The facilities and evolution of MVS/ESA

by C. E. Clark

As new processors were developed with new capabilities, the Multiple Virtual Storage (MVS) operating system was modified and enhanced to utilize the latest advances. The most recently available processors are structured on Enterprise Systems Architecture, and MVS has evolved to be a part of this architecture as MVS/ESA™. This paper describes the changes that occurred in MVS and the facilities that are currently available to support users of the latest processors.

In February 1988, IBM announced the Enterprise Systems Architecture/370™ (ESA/370™) and Multiple Virtual Storage/Enterprise Systems Architecture (MVS/ESA™). This new architecture and the supporting hardware and software offer an evolutionary step toward meeting the data processing requirements of the 1990s. ESA/370 is implemented and available on the IBM 3090 Model E and the 3090 Model S series processors and on the IBM 4381 Model Groups 91E and 92E. These processors are capable of executing software developed for both the previous architecture level—System/370 Extended Architecture (370-XA), and the new architecture, providing for a nondisruptive migration path for users of these computers. The new architecture on these processors is initially supported and utilized by the MVS operating system, MVS/ESA, which consists of two IBM licensed programs, Multiple Virtual Storage/System Product (MVS/SP™) Version 3 and Multiple Virtual Storage/Data Facility Product (MVS/DFP™) Version 3.

This paper describes how MVS/SP Version 3 supports ESA/370 and, in particular, how that support provides the user with additional virtual storage capacity, data sharing, data isolation, and increased cross-memory addressing capabilities. Basic changes were made to Version 2 of the MVS/SP operating system environment in order to introduce a new system construct, the data space, in Version 3 and to provide extended addressing capabilities to multiple address spaces and data spaces. New services were added to create address and data spaces as well as facilities to grant or limit access capabilities to these spaces.

Two new components of MVS/SP Version 3, the Virtual Lookaside Facility (VLF) and the Library Lookaside Facility (LLA), take advantage of the new hardware and software capabilities in order to reduce physical I/O operations by caching data and programs. VLF and LLA also use the additional virtual storage capabilities to provide increases in performance and throughput. Other IBM products, such as Time Sharing Option/Extended (TSO/E) and Information Management System (IMS), utilize the VLF and LLA facilities to provide performance improve-

ments for their users without requiring the users to modify any existing code.

The new hiperspaces and data window services give programs an opportunity to use expanded storage to reduce physical I/O operations in order to increase throughput. The data window services are available to higher-level language (HLL) programs and to programs coded in Basic Assembler Language (BAL). Use of data window services by the virtual storage access method (VSAM) has allowed IMS to reduce the necessary number of physical I/O operations and amount of virtual storage below 16 megabytes for its VSAM database I/O activity.

In addition to the facilities for extended addressing, program linkage mechanisms were extended by the new linkage stack facility. This facility includes the linkage stack, the new stacking program call and program return, the new branch and stack instruction, and associated recovery on the stack. The extended addressing, improved linkage facilities, and ability to use the full MVS/ESA instruction set to address multiple address or data spaces provide many of the necessary primitives to meet the large operating system requirements of the 1990s. This has been done without sacrificing users' investments in programs that execute on previous levels of MVS.

The evolution of MVS/ESA

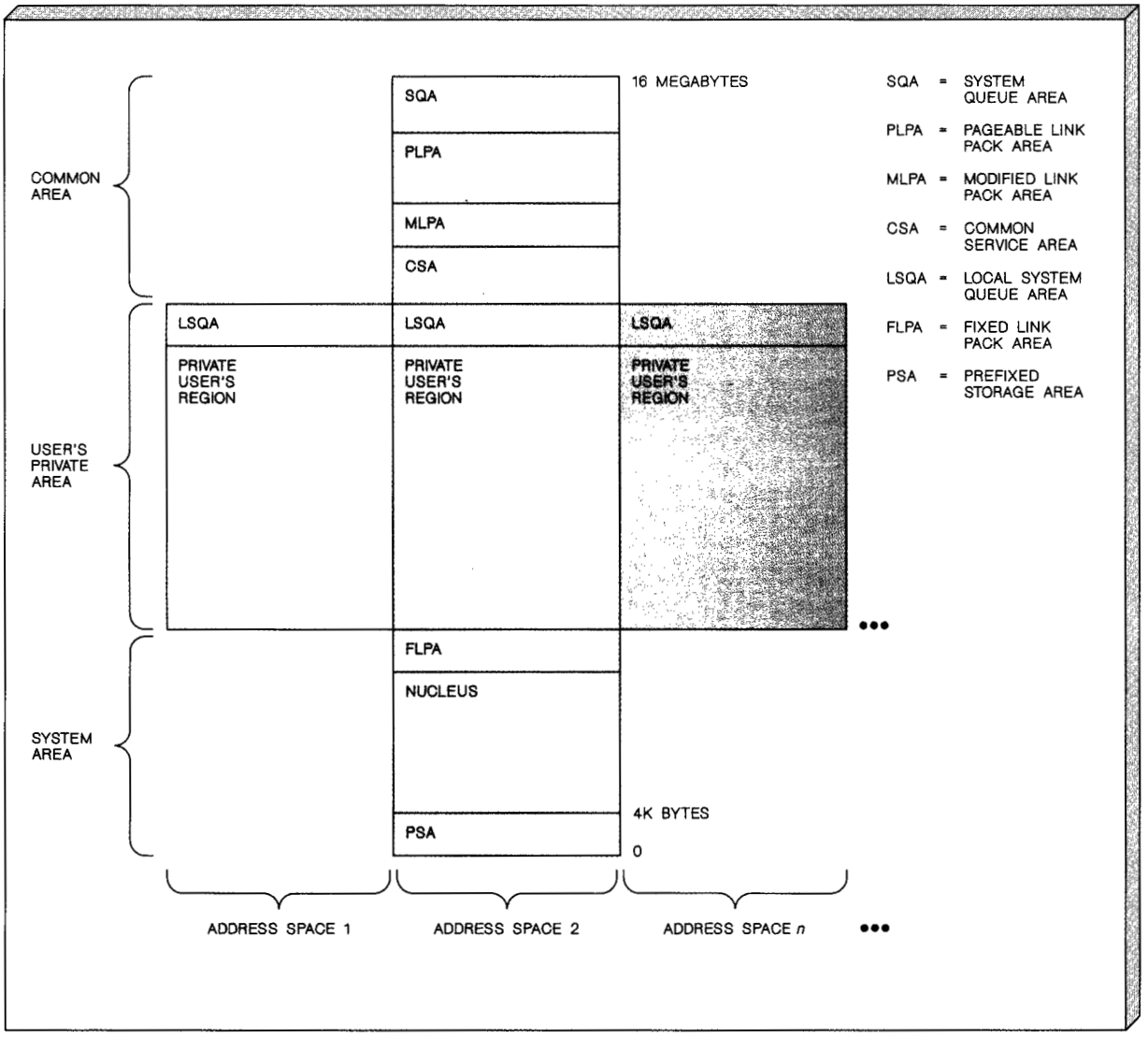
The facilities provided in MVS/ESA were designed to be extendable and to provide the required addressing capabilities before the addressing requirements became constraints in the user's environment. The facilities were also designed to remove certain limitations that existed in the previous versions of MVS. A brief look at the product history of MVS will demonstrate how MVS has changed to meet the growth in data processing capacity requirements, how the previous MVS facilities have been improved by ESA facilities, and how MVS has maintained compatibility with previous versions of users' programs.

The System/360, System/370, and ESA/370 architectures and operating systems. During the last quarter century, IBM has introduced the System/360 architecture, System/370 architecture, System/370 Extended Architecture, and most recently, the Enterprise Systems Architecture/370. In 1964 IBM announced the System/360 which had a minimum configuration of 16K characters (bytes) of main storage. The next year the Basic Operating System was announced. In the following year, the primary control program (PCP), multiprocessing with a fixed number of tasks (MFT), and multiprocessing with a variable number of tasks (MVT) were introduced, and by 1970 IBM had announced the Asymmetric Multiprocessing System (ASP), Houston Automatic Spooling Program (HASP), Remote Job Entry (RJE), MVT/M65 multiprocessor (MP), telecommunications access method (TCAM), and Time Sharing Option (TSO). IBM introduced System/370 in 1970 and Operating System/Virtual Storage 1 (OS/VS1) and Operating System/Virtual Storage 2 (OS/VS2) in 1972, extending virtual addressing capability to users of its two earlier operating systems, OS and the Disk Operating System (DOS). The MVS operating system, based on the System/370 architecture, was announced in 1973 and shipped in 1974. At the time, the first version of MVS seemed to provide a tremendous amount of virtual addressing capacity—16 million bytes for each user or job.

The 16 million bytes of virtual addressing of the MVS address space consisted of 256 segments of 64K bytes per segment. Each segment consisted of sixteen 4096-byte pages. On segment boundaries, the address range was divided into a system area, the user's private area, and the common area as shown in Figure 1. The system and common areas were addressable by all users unless the area was unallocated or protected by a storage key different from that of the user. The user's private area was unique to each user or job and only addressable by the user or job allocated to that address space. Beginning at the low address zero, some number of segments would be allocated for the system area. These segments had real storage frames permanently backing the virtual pages. Into this area, system-control program code was loaded, particularly performance-sensitive code located in the nucleus load module. Other system-control program code was loaded into the PLPA, the pageable link pack area. This area also contained subsystem, installation, and certain application code that needed to be used by more than one address space. The system queue area (SQA) and common service area (CSA) generally contained control blocks and data that had significance to more than one address space.

Since first being announced, MVS has continuously been enhanced together with the underlying hardware architecture to meet the demand for increased data processing capability. However, the user's investment in programming has been protected even with the introduction of new architecture. Programs

Figure 1 Initial address space structure of MVS



that were written for the previous levels of System/370 architecture, and indeed for System/360, are still capable of executing on today's MVS/ESA.

MVS history. The first version of MVS, OS/VS2 Release 2, supported two new System/370 processors, the Model 158 and 168 MP systems. The main storage supported by MVS was a minimum of 768K. MVS supported the maximum size of 8192K of main storage offered by the System/370 Model 168. MVS supported both a uniprocessor (UP) and MP version of the Model 158 and 168 processors, with the high-end 168 MP being approximately 2.6 times more powerful than the low-end 158 UP. As data processing requirements grew in the late 1970s and the early 1980s, MVS supported an increasingly powerful series of IBM processors: the 303Xs, 308Xs, 43XXs, and 3090s. Then in early 1988, ESA/370 became available on the 3090E models and the 4381 Model Groups 91E and 92E. Within six months of its initial announcement, ESA/370 was available on models of the 3090S series. In addition to supporting processors with more processing power, MVS made the necessary changes to support larger numbers of tightly coupled

processors, supporting the six processors of the 3090-600E and 3090-600S. This meant that under MVS/ESA the number of available processors could vary from one to six, and the range of processing power is approximately five to more than 100 times that of the first processor supported by MVS. In addition, the range of storage that MVS/ESA supports extends from a maximum of 512 megabytes of main and expanded storage for the 4381 processors, to 256 megabytes of real and 2 gigabytes of expanded storage for the 3090-600E. This growth in processing capacities and in the range of the processors that MVS has supported over the last 15 years is indicative of the potential growth and capacities that MVS must plan for in the future. This growth cannot be limited by the basic addressing capabilities of the architecture or the operating system. MVS/ESA and ESA/370 provide the primitives to allow this continued growth.

As the capabilities of the processors increased, MVS was continuously modified and enhanced.

As the processing, storage, and I/O capabilities of the processors increased, MVS was continuously modified and enhanced to meet the needs of users and the additional capabilities of the processors. As these changes emerged, multiple packaging arrangements were introduced to deliver these products to the user in as timely a fashion as possible. After the initial version of the MVS product was shipped, many of the processor and performance support packages were shipped as selectable units (SUs). They were followed by system extensions in which significant performance packages were separately priced. Finally in 1980, MVS became a priced product with the introduction of MVS/SP Release 1. By then an MVS/SP system might contain the Virtual Telecommunications Access Method (VTAM), Resource Access Control Facility (RACF), Resource Measurement Facility (RMF), TSO, Customer Information Control System (CICS), IMS, or Job Entry Subsystem 2 or 3 (JES2 or JES3). Many of these products provided services to any job, users, or application that executed in an MVS address space, and many managed resources that were global or common to all of the MVS address spaces. Because of global requirements and performance considerations, many of these products were placing programs and control blocks in common storage areas of MVS. Although this placement helped meet the performance and global resource management requirements, it was also reducing the amount of privately addressable storage available to the user, job, application, or subsystem.

