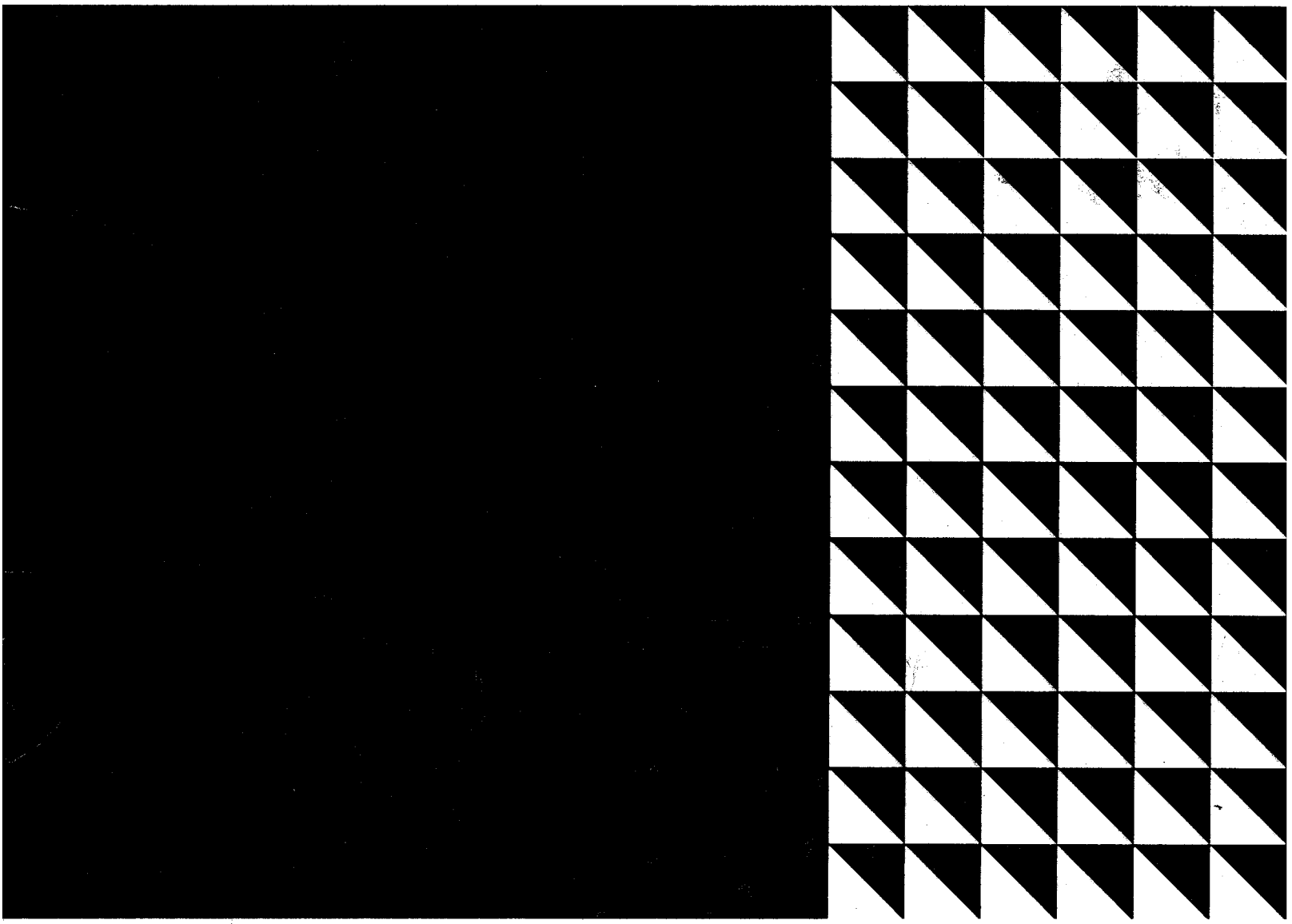
IBM

Introduction to Virtual Storage in System/370



Student Text

Introduction to Virtual Storage in System/370

Student Text

Preface

This student text contains two sections, PART I and PART II. PART I presents virtual storage concepts and relates them to a conceptual virtual storage operating system. At the end of PART I we describe the new hardware on the System/370 models that support virtual storage.

PART II describes the operating systems that support virtual storage in System/370.

We have tried to make this text usable for managers as well as for systems programmers, programmers, operators, and so forth. The reader needs a minimum of prerequisite knowledge about computer concepts. You should know what a computer is, the basics of how one works and the basics of multiprogramming as supported by operating systems like OS and DOS on System/360.

This edition of the text contains a description of OS/VS2 Rel 1 as well as the recently announced OS/VS2 Rel 2 in PART II.

Major Revision (February 1973)

This publication GR20-4260-1 is a major revision and obsoletes all previous editions.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. Address comments concerning the contents of this publication to IBM Corporation, DPD Education Development - Publications Services, Education Center, South Road, Poughkeepsie, New York 12602.

Contents

Part I.....

Lesson 1. Introduction to Part I

Before you begin to study virtual storage concepts and their implementation in the IBM System/370, let's take a brief look at the history of computers and their operating systems. The past fifteen years have seen spectacular changes in computing equipment and programs.

Fifteen years ago, central processing units (CPU's) performed instructions in thousandths of a second. Today, System/370 executes instructions in billionths of a second. In a system like the IBM 1401, main storage sizes ranged from 1400 to 16,000 locations. In System/370, main storage sizes range from 96K to over four million bytes in the range of its models. Data transfer rates for some new tape drive models are up to twenty times faster than early tape drives. Storage capacity on the new 3330 disk drive is twenty five times larger than the early models of the 1311 disk drive. Its transfer rate is eight times faster. Improvements just as dramatic have been made for printers, card readers and card punches. This overview does not even consider new devices for applications like teleprocessing that did not exist fifteen years ago.

A large amount of software development has accompanied this hardware evolution. With early computers, programmers used machine language. They had to know machine code to use these systems. Then came assemblers, compilers and input/output control systems. These had the effect of moving the programmer farther from machine code, letting him spend more time solving application problems. Computers still executed programs one at a time. Each program was set up and run independently by the system's operator.

The first operating systems enabled users to batch jobs and run them in a single stream. Jobs executed one at a time and there was a smoother transition from job to job with some assistance from the operating system. This made the operator's job, and system operation, easier. As CPU speeds became faster and main storage more abundant – for example, with the IBM System/360 – operating systems expanded in scope. These new operating systems let users run multiple jobs concurrently – multiprogramming – by sharing the CPU, real storage and other system resources among active jobs. They also made possible teleprocessing applications that control remote computing and data entry operations from one central location. OS on System/360 is an example of such a system.

These technical developments have meant significant benefits for computer users and their people – system analysts, programmers, computer operators, and so forth. System/370 users can now develop complete systems for operational control and management information in contrast to the typical payroll and billing applications of ten to fifteen years ago. System resources are shared among several jobs in multiprogramming systems, and among multiple users in time sharing systems. Programmers use application-oriented languages like PL/I, COBOL, and FORTRAN. As a result, programmers can devote more time to problem solutions. Operating systems like OS handle much of the job preparation and job-to-job transition that formerly occupied so much of a computer operator's time. Operators now spend less time handling punched cards and tapes and more time directing system activity, keeping the system *productive.* Thus, the programmers and operators who work with computers, the scope of applications developed and, in general, computer users have benefited a great amount from the past developments in hardware and software.

With System/360, the primary operating systems were OS and DOS. With System/370, you may *still* use the OS and DOS systems. We have examined how we arrived at the current mode of computer operation. Where can we go from here?

What would your objectives be if you were designing new features and functions for System/370 or OS/360? Take a moment and think about it. We will list a set of objectives; they might be somewhat different than yours, but we will have a common reference point. Our list of objectives follows:

1. Programmers should have the amount of main storage that they need for designing programs without having to use planned overlay or dynamic management techniques. Even though the size of available computer main storage has increased tremendously, users cut their storage into partitions or regions for multiprogramming efficiency. Programmers, then, are restricted to the size of the largest partition or region used in their installation. This often requires breaking a program up into separate steps or using special overlay techniques to make programs "fit" into a region or partition. All of these design requirements add overhead to solving a problem.
2. If a program is too large for main storage size, the operating system, not the programmer, should make the program "fit" into main storage.
3. Programs should use system resources – especially main storage – only as required during execution. For example, a program that needs 82,000 bytes of main storage when fully loaded may reference only

22,000 bytes during one part of processing. During this time, there is no need for the operating system to commit main storage to 60,000 unreferenced bytes. An example of such a situation is a teleprocessing application running at less than its maximum load.

4. The operating system should not allow main storage to become fragmented. Assume that several programs are executing, each in its own contiguous area of main storage. Three 15K-byte areas in main storage are idle (none of the executing programs occupy these areas). If the smallest program waiting to begin execution needs 30K bytes of main storage, it must wait until 30K bytes of contiguous main storage becomes available. Until then, a total of 45K bytes of main storage are idle because of storage fragmentation. They are wasted.
5. An operating system should control system resources like main storage in such a way that you automatically get a performance improvement by adding more main storage. For example, if you have a program that must use overlays because it won't fit into main storage, adding more main storage won't help at all unless you redesign and recode the program. It would be nice if the operating system could somehow "automatically" overlay programs and "automatically" use added main storage.
6. A computer user should be independent of the size of the main storage in which programs execute and in which the operating system is structured. He should be able to structure a system more according to his *needs* than to the size of main storage.
7. Main storage should be shared dynamically among the active jobs in the system. Programs should get the main storage that they need when they need it. In other words, the system should be adaptive to the demands of the system's activities.
8. Scheduling and operating a system should be easier. Systems like OS require a large amount of user participation to schedule jobs and control system resources. A new system should require less user participation to achieve good scheduling and operation results.

Most items in this list of objectives relate in some way to how an operating system manages a computer's main storage. These objectives can be fulfilled by an operating system that supports a virtual storage. We will present virtual storage and how it fulfills our objectives in PART I of this student text.

PART I begins with a presentation of different conceptual techniques for managing a computer's main storage. For each strategy we ask two important questions:

1. How well does the technique manage main storage?
2. How much does the technique help the computer user?

We conclude that a form of segmentation and paging is the best strategy for managing main storage, and then we describe how this technique can be used in a conceptual virtual storage system. PART I ends with a description of the new hardware features on System/370 that are used to implement virtual storage in the System/370 operating systems. PART I contains the first eight lessons of the student text.

PART II of the student text presents how System/370 supports virtual storage using the OS/VS and DOS/VS operating systems. PART II contains the last seven lessons. You should read PART II after completion of PART I.

After studying this text you should be able to answer many questions about virtual storage:

1. What is virtual storage?
2. What are the advantages of virtual storage?
3. What are some hardware features and software techniques necessary to use virtual storage?
4. How is a computer's main storage managed in a virtual storage system?
5. What is dynamic relocation?
6. What is dynamic address translation?
7. What hardware is needed on System/370 to have virtual storage?
8. How is virtual storage implemented in the System/370 operating systems?

Many lessons in PART I of this text have short exams at the end. You should take them. They are included, with scoring keys, for your personal feedback. The scoring keys for the exams can be found at the end of PART I.

The Address Space

Computer programs, whether written in COBOL, PL/I, Assembler, or another language, contain instructions, data descriptions, and input/output operations. For a moment, think of programs apart from the computer. As an example, think of a source program coded in COBOL or PL/I. Its instructions might be additions, 'move data' operations or branch operations. Data descriptions will have names; they might be names like TAX, EXEMPTIONS, or CREDITS. Any input or output operation needs file names and record names. Some instructions, or groups of instructions, have names that are referenced in a branch operation.

The source program, as a combination of symbolic instructions, data descriptions and I/O operations, exists in a space built by the programmer. We will call this space a *symbolic name space,* the combination of symbolic instructions and symbolic names built by the programmer to implement a computer application. In this symbolic name space a programmer creates, manipulates, and references the names of instructions, data elements, input/output files and input/output records as required by the logic of an application.

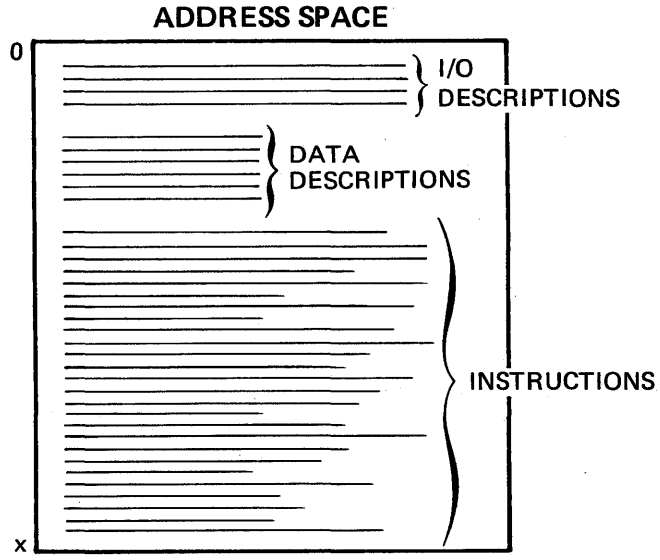


Figure 1

In preparing a source program – the symbolic name space – for execution, the programmer submits it to a language compiler. The compiler converts the symbolic elements of the language to computer instructions, data strings and control blocks. The compiler substitutes real addresses for symbolic names. This process is called translation. After translation, the range of addresses bounded by the program is called its *address space.* This is indicated by the X in Figure 1.

The compiler generally assigns addresses beginning at location zero, and all other addresses are contiguous from that beginning location. These addresses appear in the output of the compiler, the object module, in instructions or other constants. Thus, the program's address space is created during compilation (or assembly). The address space can be further resolved or combined with the address space of another program to form an even larger address space. This is usually done by a program called a linkage editor. The program's address space can be stored in punched cards or within the system in a direct access file. Once stored, the program is ready to be loaded into the computer for execution.

Real Storage

Computer storage has been called memory, processor storage, main storage and real storage. In this text we will use only the term *real storage.* Figure 2 shows a real storage that contains 256K positions. (Note: The symbol "K", when used in this text assumes a hexadecimal machine. The symbol "K" then represents 1024 storage positions with a few exceptions. These exceptions appear later in the text where "K" is used to represent 1000 storage positions only for ease of illustration and calculation. Each exception will be noted in the text where it appears.)

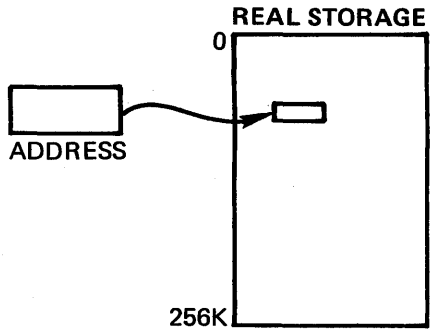


Figure 2

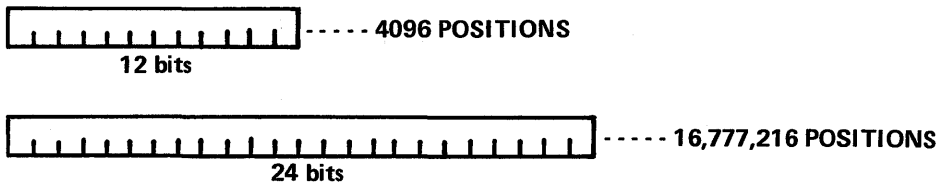


Figure 3

Each unit of real storage can be located or addressed by the addressing structure of the computer's Central Processing Unit (CPU). A computer's unit of storage contains the smallest addressable entity in real storage. In the System/370 the unit of storage is a byte. The example in Figure 2 would have 256K bytes, each byte addressable by the CPU. A computer's addressing structure sets the theoretical upper limit on its real storage size. This is shown in Figure 3.

A 12-bit address can locate only 4096 real storage positions. A 24-bit address can locate 16,777,216 real storage positions; this, in fact, is the size of the address in System/370.

A computer's real storage size is fixed with a variety of sizes to select from. Figure 2 shows a real storage of 256K positions. We could have used 128K, 512K or even 1024K. No matter what size you select, real storage is a very valuable system resource. You probably select the storage size as carefully as you select the model of your computer.

To execute a program in a computer, instructions and data in the program's address space must eventually be loaded into specified real storage locations in the computer. This process is equivalent to allocating specific machine resources to the program. In particular, part of a computer's real storage must be allocated to hold a program's instructions, data elements, control blocks, and Input/Output areas. Real storage plays a central role in a computer installation because a program cannot execute until it resides in real storage. Real storage is also expensive compared with other storage media like disk and tape. Managing this valuable resource deserves special attention. Real storage management techniques affect the programmer's job. If an operating system uses storage well, the programmer's job is easier. Otherwise, a programmer spends part of his time designing the management of real storage, compensating for the operating system.

Types of Relocation

How is the compiled program's address space translated into locations in real storage for execution? When does this translation occur? Answers to these questions relate to program relocation; and relocation techniques relate directly to real storage management.

We define *relocation* as the translation of addresses in a program's address space to specific locations in real storage. The type of relocation depends on the *time at which translation occurs.* Translation may occur at two different times:

1. When a program is loaded for execution. This is called *static relocation.* The result is a program that uses fixed real storage locations when it is run. In other words, the program is bound to real storage locations at load time.
2. During program execution. This is called *dynamic relocation.* If a computer has a dynamic relocation capability, a program is not bound to specific real storage locations during its execution.

Static Relocation

Simple program loading may occur as shown in Figure 4. This technique was used for systems like the IBM 1401.

Translation was done once at compilation or assembly time in the manner that we described before. What resulted was an address space with a zero origin as shown in Figure 4. The program's address space always used corresponding

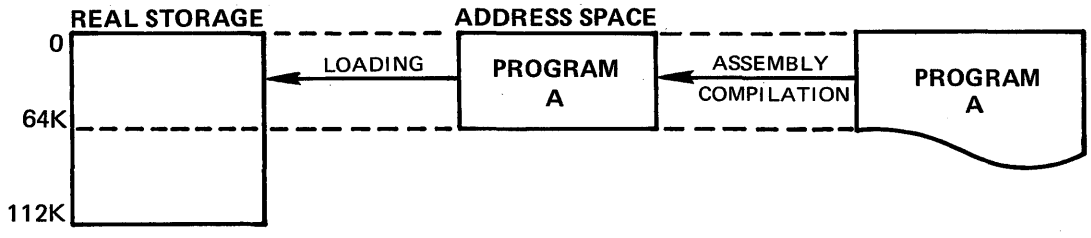


Figure 4

locations in real storage beginning at location zero. Thus, it was called an absolute program. There was no relocation of addresses in the program's address space. The program was simply loaded into real storage beginning at location zero. All programs that ran on the 1401 were loaded beginning at location zero; there was no possibility for multiprogramming. When a program's address space exceeded real storage size (Figure 5), the programmer had to solve the problem – usually by creating overlays or using multipass techniques.

