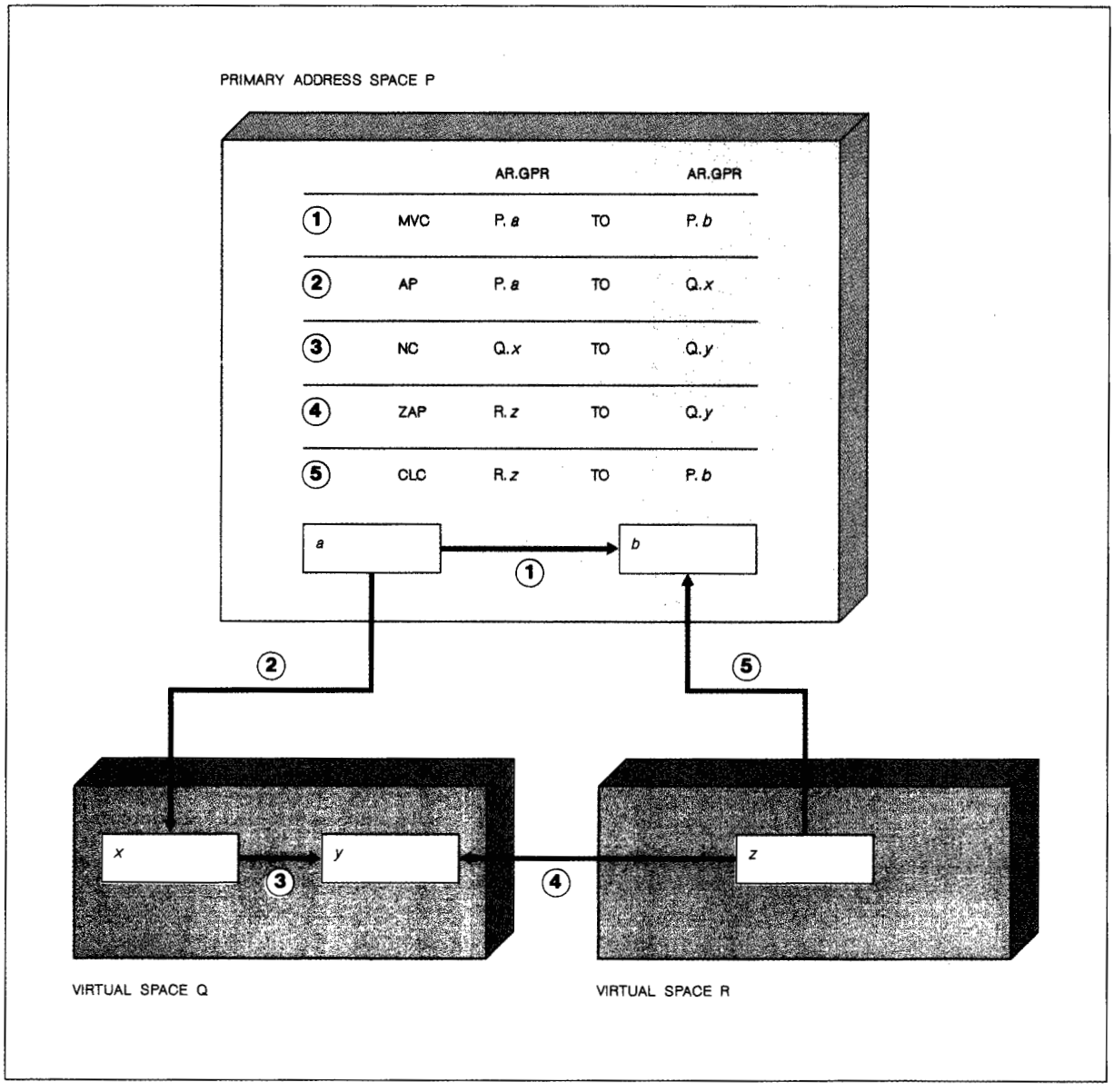
Figure 5 Address computation using address-space.offset and dynamic address translation



Access to a space may be allowed by any program running on behalf of a particular dispatchable unit (DU), or access may be allowed by a particular program regardless of which dispatchable unit the program happens to be serving. There is provision for giving different routines in the same address space different access authorities. The architecture provides an access list which indicates which spaces have been authorized for access by a specific DU or a specific address space. In other words, each dispatchable unit and each address space may have its own access list. In certain situations, the total access authority available to a program is the combination of the authority of the dispatchable unit and that of the address space containing the executing code. The creation and maintenance of access lists is an operating system responsibility. The lists reside in protected storage and cannot be accessed by application programs.

The initial MVS implementation allows each dispatchable unit and each address space to have over 250 spaces available for access. This allows the judicious placement of information so as to maintain privacy of some information while at the same time allowing sharing of other information with other dispatchable units or address spaces on a selective basis. Figure 7 illustrates the potential sharing capa-