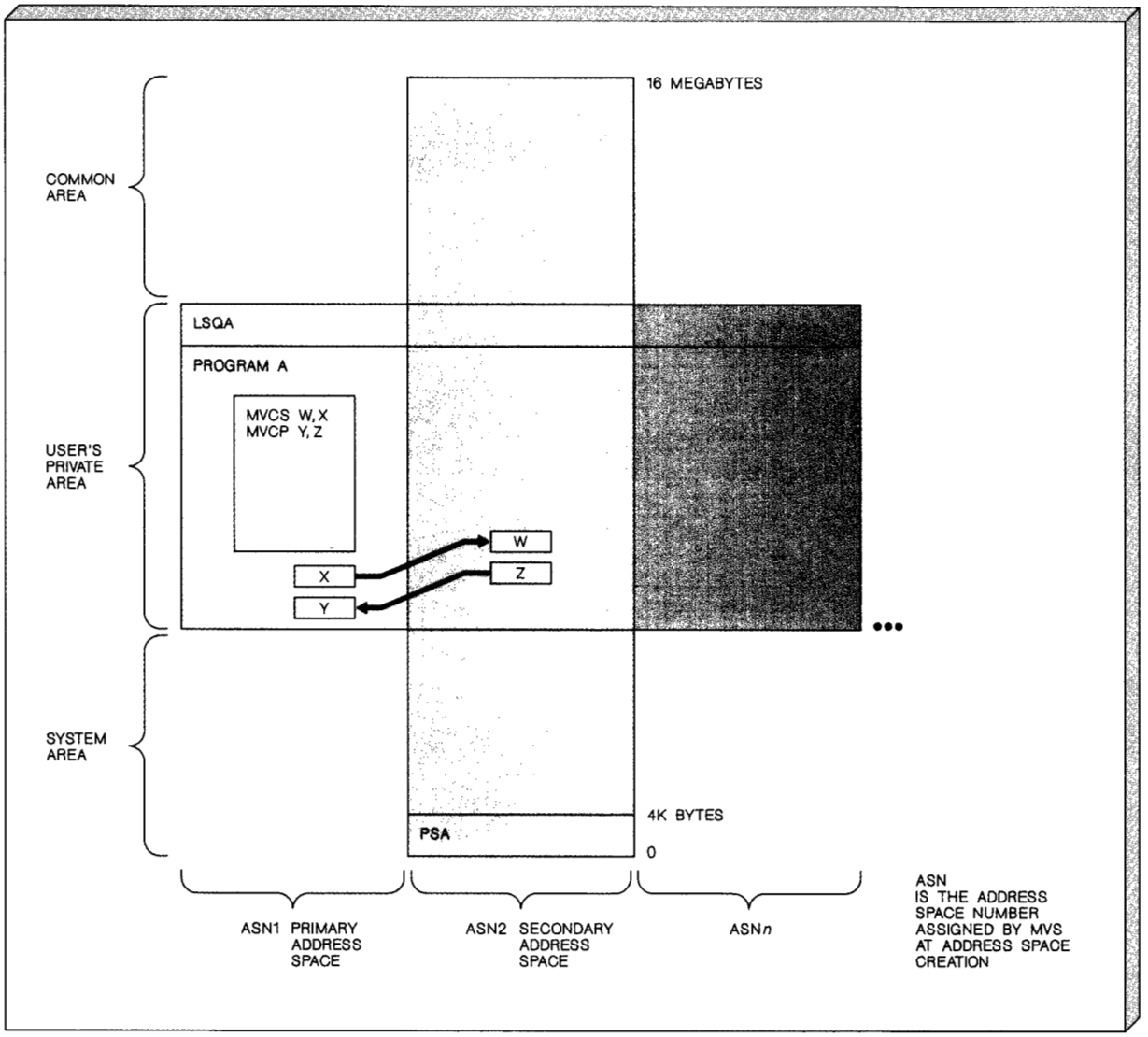
Still other products were utilizing the address space of MVS to isolate and protect their programs and control blocks, thus increasing the amount being placed in privately addressable storage and increasing the need for additional private storage. When an installation needed both types of products to meet its data processing requirements, the potential existed for virtual storage constraints. Increased function in many of these products and the additional number of jobs and users that could be supported on the 303X and 308X processors resulted in many MVS/SP users experiencing virtual storage constraints in both the common and private storage areas of the system.

MVS/SP 1.2 and 1.3—cross memory services and the DAS architecture facilities. The first products that provided some relief from the virtual storage constraints were MVS/SP Version 1.2 and MVS/SP Version 1.3, which provided cross memory services necessary to utilize the dual address space (DAS) architecture facilities. The products were announced in June 1980, signifying the first architectural change to the System/370 virtual addressing architecture since its announcement. Concurrent with MVS/SP 1.3, the System/370 dynamic address translation mechanism was changed to support extended real addressing (26-bit real addresses) and the common segment bit.

The combination of the cross memory services and DAS facilities provided two ways to help relieve use of common storage. They supplied a program with the capability to move data directly between separate address spaces and directly call another program in a different address space.

The DAS facilities permitted authorized programs to have concurrent addressability to two separate address spaces, a primary one and a secondary one. MVS and the DAS facilities provided the control structures and authorization mechanisms necessary to establish the primary and secondary relationship of address spaces.

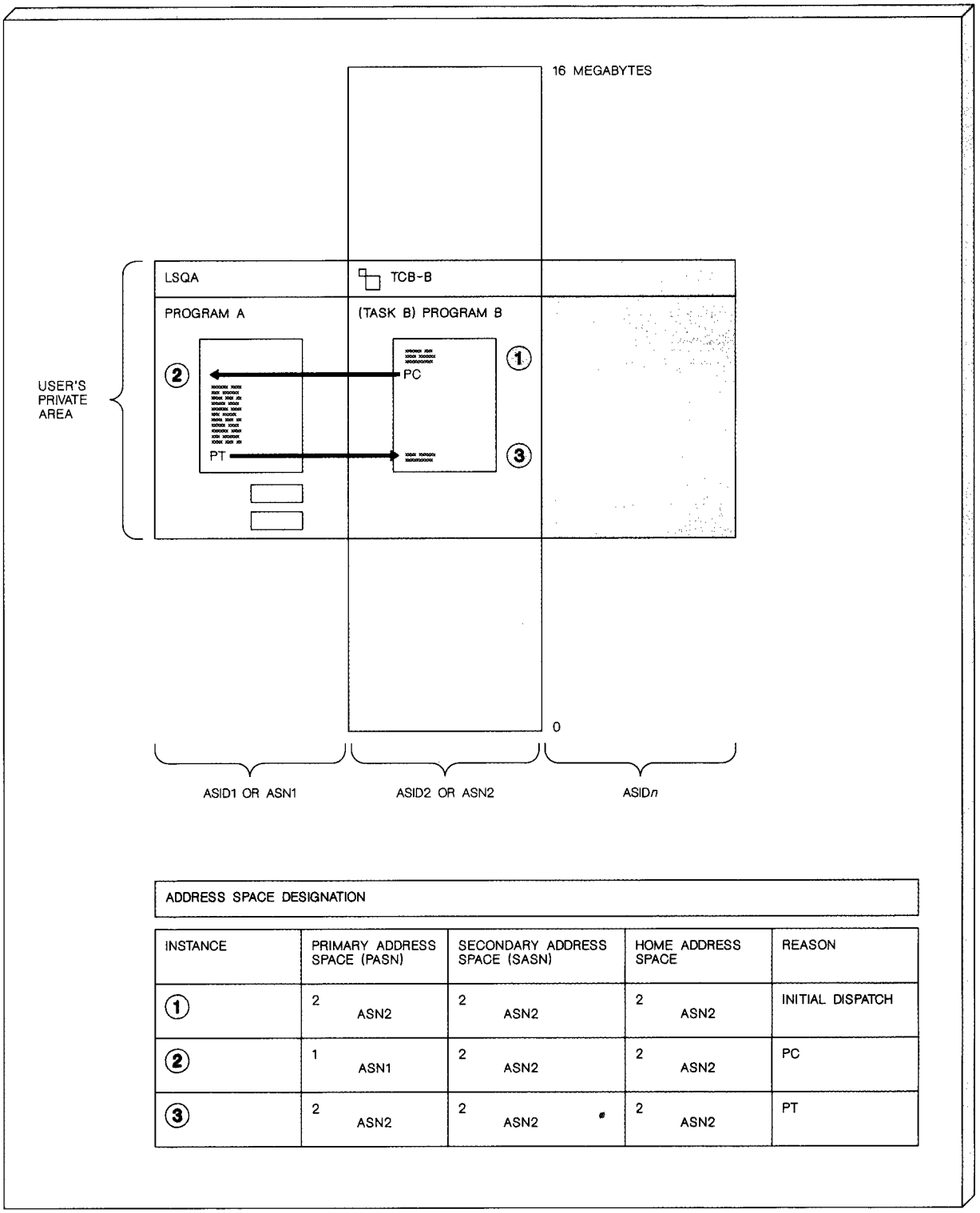
Figure 2 MVS cross-memory move of data



Two new move instructions, Move To Primary (MVCP) and Move To Secondary (MVCS), allowed data to be moved between the primary (PASN[1]) and secondary (the SASN[2]) address spaces as shown in Figure 2. Thus programs such as Program A in the figure could move an amount of data directly from its address space, the primary address space (ASN1, Address Space Number 1, shown in the figure) to or from a secondary address space (ASN2), without having to move the data or place the code in common storage. In order to move data between two address spaces before the DAS facilities were available, a program would have to move the data to common storage, then schedule a service request block (SRB) to inform a program in the second address space that the data were accessible. Such a move required multiple dispatches and synchronization points for this operation as well as the use of common storage.

Another benefit of cross memory services was the ability to perform synchronous linkage between two programs in two separate address spaces by using the program call (PC)/program transfer (PT) linkage instructions. Figure 3 illustrates this synchronous link-

Figure 3 MVS cross-memory linkage facility



ADDRESS SPACE DESIGNATION

INSTANCE	PRIMARY ADDRESS SPACE (PASN)	SECONDARY ADDRESS SPACE (SASN)	HOME ADDRESS SPACE	REASON
(1)	2 ASN2	2 ASN2	2 ASN2	INITIAL DISPATCH
(2)	1 ASN1	2 ASN2	2 ASN2	PC
(3)	2 ASN2	2 ASN2	2 ASN2	PT

Table 1 Initial users of the MVS cross-memory facilities

PCAUTH	COMTASK
GRS	ALLOCATION
JES3	IMS

age between programs in two address spaces ASID1 and ASID2. (ASID is an MVS identifier of an address space and is equivalent to ASN.) Thus, a program executing in Address Space 1 could directly call a program residing in Address Space 2. After the execution of the PC, Address Space 1 would become the primary address space (the PASN) and Address Space 2 would become the secondary address space (the SASN). Instructions and data would be accessed from Primary Address Space 1 on behalf of the program that was the target of the PC instruction. This program would use either the MVCP or MVCS instructions to access data from the program in the secondary space (space 2) when required to access its caller's data. When the called program had completed, it would return to the calling program in Address Space 2 by issuing a PT instruction which would reset the primary address space to Address Space 2, and instructions would once more be executed from the program in Address Space 2.

The combination of the cross-memory move instructions and the cross-memory linkage instructions allowed programs and data that previously required MVS common storage to be placed in the privately addressed storage of an MVS address space. This combination not only provided constraint relief of MVS common addressable storage, but also provided the isolation and protection of the MVS address space for components and products electing to utilize the cross-memory facilities. The initial components and products that utilized the MVS cross-memory facilities are listed in Table 1. Through the use of these facilities, GRS and PCAUTH were able to encapsulate their control block structures in their own address spaces. ALLOCATION and JES3 also placed some of their control data and data buffers in separate spaces. Use of the facilities not only isolated their structures to their own code, it eliminated the need for a significant amount of commonly addressable storage. COMTASK and IMS not only benefitted from the isolation but also eliminated the move of data through common storage in order to transfer data from one address space to another address space.

MVS cross memory services that allowed the establishment of a cross-memory environment required the user to be executing in a privileged state. This requirement meant that in general only system code, subsystem code, or privileged applications could take advantage of these facilities. However, this set of components and products provided much needed virtual storage constraint relief by placing their programs, data, and control blocks in their own private storage. In addition, this placement isolated data and control structures from accidental overlays by other privileged programs executing in the system.

31-bit extended addressing. Even before the DAS facilities were available, it was apparent that many major products, components, and applications needed more virtual addressing than was available in the private or common areas of a 16-megabyte address space. It was also true for some of the programs planning to use the DAS facilities. The processing power of the high-end processors had doubled between 1974 and 1978, doubled again between 1978 and 1981, and would more than double by 1983. A user could have experienced a 10- to 30-fold increase in processing capability within a single system since MVS was introduced in 1974. Many of the largest users of IBM's largest processors were experiencing difficulty in utilizing the full capabilities of the processors because of constraints, particularly in virtual storage addressing.

In 1981 IBM announced the most significant architecture and system change to System/370 since its announcement: the System/370 Extended Architecture (370-XA) and the MVS/Extended Addressing (MVS/XA) operating systems. This combination extended the virtual addressing capability of each MVS address space to 2 gigabytes (2 174 484 648 bytes). As shown in Figure 4, not only were the previous system and common storage areas significantly extended for system and subsystem code and control blocks, but the user was allocated an extended private area that could be utilized for any programs written to execute in 31-bit addressing mode.

In addition to extending the virtual addressing capability, the real addressing capability was also extended to 31 bits, and a new I/O subsystem architecture was introduced to eliminate many of the I/O constraints associated with the new processing capabilities of the processors.