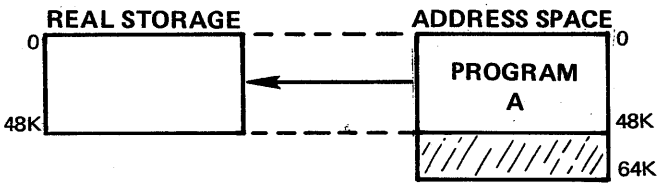


Figure 5

The Disk Operating System (DOS) on System/360 uses a variation of this process. Compilers translate symbolic names into an address space with a zero origin and in a relocatable form. The address space can then be further translated by the linkage editor into real storage locations with a different origin (Figure 6, Step 1).

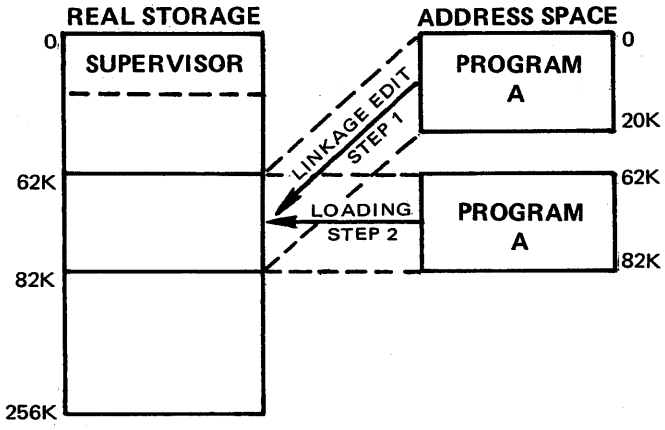


Figure 6

Figure 6 (Step 2) indicates that after the linkage edit translation Program A executes in the same contiguous real storage locations every time it is run. This is always the case unless you retranslate (or re-linkage edit) Program A. Multiprogramming is possible with this scheme; however, it requires a lot of preplanning. Consider the case in Figure 7.

Program A is running. Program B and Program C have been translated for loading into the same contiguous real storage locations used for Program A. B and C are waiting to run but they must wait until Program A completes – even though real storage locations from 90K to 120K are idle.

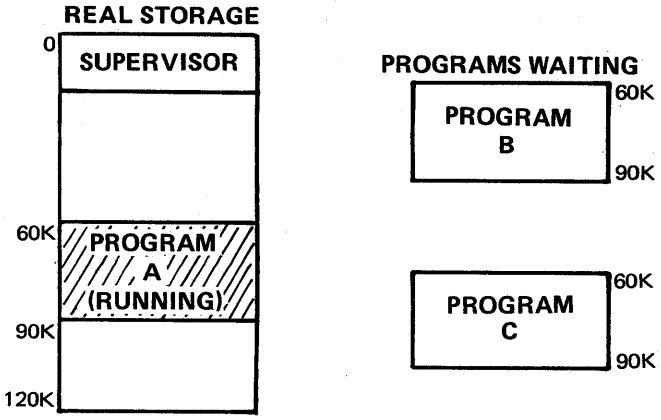


Figure 7

Techniques of static relocation eliminate this problem. With static relocation, translation of a program's address space into real storage locations does not occur until just before execution begins, when the program is loaded. It is called static because it only happens once, before program execution. Examine Figure 8.

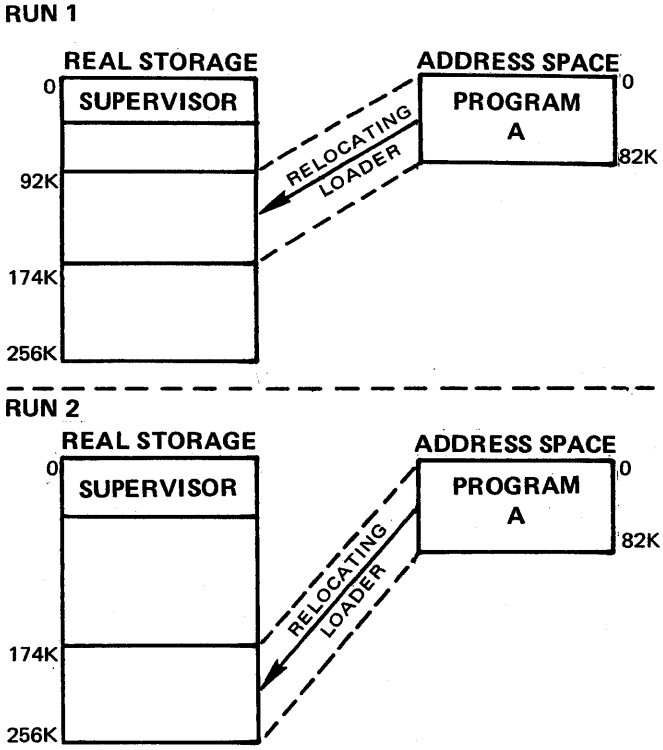


Figure 8

In Figure 8, RUN 1 shows Program A loaded into a contiguous area of real storage beginning at location 92K. This static relocation of Program A occurs just before RUN 1 begins. RUN 2 in Figure 8 shows the same Program A. For RUN 2, however, the same Program A is relocated and loaded into a different contiguous area of real storage that begins at location 174K. A system that waits to relocate

until program load time has the best chance to effectively allocate contiguous real storage areas in a multiprogramming system. OS on System/360 uses the static relocation technique for real storage management very effectively. In a static relocation system a program is bound to its real storage locations until execution ends. Thus, there is no opportunity to move a program and reallocate real storage during program execution.

Dynamic Relocation

Consider a multiprogramming system. Program A, Program B and Program C in Figure 9 represent three programs whose address spaces have been statically relocated into contiguous areas of real storage.

Figure 9 also shows Program D waiting to run. Because this is a static relocation system, Program D cannot run until 50K of *contiguous* real storage becomes available. Even though a total of 50K of real storage is available, it is not contiguous. Therefore, it is wasted. This condition is called *fragmentation* of real storage. It happens in static relocation systems like OS on System/360.

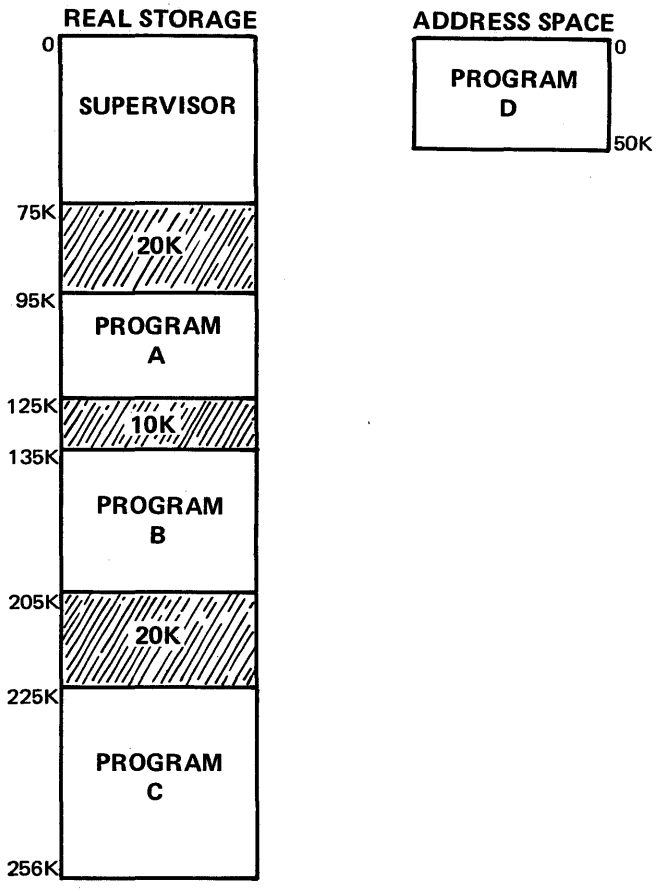


Figure 9

How can this wasted real storage be recovered? Figure 10 shows Program D of the previous example with a different structure.

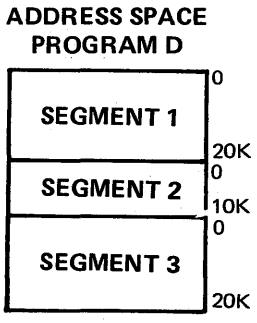


Figure 10

Assume Program D's address space has been subdivided into three parts called *segments:* Segment 1 is 20K; Segment 2 is 10K; and Segment 3 is 20K. We will assume that Program D is formatted into its segment structure by a program like the linkage editor. The real storage lost because of fragmentation in Figure 9 can now be used since the three segments of Program D will fit into the three fragments of real storage (Figure 11).

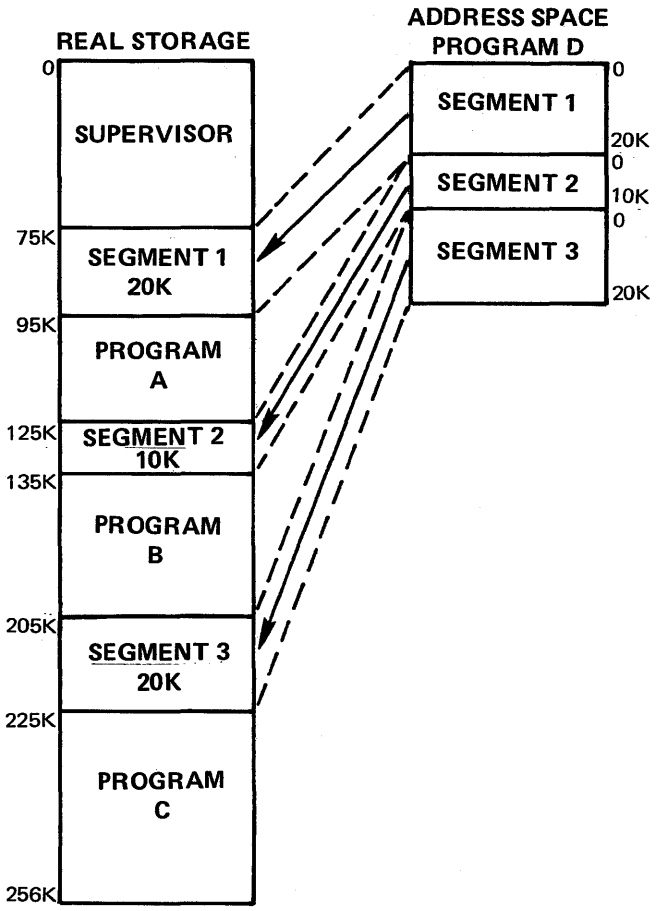


Figure 11

Assume further that Program D's segments were loaded into real storage without relocation. Thus, the addresses in each segment are related to a starting address of zero and they do not correspond to the real storage addresses in which they are actually loaded. They are related only to the origin of the segment itself and are called *relative* addresses. Another way to think of Program D's addresses is this: they are related to its address space and they are still untranslated. When are these relative addresses translated? During program execution. Only when an instruction or data element is referenced is its relative address translated into the address of its real storage location. This happens continuously during program execution for each instruction and data element referenced.

This type of translation is called *dynamic relocation* and it is performed by a special hardware feature called the *Dynamic Address Translation (DAT) Feature.*

Our example with Program D shows a type of dynamic relocation that uses a segmented program structure. With dynamic relocation, a program's address space is never bound to its real storage locations, even during execution, because the program's addresses are translated automatically by the DAT Feature during its execution. Thus, there is an opportunity to manage or reallocate real storage during program execution in contrast to static relocation. We will return to this topic later after we explain how translation works in a dynamic relocation system in the next lesson.

In summary, you should know that there are two types of relocation:

1. Static relocation which, occurs at program load time.
2. Dynamic relocation, which occurs continually during program execution.

Each type of relocation has its effect on the use of real storage in a computer system.

Lesson 2 Test Questions

1. The theoretical maximum size of a computer's real storage depends on
   A. a computer's addressing structure
   B. a computer's CPU speed
   C. a computer's real storage size
   D. a computer's auxiliary storage size
2. The process of translating addresses in a compiled program's address space to their locations in real storage is called
   A. compilation
   B. assembly
   C. relocation
   D. segmentation
3. The two different types of relocation described in this lesson are distinguished by the _____ that translation occurs.
4. Name the two general types of relocation presented in this lesson.
5. Segmentation is one type of
   A. simple program loading
   B. static relocation
   C. dynamic relocation
6. When a program's address space is relocated just before execution begins, each time the program is executed, this is called
   A. simple program loading
   B. static relocation
   C. dynamic relocation
7. When a program's addresses are translated throughout program execution this is called
   A. simple program loading
   B. static relocation
   C. dynamic relocation
8. The DOS linkage editor on System/360 is a variation of the process used for
   A. simple program loading
   B. static relocation
   C. dynamic relocation
9. The relocating loader in OS on System/360 is an example of
   A. simple program loading
   B. static relocation
   C. dynamic relocation
10. Dynamic relocation is a type of software relocation. (True/False)
11. Static relocation is a type of software relocation. (True/False)

Lesson 3. Segmentation and Paging

Organizing programs into parts called segments is a portion of one technique used for dynamic relocation. This was introduced in the previous lesson. It appears that segmentation is better than static relocation for managing a computer's real storage. But we never explained in detail what segments are and how translation occurs during execution. In this lesson you will find answers to these questions and others like: what is paging, how is paging different than segmentation, and how are segmentation and paging used together?

Segmentation

In our conceptual segmentation system, the programmer will code programs in a single symbolic name space and the operating system will automatically segment them. The result will be a program whose address space has one or more variable-size segments. What rule the operating system uses to form segment boundaries is not important to our presentation. As an example, it might be based on program modules or on the separation of instructions and data into separate segments.

For purposes of our discussion, all you need to know is that a program's address space is subdivided by an operating system into one or more segments. The programmer will concentrate on problem solutions and the operating system will segment them to better manage real storage.

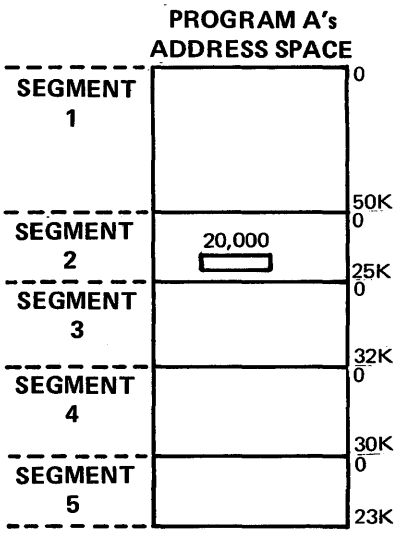


Figure 12

When a program's address space is segmented, how does the computer reference the instructions and data descriptions within a segment? What does an address look like in a segmented address space? A programmer wouldn't need to know answers to these questions to program for a segmentation system. However, for you to understand the nature of segmentation as a type of dynamic relocation, you will need to know these answers. Look at Program A in Figure 12.

Our conceptual operating system has divided Program A into five segments. Each segment has an origin address of zero – this is always the case; each segment is a contiguous space; and, in this example, each segment is different in size. How can you reference location 20,000 in SEGMENT 2 as shown in Figure 12? To say that its address is location 20,000 is ambiguous. All five segments in Program A have a location of 20,000. The address of this location, then, must also indicate the name of the segment. This is shown in Figure 13.

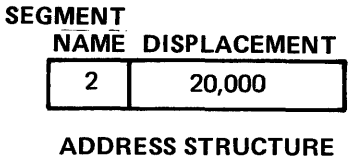


Figure 13

In any segmentation system, segments within a program must have different names. The form of a segment name is not critical. In our conceptual system, we use numbers to name segments. This is the actual form used in many segmentation systems.

In a segmented program then, the address of an instruction or a data location will have a two part structure (s,i) where

s represents the segment name, and

i represents the address of a location within a segment.

This forms a *relative address.* The address is related to the segmented structure of the address space.

If a computer system has a 24-bit address structure like the System/370, segment location addresses will be 24 bits long and split into two parts. Figure 14 shows one way to structure the address.

With this scheme a program's address space could have from 1 to 256 segments – the 8 bits used for *Segment Number.* Each segment could have from 0 to 65,535 (64K)

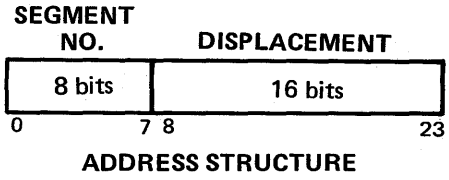


Figure 14

locations – the 16 bits used for *Displacement.* The structure of segment addresses sets two important limits:

1. The maximum number of segments that may exist in a program's address space.
2. The maximum displacement of any segment in the address space.

Figure 15 shows another way to structure a 24-bit address.

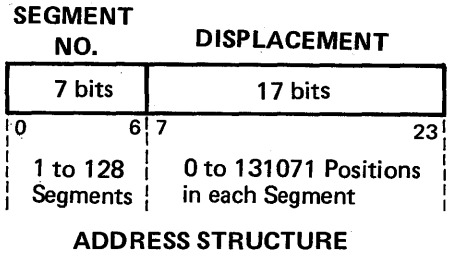


Figure 15

When compared to the address structure in Figure 14, there is a small change. One bit from the SEGMENT NUMBER part was transferred to the DISPLACEMENT part of the address. This makes a significant change in the structure of the segmented address space. The maximum size of segment displacement is doubled, and the maximum number of segments in an address space is cut in half.

At this time you should know what segments are and how addresses are structured in a segmented address space. These topics relate to the format of a segmented program's address space.

Dynamic Address Translation Using Segmentation

To execute a segmented program, address translation must occur each time an instruction or data element is referenced during program execution. To perform translation the computer must have a special CPU hardware facility or feature called the Dynamic Address Translation feature (DAT feature).

We will now describe why translation occurs and what translation involves in a segmentation system.

In the preceding lesson you examined a static relocation system executing three programs – Program A, Program B and Program C – while Program D was waiting to run. Figure 16 shows this situation.