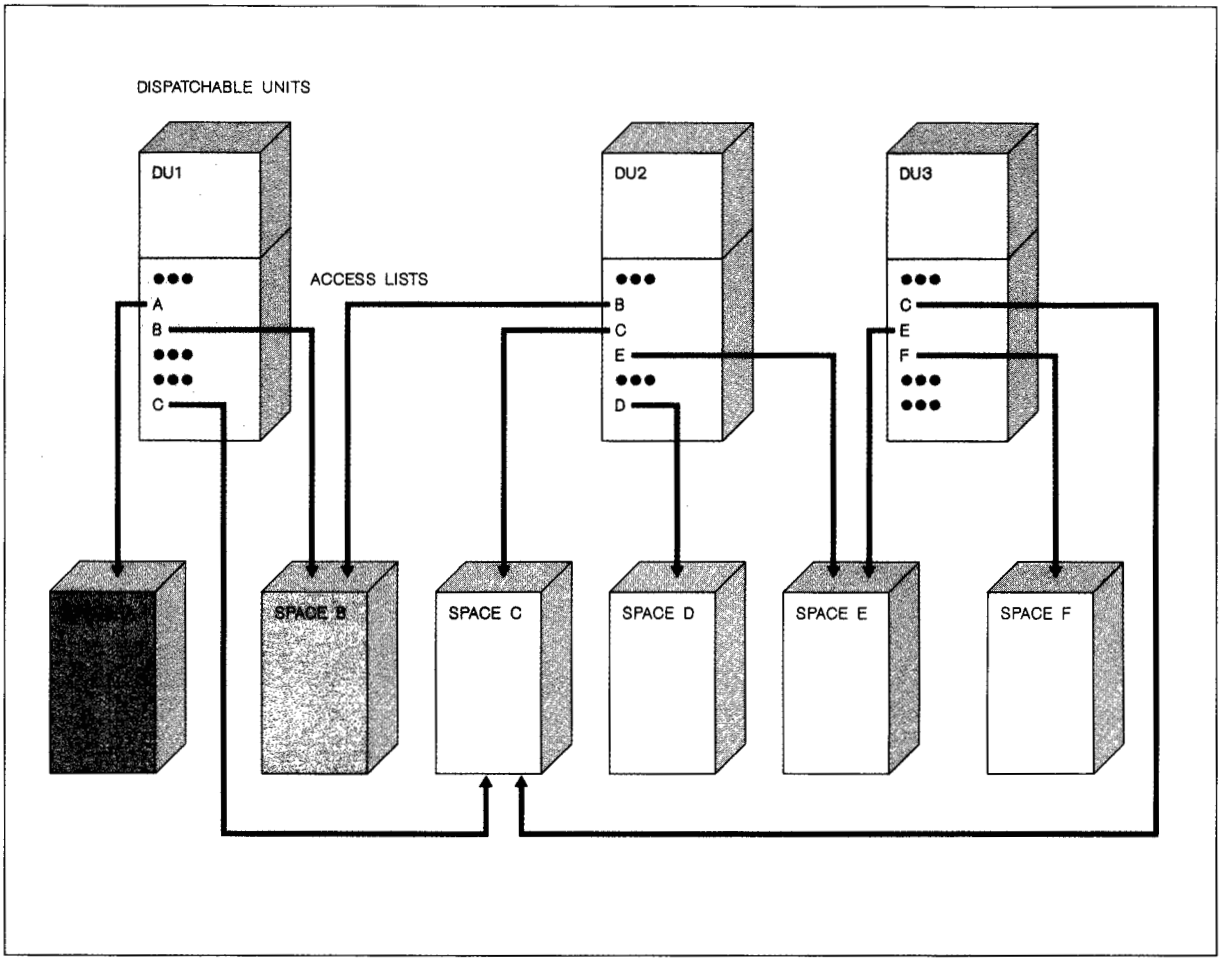
Figure 6 Direct addressing of data outside the primary space using the full processor instruction set



bility by showing three dispatchable units, each with its own access list of spaces that it can access. The illustration indicates that a particular space may be shared by all DUs. Others may be accessed only by one DU, and the sharing of others can be done between particular DUs selectively, while preventing access by other DUs.

In the translation of a virtual address within a specific address space to a real address, the ESA/370 architecture operates under the same rules as in 370-XA architecture. This is shown in Figure 8B. The virtual address is used in a two-level look-up to find the real address of the page frame containing the operand. In ESA, operands in multiple spaces are concurrently addressable. Therefore, the architectural translation process must determine which segment table designation should be used in the translation. The architectural process is shown in Figure 8A. This architecture provides for two access lists to be in force