MVS/XA and 370-XA included and enhanced the DAS facilities. Users of the DAS facilities had the ability to utilize the full 31-bit addressing. By release of MVS/SP 2.1, at least 16 major MVS products or components

Figure 4 Address space structure of MVS/XA

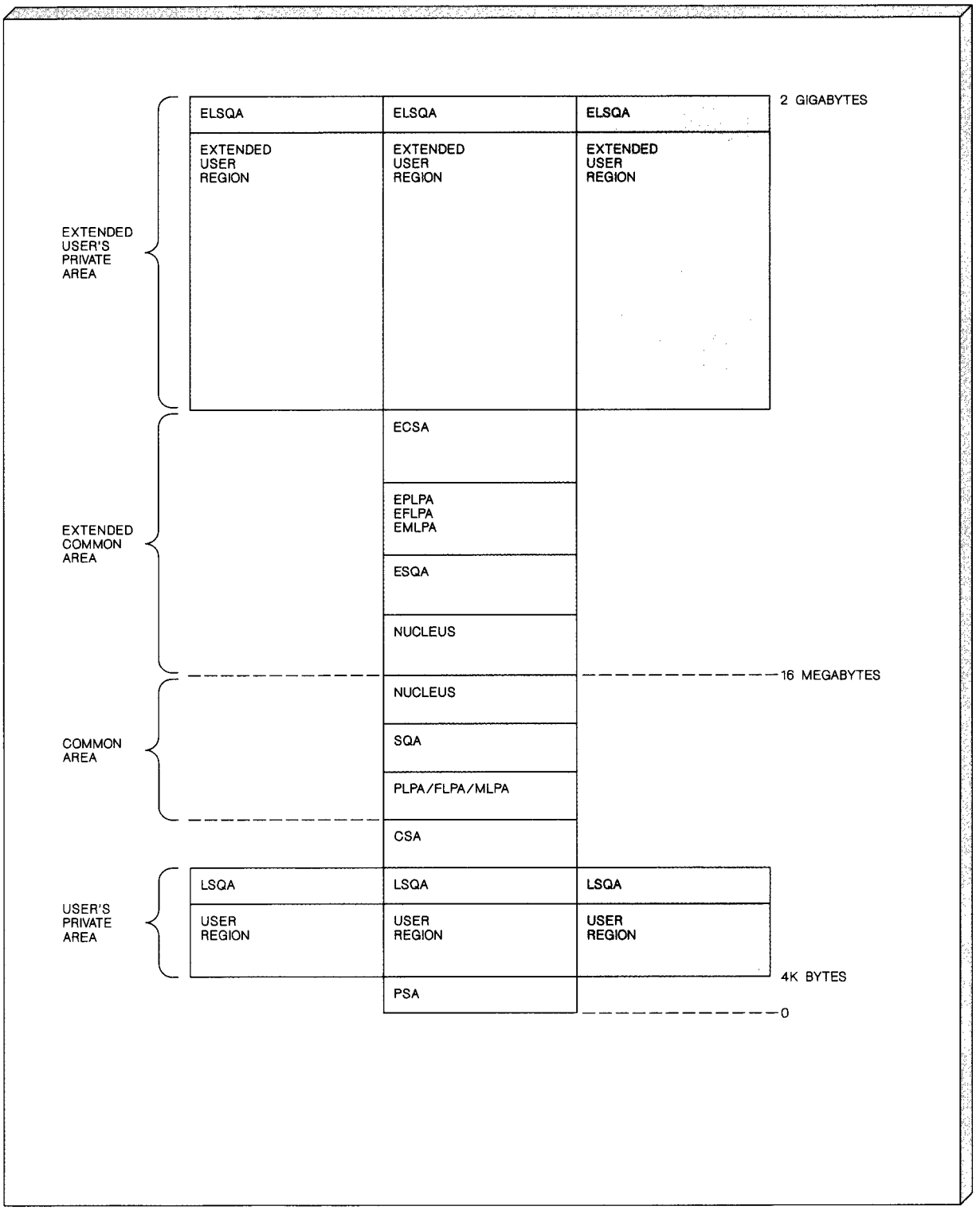


Table 2 DAS users on MVS/SP Version 2

ALLOCATION	JES2	CICS
CAS	JES3	DB2
COMTASK	LLA	IMS
DUMPING SERVICES	PCAUTH	IRLM
FSS	TRACE	RMF
GRS		

were using the DAS facilities and the extended architecture as indicated in Table 2.

A significant amount of time was needed to redesign, develop, and test the products and applications that would take advantage of these new capabilities. Even so, there were already indications that in certain environments, applications could use more virtual addressability than had been provided. In an attempt to address these concerns, improve on the limited capabilities of the DAS facilities, and yet be available before the user was seriously constrained again, ESA/370 and MVS/ESA were announced in 1988. The following sections describe the MVS/ESA primitives, facilities, and services.

MVS/ESA and MVS/SP Version 3

As previously mentioned, MVS/ESA and ESA/370 provide the following capabilities:

- Additional virtual storage addressing
- The full ESA/370 instruction set available to address multiple spaces
- Additional granularity in data isolation and sharing
- Improved linkage facilities for program call and branch linkages
- Improved linkage facilities between address spaces

It is important to note that existing programs can execute on the new system and architecture without any modifications, thus being completely compatible. For new programs and programs that are to be changed, MVS/ESA provides a new set of services and facilities that offer new capabilities to the user. Some programs will directly use the new architecture facilities to realize the new capabilities; others will benefit from utilizing the new system services. Others will not need to change to benefit—they benefit from using existing functions or services that have been modified to use the new facilities.

A very extensive description of how to use the MVS/ESA services and facilities can be found in Reference 3, particularly from a programmer's viewpoint. Additional information is available on the Virtual Lookaside Facility and Library Lookaside Facility in References 4, 5, and 6.

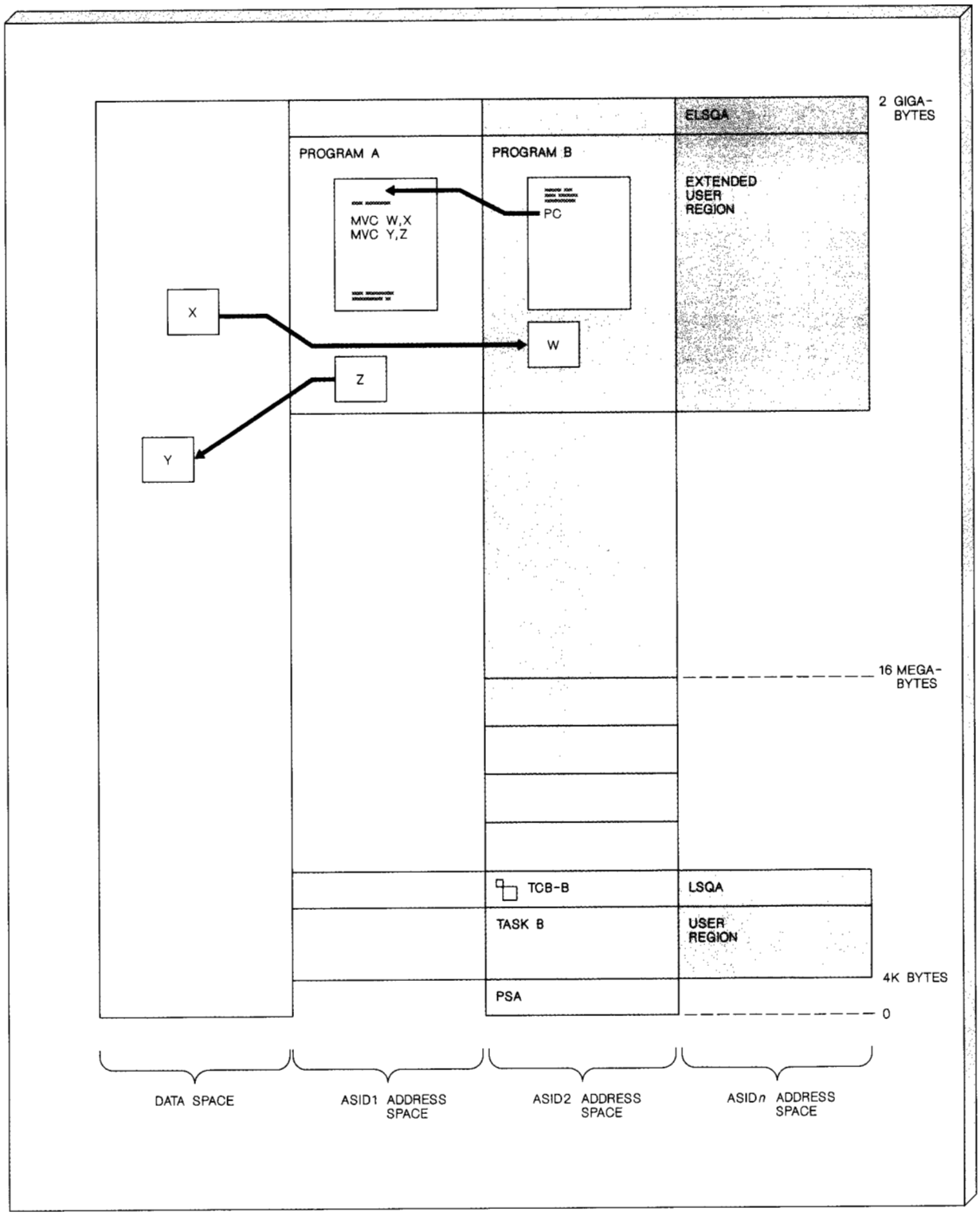
Additional virtual storage addressing. Prior to the availability of MVS/ESA, a program executing in 31-bit addressing mode had the capability to address 2 gigabytes of data and programs in its primary address space. A program using 31-bit addressing in cross-memory mode could address 2 gigabytes of data and programs in its primary address space and 2 gigabytes of data and programs in its secondary address space, but could only concurrently reference data in both address spaces via the two move instructions, MVCP and MVCS.

With MVS/ESA, a program written to execute in 31-bit addressing mode and use the ESA facilities has direct addressability to 2 gigabytes of data and programs in its primary address space. This program can execute instructions in its primary space, and with a single instruction can concurrently reference data in two other 2-gigabyte address or data spaces. The full set of ESA/370 instructions, with a few exceptions, is available for addressing data in these other spaces. This availability allows general, decimal, floating-point, and vector operations[7] to be performed on data in spaces separate from the space where the instructions are executed.

The new concepts introduced in MVS/ESA and ESA/370 were the data space, access list, access registers, and access register (AR) mode. The last three concepts are described in detail in the papers by Scalzi et al.[8] and Plambeck.[9] Data spaces are introduced in Scalzi et al.[8] and will be described here in more detail later. Briefly, a data space allows a program to have a virtual addressable space of up to 2 gigabytes of data. An access list is the construct in the architecture through which the access of a program to other address spaces or data spaces is governed. Access registers complement the general-purpose registers and identify the space where the data will be accessed. This addressing capability is only available when a program is executing in AR mode.

Figure 5 demonstrates how a program (Program A) can move data between its own address space (ASID1) and a data space, as well as between another address space (ASID2) and a data space using the System/370 instruction set, in this case, MVC. Note that the PSA (prefixed storage area), the common area, and the extended common area are not mapped in the data

Figure 5 MVS/ESA addressing to multiple data and address spaces



space. The entire 2 gigabytes of address space are available to map a user's data.

With these new constructs, a program written to execute on MVS/ESA in AR mode can address up to 16 separate spaces without changing a control register or an access register. One of the spaces must be an MVS address space, the PASN, where the instructions are executed. By simply changing the contents of an access register, a program can access up to 256 different spaces. Finally, by changing the content of an access list and an access register, a program can access more than 256 spaces. The relationships between address spaces, access registers, and access lists are presented later in Figure 8.

A data space is a new type of MVS address space.

In addition, MVS/ESA provides *hiperspaces* and *window services* for programs that cannot or will not be written to execute in AR mode. This provision is particularly important for HLL programs that must reference large amounts of virtual storage. Using hiperspaces and window services, the programs are allowed to view portions of a very large data object in their own primary address space. The current limit for a permanent data object is 4 gigabytes of data, and the size of the window is limited by the amount of storage the program can obtain with the MVS GETMAIN service. The parts of the object not being currently viewed do not occupy real storage frames and will be located on expanded or auxiliary storage. Hiperspaces and data window services are described in more detail by Rubsam.[10]

In addition to hiperspaces, data spaces, access lists, access registers, and addressing in AR mode extend the amount of virtual addressing available to a program and also provide for additional levels of data sharing and isolation. How MVS/ESA and its services provide for these capabilities will be described for each function.