In Figure 16 Program D contains three segments. Program D's address structure uses the two part relative address that we just described. We will assume that Program D is presently stored in a system library.

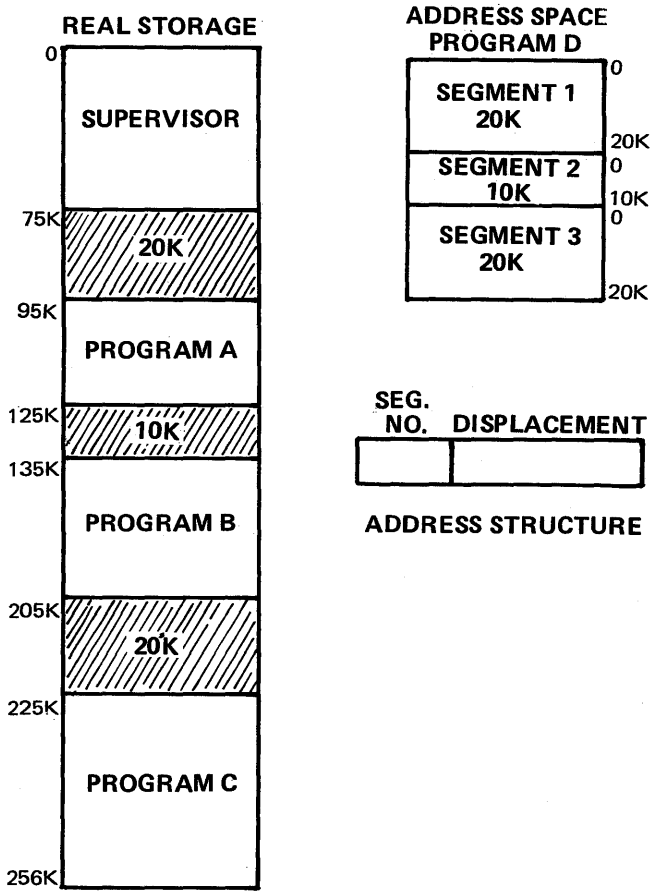


Figure 16

Figure 17 shows Program D's segments loaded. What happens at load time? The operating system moves each segment of Program D into real storage: however, no translation occurs. The segments of Program D still contain their relative addresses. The computer's CPU references real storage locations only by using an absolute address. (The address size is 24 bits for the System/370.) If the CPU uses D's relative addresses to reference real storage, it will reference incorrect real storage locations.

Figure 18 shows SEGMENT 1 of Program D in real storage. (All numbers are decimal.) To perform a certain instruction, the CPU needs data from location 15,000 in SEGMENT 1. This data is actually at real storage location 90,000. Because Program D's addresses were not translated before execution began, the CPU cannot use the relative

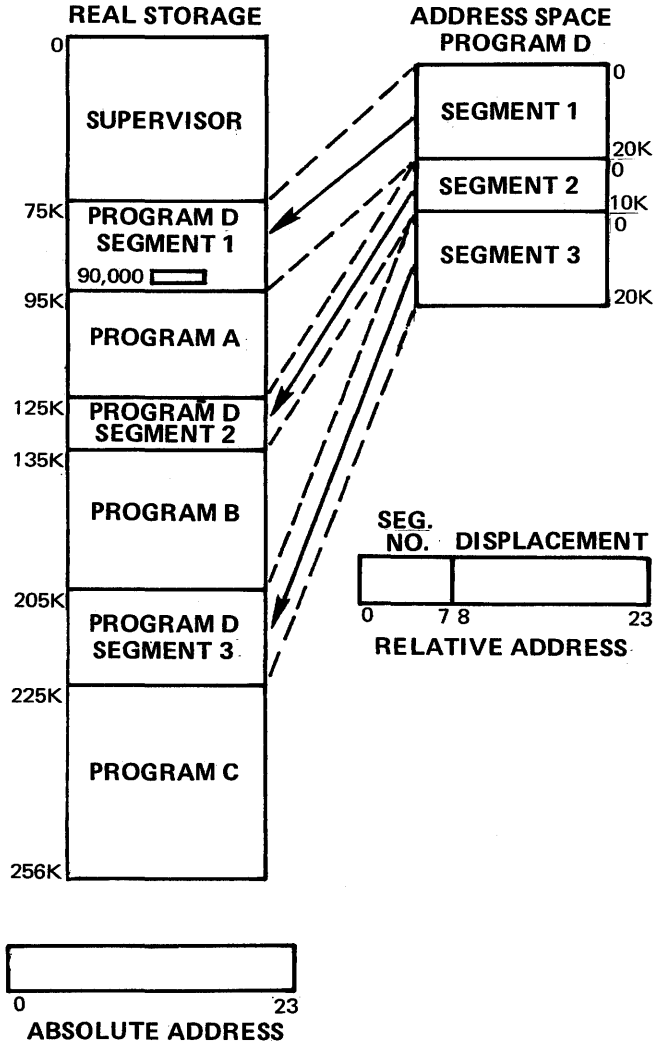


Figure 17

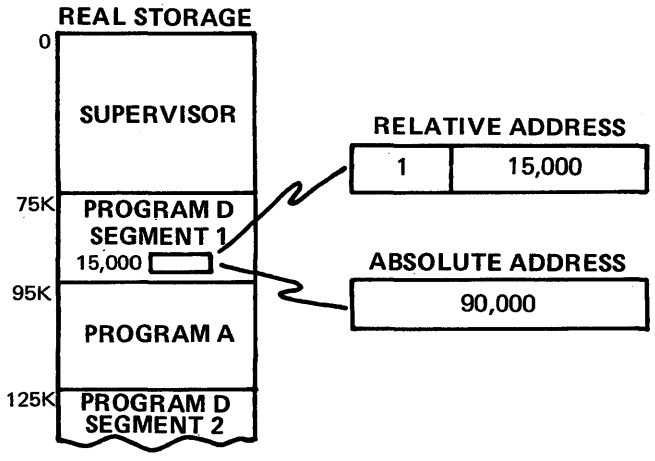


Figure 18

address to fetch the data. Clearly, then, Program D's relative address requires translation before the CPU executes this instruction. In fact, Program D's relative addresses require translation continuously during program execution. This technique – waiting to translate a program's addresses until just before executing instructions continuously during program execution – is called dynamic address translation.

How does translation occur during execution? Special hardware is required. Also, the operating system must build a table that is referenced during translation. This table establishes the correspondence between a program's segmented address space and the actual locations of segments in real storage. Consider the case of Program D again in Figure 19. Programs A, B, and C are not shown. For this example, numbers are decimal.

As segments of Program D are loaded into real storage, the operating system builds a segment table for the program. The segment table is built in real storage. Each entry in the segment table identifies the origin of a segment in real storage; thus, the second entry of the segment table in Figure 19 shows SEGMENT 2 of Program D beginning at real storage location 125,000. There is a table entry for each segment in a program's address space. Program D has three segments, thus, its segment table has three entries. In our conceptual segmentation system, each program that is executing has a segment table in real storage. This procedure – building a segment table for each executing program – is called *mapping*. In effect, the segment table maps segment origins in real storage. In addition to the segment table, a special-purpose control register, called the *Segment Table Origin Register* (STOR Register), is also used for dynamic address translation. The register's name describes its function. It points to a segment table's real storage origin address. The segment table that it locates *maps*, or represents, the program that is currently executing. Let's say Program A is executing. Program A's segment table begins at real storage location 18,000. Then the Segment Table Origin Register contains the address 18,000. If Program B interrupts Program A and begins to execute, the real storage location of B's segment table is placed into the Segment Table Origin Register.

We have described the translation tools. Let's see how they translate. We'll return to Program D and continue to use decimal numbers during this example. Program D's segment table has three entries. The segment table's origin begins at real storage location 68,000. See Figure 20.

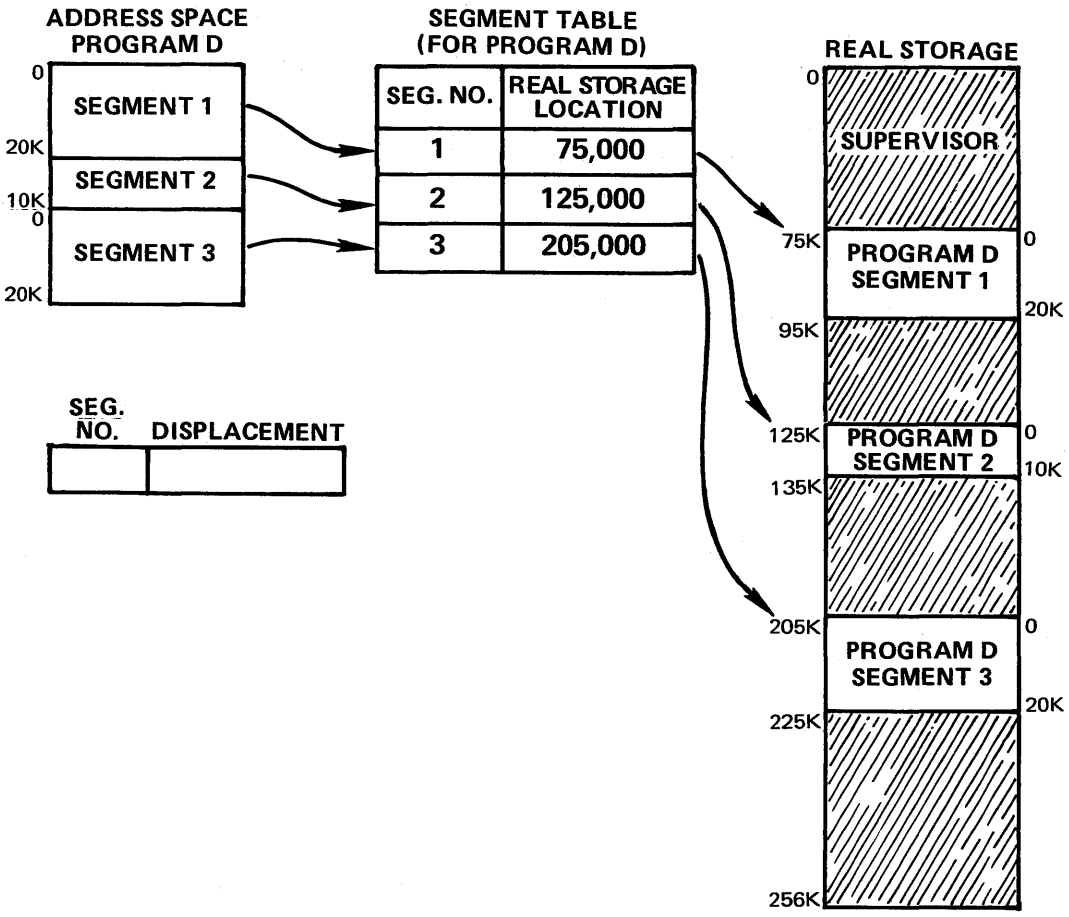


Figure 19

The relative address of location 15,000 in SEGMENT 1 will be translated. The following steps take place during translation:

1. The Segment Table Origin Register points to the origin of Program D's segment table. In our example, this is the real storage location 68,000.
2. The segment number in the relative address is used as an index into the segment table. In the example, it points to the first entry.
3. The location specified in the segment table entry indicates the origin of the segment in real storage. The displacement in the relative address is added to the origin and this results in a real storage address. In the example, the displacement of 15,000 is added to SEGMENT 1's origin – 75,000 – and this results in the real storage address of 90,000.

Figure 21 shows these three steps.

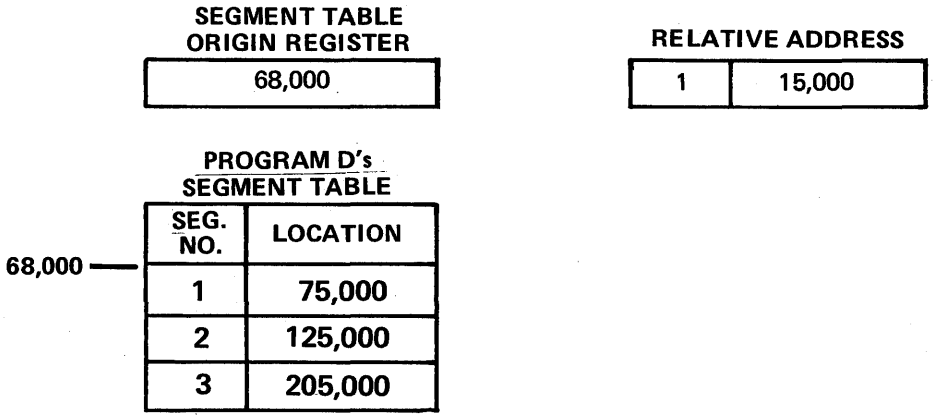


Figure 20

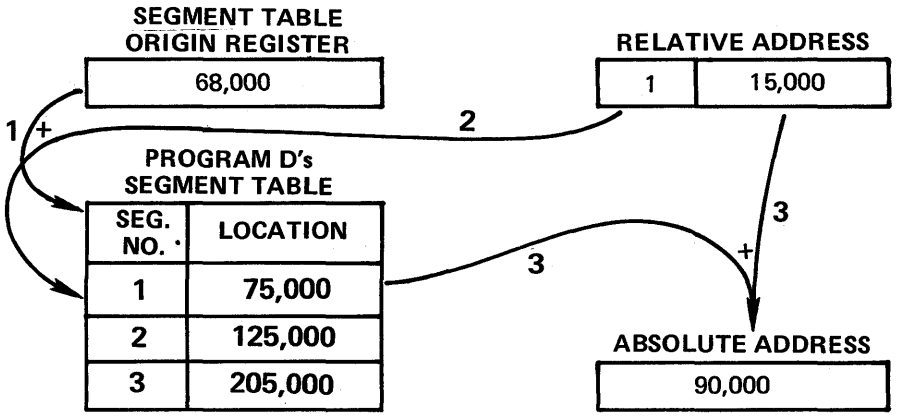


Figure 21

Using the process and segment table shown in Figure 21, take a moment to translate the relative address shown in Figure 22, using decimal arithmetic.

SEG. NO.	DISPLACEMENT
3	13,000

Figure 22

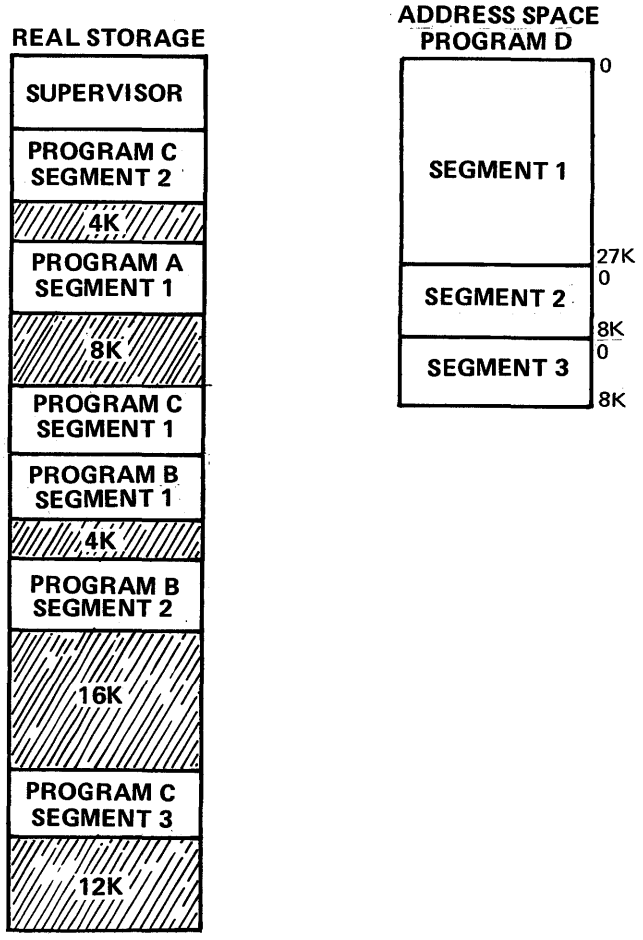


Figure 23

Your result should be an absolute address of 218,000.

Because a segmented program has all relative addresses, this address translation process happens continually during a program's execution. Translation is done automatically by the Dynamic Address Translation Feature (DAT feature) which uses the Segment Table Origin Register and the program's segment table. If the operating system needs to move a segment to another area of real storage during program execution, this presents no problem. First the segment's instructions and data would be moved. Then the segment table entry of the affected segment would be changed to indicate the new origin location, and execution would resume. This presents the possibility of dynamically managing real storage during program execution. The example shows how segmentation can be used as a type of dynamic relocation.

Until now, we have presented segmentation as a better way to manage real storage. With segmentation, we can achieve less real storage fragmentation than in static relocation systems like OS on System/360. Segments are smaller than programs and, therefore, with segmentation, fragments are smaller. However, fragmentation does still exist. In the following topic we will add a variation to our segmentation system that will greatly reduce real storage fragmentation.

Paging

Let's look at our segmentation system while it is multiprogramming. The system has been running for a while. At present it is multiprogramming three segmented programs. Figure 23 depicts such a system. The shaded areas in Figure 23 represent unused pieces of real storage between segments. These unused pieces are the fragments that we mentioned a moment ago. Fragments between segments result during system operation simply because there isn't enough contiguous real storage to load the segments in the

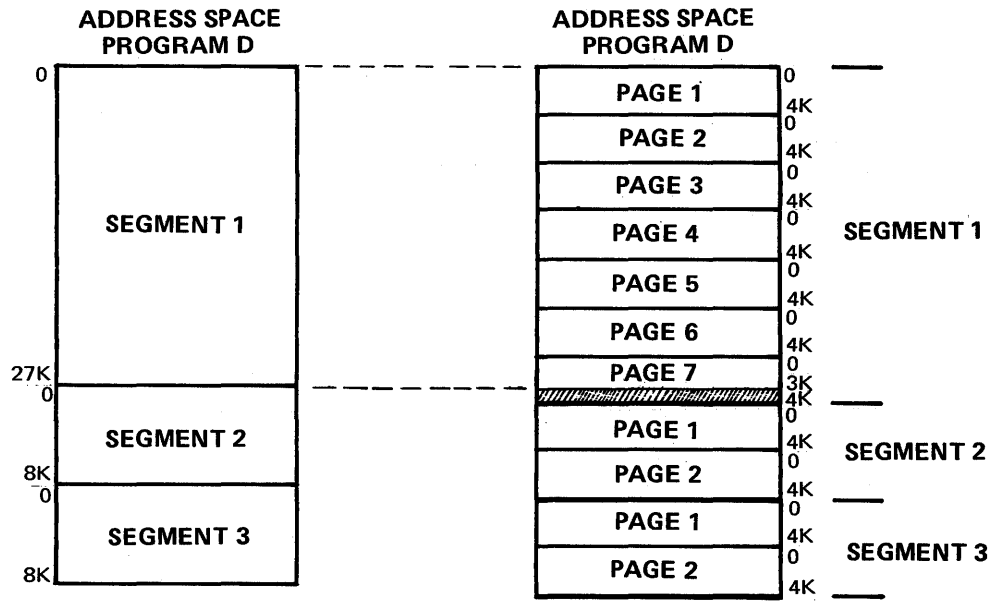


Figure 24

next program that is waiting to run. Figure 23 represents this situation.