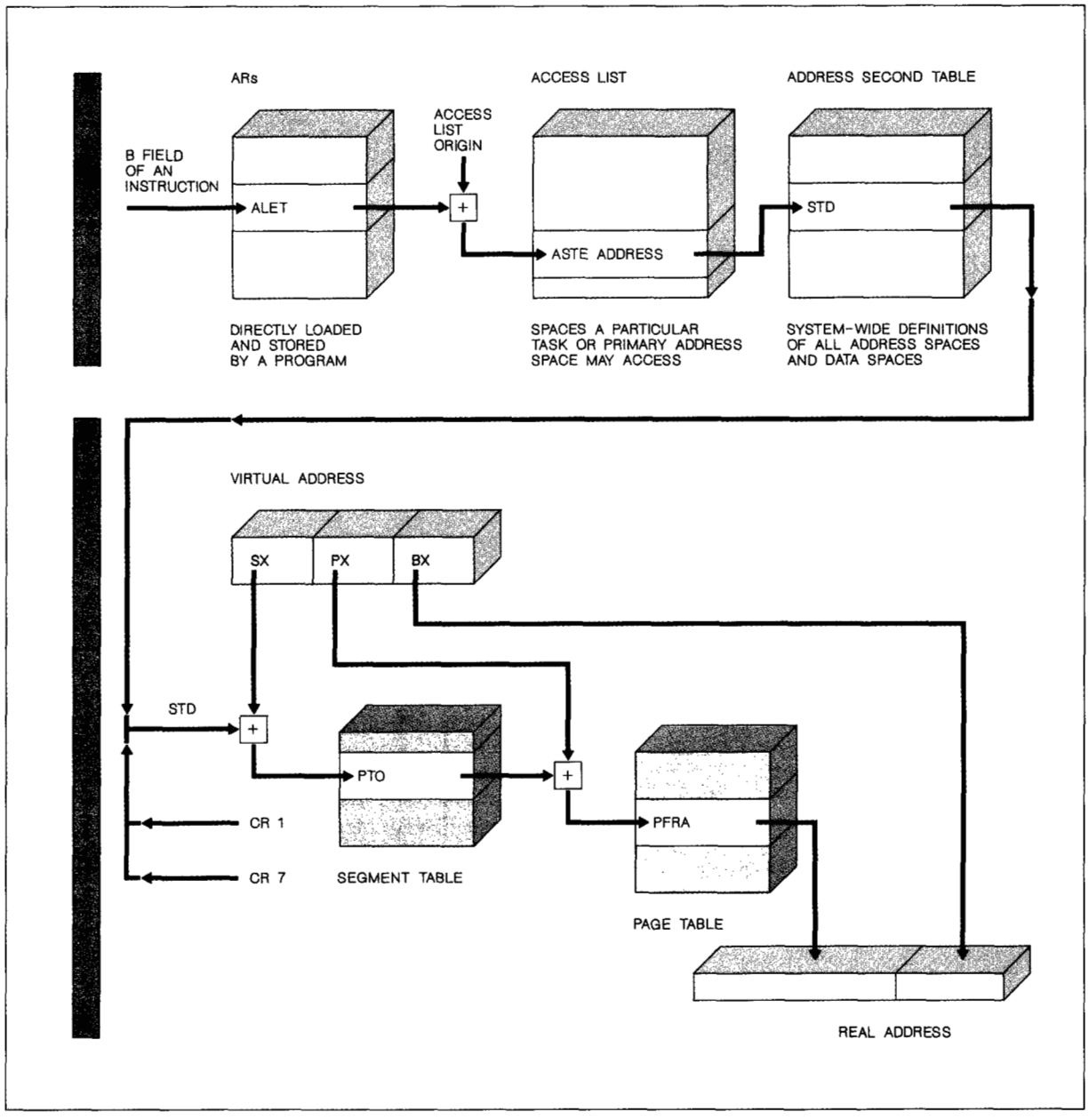
Figure 7 The sharing of spaces shown using three dispatchable units



concurrently. One, called the dispatchable unit access list (DU-AL), contains the access capabilities available to all programs executing on behalf of the task. The other, the primary address space number access list (PASN-AL), contains the access capabilities available to all programs executing in a particular address space, regardless of which task they are serving. The ALET specifies which of the two current access lists, the one of the dispatchable unit or that of the primary address space, should be used to locate the operand. Each entry in an access list represents a virtual space. The ALET identifies which member of an access list represents the space to be accessed. An access list entry (ALE) contains an address to an entry in system-wide tables containing the designations of all virtual spaces. The addressed entry provides the origin and length of the segment table to be used in the translation of the virtual address. In particular processor implementations, various look-aside buffers are used to reduce the frequency of full translation. The table containing the virtual space definition is called an address second table because, in address space number (ASN) translation, it is the second table accessed in a two-level translation process. However, the address pointer in the access list entry makes ASN translation unnecessary to find the definition of an ALET-addressed space.

The increased facility of virtual addressing provided by the architecture extensions can be exploited to

Figure 8 ESA/370 virtual-to-real address translation: (A) access register translation; (B) dynamic address translation



provide a convenient capability for a program or programming subsystem to map data stored on external DASDs to virtual storage and operate on it there without explicit I/O programming. Such use significantly increases the amount of data available to the operating system in paging storage. This will greatly enhance the opportunity to provide increased application and system performance through effective use of the electronic part of the storage hierarchy. Addressing external data in virtual will provide additional benefit where a large data structure is to be accessed at random. In this case, only the pages

containing referenced operands are brought to central storage from the DASD. At the conclusion of a transaction or job, only the pages containing changes need be written back to the external dataset. A further general benefit is increased programming ease-of-use and productivity that may result in many programming situations from the capability of operating on external data directly with processor instructions without the requirement of using access methods, I/O programming, and buffer management. The system manages physical residence of the data based on usage. This is particularly effective in systems with expanded storage.

Data spaces. Increased isolation and separation of data objects, as well as the requirement to share objects among multiple users, encourages the use of identifiable data areas that contain no code. In MVS, all virtual spaces are uniform in size and map both code and data in their addressing range, including all commonly addressable areas. Such virtual spaces are called *address spaces*. As protection requirements evolve to finer levels of distinction, the need arises to isolate objects much smaller than the 31-bit, 2 gigabyte address space of System/370-XA. Because the system overhead for supporting address spaces in terms of real-storage consumption is related to the space size, it is worthwhile to provide smaller spaces for smaller objects. In view of the fact that there is a certain amount of system overhead associated with supporting functional address spaces because of the dispatchable unit structures that must be provided, it is beneficial to provide virtual spaces earmarked for data only.