Data spaces

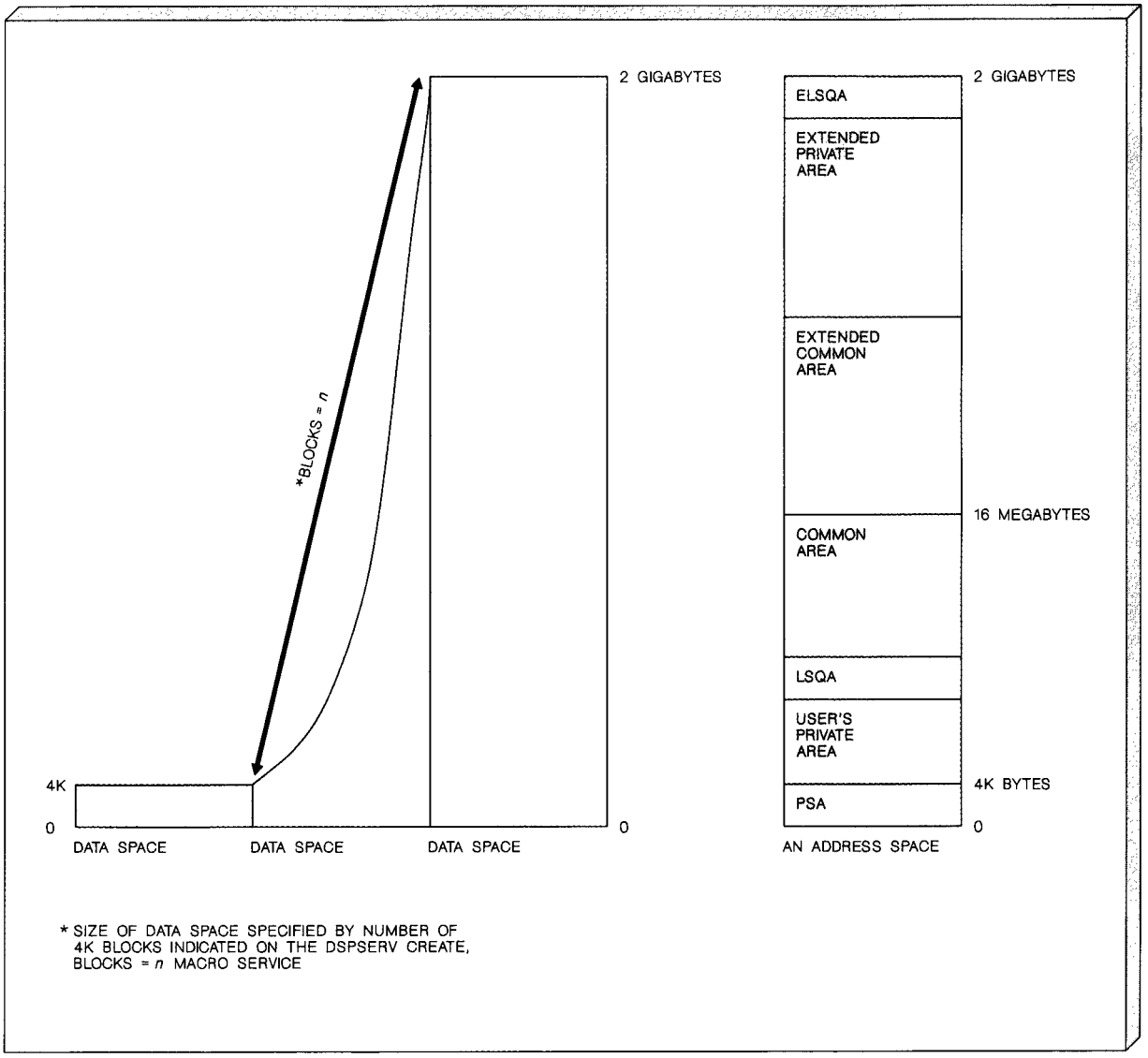
A data space is a new type of MVS address space that contains only data; that is, no facility exists to execute instructions in a data space. It only contains the data the user places in the space. The entire space is available to the user, contains no system control blocks, and maps none of the MVS/ESA common area. A data space is a linear mapping of virtual storage that begins at virtual address 0 or 4K bytes, depending on the processor model on which MVS/ESA is executing, and can extend up to 2 gigabytes. Figure 6 is a comparison of an MVS address space and a data space.

The newly created data space will appear to the user to be initialized to zeros (X'00'). It does not require any real storage until the user first attempts to use the space and then only when a new page is used. If pages in the data space are never accessed, they never need to exist in real storage. The data space is assigned a storage protection key of the requestor. However, privileged programs can request that a specific storage protection key be assigned. In addition, privileged programs can request that the data space be fetch-protected, and they can also create data spaces that may be shared between tasks or address spaces. The nonprivileged program can share its data spaces only with subtasks that it created. See Figure 7 for an example of how spaces may be shared or isolated between tasks and address spaces.

Data spaces, MVS/ESA, and ESA/370. Data spaces have some very unique properties that are provided by the design and constructs of MVS/ESA and its use of ESA/370. When executing in AR mode, data and instructions may be accessed from the same or separate spaces. This capability allows programs to reference data residing with the program (e.g., data constants and relocatable address constants). It also provides for a data space that can be completely void of any programs and still be accessed with the full ESA/370 instruction set. A data space, however, cannot be the target or source of an MVCP or MVCS instruction because a data space can never be a primary or secondary address space. MVS/ESA prevents data spaces from being eligible as primary and secondary address spaces. This constraint along with the architecture prevents data spaces from being designated spaces for a load program status word (LPSW), PT, PC, or branch instructions and, hence, prevents any (accidental or otherwise) execution of instructions in data spaces.

Creating a data space. The data space manager is the entity by which a data space is created. The data

Figure 6 MVS/ESA data space versus MVS address space



space manager resides in a separate address space and uses its own data spaces to contain the control structure for all other data spaces in the system. This arrangement isolates the control structures that identify and map the resource (the data space) to the manager of the resource (data space manager). It also avoids using any of the virtual addressing space in the requestor's data space to map the data space. The DSPSERV macro facility is the MVS/ESA interface to the data space management services. The macro interface will provide the necessary program call to the appropriate data space service in the address space of the data space manager. The macro is available to programs written in BAL. HLL programs may call user-written BAL programs that will create, delete, and manage data in data spaces.

The ability to create a data space is provided by the DSPSERV CREATE macro. When a request for a data space is made, the service will create the data space and return a unique token, named a STOKEN, to the requestor. At this point, the data space has no ad-

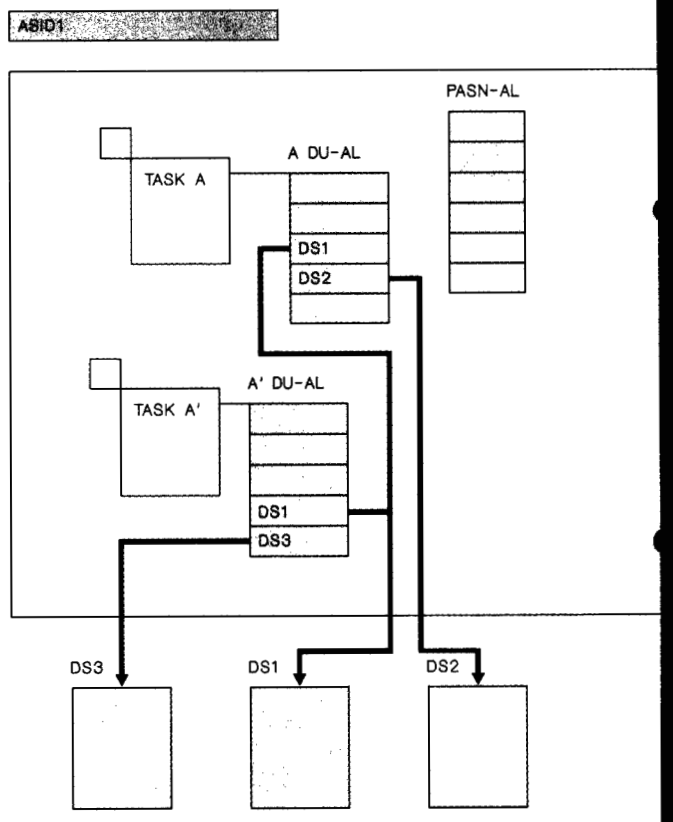
dressability. The STOKEN must be used as input to the access list services (ALESERV) to add the data space to an access list in order to establish addressability to the space. Both privileged and nonprivileged programs can create data spaces. The size of the data space, the number of data spaces, and the total amount of storage available for all data spaces created on behalf of a single address space (typically, a job or a user) is limited by the system. These values can be established on an installation basis and can be overridden in the SMF exit, IEFUSI, of the installation. Details are described in References 5 and 11.

Data can be placed into a data space by programs executing in AR mode and using ESA/370 instructions to move data from one space to another. Data can also be placed into a data space by I/O operations. Current support for I/O operations in a data space is limited to data-in-virtual (DIV) services. The DIV services are described in detail by Rubsam.[10]

Data space sharing and isolation capabilities. A data space may be assigned to a single task and hence be addressable only to programs executing under that task. A data space may be shared between a task and any subtasks that it creates, or it may be shared between any number of tasks and/or address spaces. The creator of the data space must indicate on the DSPSERV CREATE request that SCOPE=ALL is the option if the data space is to be shared between address spaces and unrelated tasks. This option is only available for privileged programs, and the owner of the SCOPE=ALL data space must be a nonswappable address space. A data space created with the SCOPE=SINGLE option can only be added to the access list of the creating task. However, the owning task can share the data space with any subtasks it creates by specifying ALCOPY=YES on the ATTACH request. This option will provide the subtask with a copy of the access list of the creating task at the time of the ATTACH.

Figure 7 shows some of the combinations of data space sharing and isolation that are available in MVS/ESA. Data spaces (DS2, DS3, DS7, and DS8) are only accessible from single MVS tasks whose (DU-AL) dispatchable unit access list contains those data spaces. DS1 can be accessed by both Task A and Task A′ in Address Space 1. However, other tasks in Address Space 1 would not have access to DS1. DS9 can be accessed by Task B in Address Space 2 and Task C in Address Space 3. DS4 can be accessed by all tasks in Address Space 2, and DS6 can be accessed by all tasks in Address Space 3 because these data

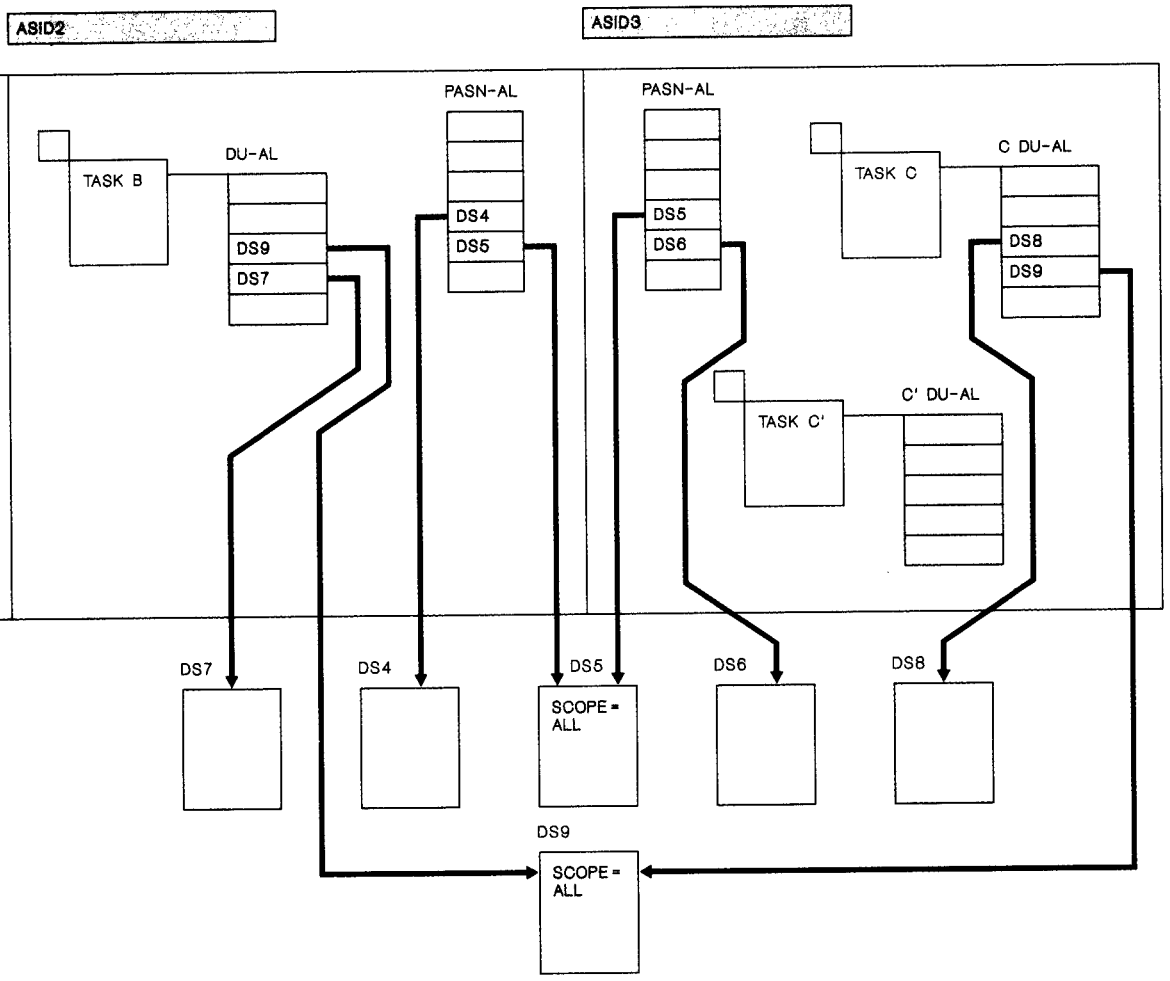
Figure 7 Data space isolation and sharing capabilities of MVS/ESA