In Figure 23 Program D is waiting to run. For the moment we will assume that all of the segments in a program must be loaded before it can run. Program D needs 43K of real storage in three contiguous areas. These three real storage areas must be at least 27K, 8K and 8K in size to load the three segments in Program D. In this situation, depicted in Figure 21, these real storage areas are not available and Program D must wait. 44K bytes of real storage are wasted during this condition. We face the same kind of problem that existed with static relocation. How do we solve the problem? There are several ways to approach it:

1. Stop all execution and reorganize the real storage layout by dynamically relocating the executing programs. This requires moving the segments in real storage, changing their segment table entries, loading Program D's segments and restarting execution. This approach can create a large amount of system overhead.
2. Let the programmer arbitrarily create small segments. This would destroy our conceptual idea of segmentation – a logical substructure of a program's address space created by an operating system – and place the burden on the programmer.
3. Let the operating system cut each segment into smaller pieces. This, in fact, is done with a *segmentation and paging* system.

In a segmentation and paging system, segments are subdivided into one or more units called *pages.* Pages are fixed in size. Page size selection is not an arbitrary decision. One factor – the one we're now considering – is the reduction of fragments in real storage. If page size is small enough compared to the typical size of segments, fragments between segments are reduced considerably. In our discussion of concepts, we will assume a page size of 4K.

Paging looks like a good approach to reduce real storage fragments. Let's try a paging technique to solve the fragmentation problem that exists in a pure segmentation system. Figure 24 shows Program D's address space using two formats – segments only and a combination of segments and pages.

When paging is used, each segment is divided into fixed size pages. In our case each page is 4K in size. In Program D, SEGMENT 1 has seven pages, SEGMENT 2 has two pages and SEGMENT 3 has two pages. Some pages may have wasted space because the instructions and data in a segment may not completely fill the last page of the segment. Note the shaded area in Figure 24. SEGMENT 1 has 27K storage locations. The first six pages contain 24K locations and 3K are contained in PAGE 7. Because page size is fixed, this results in 1K of unused locations in PAGE 7. This condition can exist only in the last page of a segment if segment size is not an exact multiple of page size. When such a page is loaded into real storage, its unused portion will result in the same amount of unused real storage. This is called intra-page fragmentation. Although this results in some waste, the waste is insignificant when compared to the increased utilization of real storage that is possible when we eliminate the unused fragments between segments.

Paging adds another level of structure to a program's address space. Instead of segments and locations within

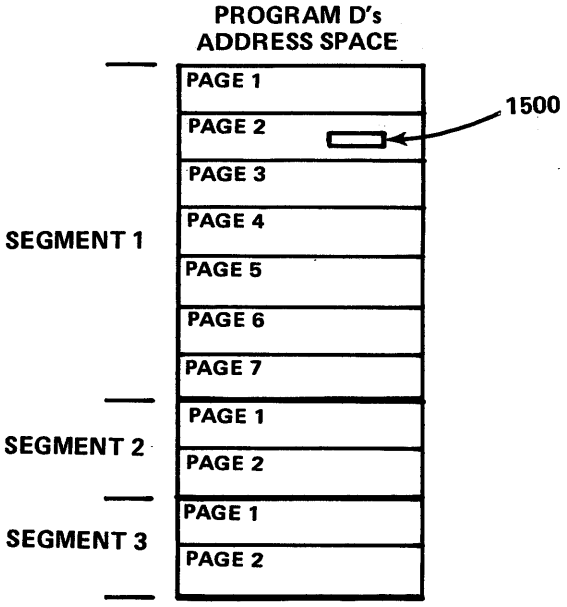


Figure 25

segments, the address space is structured into segments, pages within segments and locations within pages. This new level also requires a new address structure to reference instructions and data locations within the address space. Consider Program D in its segment-page address space as shown in Figure 25.

We would like to address an instruction that begins at location 1500 in PAGE 2 of SEGMENT 1. An absolute address of 1500 is completely ambiguous. All pages contain locations from 0 to 4K, therefore, every page in Program D has a location 1500. Even a two-part address structure is ambiguous.

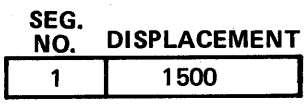


Figure 26

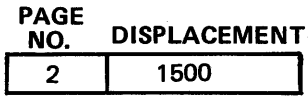


Figure 27

If the first part of the address structure identifies SEGMENT NUMBER (Figure 26) the displacement of 1500 would really point to location 1500 in PAGE 1. In reality, it is in PAGE 2 of SEGMENT 1. If the first part of the address structure identifies PAGE NUMBER (Figure 27), this also is ambiguous. There are three PAGE 2's in Program D, one in each segment. To clearly identify the location indicated in Figure 25 requires a three-part address structure. SEGMENT NUMBER, PAGE NUMBER (within segment) and DISPLACEMENT (within page) must all be indicated. Figure 28 shows this kind of structure.

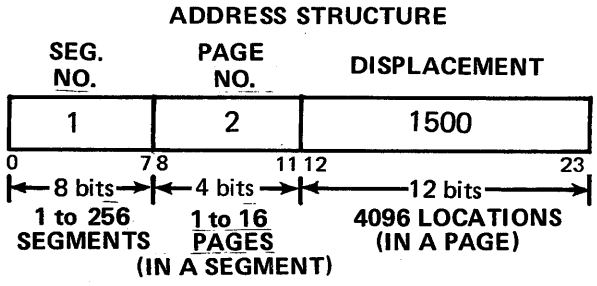


Figure 28

The address still contains a total of 24 bits. The first 8 bits contain the segment number. This allows up to 256 segments in a program's address space. Since four bits are used for PAGE NUMBER, each segment may contain up to 16 pages. The DISPLACEMENT portion of the address indicates, within the page, the location of the instruction or data being addressed relative to the first location in the page.

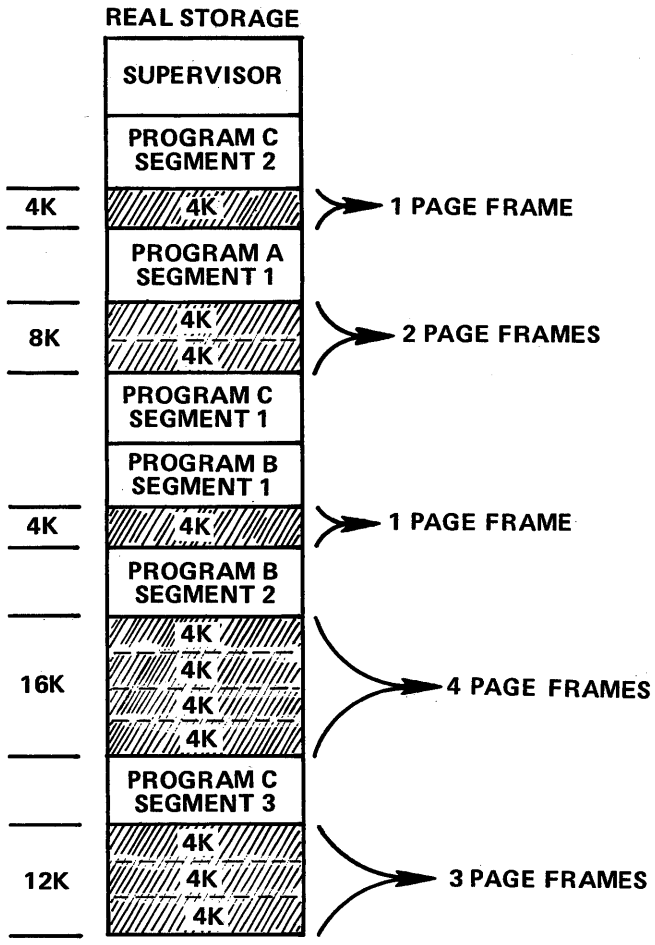


Figure 29

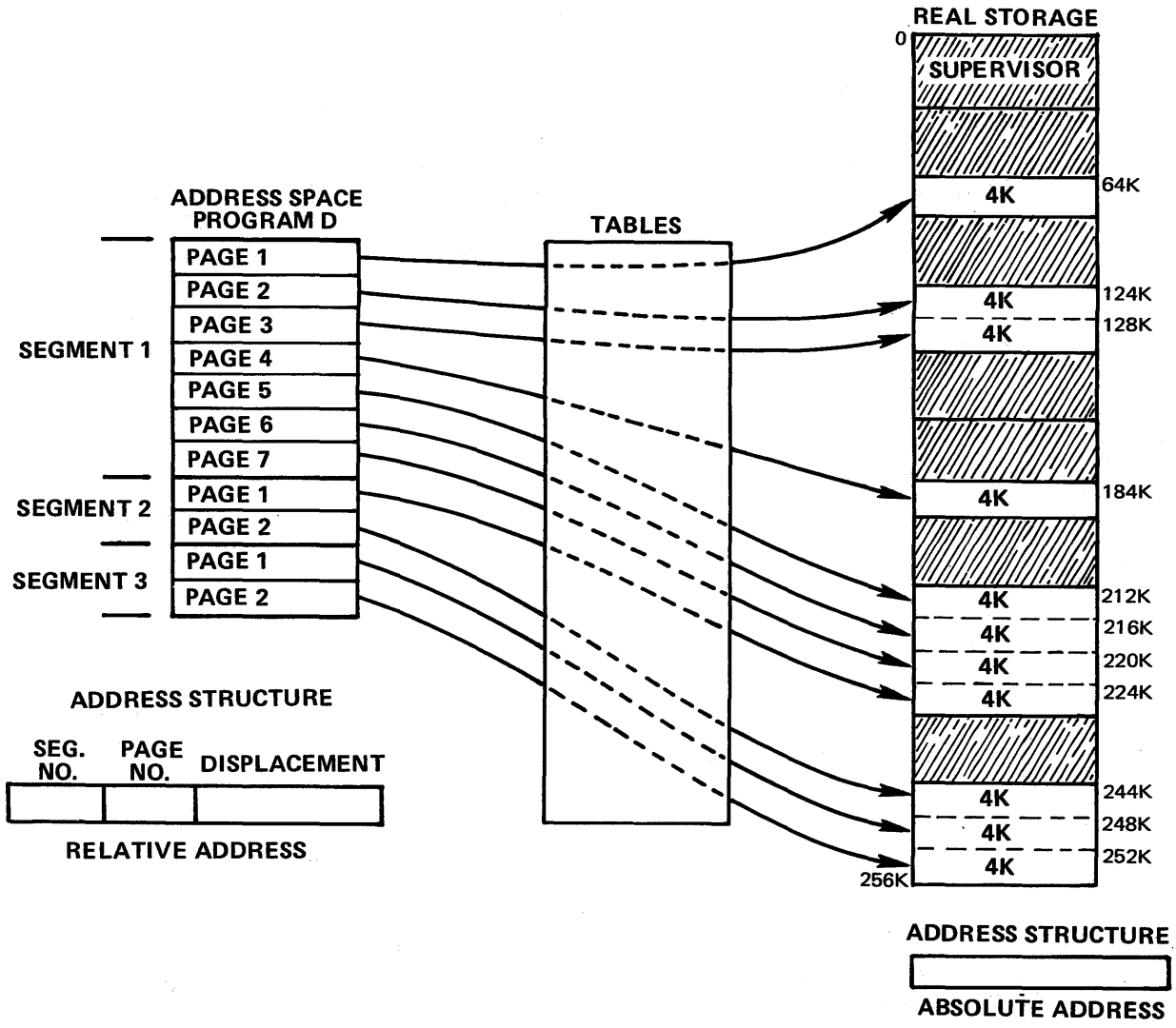


Figure 30

Thus, a three-part address is used to locate, or reference, instructions and data in a segment-page address space. The address has the structure (s,p,i) where:

- s identifies the segment name of the location within a program's address space,
- p identifies the page name (within the segment) of the location, and
- i points to the exact location being addressed within the page.

Formatting a program's address space into a segment-page structure is done automatically by our conceptual operating system. The programmer merely codes a program related to the needs of the problem being solved.

Dynamic Address Translation Using Segmentation and Paging

We will now demonstrate the advantage of a system that uses segmentation and paging. To do this we'll return to our multiprogramming system that used segmentation only.

Real storage is pictured in Figure 29. Three segmented programs are executing and a total of 44K real storage locations are unused. This unused real storage has been subdivided into *page frames.* Every real storage *page frame* is 4K in size, the same size as a page. When pages are loaded into real storage from a program's address space, they are placed into page frames of real storage. Both are identical in size. Let's load Program D into page frames.

The loading process is shown in Figure 30. Segments are loaded, one page at a time, into page frames. As loading proceeds, tables are built to identify, or map, the segment and page locations in real storage. These tables will be used

for dynamic address translation during program execution. Figure 30 shows how pages in Program D might be loaded. these pages can be placed in any page frames except those that are shaded (they are already occupied). Only one rule applies. Pages may be placed in any *free* page frame.

What types of tables do you need for dynamic address translation? Pure segmentation required a segment table to map a program's address space. Segmentation and paging systems use one segment table and multiple *page tables* – one page table for each segment in a program's address space. Let's first take a look at this new table, the page table. We will use decimal numbers during this example. Because pages are placed in page frames of real storage, the page table indicates what page frame contains each page. In Figure 30 the seven pages in SEGMENT 1 were loaded into page frames. Figure 31 shows the page table that was built to identify their real storage locations.

PAGE NO.	REAL STORAGE LOCATION
1	64,000
2	124,000
3	128,000
4	184,000
5	212,000
6	216,000
7	220,000

PAGE TABLE
(for SEGMENT 1 of PROGRAM D)

Figure 31

The page table, then, tells the computer where a page is located in real storage during dynamic address translation. The page table itself is in real storage, and each segment in a program has a page table.

What tells the computer where to find the page table? This is done with an entry in the program's segment table – because each segment has its own page table. Program D, in Figure 30, has three segments and, therefore, three entries in its segment table. It will have three page tables – one to describe each segment. Let's assume that Program D's page tables are at the following locations:

1. The page table that identifies SEGMENT 1's page locations begins at real storage location 30,000.
2. The page table that identifies SEGMENT 2's page locations begins at real storage location 30,200.
3. The page table that identifies SEGMENT 3's page locations begins at real storage location 30,300.

Program D's segment table will have the structure and entries shown in Figure 32.

PROGRAM D's SEGMENT TABLE

SEG. NO.	PAGE TABLE LOCATION
1	30,000
2	30,200
3	30,300

Figure 32

Thus, in a segmentation and paging system, segment table entries point to (map) the real storage locations of their corresponding page tables.

Figure 33 contains all the tables needed to map Program D as it was loaded in Figure 30. This mapping assumes that the segment table begins in real storage location 28,000.

In Figure 33, the real storage location for each table is indicated in the table's upper left hand corner. We'll assume Program D is executing. The segment table origin register points to the real storage origin of Program D's segment table. The segment table origin register and the segment table and page tables are the tools used by the computer's Dynamic Address Translation feature (DAT feature). Let's see how the computer translates relative addresses during program execution. We will continue to use decimal numbers.

Figure 34 contains Program D's translation tables and a relative address that references a location in its address space. (We'll assume that this is the location of an instruction.)