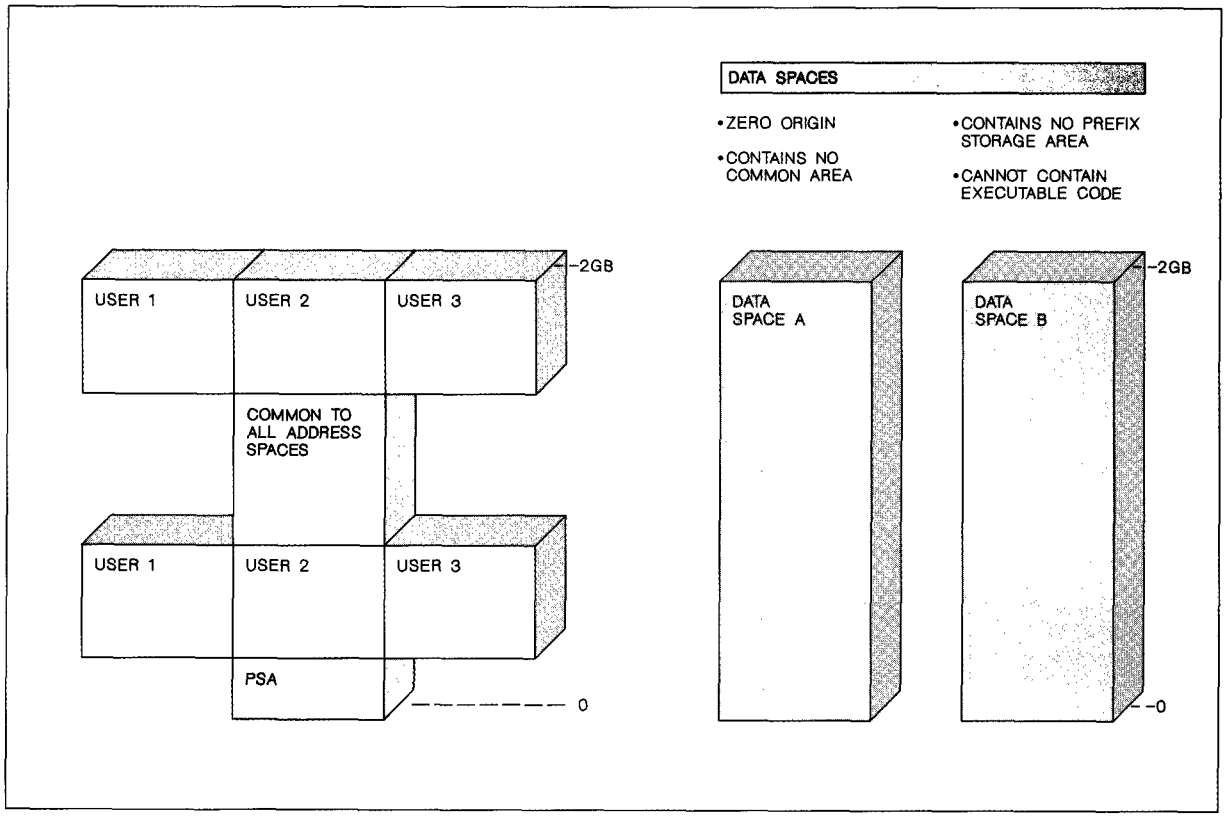
Virtual spaces with these characteristics, called *data spaces*, can provide increased virtual storage, data isolation, and an entity conducive to being shared among multiple processes or domains. They are also useful containers for mapping data-in-virtual storage. The size of the addressing range of data spaces can be variable, allowing for objects of as small as one page. In the context of ESA/370, the maximum is limited architecturally to 2 gigabytes for a single virtual space.

Data spaces can provide increased virtual storage, data isolation, and an entity conducive to being shared among multiple processes or domains.

In the MVS support of ESA, the basic MVS virtual address space is, of course, supported in addition to the new data space. A data space does not contain those areas common to MVS address spaces: a prefix save area (PSA), the MVS nucleus, or the MVS common area. Because the control program cannot execute within a data space, it need not contain the PSA, the hardware-software interface area to communicate interruptions and other information, nor need it contain the MVS common area for interprogram communication. A data space starts at a specified origin, expected to be zero in most cases, and may be addressed by the full 31-bit addressing range of the architecture. The MVS addressing structure with the addition of data spaces is shown in Figure 9. MVS provides services, described by C. E. Clark[4] in this issue, to create and delete spaces. A space may be used as a temporary work space or file, as an area for communication with other program address spaces, or it may be the vehicle for addressing permanent data on DASD. To support the DASD function, the data-in-virtual (DIV) component of MVS has been extended so that its services can apply to data spaces as well as MVS address spaces. This allows permanent collections of data to be defined as data objects and be operated on directly by all processor instructions, with the actual physical residence of pages controlled by the system based on actual reference patterns. Commitment of any changes to the original dataset is under control of the using program.

The mapping of datasets in page-format to virtual spaces is illustrated in Figure 10. This figure illustrates the operation of an add instruction whose two operands are each in a different external data file, each of which has been mapped into virtual storage. In this example, the add instruction in the primary space specifies two operands, each of which is in a different DIV object that has been mapped into its own data space. Operands *a* and *b* are to be obtained and the result replaces operand *a*. If the pages containing the operands had not been referenced before, they will be obtained from the DASD datasets by the page-fault resolution process. The physical residence