ACCESS CAPABILITIES (A = ACCESSIBLE)

	DS1*	DS2	DS3
TASK A	A	A	
TASK A'	A		A
TASK B			
TASK C			
TASK C'			

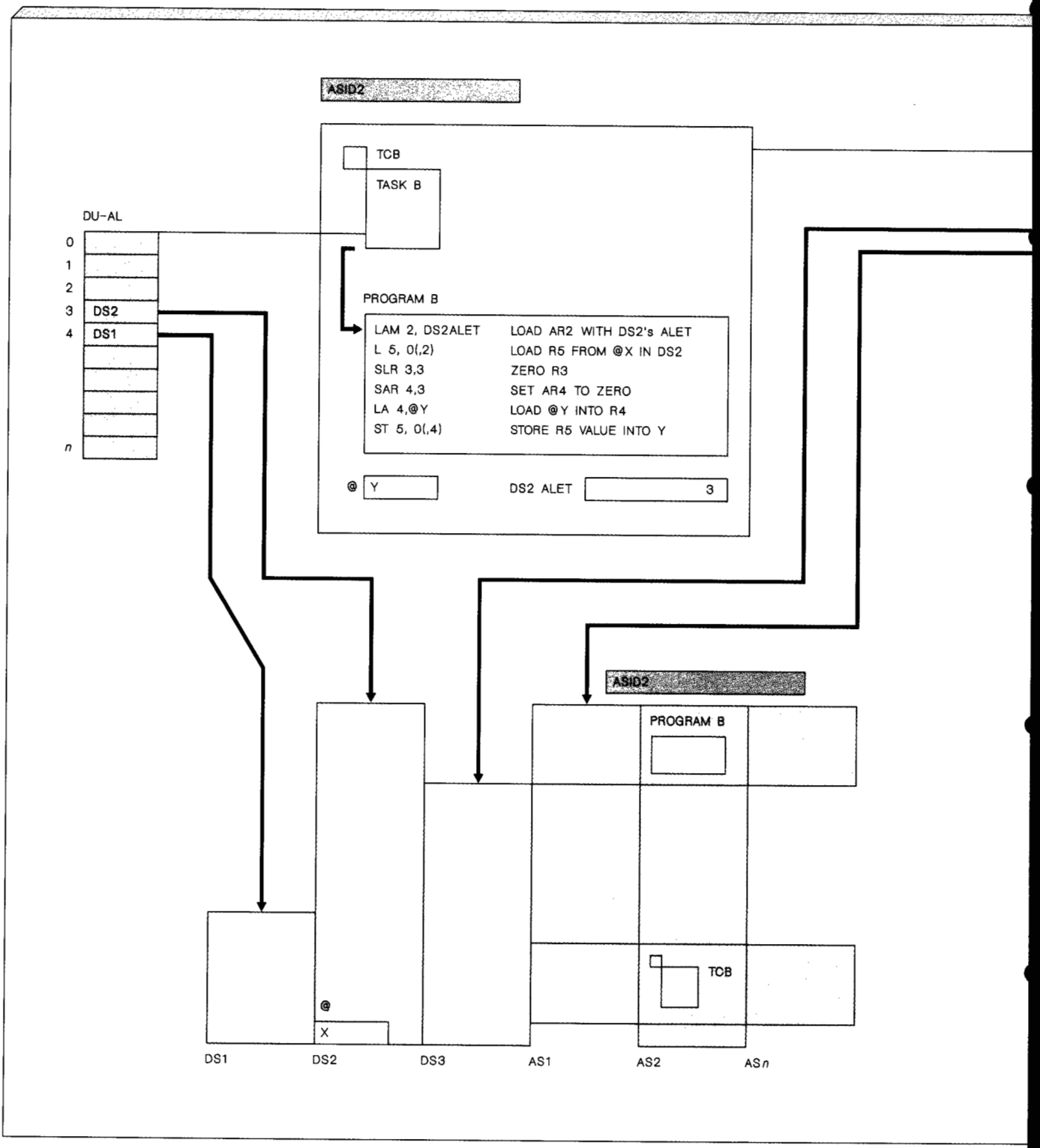
*DS1 IS SHARED VIA ATTACH W/ALCOPY

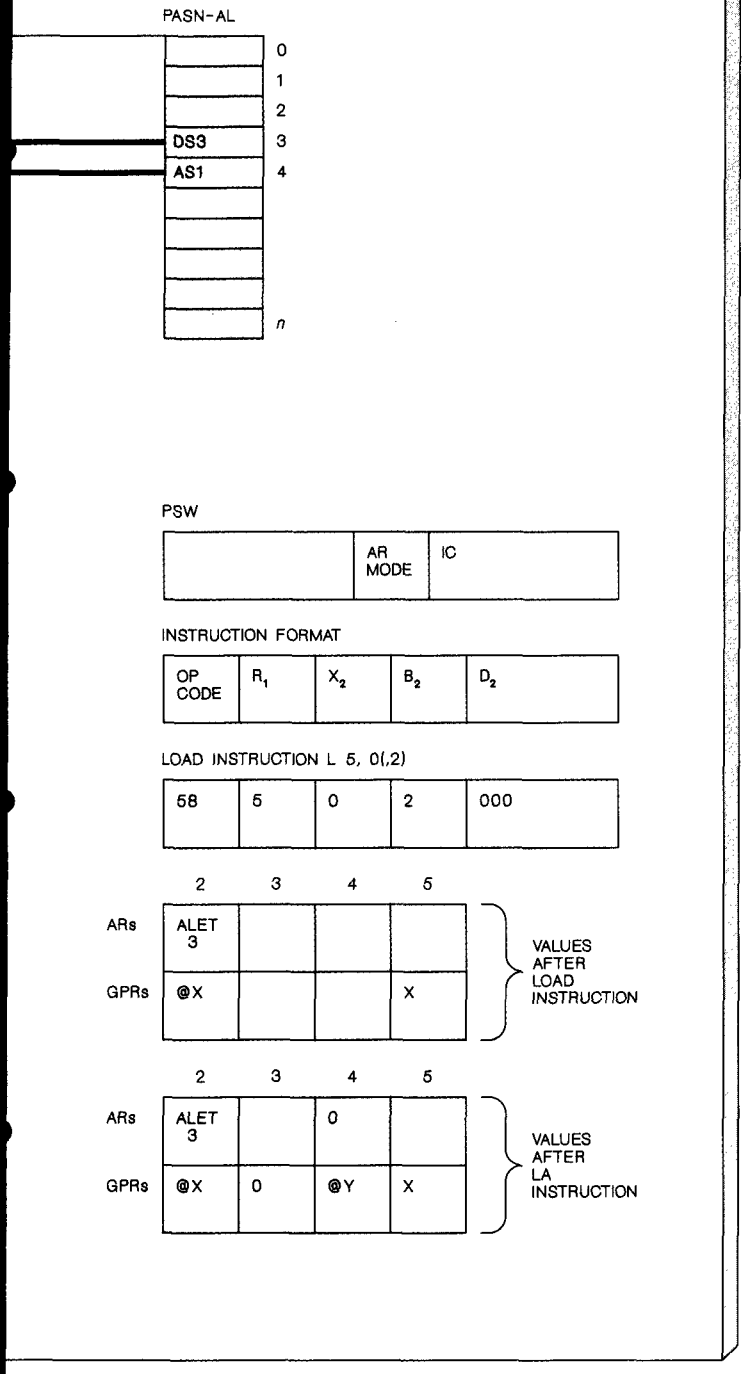


ACCESS CAPABILITIES (A = ACCESSIBLE)

	DS4	DS5	DS6	DS7	DS8	DS9
TASK A						
TASK A'						
TASK B	A	A		A		A
TASK C		A	A		A	A
TASK C'		A	A			

Figure 8 DU-AL and PASN-AL address capability to multiple spaces (DS = data space, AS = address space)





spaces have been added to the PASN-AL (the primary address space access list). DS5 can be accessed by all tasks in both Address Spaces 2 and 3 as the data space has been added to the PASN-ALs of both address spaces.

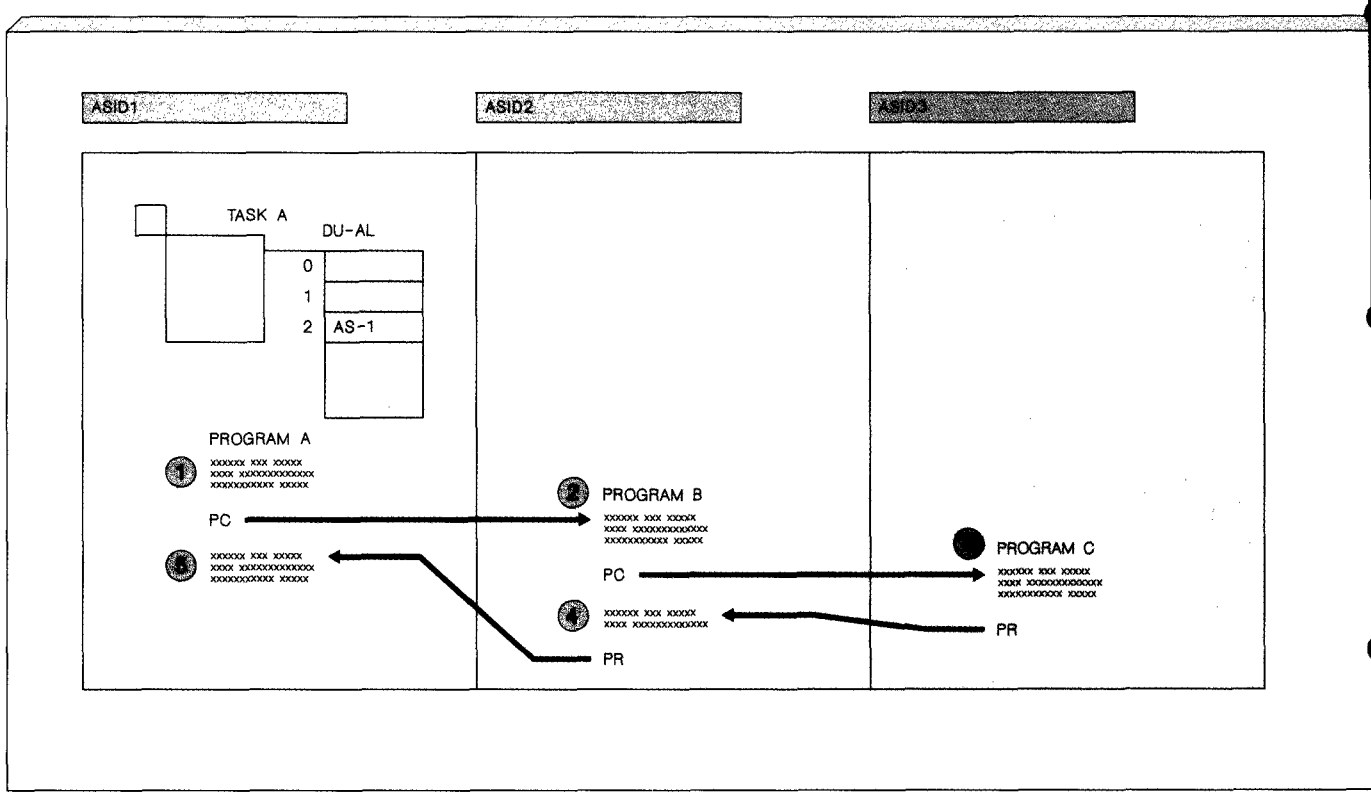
Access list

The access list is the construct through which the ESA/370 hardware and MVS/ESA determine which spaces a program is authorized to access. In addition, the hardware uses the access list in its access-register translation process to select a segment table descriptor (STD) for resolving the virtual address through its dynamic address translation mechanism. The ESA authorization and address translation mechanisms are utilized when a program executing in AR mode must reference a virtual address in spaces other than the primary space where instructions are being executed. A program can add an entry to the access list for an address space or data space if the program is authorized to that space.

The ALESERV macro facility is the MVS/ESA interface to add spaces to the access lists. The STOKEN of the address space or data space the user wishes to add to the access list must be specified as input to ALESERV. The STOKEN is a unique identifier of an address space or a data space. The ALESERV service returns an access list entry token (ALET) that can be loaded into an access register by the program. In ESA/370, each general-purpose register has a corresponding access register. When executing in AR mode, the System/370 hardware will decode the instruction. If the instruction designates a base register, the corresponding access register will be used to determine the STD of the space (address or data space) in order to resolve the virtual address (the base register value plus the displacement and index register value if appropriate). The program thus has the capability in AR mode to access the space with any instruction by loading an access register with the desired ALET and using the corresponding general-purpose register (GPR) as a base register.