This relative address references an instruction in SEGMENT 1, PAGE 3, at location 1564. To translate the address, the following sequence occurs as noted in Figure 34:

1. The segment table origin register points to the origin of Program D's segment table, real storage location 28,000.
2. The segment number in the relative address is used as an index to its segment table entry. This entry identifies the origin of the segment's page table, real storage location 30,000.
3. The page number in the relative address is used as an index to its page table entry. This entry identifies the origin of the page frame that contains this page, in our example, location 128,000.
4. The displacement in the relative address and the page frame location are combined to form the absolute address 129564. Although we use addition in our example, in actual systems the page frame address and the displacement are linked – simply joined together – to form an absolute address. In other words, they are concatenated. Our example uses decimal numbers

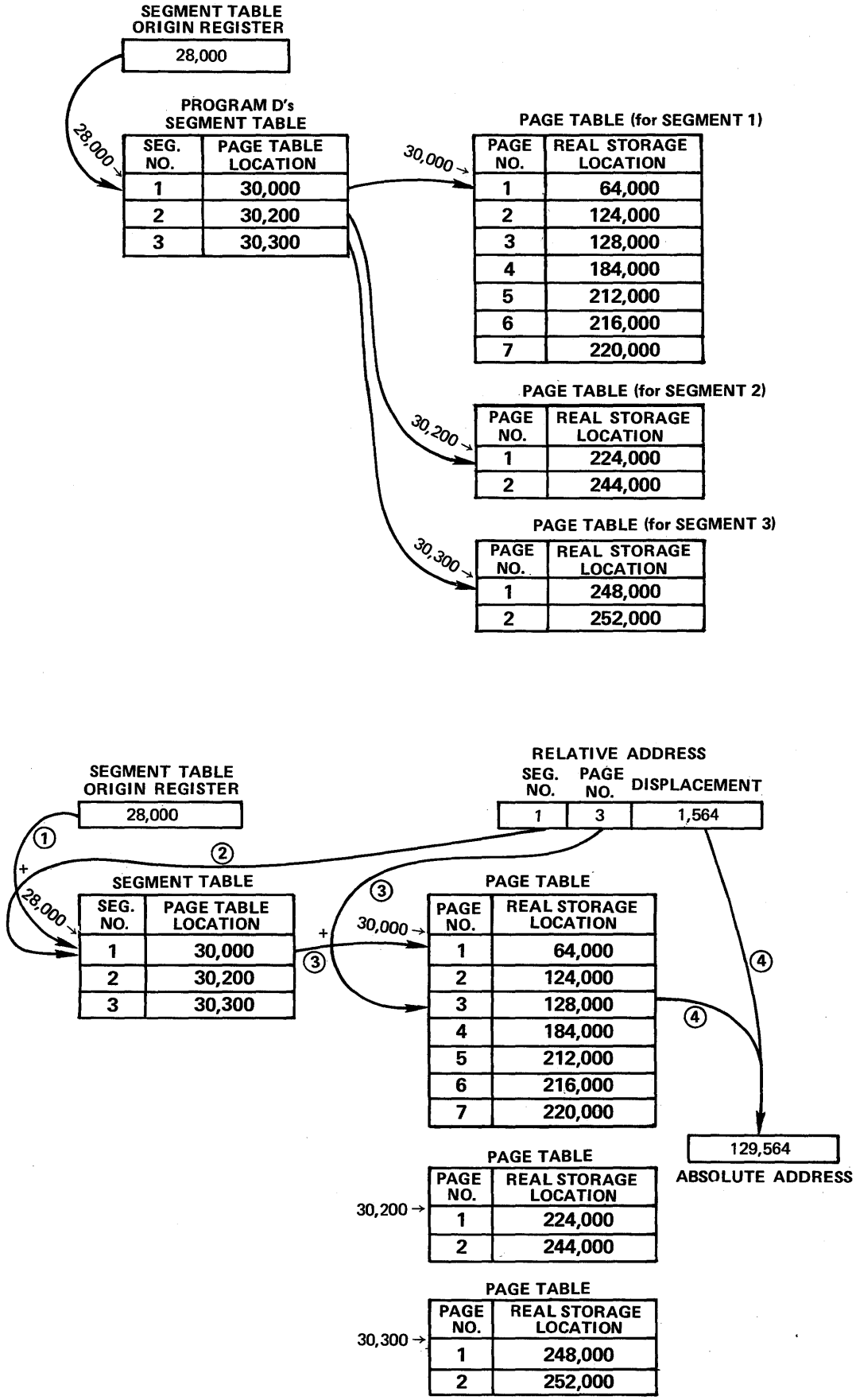

SEGMENT TABLE ORIGIN REGISTER
28,000
PROGRAM D's SEGMENT TABLE
28,000
SEG. NO.
PAGE TABLE LOCATION
1
30,000
2
30,200
3
30,300
30,000
PAGE TABLE (for SEGMENT 1)
PAGE NO.
REAL STORAGE LOCATION
1
64,000
2
124,000
3
128,000
4
184,000
5
212,000
6
216,000
7
220,000
30,200
PAGE TABLE (for SEGMENT 2)
PAGE NO.
REAL STORAGE LOCATION
1
224,000
2
244,000
30,300
PAGE TABLE (for SEGMENT 3)
PAGE NO.
REAL STORAGE LOCATION
1
248,000
2
252,000
RELATIVE ADDRESS
SEG. NO.
PAGE NO.
DISPLACEMENT
1
3
1,564
SEGMENT TABLE ORIGIN REGISTER
28,000
①
②
+
28,000
SEGMENT TABLE
SEG. NO.
PAGE TABLE LOCATION
1
30,000
2
30,200
3
30,300
③
+
30,000
PAGE TABLE
PAGE NO.
REAL STORAGE LOCATION
1
64,000
2
124,000
3
128,000
4
184,000
5
212,000
6
216,000
7
220,000
④
④
129,564
ABSOLUTE ADDRESS
30,200
PAGE TABLE
PAGE NO.
REAL STORAGE LOCATION
1
224,000
2
244,000
30,300
PAGE TABLE
PAGE NO.
REAL STORAGE LOCATION
1
248,000
2
252,000

Figure 34

and addition for our convenience in demonstrating this process to you.

This entire sequence in dynamic address translation is done automatically by computer hardware (called the DAT feature). In segmentation and paging systems, translation occurs during the entire program execution. To be sure you understand the process, translate the two relative addresses that appear in Figure 35 using the tables in Figure 34. Assume decimal numbers.

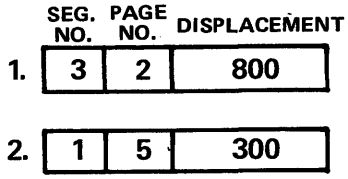


Figure 35

The first relative address results in a real storage location of 252,800, the second results in location 212,300.

It might occur to you that a 'paging only' system – instead of segmentation and paging – would be just as effective for address space management as well as real storage management. As we proceed, it will become more evident that segmentation is a better way to manage a program's, or a system's address space. Paging is a good way to manage a system's real storage; it results in a minimum loss due to fragmentation. Segmentation permits several additional benefits, such as segment protection and less page table space requirements, that we will consider in later lessons. The combination of segmentation and paging results in more benefits for a system's users than either a pure segmentation or a pure paging system.

The implementation of segmentation and paging in a multiprogramming system could be done in two ways:

1. The entire system would use one segment table; it would be one large address space, the system's address space. The supervisor and each executing problem program would have entries in the segment table to describe their structure and locations in the system's address space. Each segment table entry would be mapped (described) by a page table.
2. Each program running in the system would have its own segment table; thus, each program would have a separate address space. For every program, each segment table entry would be mapped by a page table.

We shall present more information on these two approaches later in the text.

All the dynamic relocation benefits using segmentation and paging are achieved by system hardware and software, not by the people who use the system. The software organizes a program's address space into segments and pages, builds its relative addresses, and builds segment and page tables at load time. The DAT feature translates addresses automatically during program execution. In the following lesson we'll discuss some fine points about segmentation and paging concepts, and introduce some additional hardware used to assist the translation process.

Lesson 3 Test Questions

1. Segmentation is one type of
   A. simple program loading
   B. static relocation
   C. dynamic relocation

2. A segmented address space contains
   A. relative addresses
   B. absolute addresses
   C. real addresses
   D. normalized addresses

3. Parts of real storage lost for some period of time because they are too small to be used by programs waiting to begin execution are called
   A. partitions
   B. regions
   C. fragments
   D. sectors

4. The hardware translation of relative addresses in a segmented or a segment-page format address space to their real storage locations during program execution is called_____.

5. In a segmentation and a paging system, page size is large compared to the size of a segment. (True/False)

6. In a segmentation and paging system page size is
   A. variable
   B. fixed
   C. unlimited

7. During dynamic address translation, the control register that contains the real storage location of the segment table is called the_____.

8. To perform dynamic address translation a segmentation and paging system requires
   A. one segmented table and one page table for each segment
   B. two segment tables and one page table for each segment

C. one segment table and two page tables for each segment
D. two segment tables and two page tables for each segment

9. In a segmentation and paging system, each segment table entry points to the real storage location of its
A. page displacement
B. page boundary
C. page frame
D. page table

10. Page table entries point to the real storage location of
A. page pools
B. page frames
C. page displacements
D. segment boundaries

11. Using the segment and page tables in Figure 34 translate the following relative addresses using decimal arithmetic.

	SEG. NO.	PAGE NO.	DISPLACEMENT
A.	2	2	3230
B.	1	3	0032

12. In a segment-page formatted address space, segment size and page size are determined by the
A. relative address size
B. real storage size
C. auxiliary storage size
D. relative address structure

Compared to a static relocation system like OS, a segmentation and paging system appears quite complex. OS manages real storage using regions or partitions that contain an entire relocated program in one contiguous space. A segmentation and paging system manages real storage using page frames that contain pages with relative addresses. These relative addresses require translation throughout execution. As we proceed, however, it will become more evident that a more sophisticated system that effectively manages real storage is more desirable than requiring sophisticated programming techniques (from the programmer) to manage real storage.

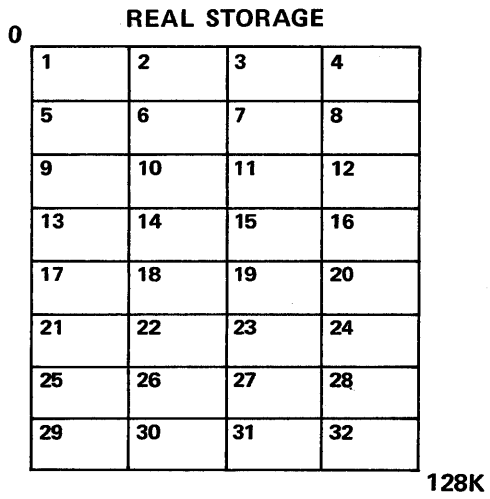


Figure 36

Page Frame Table

When we introduced paging in the last lesson, we showed how pages are placed in page frames of real storage. In a segmentation and paging system all of real storage is subdivided into page frames. They are fixed in size and they are allocated to users by the system's software. Figure 36 is a schematic presentation of real storage. Its size is 128K. Page frames are 4K in size.

In this example there would be 32 page frames available for allocation. The status of each page frame, whether it is in use – contains an active page – or whether it is available for use, can be indicated in a table. The size of the table will depend on the number of page frames in the system. In our example with 32 page frames the Page Frame Table would have 32 entries. See Figure 37.

In the Page Frame Table each entry has four parts:

1. The *Page Frame Number* identifies the page frame described in the entry. It is an index to the table.
2. The *Program ID* contains the name of the program whose page resides in the page frame. It simply identifies the program currently using the page frame.
3. The *Segment and Page Number* specifically identifies the page contained in the page frame. This part and the Program ID clearly identify the page.
4. The *Status* indicates whether the page frame is available. A zero indicates a free page frame; a one indicates a page frame in use. In Figure 37, all status entries are zero; all page frames are free.

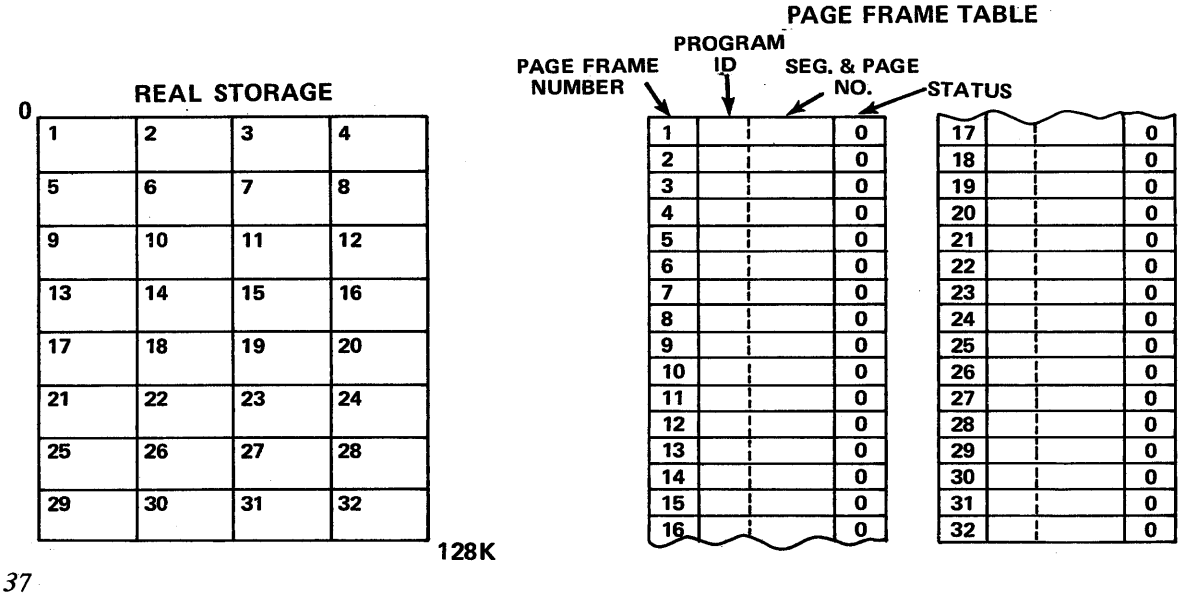


Figure 37

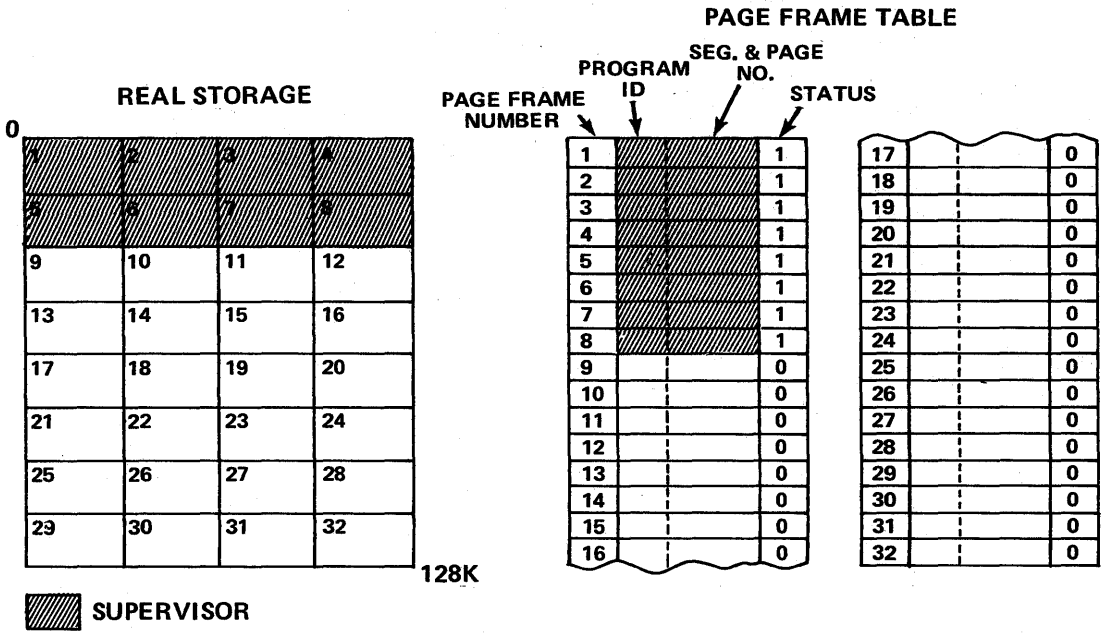


Figure 38

The Page Frame Table, then, is used to record the allocation of page frames of real storage to user programs in a segmentation and paging system. While segment and page tables are used by the DAT feature for dynamic address translation, the Page Frame Table is used by the operating system for real storage management.

Let's put a supervisor in real storage. Assume that it needs 32K. We'll use the first eight page frames. Figure 38 shows real storage and its related Page Frame Table.

The shaded area of real storage indicates the page frames used by the supervisor; the first eight entries in the Page Frame Table are also shaded. Their status entry contains a one, indicating in-use.

Now we'll load two programs into the system, Program A and Program B. Program A requires 11 page frames – it has a total of 11 pages; Program B needs 9 page frames – it has 9 pages. Figure 39 shows how these pages might be loaded into real storage and their effect on the Page Frame Table. Notice that the pages in a program need not be contiguous in real storage since their locations are mapped in page tables.

When examining the Page Frame Table in Figure 39, it is

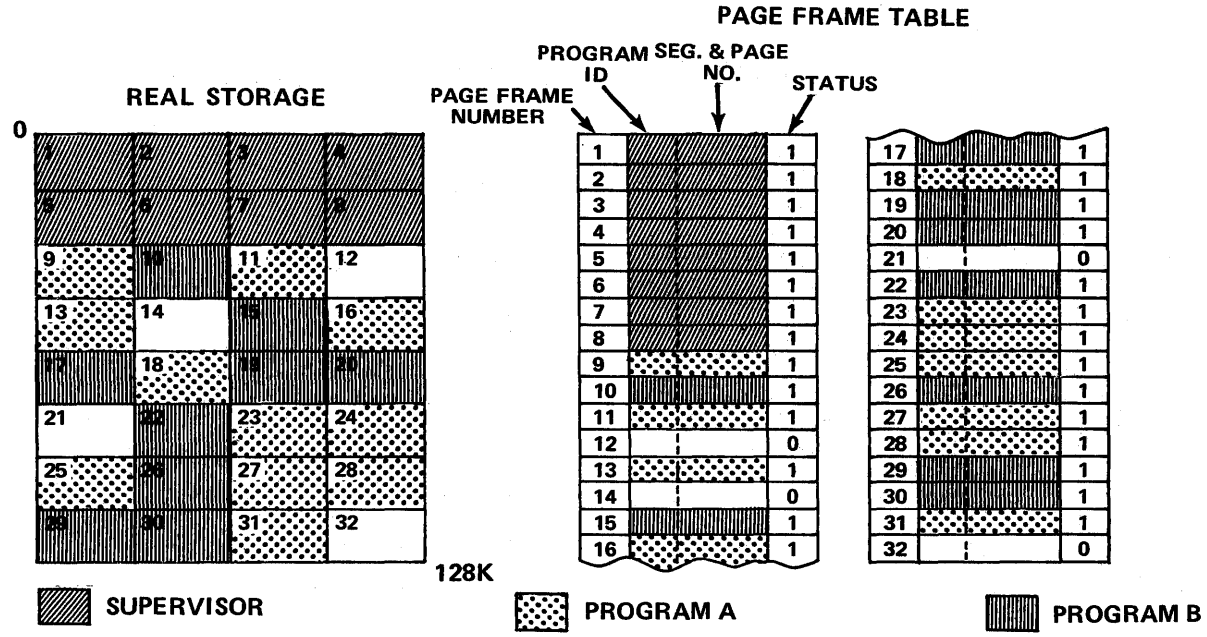


Figure 39

immediately evident what page frames are free and how those in use are allocated. When Program A ends, its status entries in the Page Frame Table are simply set to zero; the same is true for Program B. As new programs are loaded into real storage, page frames are allocated using the Page Frame Table status entries. Thus, the Page Frame Table is used by our conceptual operating system to indicate the use of real storage at all times.

Segment Protection

Multiprogramming systems require some sort of storage protection to prevent a program from referencing beyond its address space and perhaps altering or destroying another user's program. In System/360 and System/370 storage protect keys provide this type of protection. In a segmentation and paging system another higher level of storage protection can be implemented in the form of *segment protection.*

Because all storage references by a problem program go through the segment and page tables for translation, a protection mechanism can be built into the segment table entries. Thus, if a program refers to a segment that is protected, during address translation the system can generate a program interrupt and prevent the illegal reference. By using both segment protection and the storage protect keys a segmentation and paging system can provide a hierarchy of storage protection.

Associative Array Registers

It may have occurred to you by now that programs will execute more slowly if they use a segmentation and paging form of dynamic address translation. After all, relative addresses must be translated throughout program execution. During translation, the CPU's DAT feature must reference the segment and page tables and they reside in real storage. Real storage speed is slow compared to the speed of the CPU. It appears that the storage management improvements gained in a segmentation and paging system will be counter balanced by this reduction in execution speed. This would be the case if it were not for some special hardware – called *associative array registers* – in these systems.