Figure 9 MVS addressing structure with the addition of data spaces

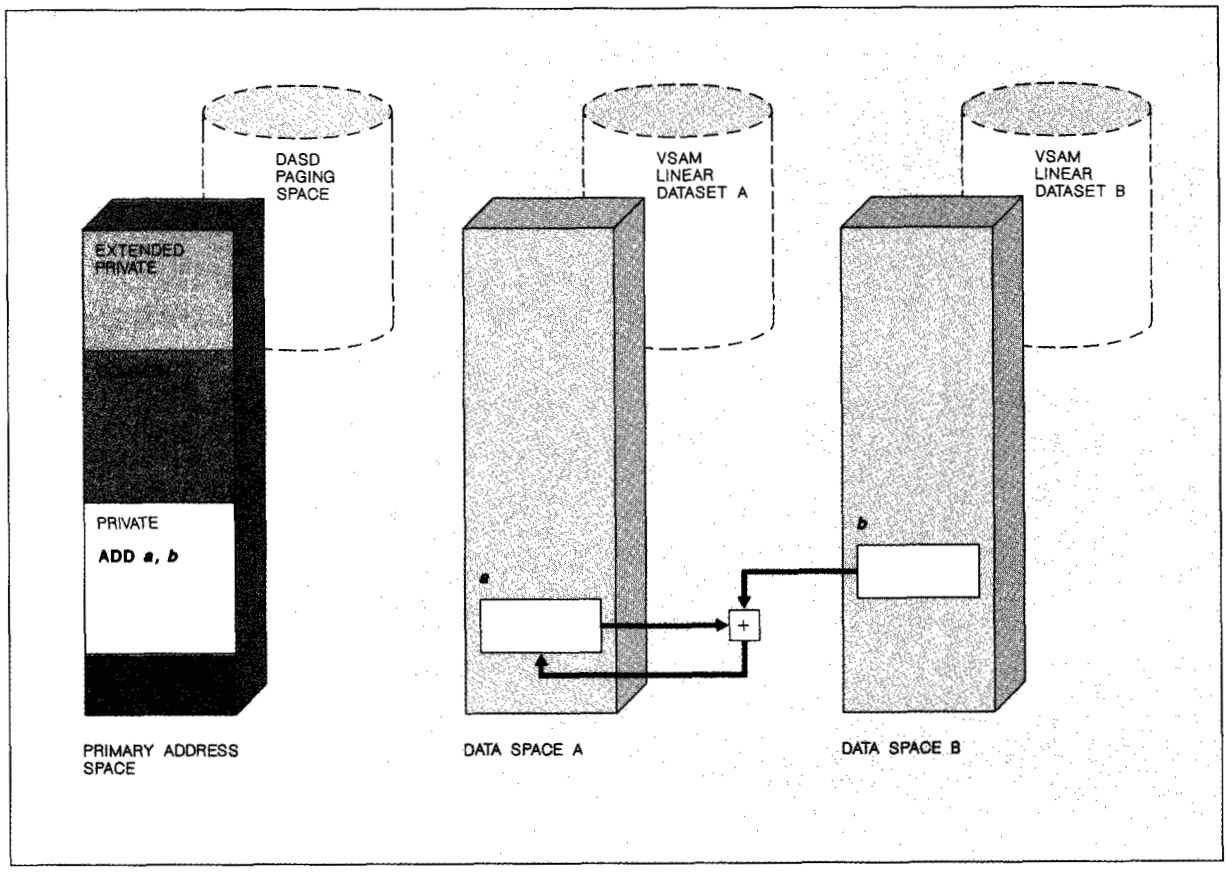


of the pages containing operand *b* and the result *a* are subsequently under control of the system paging operation. The new contents of *a* will be used to update the external DASD dataset only on command of the program. Of course, if either or both operands had been previously referenced, they will be obtained from system paging space if they are not in real storage. The example makes use of multiple spaces addressed through access registers, though it would work similarly if the data objects were mapped into portions of the primary space. Use of access registers allow each data object to occupy its own 2 gigabyte space. Also, this allows shared access in virtual storage of the DIV objects with other authority domains, separately, if desirable or required, and without requiring the primary space to be shared with the other domains. Addressing of such DIV objects can be natural to their internal structures, because each may be addressed in its own space, starting from zero. For example, a DIV object may contain internal pointers to its parts that remain valid without programmed relocation when the object is made addressable within a processor. DIV supports objects mapped to data spaces with its full services, which include writing back only changed pages when specified by the using program. This is further described by K. Rubsam[2] in this issue.

The mapping of data to virtual makes good use of the storage hierarchy, because the hierarchy can be managed on a system-wide basis to meet overall performance goals. Frequently accessed data is in effect cached in main storage or in expanded storage, so that references can be satisfied with high performance. Much more data can take advantage of the speed of expanded storage (ES), for example, by using this technique. This puts the physical residence of data pages under system control and allows ES to be used as an effective cache for frequently accessed data.

There are some cases where an application or subsystem has an algorithmic reference pattern to data that, if exploited, provide superior performance to

Figure 10 Mapping of page-format datasets to data-in-virtual (DIV) spaces



general algorithms that assume random access. Such a program benefits by treating the ES effectively as an I/O device, directing the staging of data to and from it and main storage. MVS supports such operation through the provision of *hiperspaces*. The physical residence of pages in a hiperspace is primarily ES. Thus data are addressed in a hiperspace only in move mode. The data are operated on arithmetically and logically only in buffers in the primary space. Provision is made for multiple page moves in either direction to reduce the frequency of calling the system services. This allows a program to manage the content of processor real storage through use of logical programming constructs, while obtaining the performance benefit of the ES as a cache for a very large amount of data. Hiperspaces and the services useful in processing them are described further in Reference 2 in this issue.

It is interesting to note that the set of facilities just described are believed on the whole to meet the LSRAD objective of a hierarchical storage manager. The objectives of device-independent I/O are addressed by the data facilities product system managed storage component. Technical objectives include removing dependencies on, and management of, physical device characteristics from the programming environment. Taking the LSRAD report as a comprehensive statement of customer requirements in the use of virtual addressing for data, ESA and the associated programming support takes the system a very long way toward a complete solution.