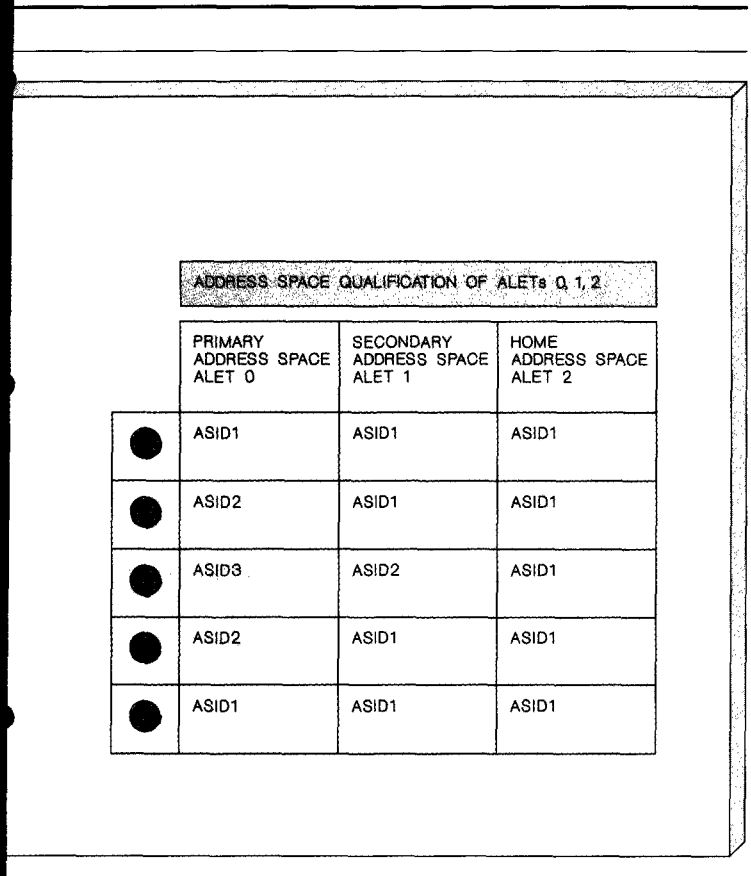
Figure 8 shows how a program (Program B) executing in AR mode loads the value of a word (X) from Data Space 2 into GPR 5 and then stores the value at the address of word (Y) in the address space of Program B. Access Register 2 is loaded with an ALET that represents Data Space 2. Assume the data space was created, added to the DU-AL via the ALESERV, and was assigned the ALET 3 via ALESERV services. GPR 2 is used as a base register for the load instruc-

Figure 9 MVS/ESA addressing with ALETs



tion, and since the program is executing in AR mode, AR 2 is used to locate the access list entry that defines the space designation. The hardware will use the contents of GPR 2 and the displacement value (D2) to calculate the address within the data space. If an index register had been specified, the contents of that GPR would have been added to the D2 and GPR 2 values to determine an effective virtual address in Data Space 2. The next two instructions are used to set AR 4 to a zero value. An ALET value of zero is always qualified to the current primary address space (the space where the current instruction is being fetched). The STORE instruction stores the value in GPR 5 into the address specified by GPR 4 in Address Space 2. The ALET value of zero in AR 4 designated the primary space, Address Space 2, as the space where the address was to be resolved. At this point in the example, the use of Registers 2 and 4 as base registers will continue to result in address resolution to the data space and address space, respectively, as long as AR mode is maintained and the contents of ARs 2 and 4 or Access List Entry 3 are not changed. If AR 4 had been set to ALET 4 of the DU-AL or ALET 3 or ALET 4 of the PASN-AL, the STORE would have been made into DS1, DS3, or AS1, respectively.

In MVS there are two types of dispatchable work units. One is represented by a task control block, or TCB. The TCB preceded MVS but is still used to represent work units within an address space. The other unit of work is represented by a service request block, or SRB. The SRB was introduced with MVS and is often used to request work in another address space. MVS/ESA provides each work unit (TCB or SRB) its own access list and a dispatchable unit access list (DU-AL). Each address space also has its own PASN access list (PASN-AL). Any program executing under the work unit can add spaces to the DU-AL if it has the authority. However, only privileged programs can add spaces to the current PASN-AL. Any program executing under a task has access to any of the spaces on the DU-AL and any spaces on the PASN-AL.



An exception to the access authority is that address spaces may be added with a *private* attribute. These address spaces may only be accessed by programs executing with an extended authorization index (EAX) that is authorized to access the address space. A unique EAX value can be associated with a PC routine when the routine is defined to the system by a privileged program. This provides the PC routine with greater authority to address private address spaces than the task or address space that invoked the PC routine.

The DU-AL is initialized as an empty list except as follows: Entry 0 and 1 are not used. ALET 0 and 1 have special meaning. ALET 0 always refers to the current primary address space, and ALET 1 always refers to the current secondary address space. ALET 2 is initialized to the home address space, which is where the work unit is initially created or dispatched. The home address space of a work unit never changes, whereas the primary and secondary address spaces may change as a result of space-switching PC/PRs.

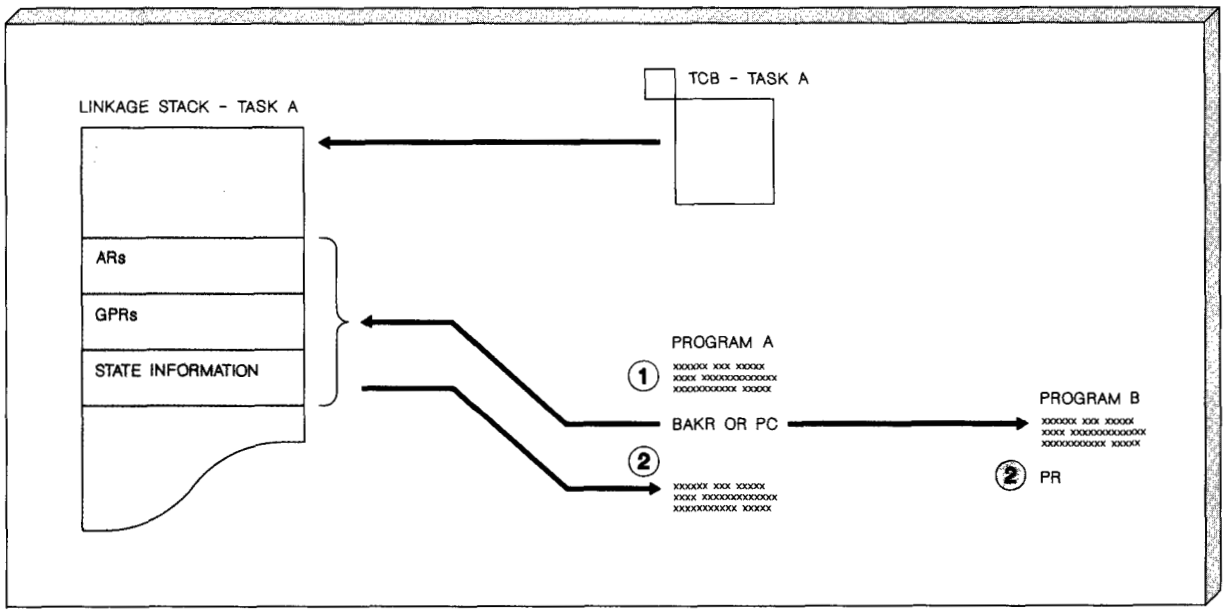
Figure 9 shows how the address space qualification of ALET 0 and 1 change across space-switching program calls and how ALET 2 remains qualified to the home address space.

The linkage stack facility

Although the dual address space facility and the cross-memory mode support are very restrictive, a significant number of products and components were using the facilities to provide their services in a separate address space. These services were callable by the space-switching program call/program transfer (PC/PT) instructions. The services utilized the separate address space to increase the amount of virtual storage available to them, provide for the isolation and protection of their data, and have their functions only accessible through well-defined interfaces (the space-switching PCs for the requested function). The PC of the DAS facility required the called function to save a great deal of control information about the caller (e.g., caller's addressing mode, authorization state, program key mask [PKM], and return address). This control information was required for the PT instruction to return to the caller and was in addition to the general-purpose registers that the service had to save and restore. MVS offered services to assist these functions in status saving and restoring, storage management, and functional recovery, but those services were limited to key 0, supervisor state routines. ESA/370 and MVS/ESA provide many improvements for these cross-memory mode servers and other programs as well.

In addition to the full ESA/370 instruction set for managing data across multiple spaces, MVS/ESA has a linkage stack facility to assist in managing program calls. The linkage stack facility provides an automatic status saving and restoring facility for both privileged and nonprivileged programs across program call and program return linkages and the new branch linkages. This facility relieves the called programs from managing return status. In addition, it provides for more ready usage of registers since the contents of the registers have been saved on a stack. Saving is automatic on execution of the new calling instructions. MVS/ESA also provides the ability to establish and remove a recovery environment that is associated with the call and return of a program. Because it is associated with the MVS/ESA PC/PR, this recovery is available at no additional cost to the path length of the mainline program.

Figure 10 MVS/XA linkage stack facility



Stacking program call, branch and stack, and program return. MVS/ESA provides each work unit (TCB or SRB) with its own linkage stack. The linkage stack is used to save information about the execution state of a program at the point of the call to another program. The information that is saved in a linkage stack entry on a call includes the general-purpose registers, the access registers, and certain state information (e.g., program status word [PSW], PASN, SASN, and PKM). Most of this information is restored from the stack entry whenever the called program returns to its caller. Figure 10 illustrates the linkage stack facility between calling programs.

Two new instructions, the branch and stack (BAKR) and the program return (PR), were provided in ESA/370 to utilize the stack and effect a call/return linkage. The program call (PC) instruction was modified to recognize a stacking option and to perform a stacking program call. A stacking PC can be made to a program in another address space or to a program in the same address space as the caller. It can also be used to establish an associated recovery routine for the called routine. The BAKR instruction only permits branching to a program in the same address space as the calling program. PR must be used to return from the stacking PCs (space switching and nonspace switching) and all BAKR linkages. The PR will automatically result in the deletion of the associated recovery environment established on the PC. Since these new linkages automatically save and restore status on the stack, existing linkage conventions of saving and restoring registers are redundant.

Note that the conventions for writing a PC routine and the services required (LXRES, ETDEF, ETCRE, ETCON[12]) to reserve a PC number and make the PC routine available to the system are described in detail in Reference 3.

Both the stacking program call and the branch and stack instructions result in a new stack entry being created and the following information being saved in that stack entry: GPRs 0-15, ARs 0-15, the current PASN, SASN, PKM, EAX, and PSW (including the return address), and the PC number if the instruction was a PC. If the BAKR instruction was issued with a nonzero R1 operand, the R1 operand will be saved as the return address. The high-order bit of R1 will indicate the addressing mode of the return. If the R1 operand specifies GPR 0, the return address will be to the instruction following the BAKR in the same addressing mode that existed at the time the BAKR was issued. BAKR is much like the branch and link (BALR) instruction, in that, if the R2 operand specifies GPR 0, the next instruction to be executed will be the instruction after the BAKR. If the R2 operand specifies an address, the BAKR instruction will branch to that address to execute the next instruction.

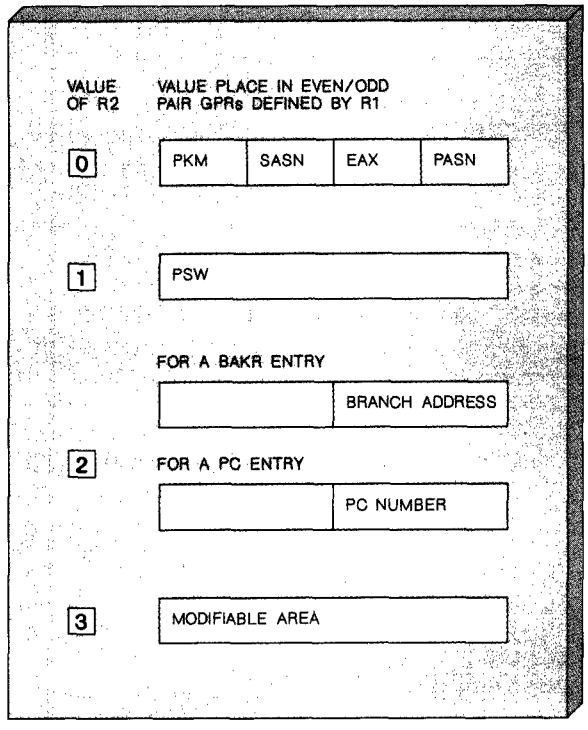
The return mechanism for both of the stacking calls is the PR instruction. This instruction results in restoring GPRs 2-14 and ARs 2-14 from the values saved in the current stack entry. The 0, 1, and 15 GPRs and corresponding ARs are not restored and are unchanged from the values in those registers at the time of the PR, thus permitting the called routine to pass back return information to the calling routine. As a result of the PR, the PSW that was saved on the BAKR or PC instruction in the current stack entry is loaded, thus restoring the return address and the caller's addressing mode and PSW KEY. If the call was a PC, the PR will also result in the PASN, SASN, PKM, and EAX being restored from the stack entry.

Additional stacking operations. In addition to the linkage instructions that utilize the stack, extract instructions allow a program to obtain the data that were stored in the current stack entry. The Extract Stacked Registers instruction, EREG R1,R2, permits a range of access and general-purpose registers to be specified and will load those registers with the corresponding register values saved in the current stack entry. The Extract Stacked State instruction, ESTA R1,R2, will extract one of four eight-byte state entry fields in the current linkage stack entry and place the information in the pair of general registers designated by R1. The selection of the eight-byte entry field is based on the value in R2. Figure 11 illustrates the values a program can obtain from the state entry information.