Associative array registers are special purpose devices, much faster than real storage, that are used to compensate for the relatively slow speed of table translation. In our conceptual system we will use eight associative array registers. System/370 models (with the DAT feature) have no less than eight special purpose devices to assist translation. The larger models have a special device called the translation look-aside buffer for fast translation. Figure 40 shows the structure of our conceptual system's eight associative array registers.

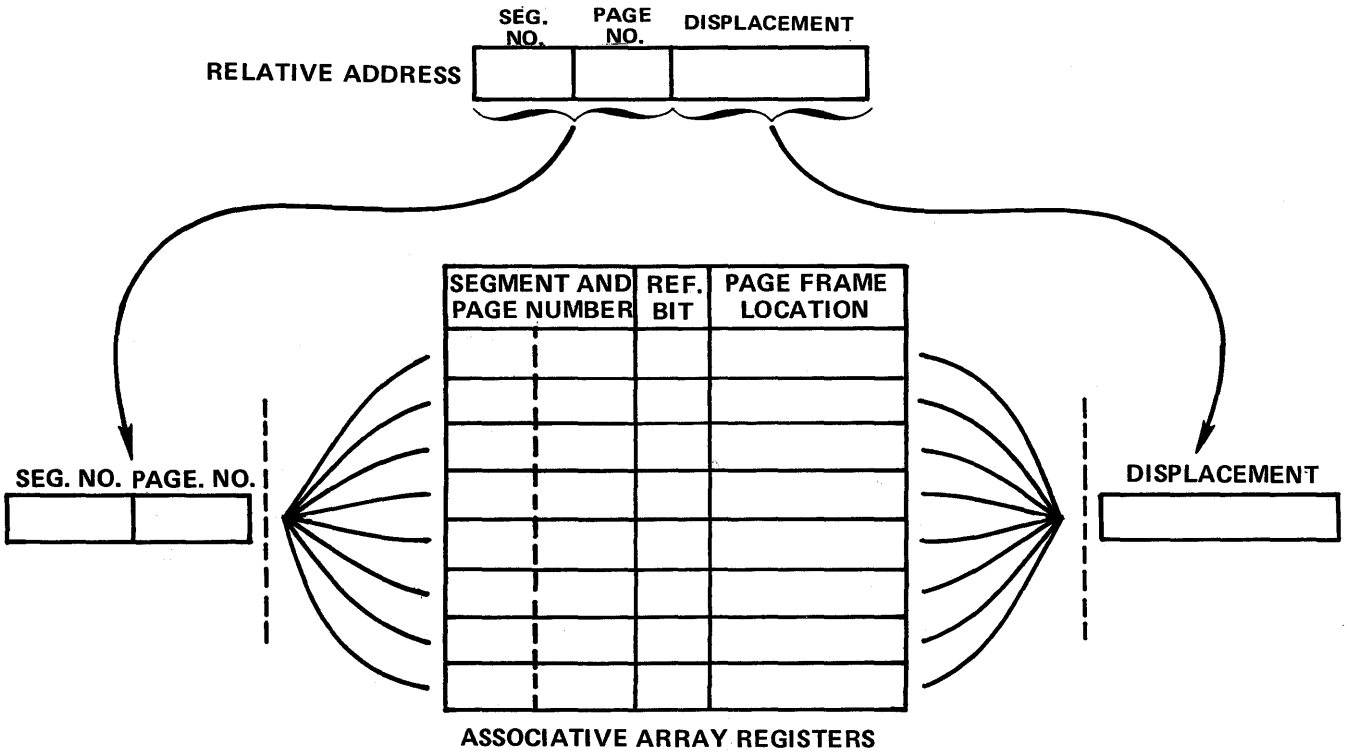


Figure 40

RELATIVE ADDRESS

	SEG. NO.	PAGE NO.	DISPLACEMENT
1.	3	6	1024
2.	1	4	324

SEGMENT AND PAGE NUMBER		REF. BIT	PAGE FRAME LOCATION
2	3	0	32,000
1	3	1	44,000
3	6	1	92,000
3	7	1	64,000
3	5	1	84,000
2	4	1	80,000
1	2	1	56,000
3	8	1	40,000

ASSOCIATIVE ARRAY REGISTERS

Figure 41

The segment and page number portion of a relative address, when referenced in a program, is compared to the first part of the registers. The registers contain the eight most recently referenced pages from the executing program identified by their segment and page numbers. If there is a match with an entry in any one of the registers, the corresponding page frame location is LINKED (or concatenated) to the relative address displacement to result in the required real storage address. Also, the reference bit of the register is turned on. These eight comparisons occur at the same time, in parallel. Because the associative array registers are very fast devices, translation time is negligible.

What happens if there is no match between the relative address and the associative array registers? Translation will occur through the segment and page tables. In fact, translation begins in the tables at the same time it begins in the associative array registers. If there is no match in one of the registers, translation is completed with the tables. The resulting page frame location and its related segment and page number would then be placed in one of the associative array registers. The register selected is determined by using the reference bits.

Let's look at an example. Figure 41 shows the associative array registers. Program A has been executing and its eight most recently referenced pages are contained in the registers. We use decimal notation in this example.

When the first relative address in Figure 41 is referenced, translation will occur through the associative registers. Even though translation also began through the tables, it is automatically stopped. Page 6 in Segment 3 is located in the third register. Its page frame location, which begins at 92,000, will be combined with the relative address displacement 1024 to form a real address of 93,024. This translation is extremely fast.

When the second relative address in Figure 41 is referenced, translation again begins through the registers and the segment and page tables. Very quickly, the system finds no match in the registers and translation continues uninterrupted through the tables. Let's assume that Page 4 of Segment 1 has an origin at real storage location 68,000. This results in a real address of 68,324. Because this is the most recently referenced page in the program, we should place its page frame location into one of the associative array registers. Examine the reference bits in Figure 41. Page 3 in Segment 2 – in the first associative register – has not been referenced lately. Its reference bit is off (zero). Therefore, the system will select this register to hold the page frame location of Page 4 in Segment 1 – that contains the relative address just translated. Figure 42 shows the new state of the associative array registers.

SEGMENT AND PAGE NUMBER		REF. BIT	PAGE FRAME LOCATION
1	4	1	68,000
1	3	0	44,000
3	6	0	92,000
3	7	0	64,000
3	5	0	84,000
2	4	0	80,000
1	2	0	56,000
3	8	0	40,000

ASSOCIATIVE ARRAY REGISTERS

Figure 42

Note that the hardware has also turned off all the remaining reference bits. The system does this whenever all the reference bits become one. Otherwise, the reference bits would eventually all become one, and the system would have no way to identify the most recently referenced pages in the executing program. This technique – keeping the most recently referenced pages in the associative array registers – is called the Least Recently Used (LRU) algorithm or rule. With the associative array registers the LRU algorithm is implemented by system hardware. We will discuss the LRU algorithm again in a later topic on page frame management.

Because the associative array registers identify the eight most recently referenced pages in an executing program, they map the most recently referenced 32K in the program assuming 4K pages. Most address translation then occurs through the associative array registers. There is very little increase in instruction execution time caused by translation through segment and page tables.

Until now our discussion of dynamic relocation has been directed at two topics:

1. The structure of a program's address space.
2. Dynamic address translation during program execution.

We have found that a segment-page structured address space results in a significant reduction of real storage fragmentation. You have seen how dynamic address translation, using segment and page tables and associative array registers, functions in this type of system. Throughout this discussion we have assumed that no program can begin execution unless there is enough real storage to hold its entire address space. We have not considered a program with an address space larger than available real storage (handled by program overlays or multipass techniques in a static relocation system). A segmentation and paging system has the ability, in effect, to "automatically overlay programs". (In fact, this concept can be extended well beyond the function of "automatic overlays".) We will begin to present this more powerful use of dynamic relocation in a segmentation and paging system in the following lessons. For now, we'll present one more topic pertinent to this expanded use of segmentation and paging.

Levels of Storage

Purely for economic reasons, computer systems have more than one level, or type, of storage. Real storage is more expensive than direct access devices, magnetic tape devices and unit record equipment. In systems that use dynamic address translation at least two storage levels are significant, real storage and auxiliary storage. These two levels form a hierarchy used in the dynamic relocation process.

Real storage may contain programs and data. All locations in real storage are directly addressable by the computer's central processing unit. The CPU may fetch instructions and data from real storage and return results. For this reason, programs can execute only when they reside in real storage. Auxiliary storage, for purposes of our presentation, will consist only of direct access devices, typically disk devices. Their capacity is much larger than real storage. But the CPU cannot directly access instructions or data that reside in auxiliary storage. For this reason, programs cannot execute directly from auxiliary storage. They must first be loaded into real storage. How a program's address space is structured and when its addresses are translated has been the major concern of this text. Until now, we have been interested in the real storage management effect of this process. We will begin to include the role of auxiliary storage as we expand the possibilities of segmentation and paging systems.

Lesson 4 Test Questions

1. When loaded into real storage, pages are placed in
   A. page frames
   B. storage blocks
   C. page slots
   D. storage slots

2. In a system that uses segmentation and paging, the table that indicates real storage availability is called the
   A. page table
   B. page frame table
   C. real storage table
   D. segment table

3. Dynamic address translation through associative array registers is slower than translation through segment and page tables. (True/False)

4. Translation through associative array registers occurs in parallel with translation through segment and page tables. (True/False)

5. If translation fails using the associative array registers, translation will continue to completion using the segment and page tables. (True/False)

6. To decide what associative array register to select for storing the segment and page number and the associated

page frame location of a newly translated address, the system checks the

A. change bits
B. reference bits
C. protect bits
D. storage bits

7. When deciding what associative array register to replace, the system tries to replace the most recently referenced page. (True/False)

8. Using the contents of the associative array registers in Figure 41 of this text, translate the following relative addresses using decimal arithmetic.

	SEG. NO.	PAGE NO.	DISPLACEMENT
A.	1	3	1024
B.	3	3	0240

9. Name the two levels of storage described in this lesson.

10. In a computer system the CPU may fetch and store only data or instructions that reside in real storage (True/False)

Lesson 5. Virtual Storage

In the beginning of this text we introduced the idea that a program is associated with its address space, while a computer system is associated with its real storage space. The program's address space is contiguous, or linear, and it contains the set of addresses generated for its program. The real storage space is also linear. It can be regarded as the set of physical locations addressable by the CPU. Until now, our entire presentation has related to two questions:

1. How does translation occur from a program's address space into real storage locations?
2. When does this translation occur?

We concluded that we can best manage the real storage resource by structuring a program's address space into segments and pages, by structuring real storage into page frames the same size as pages, and by allocating free page frames to a program's pages. Using this technique, we saw that translation occurs, through segment and page tables or associative array registers, at execution time. Translation continues throughout program execution. During this entire discussion, we have arbitrarily restricted the size of a program's address space to an equal amount of available real storage for execution to begin. This implies that a program's address space can never be larger than real storage size, but this is not true. Early in this text we demonstrated that the maximum size of an address space is established by a computer system's address structure. A computer system with a 24-bit address structure may have an address space up to 16,777,216 addressable positions (16 megabytes). This concept of an address space that may be much larger than real storage is called *virtual storage.*

Virtual Storage Size

A virtual storage system can be used to simulate a large real storage. Figure 43 illustrates this concept of virtual storage and places it next to a 512K real storage.

The address structure in Figure 43 is 24 bits long. The range of addresses that can be referenced with 24 bits spans from 0 to 16,384K (2^{24} equals 16,384K). Thus, with a 24-bit address, virtual storage size can be 16,777,216 locations (16 megabytes), 32 times larger than the 512K real storage shown in Figure 43. System/370 has a 24-bit address. Its virtual storage can be 16 megabytes (16 megs). Notice that we say *can* be 16 megs. This is maximum virtual storage size. You can use a smaller virtual storage. We will discuss why you might use a smaller virtual storage in later topics. Virtual storage gets its name from the fact that it

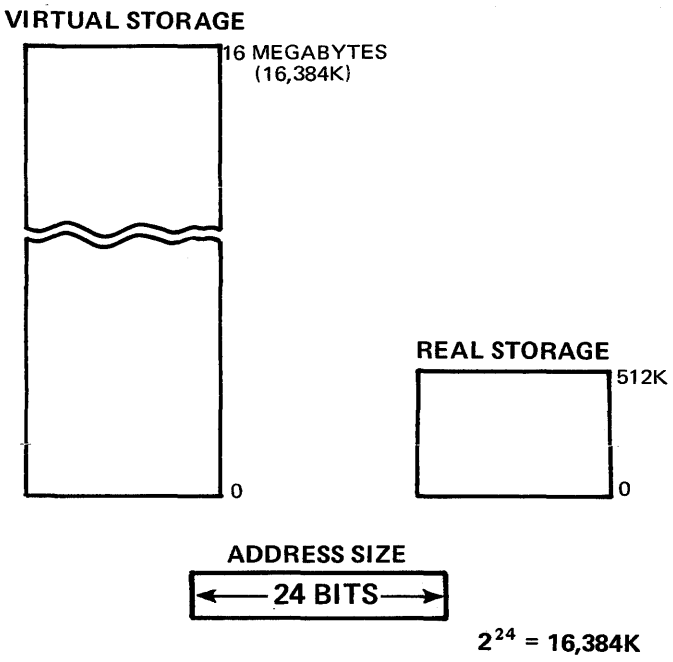


Figure 43

extends well beyond the size of a computer's real storage. You can see real storage, but you can't see virtual storage. Virtual storage is a large address space. Where does virtual storage physically exist, if anywhere? We will answer this question in a moment. First, let's take a look at a way to structure virtual storage, the large address space.

Virtual Storage Structure

We found that a segment-page structure is good for a single program's address space. When multiprogramming, fragmentation of real storage is reduced, and segments can be protected. A segment-page structure is also very suitable for virtual storage. Virtual storage can be allocated, or managed, in segment size increments. Real storage can be allocated, or managed, using page size increments. We will divide virtual storage, the large address space, into fixed size segments, and its segments into fixed size pages. The resulting structure of virtual storage will then determine how a system's DAT feature must interpret the addresses that reference virtual storage – virtual addresses. Look at Figure 44.

Each segment is 64K. Because there are eight bits in the segment number portion of our virtual address, there may be from 1 to 256 segments. There are 256 segments in 16 megs of virtual storage that we have numbered 1 to 256. In

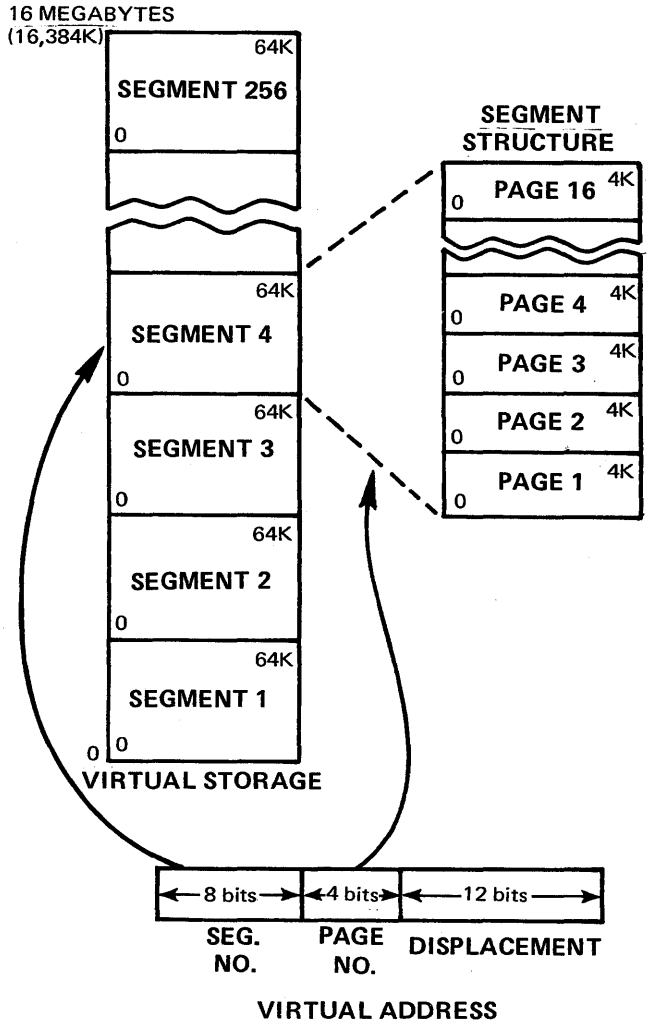


Figure 44

Figure 44 SEGMENT 4 is enlarged to show you the page structure within segments. Segments are 64K. Pages are 4K. Thus, there are sixteen pages in each segment, that we have numbered 1 through 16. The page number portion of our virtual address needs four bits to reference pages 1 through 16 in each segment. The remaining twelve bits in our virtual address indicate the displacement of an instruction or data within a page. Each page has 4K locations from 0 to 4095. Although we have assumed that page size is 4K and segments 64K, we could describe a virtual storage with 1024K segments and 4K pages or 64K segments and 2K pages. Segment size and page size determine the way the DAT feature interprets a virtual address – number of bits for segment number, page number and displacement. In this topic we have described the structure of virtual storage in a segmentation and paging system. As we continue its description and discuss its functions and attributes you might think that we are describing a computer's real storage. In a virtual storage system, virtual storage replaces our concept of real storage in a conventional computer, and the computer's real storage becomes a resource managed by the operating system. We will return to this concept as we unfold our description of virtual storage. Now let's return to our unanswered question. Where does virtual storage exist, if anywhere?

The Relationship of Virtual Storage to Real and Auxiliary Storage

Because virtual storage might be 32 times larger than real storage it can't *all* be represented in real storage at one time. It must exist in one of the system's two levels of storage – either real or auxiliary storage. However any virtual storage not in use need not exist anywhere. It is *potential* address space. What do we mean by potential address space?

Suppose that you have a program, Program A. It needs 80K of storage. Because we will allocate virtual storage to programs in whole segments we give Program A two segments (128K). Figure 45 shows Program A loaded in virtual storage. This is the shaded area in segments one and two. Notice that all of Program A's pages (the 80K that Program A uses) are also contained in *slots* of *external page storage.* External page storage is the part of auxiliary storage that is used to store pages in a virtual storage system. A slot is a record area in external page storage. It is the same size as a page. Therefore pages in use are shaded in slots of external page storage. Because Program A has been loaded into virtual storage it must exist somewhere in the system. All of its pages are in slots of external page storage.