ESA/370 linkage architecture

ESA/370 supports the use of virtual spaces as isolated objects with individual authorization and also pro-

vides the capability of establishing separate authority domains, each with its own access authority. It does this through extensions to the program call (PC) instruction, the addition of a program return (PR) instruction, provision of a linkage stack, and architecture facilities for controlling the use of access lists and the entities within them. The program call instruction was defined in DAS architecture to be a function-calling mechanism, whereby code of different operating authority—possibly in a different address space—could be accessed with a synchronous instruction. The operating environment in which the code that provides the called function is to operate is initialized through operating system services in an entry table, used by a processor in performing the PC instruction. In ESA/370, the authority mechanism architecture is enhanced so that such a called function can operate at greater, less, or completely different access authority than its caller. In addition to the hierarchical key handling provided in DAS, whereby the called program has access to the key of its caller as well as its own potentially different key, the called program may be restricted to its own different key. Also, because of the capability to establish separate shared virtual spaces for communication, an option exists to prevent access by the called program to the address space of its caller. Isolation of caller and called programs is provided by the linkage stack, which can be specified to receive the caller's operating state information during the execution of the PC instruction. These include its PSW key, PSW key mask, primary and secondary space designations, general-purpose register and access register contents, return address, and PSW operating mode. A program return instruction in the called program restores the calling program's environment from the linkage stack except for those registers defined for intercommunication between calling and called programs. Each dispatchable unit (DU) has its own linkage stack. In summary, the entry table for a particular callable program contains the conditions under which the called program is allowed to operate. The linkage stack dynamically receives the operating state of the caller so that it may be restored on return from a called program. The linkage operation is illustrated in Figure 11.

In the execution of a PC, a new primary space can be established that may have its own primary address space access list (PASN-AL) containing spaces that can potentially be accessed by code running in that primary space. The task executing a PC has its own dispatchable unit access list (DU-AL) containing spaces that any code operating under that dispatchable unit may potentially access. On a call to a function provided in another address space, the spaces in the dispatchable unit's access list may be accessed in providing the requested function, but the architecture contains controls to constrain such access. Spaces designated in an access list may be specified as "public," meaning that any code running with the access list may access them regardless of where the code resides. To emphasize this, such spaces are public only in the context of the access lists containing them, but no other authority is required. To provide controlled sharing of access, an extended authorization index (EAX) has been defined. All code runs with a particular EAX, which constrains its access to spaces. The EAX controls which nonpublic spaces may be accessed. Each access list entry can contain an EAX to restrict access to the space it represents. Access to nonpublic spaces is permitted in either of two ways:

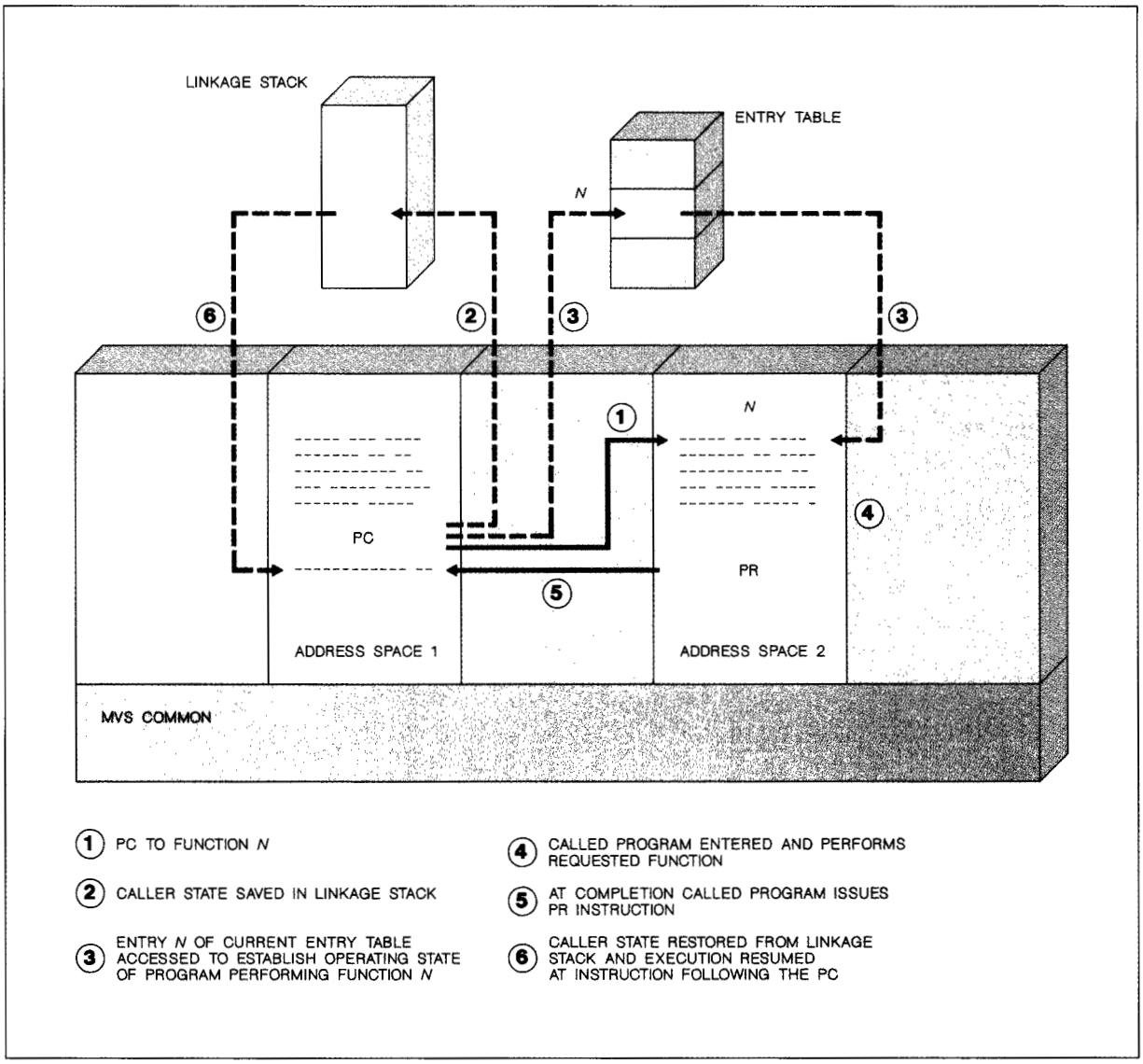
- The EAX of the running code equals the EAX in the access list entry defining the space.
- Otherwise, the authority table entry for the space to be accessed allows such access for the particular EAX attempting access. (The authority table was added to the architecture as a part of DAS to provide a control mechanism over multiple-space access.)