The modified area in the stack is a double word in each stack entry that is available to the program to make modifications through a Modify Stack State instruction, MSTA R1. The R1 field specifies the even-number register of an even-odd pair of general-purpose registers whose values will be stored in the modified area of the current stack entry. A typical use of this field would be to allow a program to coordinate its mainline function with its recovery process. For instance, the mainline function might request storage using the GETMAIN service and place the address of the storage in the stack. The mainline function could record information in the storage area to aid in its recovery. When the recovery retry routine received control, it could extract the data (the address of the storage) from the stack by the ESTA instruction.

These instructions permit programs utilizing the stack to have access to data in the stack without needing direct addressability to the stack. MVS/ESA furthers this isolation of the stack by locating the

Figure 11 MVS/XA linkage stack extract stacked state



stack in one of two new storage subpools. For tasks, the linkage stack segments are obtained from disable reference storage (DREF)[13] subpool 215 in extended private storage ELSQA, the extended local system queue area. For service request blocks (SRBs), the linkage stack segments are obtained from DREF subpool 247 in extended common storage, ESQA. Both of these subpools have the attributes of key 0, fetch-protected storage. The storage backing these subpools may be paged to expanded storage if it exists. However, the data may never be paged to auxiliary storage. This attribute allows the system to page the storage, backing the linkage stacks to expanded storage, and yet allows disabled callers to utilize the stack. This is permitted since page faults to expanded storage can be resolved synchronously without requiring the suspension of the faulting program.

Linkage stack and MVS control structures. The normal linkage stack for each work unit has the capacity to hold 96 entries which would allow up to 96 calls to be held without an intervening return. If a program determines that the nesting level of calls would exceed this value, the maximum number of entries can be extended to 16 000. The LSEXPAND

NORMAL=nnn macro invokes this service. However, this must be done prior to reaching the current maximum limits. If the current maximum limit is reached, a recovery stack with the capacity to hold 24 entries is provided such that the recovery routines can use the stack while cleaning up from a stack-full exception condition. The recovery stack can also be extended up to 4000 entries if the expansion is done while executing under the normal stack.

MVS/ESA coordinates the linkage stack with the previous execution work unit structures that exist in MVS.

MVS/ESA utilizes the ESA/370 linkage stack architecture for these large stack capabilities. The entire stack does not have to be initialized, as the stack manager will process stack segments of 4096 bytes. After the first segment has been initialized, subsequent stack segments will only be used and connected to the previous stack segment if required by the depth of the calling sequences that use stacking calls.

MVS/ESA coordinates the linkage stack with the previous execution work unit structures that exist in MVS. For many years, programs have been represented by request blocks (RBs) that have been chained to the task, TCB, under which they were invoked. This representation has often been referred to as the TCB/RB queue or stack. These RBs could have been created by a program making a synchronous request, a supervisor call, or SVC, instruction which would result in a supervisor request block (SVRB) being placed on the queue for the SVC routine. A synchronous request for a program made by a LINK request would have placed a program request block (PRB) on the queue for the called program. Asynchronous events such as a timer event would have placed an Interrupt Request Block (IRB) on the RB queue of the execution unit. These structures are still a basic part of the MVS task structure. When the linkage stack was designed, there had to be a corresponding relationship of programs called by stacking facilities and programs called by the previous calling conventions. MVS/ESA keeps track of which linkage stack entries are formed under which RB. Programs executing under one RB are not permitted to delete stack entries created under previous RBs. If a program represented by an RB exits without having deleted all the stack entries created in its behalf, MVS/ESA will purge the linkage stack back to the entry that makes it consistent with the next RB on the MVS TCB/RB queue.

Recovery and the linkage stack. In addition to maintaining the correspondence between RBs and the linkage stack, MVS/ESA correlates recovery routines and the appropriate entry on the linkage stack. If specify task abnormal exit (ESTAE) recovery routines are required, the ESTAE must be created and deleted under the same linkage stack entry, just as previously ESTAEs had to be created and deleted under the same RB. If an ESTAE is created by a stacking program (e.g., a program invoked by a stacking PC) and that program attempts to return to its caller via an unstacking operation (the PR), an exception will be recognized by MVS/ESA. The system will delete the ESTAE associated with that stack entry and then permit the return to complete such that the stack and the recovery are kept in synchronization.

ESTAE routines will receive control without any assurance of the position of the linkage stack. However, the system will keep track of the current stack entry and the stack entry that existed at the time the ESTAE was created. If the ESTAE routine elects to retry, the retry routine will be given control with the current linkage stack entry being the same entry that existed whenever the ESTAE was created. Function recovery routines (FRRs) have the option to retry with the linkage stack entry that was current at the time of the error or with the entry that was current at the time of the SETFRR invocation.

MVS has many services available for establishing recovery routines. Not all of these services and not all recovery mechanisms are available in AR mode. The SETFRR, ESTAEX (the specify task abnormal exit facility for AR mode programs), and the associated recovery routines are available for access register mode programs.

Additional services for cross-memory mode programs. As previously discussed, the services available to cross-memory mode programs initially provided support only for key 0 programs. In MVS/ESA SP3, the ESTAEX service provided conditions where nonprivi-

leged programs executing in a cross-memory environment could establish ESTAE recovery routines. The LINKAGE=SYSTEM options were added to wait and post services, providing a wait and post function that could be invoked in cross-memory mode and by nonprivileged callers. A new storage service was provided to allow privileged and nonprivileged callers to acquire virtual storage in either cross-memory or access register mode and in either problem or privileged state. A new recovery termination service, RESMGR, is provided for privileged programs to establish a dynamic resource manager exit that will be given control on termination of tasks or address spaces. The service will allow the user to specify an exit to receive control if a task terminates, be it a particular one or not. Like the task termination exit, an address space termination exit can be specified for a specific address space or all address spaces. The service will also allow the deletion of the resource manager from a list of exits that are called by the resource termination recovery manager of the system.

Additional enhancements to the stacking program call. A new macro interface has been provided to assist the programmer in defining the options available on a program call. This new macro, ETDEF,[12] assists in defining the entry table entry fields for the program call. Several new options are available on the program call to assist in establishing the desired program environment. One option previously mentioned is the ability to specify an associated recovery routine exit (an address of an exit or a name of an exit that resides in the link pack area). By specifying an associated recovery exit, the mainline function of the PC routine does not have to establish or delete a recovery routine in its mainline processing. Whenever the stacking PC places the PC number in the stack, the system has enough information to locate the recovery exit if there is a failure in the PC routine. The recovery routine is automatically deleted when the routine returns via program return.

Several new options are available to the stacking PC that will simplify the programming for a PC called routine. Three options that all PC routines may want to consider are

1. The address space control (ASC) mode that the PC routine will be given control of may be specified as primary or AR mode. The PC will set the new ASC mode. The old mode is saved in the stack entry.
2. The PC routine can be entered with the SASN set to the PASN of the PC routine or the caller's PASN. Selecting the PASN of the PC routine eliminates the requirement that the called space must be authorized for the caller's space.
3. The execution authorization index, EAX, may be specified to a designated value. The address space issuing the ETCRE macro must own the authorization index (AX) value.

The Virtual Lookaside Facility

The Virtual Lookaside Facility (VLF) is new in MVS/ESA and utilizes ARs, data spaces, and linkage stack facilities, and other MVS/ESA services. VLF exists in its own address space. The VLF services are invoked by the PC/PR program linkages. When invoked, ARs and data spaces provide VLF with addressing capabilities that far exceed the 2 gigabytes of virtual addressing in its own address space. These new capabilities allow VLF to provide its users significant improvements in response time and throughput by eliminating I/O activity.

VLF provides a set of services that will manage named temporary objects in the storage of VLF. These services create and delete named objects that can be retrieved by one or more users (here meaning a program or component that is utilizing some facility) with minimum cost. The services to define users and object classes and to create and delete objects are limited to privileged callers. Once identified as a user for a class of objects, the user, whether privileged or nonprivileged, may retrieve objects from that class. The user must recognize that the objects managed by VLF on its behalf are temporary in nature, because VLF is a cache-like facility. In managing its storage, VLF may decide to delete an object from its storage on the basis of demand for that object versus other objects VLF is managing. If VLF does not exist or if VLF indicates the object no longer exists as an object it is managing, the user must be prepared to obtain the object in the manner the user would have originally acquired the object. However, users that require repeated access to objects that reside on storage volumes can eliminate I/O delays and contention by allowing VLF to manage these objects.

As an example, the Library Lookaside Facility (LLA) uses VLF to cache frequently invoked programs. If VLF is managing the desired program as a named object in its storage, LLA will have access to a copy of the program. The result can be the elimination of I/O activity to find the location of the program on a direct-access storage device (DASD) and to fetch the program into main storage.

In another example, TSO/E uses VLF to cache frequently invoked command lists (CLISTs). Since an invocation of a TSO CLIST involves a preprocessing phase, the output of this phase is given to VLF as a named object to manage. On subsequent requests for the same CLIST, TSO/E invokes VLF to retrieve the named object. If retrieved, TSO/E can generally avoid the I/O operation to read in the CLIST and the resulting contention on the SYSPROC dataset[14] and volume. In addition, TSO/E can avoid the overhead of the preprocessing phase of the CLIST process.

In both of these examples, the system and its end users benefit from VLF. The end user and the products that service the user derived these benefits without having to understand or produce code for the unique facilities of ESA. However, VLF uses most of the features previously discussed to provide these benefits.

VLF use of ESA. VLF is initialized in its own address space as the result of a START VLF,SUB=MSTR command by an operator or automatically by an IPL. If the installation desires to provide control information to VLF, a new parmlib member (a partitioned dataset member of a system parameter library) can be specified instead of the default parmlib member, COFVLF00, by adding a keyword, NN=XX, to identify the new COFVLFXX parmlib member. Through the parmlib control information, the installation can specify the classes of objects eligible for VLF management and the amount of virtual storage to be allocated for each class.

For each specified object class that is actually activated, VLF will create two data spaces per class to be managed. One data space will contain control information about the class, and one data space will contain the named objects associated with that class. Because VLF exists in its own address space, VLF creates the necessary program call entry table entries for the VLF services. VLF connects the entry table to a system level linkage table entry it has reserved, thus providing its services to every address space in the system.

The services are invoked through VLF-supplied macros that result in the appropriate program call being issued. VLF defined the program calls to be the ESA stacking program calls, thereby using the PC/PR instructions to save and restore the caller's status (GPRs, ARs, PSW, XM status, where XM is cross memory, generically used to refer to programs executing in a cross-memory environment) on the linkage stack. In addition, using the associated recovery option on the stacking program call, VLF has its recovery environment automatically established on the call and deleted on the Program Return at no extra performance cost.

The data spaces that are created for each class are added by VLF to its PASN-AL. Thus, as soon as the VLF program calls are issued, VLF receives control in its address space with the capability to access these data spaces. When VLF completes its processing and returns to its caller via program return, the capabilities to access the data spaces are revoked automatically on the program return, thus providing VLF with complete encapsulation of its programs and data.

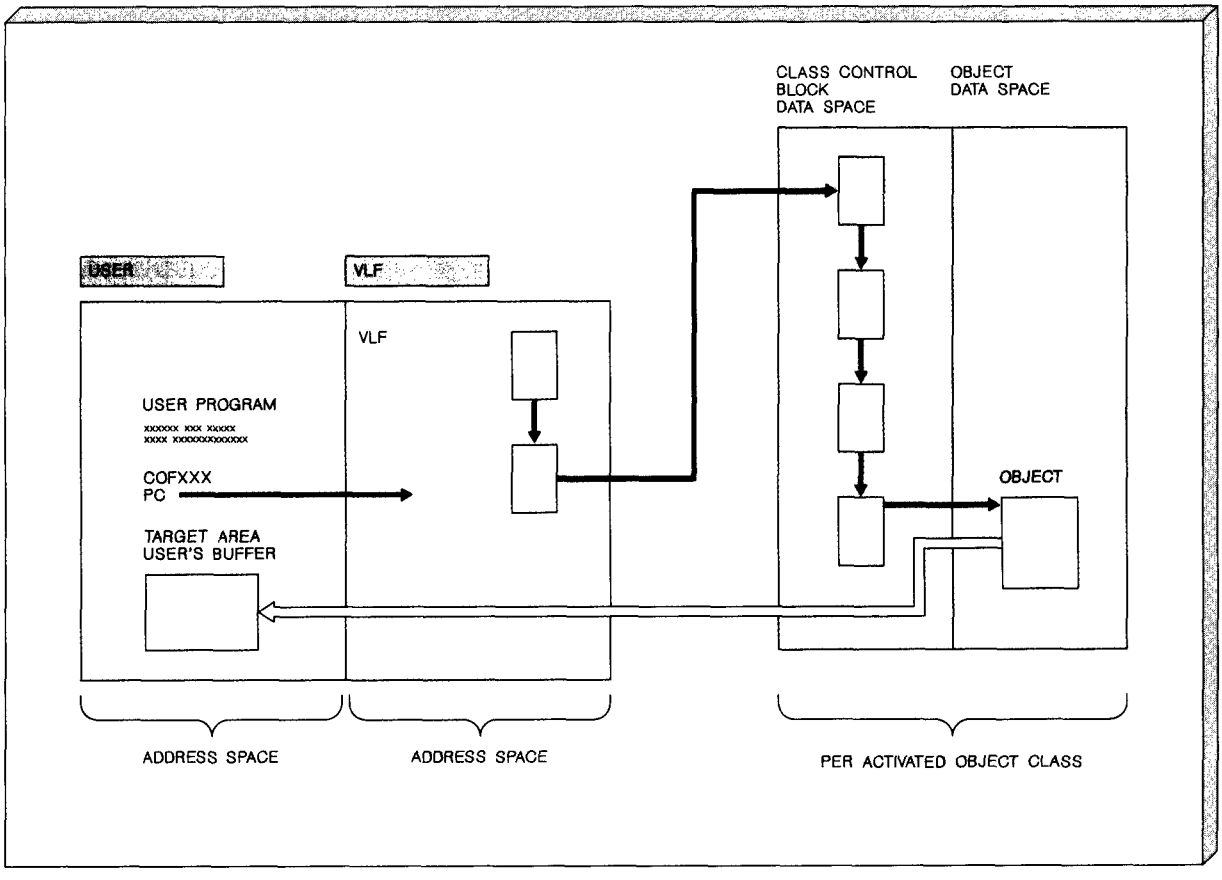
Names of objects managed by VLF can be viewed as having a three-level name.