At this point in time, we assume that Program A is executing. Some of its pages are also in page frames of real storage. See the shaded portion of real storage in Figure 45. To be executed, all referenced instructions and data must reside in real storage. If, during the translation of a virtual address in Program A, the system detects that the page being referenced is not in real storage, the system must transfer a copy of that page from external page storage into real storage. The page is literally brought into real storage on demand. This technique is called *demand paging.* How the system detects this condition, and how the page is moved will be covered in the next lesson. Also covered will be questions like "how is Program A loaded into virtual storage?" But now, let's return to Figure 45. We still haven't said what we mean by potential address space. Program A has the first two segments of virtual storage. However, only four pages are in use in SEGMENT 2. The remaining twelve pages in SEGMENT 2 are not used. The remaining segments in virtual storage are not allocated. These pages and these segments are potential virtual storage, or potential address space. In a virtual storage system in which all active programs share one virtual

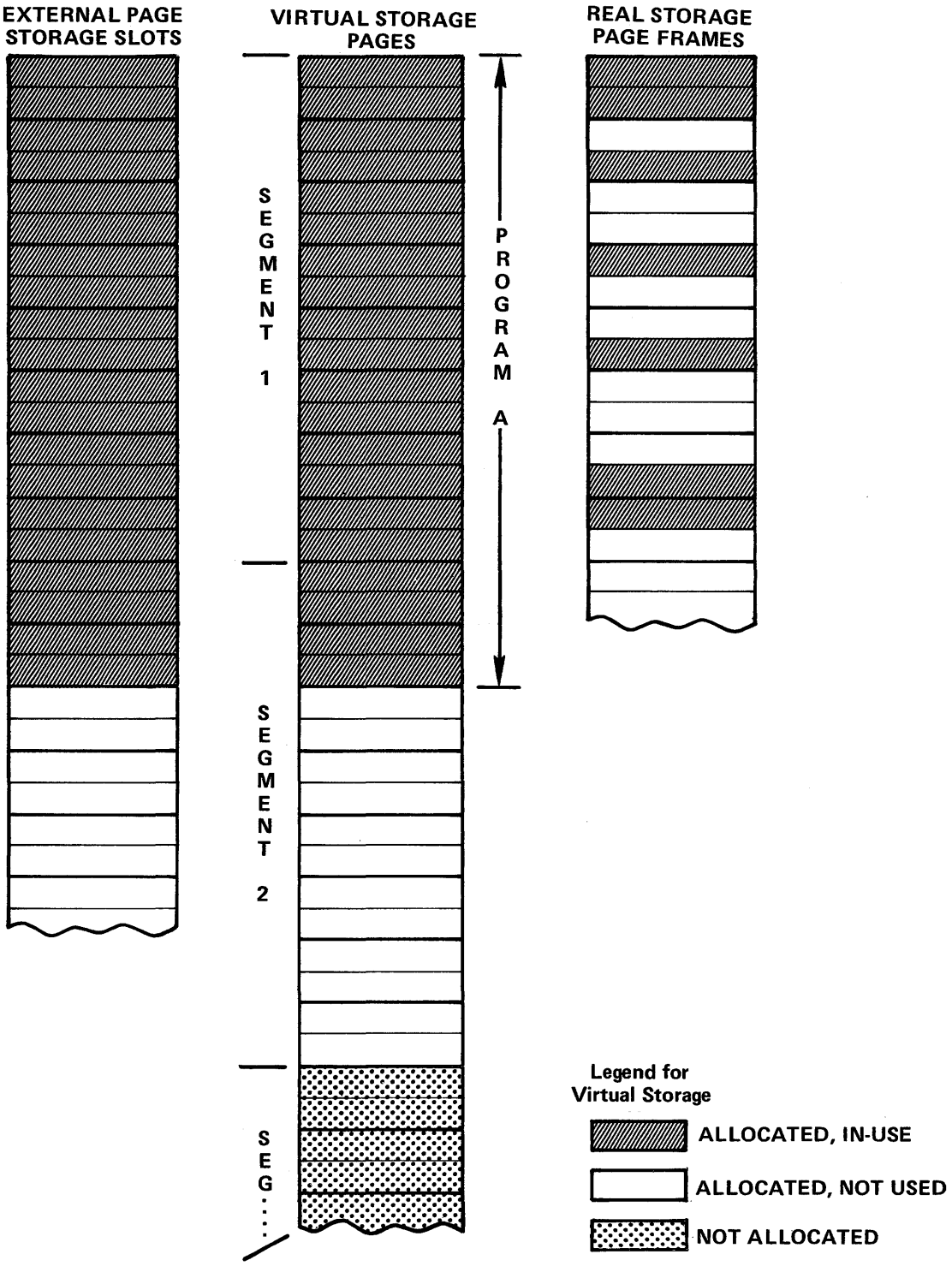


Figure 45

storage, unallocated segments are available for new programs. The remaining pages in SEGMENT 2 could be used by Program A, if required, but they are not available to other programs since the whole segment has been allocated to Program A. In a virtual storage system in which each active program has its own virtual storage, Program A could use the remainder of virtual storage – all pages and segments – if required. We will return to the idea of one or multiple virtual storages in a moment.

Because virtual storage is typically larger than real storage, a programmer has much more space to design and code a problem solution. The need to use overlay structures or multistep jobs – the programmer's problem because of a lack of real storage – is greatly reduced, if not eliminated completely. By use of dynamic address translation and demand paging the system can be thought to "automatically overlay" programs in page size increments. Demand paging

requires that only part of a program reside in real storage at any time during execution. This is true even if the entire program could easily fit into real storage. This allows a virtual storage system to allocate the real storage resource among more programs according to their actual needs, and to start more programs running when required.

Single Virtual Storage and Multiple Virtual Storages

Virtual storage systems can be implemented with a *single virtual storage* or with *multiple virtual storages.* The single virtual storage system has the following characteristics:

1. The maximum size of virtual storage is set by the computer's address structure. As we said, a 24-bit address results in a maximum virtual storage of 16 megabytes. In some implementations you can use a virtual storage smaller than the maximum and in many situations this may be desirable. We will discuss this topic again in PART II.
2. The single virtual storage is mapped by one segment table with a page table for each allocated segment.
3. In a single virtual storage system, virtual storage is considered as the system's address space (contrasted to the idea of a program's address space that we have presented this far). All active programs in the system, including the system control program, are mapped into the single virtual storage, the system's address space. Virtual storage, or the system's address space, is allocated to user programs in segment size increments.
4. In a single virtual storage system the supervisor and all active programs are structured in virtual storage just as they are structured in real storage in a static relocation system like OS/MFT or OS/MVT.

In a multiple virtual storage system there is one essential difference. Each user, whether a batch job or a timesharing user, may have his own virtual storage. Some characteristics of a multiple virtual storage system are as follows:

1. The maximum size of virtual storage is set by the computer's address structure. No user's virtual storage may exceed the maximum. However, some users may have a virtual storage size that is smaller than the maximum.
2. In a segmentation and paging system, each virtual storage is mapped by a segment table with page tables for all allocated segments.
3. In a multiple virtual storage system, virtual storage is considered as the user's address space (similar to the idea of a program's address space that was presented earlier). Some of each user's virtual storage or address space may be used by system control programs, but only the assigned user will control the remainder of the address space. We will discuss this consideration again later.
4. Because each user has his own address space, a special purpose control register (that we described in Lesson 3 as the Segment Table Origin Register, *STOR register*) must direct the DAT feature to the segment table of the user that is executing for proper address translation.
5. Also, because each user has his own address space, system protection is improved. A user may only reference within his own address space. There is no way that one user may "damage" another user's address space.

We have expressed several times the theoretical limits of virtual storage systems. Virtual storage size is limited only by the computer's address structure. In a multiple virtual storage system the number of virtual storages is theoretically unlimited. However, there are practical limits. The size of real storage, the size and speed of the auxiliary storage used for storage management, the speed of the dynamic address translation process and even programming style are all parameters that have an effect on a successful virtual storage system implementation. We will expand the discussion of these parameters as we continue our presentation of virtual storage concepts.

The combination of segmentation and paging for virtual storage implementation is a very successful one. Segmentation provides a good technique to manage and allocate virtual storage. Segments may be protected and in some cases shared. Paging is an excellent way to manage and allocate both real and external page storage. It results in minimum waste and dynamic sharing of one of the system's key resources, real storage.

In the following lesson we will describe how a virtual storage operating system works, for both a single and multiple virtual storage system, and how a program actually executes in a virtual storage system.

Lesson 5 Test Questions

1. If a computer's address register is 24 binary bits long its maximum virtual storage size, or address space size, is
   A. 4,194,054
   B. 8,388,108
   C. 16,777,216
   D. 33,554,432

2. Maximum virtual storage size is directly dependent on the size of a computer's
   A. real storage
   B. address register
   C. external page storage
   D. control registers

3. Virtual storage structure is directly dependent on the _____.

4. In a virtual storage system in which virtual storage has a segment-page structure, the virtual address has a three-part structure in this order
   A. page number, segment number, displacement
   B. program number, segment number, displacement
   C. segment number, page number, displacement
   D. page number, displacement, page number

5. In our conceptual virtual storage system both segments and pages are fixed in size. (True/False)

6. In our conceptual virtual storage system segments and pages are the same size. (True/False)

7. In a virtual storage system that uses segmentation and paging, segmentation is a good technique for managing and allocating
   A. real storage
   B. auxiliary storage
   C. external page storage
   D. virtual storage

8. Paging is a good technique for managing and allocating
   A. real storage
   B. auxiliary storage
   C. external page storage
   D. virtual storage

9. In a single virtual storage system, all active jobs share the same virtual storage. (True/False)

10. In a multiple virtual storage system, each user has his own virtual storage. (True/False)

11. Unused virtual storage, or "potential" address space requires the use of external page storage. (True/False)

In our discussion of virtual storage, we said that pages being used must reside in either real or external page storage, external page storage being represented on direct access storage. In addition to storing pages, auxiliary storage has many traditional uses as a second level of storage in a computer system:

1. It contains a copy of the operating system's nucleus or supervisor.
2. It holds the operating system's program libraries.
3. It contains user program libraries and data sets.

As we said before, the part of auxiliary storage that is used to store pages is called *external page storage.* External page storage is organized into *paging data sets* where copies of all pages in use are stored. Record areas in paging data sets are called *slots.* Thus, a slot is the same size as a page, 4K in our examples. All loaded pages of virtual storage reside in slots of external page storage. Some *active pages,* that is those that have been recently referenced during execution, will also reside in real storage.

External Page Tables

When we described dynamic address translation using segment and page tables, we considered only programs that would fit into real storage. With a virtual storage system in which pages may be in real storage or external page storage, we must expand our mapping concept. This requires an additional item in the page table entry and a new type of table which we will call the *External Page Table.*

Consider Program A. Assume that it has been loaded into virtual storage and it is executing. Three segments have been allocated to Program A. Figure 46 shows Program A's segment table and one of its page tables.

Each page table entry has a new item, the *invalid bit.* Its function is to tell the DAT feature whether or not its page resides in a real storage page frame; if the invalid bit is off (zero), the page is in real storage; if the invalid bit is on (one), the page is in a slot of external page storage. In Figure 46, PAGE 3 and PAGE 5 are in page frames; the remaining pages are in slots of external page storage.

Figure 47 contains two virtual addresses.

What will result if we translate the addresses in Figure 47 using the tables in Figure 46? The first virtual address can be translated successfully. During translation of the second address the DAT feature generates a program interrupt. The invalid bit is on. This page resides in a slot of external page storage; but what slot? Its slot location is contained in another table called the *external page table.*

	SEG. NO.	PAGE. NO.	DISPLACEMENT
1.	2	3	1024
2.	2	4	2048

VIRTUAL ADDRESSES

Figure 47

In a virtual storage system, each page table has a corresponding external page table. See Figure 48. The external page table maps the slot locations for all pages that reside in the system's external page storage. In our preceding example, translation was unsuccessful. The page identified by the second virtual address did not reside in a page frame of real storage.

SEGMENT TABLE

SEG. NO.	PAGE TABLE LOCATION
1	64,000
2	65,024
3	66,026

PAGE TABLE

PAGE NO.	PAGE FRAME LOCATION	INVALID BIT
1	84K	1
2	——	1
3	80K	0
4	112K	1
5	120K	0
6	——	1

Figure 46

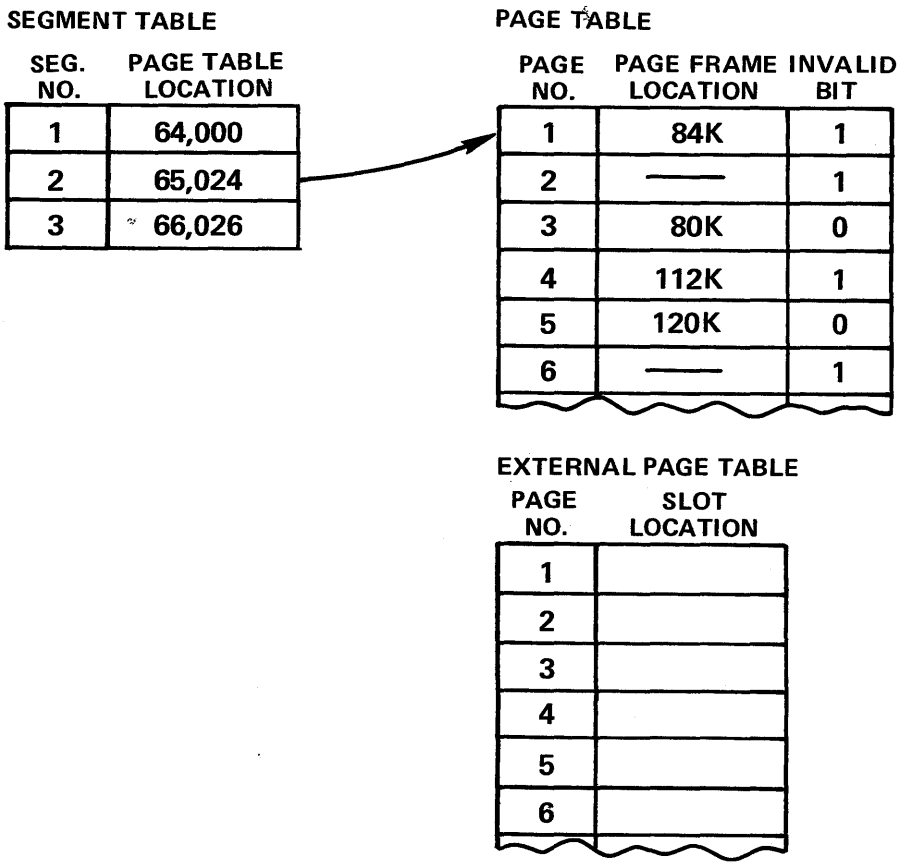


Figure 48

In this situation, with the addition of the external page table, the system can now:

1. Locate the page in external page storage.
2. Select a page frame of real storage to hold the page.
3. Move a copy of the page from its slot in external page storage to the page frame in real storage.
4. Update the page table entry to show that the page is resident in real storage.
5. Complete the virtual address translation.
6. Resume program execution.

With the external page table, we have introduced the final tool required in a two-level virtual storage system that uses segmentation and paging. We're ready to see how all these tools – dynamic address translation hardware, segment tables, page tables, external page tables, associative array registers and the page frame table – are used in the implementation of a virtual storage operating system. We will approach this by examining how to load and execute programs in a virtual storage system.

Program Execution – Demand Paging

First we will examine program execution in a virtual storage system. We will then describe a virtual storage operating system that uses a single virtual storage, the system's address space, to show how programs are loaded in such a system. The final part of this lesson will describe a multiple virtual storage system and how the virtual storage concept may be expanded into the concept of a *virtual machine*.

To begin our discussion of program execution, we will assume that the program is already loaded in virtual storage and return to the demand paging technique. To begin execution, one or more of a program's pages are loaded into available page frames, the page frame table is updated, the appropriate page table entries and their invalid bits are updated and execution begins. The program's virtual addresses are translated using the DAT feature and the associative array registers. So long as these virtual addresses refer to pages that reside in real storage, execution continues. What occurs when the program references an instruction or data in a page that does not reside in real storage? This condition is called a *page fault.* The referenced page must now be loaded into a page frame of real storage. This is called a *page-in* operation. We have already seen how a page fault is detected. The hardware detects this situation during translation by automatically checking the invalid bit of the page table entry. In the corresponding external page table the system can find the address of the slot in which the page resides in external page storage. The system must now

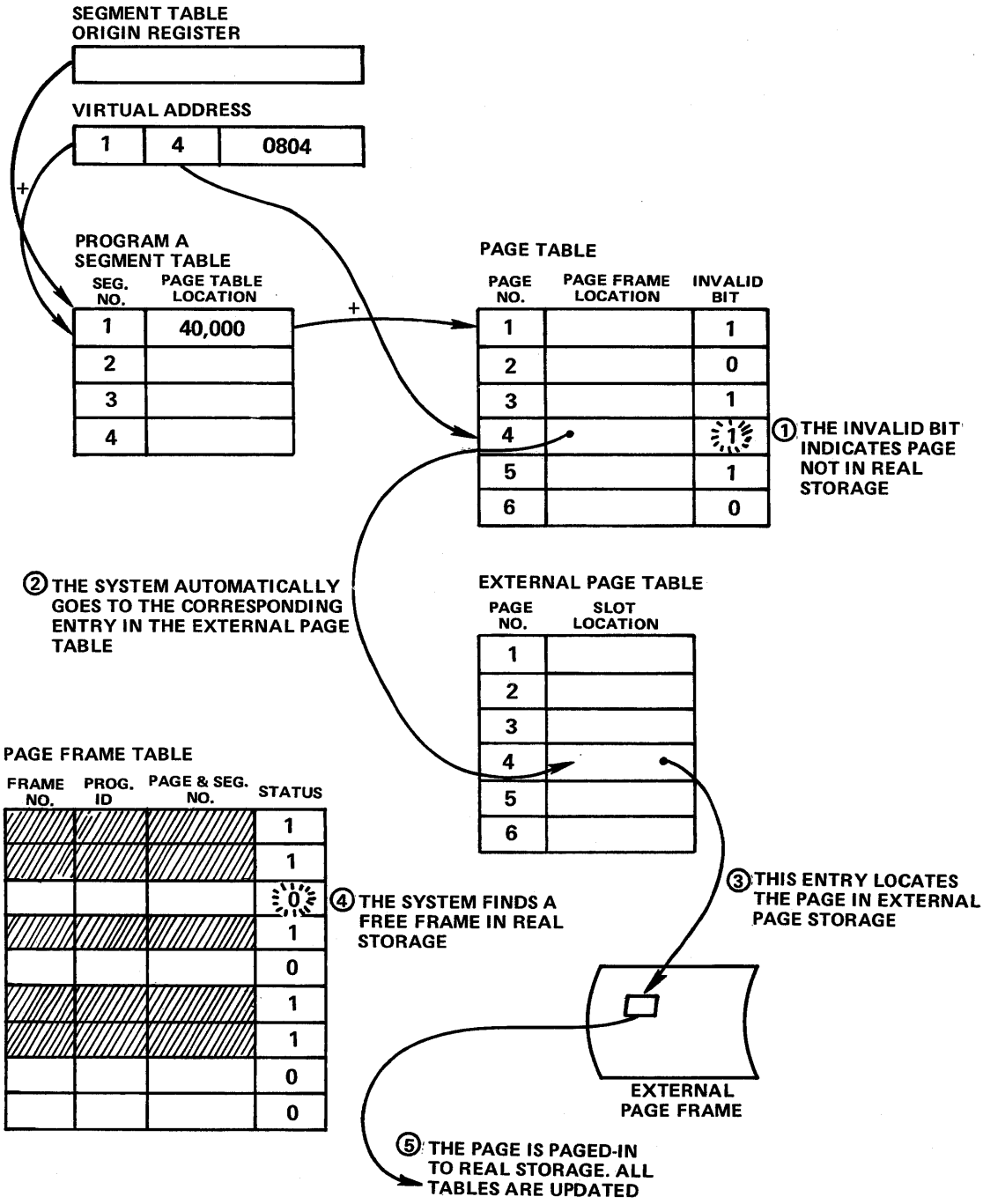


Figure 49

select a page frame of real storage to hold the required page. To do so it must reference the page frame table and find a free page frame. For now, let's assume that free page frames will exist, and that we'll use the first available page frame found in the table. The page is transferred or paged in from its slot in external page storage to the page frame in real storage; the page frame table is updated; the page table entry is updated; the page's real storage page frame location is placed into an associative array register. The page-in operation is complete, and program execution may resume.