The first case can be thought of as the owner of a space accessing it. The second case can be thought of as the owner of a space selectively allowing access to it for one or more functions operating with their own different EAX. The EAX is a characteristic or capability of executing code and may be specified to change during a PC and be restored from the linkage stack on a subsequent PR. It is designed to provide controlled, shared access to spaces in an authority domain environment. MVS supports the use of the EAX for address spaces but not for data spaces.

Concluding remarks

The basic ESA/370 system structure direction is to provide a programming environment in which multiple virtual spaces can be used effectively to allow extensive and practically unconstrained use of virtual storage. This usage is intended not only to overcome current and anticipated virtual storage constraints, but also to extend the System/370 systems to take advantage of technological progress in electronic storage hierarchies and allow enhanced isolation, protection, and sharing of programs and data. The operating system environment necessary to satisfy

Figure 11 The linkage operation



these requirements is one in which the key logical entities of the operating system environment are segregated into distinct virtual spaces, where the virtual space becomes the basic unit of protection.

ESA/370 provides the needed addressing capability to avoid constraints for the foreseeable future. As virtual storage requirements increase, new address spaces or data spaces easily may be added to accommodate more addressability. The architecture provides a rich instruction set with which to manipulate storage across a large number of spaces. In addition, a robust authorization control mechanism with hardware enforcement is provided to allow a flexible environment for access control with good performance. This is significantly important in improving security and integrity.

With ESA/370, this system direction will be achieved in an evolutionary manner. Existing environments

must be continually supported for compatibility with prior systems. The approach to accomplish this evolution must provide new functions and capabilities while still supporting existing environments. Gradually, the advantages and extended function offered in new environments will encourage more pervasive usage of the multiple virtual space addressing features. ESA/370 has been designed to enable a strategy of supplying new facilities without disturbing existing user environments. By extending System/360, System/370, DAS, and System/370-XA architectures in a nondisruptive way, usage of the new facilities can coexist with existing programs and operating system environments. Exploitation of the enhanced architecture will initially materialize in subsystems and system components in ways that are, for the most part, transparent to users. Over time more software services will be made available that facilitate direct usage by the end user of the enhanced architecture.

ESA/370 is designed with extendability in mind. It provides a new architectural base that can be extended in an evolutionary way as technological advances occur in processor and electronic storage hardware and accommodate evolving software technologies in operating systems, subsystems, and applications.

Acknowledgments

Many persons have supported and contributed technically to the early stages of the development of ESA/370. The project was initiated by James T. Brady. The architecture was developed with significant joint contributions from MVS Design, Central Systems Architecture, 3090 Engineering, and Processor Architecture. Individuals other than the authors who made contributions are listed below by organization.

Richard I. Baum, Justin R. Butwell, Terry L. Borden, Carl E. Clark, James Lum, Michael G. Mall, Thomas Springer, Rick Reinheimer, Ruth Allen, and David Page participated in the early development of ESA/370, representing the MVS Design organization. Henry Brandt, Ronald F. Hill, Ben Messina, and Julian Thomas participated in the early development of ESA/370, through the 3090 Engineering group. Kenneth E. Plambeck represented Central Systems Architecture for the duration of the ESA/370 design and development. Catherine Eilert, Kenneth G. Rubsam, and Robert B. Seaborg were responsible for the MVS design for data spaces, hiperspaces, and related services. Bernard M. Goldman, from Intellectual Property Law, wrote the two patents referenced.

We especially acknowledge the product support and guidance of James A. Cannavino, Carl A. Caricari, the late Robert E. Crosby, then Poughkeepsie Laboratory manager, and the late John E. Bertram, then the Data Systems Division president.

Cited references

1. E. I. Cohen, G. M. King, and J. T. Brady, "Storage hierarchies," *IBM Systems Journal* **28,** No. 1, 62–76 (1989, this issue).
2. K. G. Rubsam, "MVS data services," *IBM Systems Journal* **28,** No. 1, 151–164 (1989, this issue).
3. *Toward More Usable Systems*, Report of the Large Systems Requirements for Application Development Task Force, SHARE, Inc., One Illinois Center, 111 E. Wacker Drive, Chicago, Illinois, 60601 (December, 1979).
4. C. E. Clark, "The facilities and evolution of MVS/ESA," *IBM Systems Journal* **28,** No. 1, 124–150 (1989, this issue).

General references

J. R. Butwell, C. A. Scalzi, and R. J. Schmalz, *Address Generating Mechanism for Multiple Virtual Spaces*, U.S. Patent 4,355,355, October 19, 1982.

P. J. Denning, "Fault-tolerant operating systems," *Computing Surveys* **8**, No. 4, 359–389 (December 1976).

J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computation," *Communications of the ACM* **9,** No. 3, 143–155 (March 1966).

R. S. Fabry, "Capability-based addressing," *Communications of the ACM* **17,** No. 7, 403–412 (July 1974).

IBM Enterprise Systems Architecture/370 Principles of Operation, SA22-7200, IBM Corporation; available through IBM branch offices.