For each object class, VLF maintains the named objects in one data space and the control structures in another data space. This separation makes dumping of error data easy and allows VLF to disconnect the data and control structures on a class basis. Because data spaces are unique instances in the life of the system, VLF does not have to worry about latent binds or problems of address reuse. Any reference to the old data space will result in an error, which VLF has recovery routines in place to handle. This is then turned into a return code that the user can process as part of the calling sequence.

VLF uses access registers both to address the data residing in data spaces and to address the caller's information and data areas residing in the caller's space(s), thereby affording VLF the full ESA/370 instruction set to manage the data it needs to reference, regardless of location.

Figure 12 shows the VLF structure that utilizes MVS/ESA to meet its requirements of managing large amounts of virtual storage. This storage and the

Figure 12 Use of MVS/ESA address space and data spaces by VLF



objects it contains are completely isolated from the VLF users and other products in the system. Still, VLF provides a high-performance retrieval mechanism to its users for the objects it is managing.

VLF facilities and services

VLF object naming. Names of objects managed by VLF can be viewed as having a three-level name: CLASS.MAJOR.MINOR. VLF allows similar objects (e.g., CLISTs) to be grouped and managed by a class. Within a class, an object is distinguished by its major and minor name. For each major name, objects must have unique minor names. This requirement mimics the structure of a partitioned dataset (PDS), where for each dataset name the member name must be unique. As with PDS member names, duplicate minor names may exist in other major names. Therefore, before retrieving objects, each user must indicate the search order of major names. This search order is defined by the user, when the user is first identified to VLF. If VLF is to manage objects that correspond to members of a partitioned dataset, the search order is implied by identifying the DDNAME of the concatenated dataset list, which is the source of the data for the VLF object for this class.

The following subsections describe briefly some of the concepts and features of the invention, Virtual Lookaside Facility,[15] on which VLF is based.

VLF object storing and retrieving. Prior to retrieving or storing objects in VLF, a user and the user's order of searching the major names must be identified to VLF.

An object can be retrieved only after it has been saved. At the time an object is saved, the user provides the major name (indirectly) and the minor name identifying the object being saved. Once saved,

Table 3 Initial services provided by VLF

Function	VLF Macro
DEFINE CLASS	COFDEFIN
PURGE CLASS	COFPURGE
IDENTIFY USER	COFIDENT
REMOVE USER	COFREMOV
CREATE OBJECT	COFCREAT
RETRIEVE OBJECT	COFRETRI
NOTIFY	COFNOTIF

the object is available to all users with a search sequence that results in the saved object being located before any other object with the same name.

The minor name is manipulated by a hash routine.[16] If a duplicate for the major is found, the duplicate object is not saved, since the existing object is known to be the same. The invention disclosure describes the technique that VLF used for preserving dynamically obtained information about an object and its major and minor name relationship. The technique provided the following benefits to VLF:

First, a complete list of all minor names in every major name was unnecessary, and thus, it was not necessary to read and maintain copies of partitioned dataset directories. This kept VLF out of the data management business and allowed it to keep information only for minors actually in the lookaside facility.

Second, this arrangement enables a very fast determination of the correct object for a particular search sequence.

The VLF services. VLF provides interfaces to its services through the macros listed in Table 3. Each function is discussed below.

Define class. This function activates a new class of objects if none were previously activated. The class had to be previously defined in the initialization parmlib member. Through the define class service, the maximum major and minor name length can be specified. An option exists to indicate whether VLF should "trim" or remove objects from the class if needed for space reclamation. If requested, this will be done on the LRU algorithms of VLF. Otherwise VLF will only remove objects when explicitly requested. This service is only available to privileged programs.

Purge class. This service immediately deletes all objects associated with the specified class. The service is only available to privileged programs.

Identify user. For the specified class, the service identifies an end user to VLF and the major name search order for that user. The identify user function returns a UTOKEN to be used on subsequent requests. Through identify user, the scope of the UTOKEN can be specified as HOME or SYSTEM, thus restricting all subsequent requests to be invoked only when the user's space is the HOME space. SYSTEM will allow retrieves to come from a user when the home address space is different than the address space that issued the identify user. This allows the server address space to utilize VLF on behalf of many users, requiring only the server address space to identify the user to VLF. This service is only available to privileged users.

Remove user. This service removes an end-user access to VLF services for the class of objects associated with the UTOKEN. Remove user is only available to privileged users.

Create object. The create object service must be invoked while executing under a task in the same home address space as the issuer of an identify user. Create object must specify the major and minor name of the object. For those classes of objects with the major-name-to-PDS-name correspondence, the concatenation index of the major name must be specified. For concatenated partitioned datasets, the CINDEX value is the same as the "K" concatenation index value returned when a build list (BLDL) is performed to locate a member. For each object, a parts list must be specified that can identify up to 255 separate parts of the object that VLF will combine into a single continuous object. Prior to issuing a create object, a retrieve object for the same minor name must be attempted on behalf of the user. Only if the retrieve object service returns a "no object found," "best available object found," or "best object found, but target area is too small for retrieve" message will the create object function be performed and then only if the concatenation index values do not match. This service is only available to privileged users.

Retrieve object. The retrieve object service must specify the UTOKEN of the requestor and the minor name. If the object exists, the object and its size will be returned to the user's specified area if the area is of sufficient size. The CINDEX of the major name where

the object was located will be returned. If the object will not fit in the user's receive area, the size of the object will be returned. Only if the object is returned with a return code of zero can the user be guaranteed that the object represents the highest occurrence of the object in the concatenation list. For the other return codes (2, 4, 6, and 8), the conventional mechanism must be invoked to find the highest level of the object. Then creation of that object with the CINDEX value will allow VLF to update its existence table information for those major names where information was not previously known.

Notify. The notify service allows the user to notify VLF when a change in the external source of objects would invalidate the caching and existence knowledge that VLF has about that object. Notify is the facility that DFP services (STOW, SCRATCH, CLOSE, RENAME) or data facility dataset services (DFDSS) COPY uses to inform VLF of changes to partitioned datasets. VLF also informs ALLOCATION of what DDNAMEs it is interested in, and ALLOCATION informs VLF of any change to those DDNAMEs.

Concluding remarks

MVS/ESA developed from many requirements that were generated as MVS evolved to meet the increasing demands for data processing capacity. Not only were the requirements stemming from the current MVS products and system considered, but a set of primitives were included in MVS/ESA to allow for unconstrained growth well into the future. The design of MVS/ESA encompassed these considerations without sacrificing users' investments in existing programs.

MVS/ESA provides facilities that enable applications to utilize the full capabilities and capacity of IBM's largest systems. These facilities extended the addressing capabilities and capacity far beyond those of the previous architecture. The facilities in the previous dual address space facility and the Extended Architecture were incorporated, extended, and enhanced to allow the full use of expanded storage, main storage, and the complete ESA/370 instruction set to manage data in multiple address and data spaces.

Basic MVS/ESA services were enhanced to utilize unique features so as to benefit many applications. Other MVS/ESA services were offered that provided benefits to applications and required minimum changes.

Some products (e.g., TSO/E, IMS) benefit from throughput and response time improvements provided by the LLA facility without any design or code changes. Products that use VSAM (e.g., IMS) may obtain performance improvements and realize virtual storage constraint relief through VSAM's use of hiperspaces. Other products may require minor changes to them to realize the performance advantages of the VLF of MVS/ESA. Still other products may have to be redesigned or recoded in order to take advantage of the large-system capabilities offered by the MVS/ESA facilities.

MVS/ESA provides a range of capabilities to meet a range of application requirements. It is anticipated that MVS/ESA will continue to expand these services at each level, enabling a wide range of users' applications to fully utilize the capacity and capabilities that were introduced with MVS/ESA, the ESA/370 architecture, and IBM's ESA/370 processors.

Acknowledgments

In addition to the authors of other papers in this issue of the *IBM Systems Journal*, I would like to recognize the efforts of several people who have contributed so much to the MVS/ESA product from the design phase until it was shipped. Their contributions were not only to the design and development of the product, but also to the level of documentation that has been prepared for use of MVS/ESA by applications. I particularly cite Mike Mall, Rich Howarth, Jeff Frey, Rick Reinheimer, Kathy Eilert, Bob Seaborg, and Peter Cochrane among those who have contributed to the design and development of MVS/ESA as well as to the documentation described in the cited references. Duna Williamson from MVS information development worked unrelentingly on describing these very technical concepts and services in many of the excellent detailed descriptions found in the cited IBM publications. There are, of course, many more people and areas of MVS design, development, performance, test, and management that I have not mentioned, but whose contributions were invaluable to the success of the MVS/ESA product.

Cited references and notes

1. In the strictest architectural sense, PASN is the primary address-space number. It also sometimes refers to the primary address space. See *IBM System/370 Principles of Operation*, GA22-7000, IBM Corporation; available through IBM branch offices.

2. In the strictest architectural sense, SASN is the secondary address-space number. It is sometimes used in referring to the secondary address space. See *IBM System/370 Principles of Operation*, GA22-7000, IBM Corporation; available through IBM branch offices.
3. *MVS/ESA System Programming Library: Application Development—Extended Addressability*, GC28-1854, IBM Corporation; available through IBM branch offices.
4. *MVS/ESA System Programming Library: Initialization and Tuning*, GC28-1828, IBM Corporation; available through IBM branch offices.
5. *MVS/ESA System Programming Library: User Exits*, GC28-1836, IBM Corporation; available through IBM branch offices.
6. *MVS/ESA System Programming Library: Application Development Guide*, GC28-1852, IBM Corporation; available through IBM branch offices.
7. W. Buchholz, "The IBM System/370 vector architecture," *IBM Systems Journal* **25,** No. 1, 51–62 (1986).
8. C. A. Scalzi, A. G. Ganek, and R. J. Schmalz, "Enterprise Systems Architecture/370: An architecture for multiple virtual space access and authorization," *IBM Systems Journal* **28,** No. 1, 15–38 (1989, this issue).
9. K. E. Plambeck, "Concepts of Enterprise Systems Architecture/370," *IBM Systems Journal* **28,** No. 1, 39–61 (1989, this issue).
10. K. G. Rubsam, "MVS data services," *IBM Systems Journal* **28,** No. 1, 151–164 (1989, this issue).
11. *MVS/ESA System Programming Library: System Modifications*, GC28-1831, IBM Corporation; available through IBM branch offices.
12. LXRES is defined as reserve a linkage index, an MVS service that operates through a macro facility to reserve a linkage index (an index to the linkage table). ETDEF is the entry table definition, an MVS service that defines the contents of entries in an entry table. It is used for program call linkages. ETCRE is entry table create, an MVS service for creating an entry table. ETCON is entry table connect, an MVS service for connecting an entry table to a linkage table entry.
13. Disable reference storage is a new MVS storage attribute in which storage can be referenced by disabled routines. The storage is not fixed; it is pageable. But if it is paged, it will be resolved by synchronous page faults. It cannot be used as storage for input/output.
14. SYSPROC is the set of partitioned datasets whose members represent procedures that can be executed, e.g., CLISTs.
15. D. D. Brown, W. J. Morschhauser, R. F. Reinheimer, and M. D. Swanson, *Virtual Lookaside Facility*, U.S. Patent Application, P09-88-006, IBM Corporation.
16. A hash routine is a programming technique to reduce a key or string of characters to a unique smaller representation of the same.

Carl E. Clark *IBM Data Systems Division, P.O. Box 390, Poughkeepsie, New York 12602.* Mr. Clark is a senior technical staff member at the Myers Corners Laboratory. He joined the IBM Data Processing Division in 1967 after receiving a B.B.A. in accounting from George Washington University. During his IBM career, Mr. Clark has been involved in the development and design of the MVS control program. In 1979 and 1988 he received Outstanding Innovation Awards for MVS supervisor performance enhancements and for the Enterprise Systems Architecture/370 respectively, and in 1987 he received his Second Level Invention Achievement Award. He currently is working in MVS system architecture.

Reprint Order No. G321-5351.