IBM MVS/ESA: SPL Application Development Guide, GC28-1821, IBM Corporation; available through IBM branch offices.

IBM MVS/ESA: SPL Application Development—Extended Addressability, GC28-1854, IBM Corporation; available through IBM branch offices.

IBM MVS/Extended Architecture: Overview, GC28-1348, IBM Corporation; available through IBM branch offices.

IBM System/38 Technical Developments, G580-0237, IBM Corporation; available through IBM branch offices.

IBM System/370 Extended Architecture Principles of Operation, SA22-7085, IBM Corporation; available through IBM branch offices.

IBM Time Sharing System: Concepts and Facilities, GC28-2003, IBM Corporation; available through IBM branch offices.

IBM VM/SP System IPO/E: General Information Manual, GC20-1890, IBM Corporation; available through IBM branch offices.

J. K. Iliffe, *Basic Machine Principles*, American Elsevier, Inc., New York (1968).

B. W. Lampson, "Protection," *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, March 1971, pp. 437–443. Reprinted in *Operating Systems Review* **8,** No. 1, 18–24 (January 1974).

H. M. Levy, *Capability-Based Computer Systems,* Digital Press, Bedford, Massachusetts (1984).

T. A. Linden, "Operating systems structures to support security and reliable software," *Computing Surveys* **8,** No. 4, 409–445 (December 1976).

J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE* **63,** No. 9, 1278–1308 (September 1975).

C. A. Scalzi and R. J. Schmalz, *Mechanism for Accessing Multiple Virtual Address Spaces,* U.S. Patent 4,521,846, June 4, 1985.

M. V. Wilkes, *Time-Sharing Computer Systems,* American Elsevier, Inc., New York (1968).

W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The kernel of a multiprocessor operating system," *Communications of the ACM* **17,** No. 6, 337–345 (June 1974).

Casper A. Scalzi *IBM Data Systems Division, P.O. Box 950, Poughkeepsie, New York 12602.* Mr. Scalzi is a senior programmer in the Processor Architecture and System Structure department in the Poughkeepsie Development Laboratory. His responsibilities include the definition and evaluation of architectural extensions for IBM's large processor systems. Mr. Scalzi joined IBM in Poughkeepsie in 1956. He was part of the team that defined the architecture for the IBM 7030 (Stretch) and later was part of the Stretch Experiment in Multiprogramming (Project STEM). He was part of the initial design team for Operating System/360 (OS/360). He managed the design and development of the Linkage Editor E, the primary control program improvement program, he was OS/360 Scheduler design and development manager, and OS/360 design manager. He has been a member of the IBM corporate programming staff. Mr. Scalzi has been a part of the access register definition team since its inception and has been an active participant during the development and definition of the architecture. Mr. Scalzi received an IBM Outstanding Contribution Award for his role in OS/360, an Outstanding Innovation Award for his contribution to ESA/370, an Outstanding Innovation Award for his work on PR/SM architecture, and a Second-Level Invention Achievement Award. Mr. Scalzi is a coauthor of a chapter in the book *Planning a Computer System,* edited by W. Buchholz, McGraw-Hill Book Company, 1962. He is a coauthor of "The history of IBM memory management technology," *IBM Journal of Research and Development,* 25th anniversary issue, 1982. He is the holder of three issued U.S. patents. Mr. Scalzi has a B.S. degree from Fairfield University, Fairfield, Connecticut.

Alan G. Ganek *IBM Data Systems Division, Neighborhood Road, Kingston, New York 12401.* Mr. Ganek is manager of VM/XA Advanced Systems. He received his M.S. in computer science from Rutgers University in 1981. Mr. Ganek joined IBM as an associate programmer in 1978 in Poughkeepsie, New York. His first assignments involved the implementation of the MVS operating system software support for the cross-memory and System/370-XA architectures. In 1981 he joined the MVS System Structure Technology department, where he contributed to the definition of the Enterprise Systems Architecture/370, leading to a first patent award. He later became team leader for the MVS software support design for ESA/370. Mr. Ganek was the recipient of an Outstanding Innovation Award for his contributions to ESA/370. In June 1983, Mr. Ganek was named development manager of the MVS System Design department, involving work on a variety of advanced technology activities. In 1985, he was appointed manager of MVS Design and Performance Analysis, where he was responsible for the technical plan and content of the MVS base control program future releases and, in 1986, he was promoted to program manager. Later that year, Mr. Ganek joined the Information Systems and Storage Group staff as a technical assistant. In 1987, he came to the IBM corporate programming staff where he contributed to a number of efforts concerning IBM's programming strategy and Systems Application Architecture. During that assignment, he coauthored "Introduction to System Application Architecture" with E. F. Wheeler, which was published in the *IBM Systems Journal* in 1988. He began his current assignment in June 1988, in which he is responsible for VM/XA advanced systems strategy, design, planning, and product introduction support.

Richard J. Schmalz *IBM Data Systems Division, P.O. Box 950, Poughkeepsie, New York 12602.* Mr. Schmalz is a senior planner in the Processor Architecture and System Structure department of the Poughkeepsie Development Laboratory. He joined IBM in 1956 in St. Louis, Missouri, and has held numerous marketing, programming, and engineering positions throughout his IBM career. He has an Outstanding Innovation Award for his contribution to ESA/370, an Outstanding Innovation Award for his work on Expanded Storage, an Outstanding Innovation Award for hiperspaces, a Second-Level Invention Achievement Award, and three issued U.S. patents. Mr. Schmalz received his B.S. in mathematics from Washington University, St. Louis, Missouri.

Reprint Order No. G321-5346.