Figure 49 outlines this process. In a multiprogramming system other programs will be executing during page-in operations to avoid lost system time. This is because a page-in operation is an input/output operation. The computer's CPU is free to process other programs during the time needed to transfer the page. During the execution of a program the demand paging technique continues to load new pages as they are required. If a page is never referenced during execution, it will never be paged in.

Page Replacement

Servicing page faults, even though it requires several steps, is straightforward so long as free page frames are available. In a multiprogramming system, where virtual storage is larger than real storage, all of the free page frames of real storage will eventually be filled. A page fault in this situation requires a *page replacement* operation. The replaced page may first have to be *paged out.* In this page fault condition, with no available real storage, the page to be brought into real storage may replace any page residing in a frame allocated to users (some frames may be reserved for supervisor residence and other special purposes). This is true even if the page being replaced belongs to a different user. The page replacement strategy is an important issue because it affects system performance in two ways:

1. If you replace a highly referenced page you may need to bring it back into real storage in a short time, causing perhaps unnecessary system work.
2. If you replace a page whose contents have been changed in any way since it was paged in, you must first perform a page-out operation; you must save the altered page in external page storage.

Let's consider the first item. There are a variety of rules that could be used to select a page for replacement. You could use a FIFO replacement rule, that is, the *first* pages *in* real storage are the *first out.* This method would replace those pages longest resident in real storage. A second rule would give ideal results. If you know the pattern of future page references, you can replace a page that will not be referenced again at all, or for a long time. Although this ideal replacement rule would be most effective, it is unusable because the system can't predict future events. It is useful in simulation, however, for comparison to replacement rules that are usable. One that compares well with the ideal is called the Least Recently Used (LRU) rule. It will replace first those pages that haven't been referenced in a long time; that is, the least recently used pages. In other words, the LRU rule attempts to keep the most recently referenced pages in real storage, assuming that they are more likely to be referenced again in the near future. The LRU rule is used with associative array registers as we mentioned earlier. Several System/370 virtual storage systems use the LRU replacement rule to manage real storage. We will not say how in detail; only that they replace a single page frame table with a set of page frame queues, the lowest priority queue containing the least recently used pages and so forth. In our discussion, we will continue to use a single page frame table.

Reference and Change Bits

We haven't discussed yet how the system knows that a page is referenced or how the system knows that a page is changed. This is done by hardware control bits contained in each page frame's storage protect key. One bit in the storage protect key is called the *reference bit,* one bit is called the *change bit* and the others are used for storage protection. The reference bit is turned on (automatically by the hardware) whenever data is referenced (fetched) from or stored in the contents of its page frame, that is, from the page stored in its page frame. The reference bits are used to implement the LRU replacement rule in a manner similar to the reference bits in associative array registers. The change bit is turned on whenever data is stored into the page residing in its real storage page frame. Change bits are used to decide if a page-out operation is necessary. The reference and change bits are represented as new entries to our frame table as shown in Figure 50.

With this foundation we can return to our problem. How do you handle a page fault condition that requires page replacement?

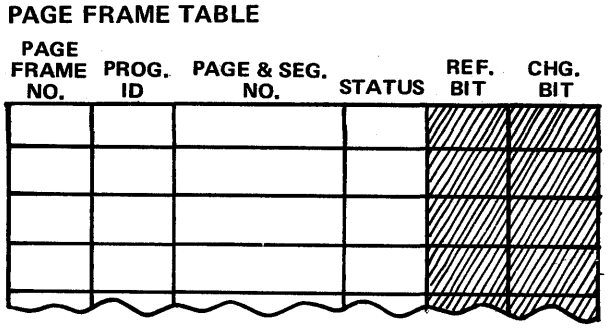


Figure 50

The page-in operation remains the same, but first the system must select a page for replacement. We'll assume that the system is using the LRU replacement rule and examine two examples:

1. In the first example the page to be replaced is unchanged since being paged in.
2. In the second example the page to be replaced has been changed since it was paged in.

Our first example doesn't add many new steps for handling page faults. The sequence of events occurs in the following way. During translation of a virtual address a page fault occurs; the invalid bit for the required page is on because the page is not in real storage. The system must interrupt program execution to service this page fault. The system must first find a page frame of real storage. To do this, it examines the page frame table. If there is a free page frame, the system will use it. In our case, all page frames are occupied. The system, using the LRU rule, selects a page for

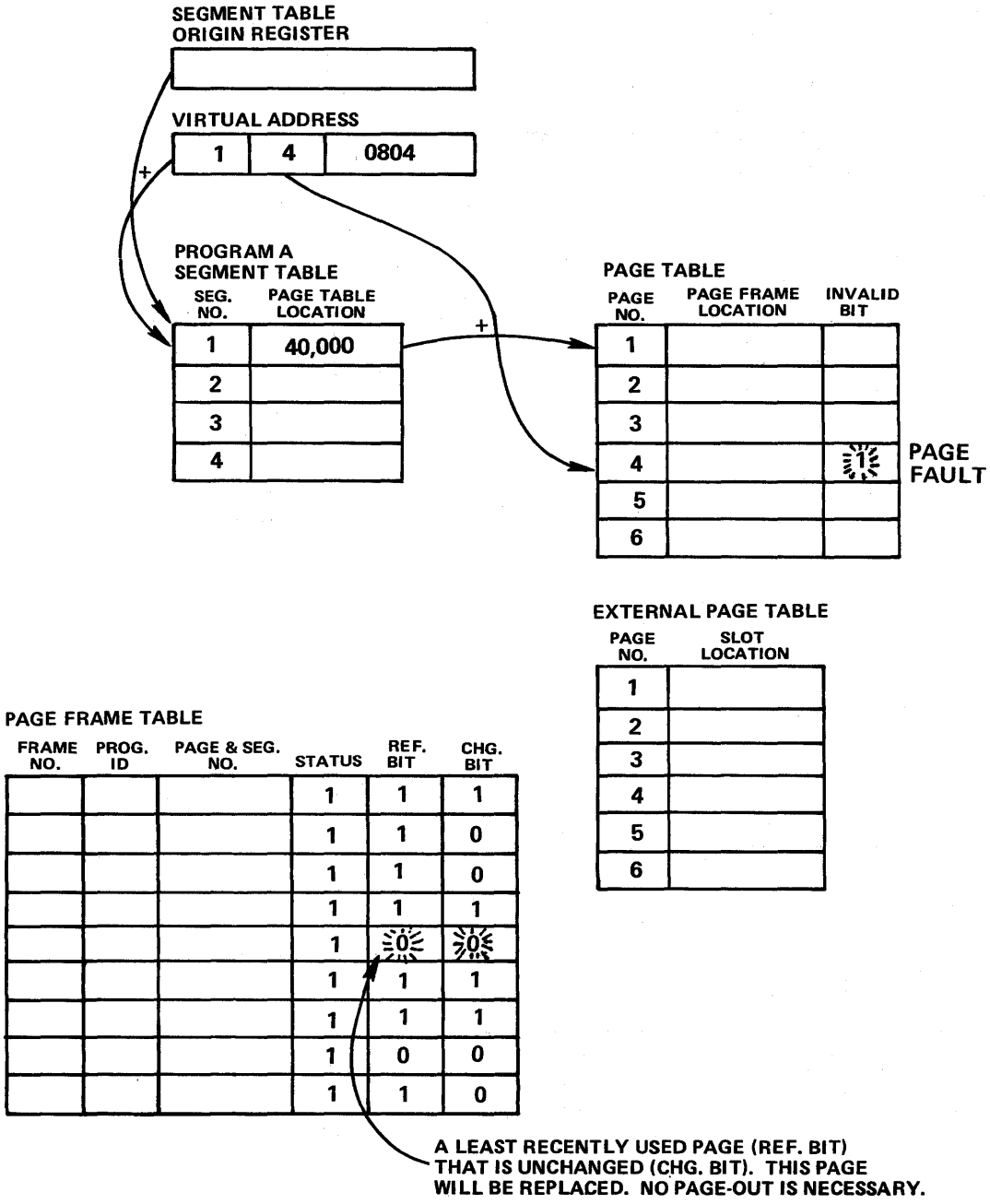


Figure 51

replacement. As we have stated in this example, the contents of the page are unchanged since page-in time. This is indicated by the change bit; it is turned off (a zero). In this situation, there is a duplicate copy of the page to be replaced on external page storage and no requirement for a page-out operation. The system only needs to turn on the invalid bit in the replaced page's page table entry. It may then page in the page that caused the page fault, update the appropriate tables and resume program execution. Figure 51 indicates the highlights of this operation.

In our second example of a page replacement condition, the page to be replaced has been changed since it was paged in. The system must save a copy of the page by means of a page-out operation. Otherwise, if the replaced page is again referenced the old copy will be paged in and it will not reflect the latest state of the program. The page-out operation will transfer the page from its page frame in real storage to a slot in external page storage. The slot selected need not be the one that contains the old copy of the page. The system only needs to update the external page table entry to designate the new slot location. After the page-out operation, the page that caused the page fault will be paged

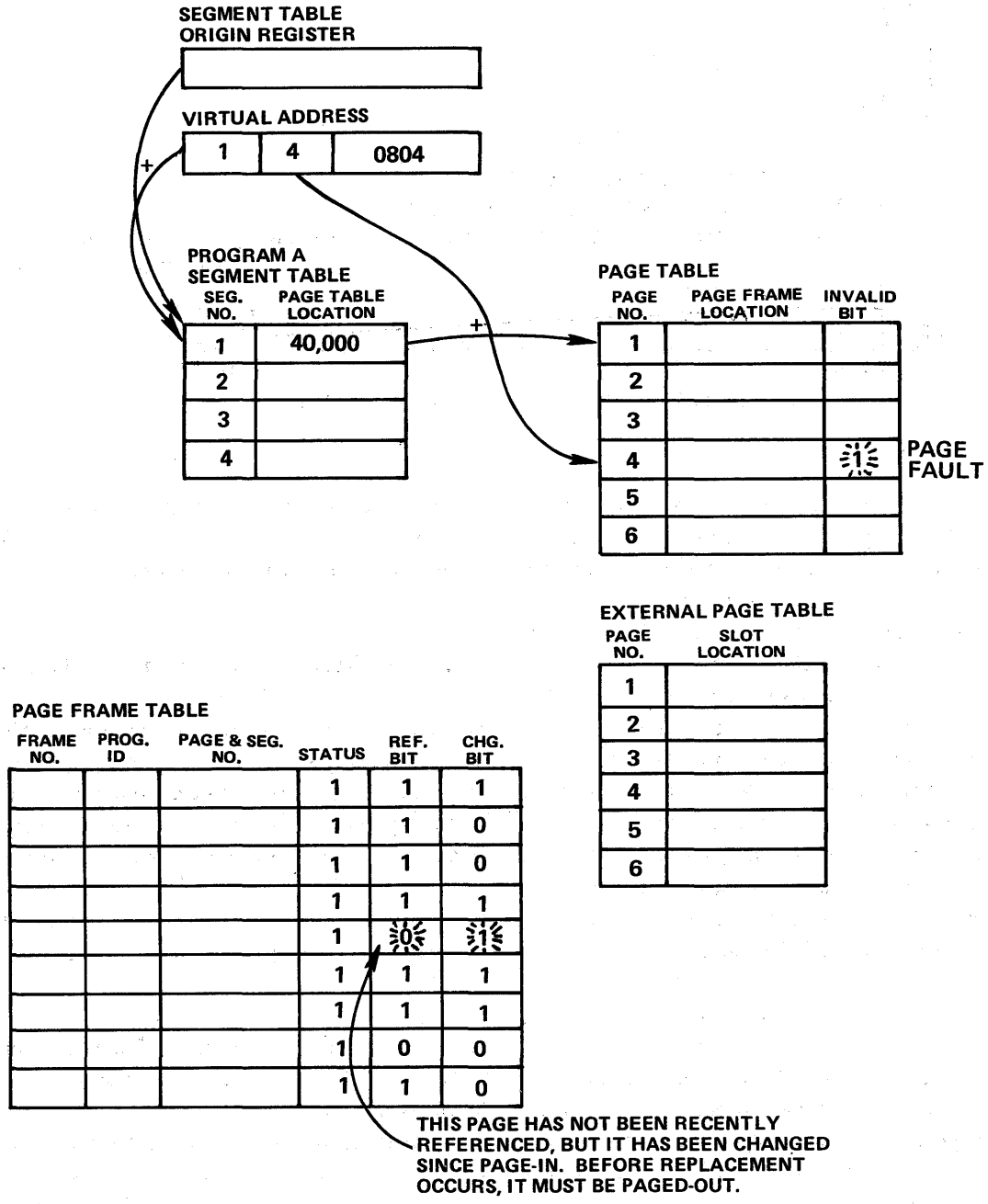


Figure 52

in, appropriate tables will be updated and program execution will resume. This procedure is generally outlined in Figure 52. This "page in – page out" activity is commonly referred to as *paging*. Because pages are loaded only when required, the activity is called *demand paging*. Through this technique a virtual storage system is able to dynamically share real storage among many users.

Locality of Reference

When programs are written they must include all functions (whether common routines, one-time routines or exception routines) required to solve an application. When programs execute, they perform only one operation at a time. In static relocation systems, in principle, entire programs must be loaded into real storage before execution begins. During execution, some parts of the program may never be executed, for example, an exception routine not encountered during a particular execution. Some code may be executed only once or twice such as an initialization routine. What

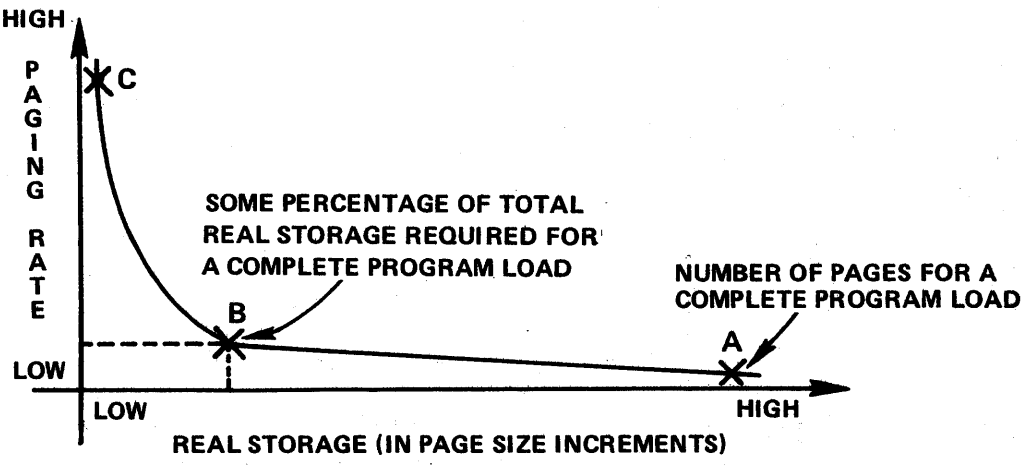


Figure 53

code is executed depends entirely on the reference pattern of a program, which may actually be different every time a program is run. Some studies of program reference patterns have been made to determine how much real storage is required – in page size increments – to give the optimum combination of program execution time and program storage requirements. Both commercial and scientific applications have been used. The results generally produce a performance curve such as the one in Figure 53.

Location A on the graph indicates the condition in which pages are never paged out after they are loaded during program execution. As you might expect, this gives the minimum program execution time, or run time, since there are no delays due to paging, except for page-in activity. Location B on the graph indicates a demand paging execution with "page in – page out" activity. Less real storage was available but with only a small increase in the paging rate and therefore program run time. As real storage size is made even smaller than at Location B the paging rate increases rapidly resulting in high paging rates and large increases in program run time. This is indicated as location C in Figure 53. These characteristics of program performance are due to a phenomenon called *locality of reference*. That is, during a program's execution, its reference pattern will dwell within a relatively small number of pages (compared to the total in a program) for relatively long time periods. The number and combination of pages needed for satisfactory performance (low paging rate) during a given time period is called a program's *working set*. The working set size and its contents may change during execution, but it is usually smaller than a program's total size. This is one big reason why virtual storage systems work to a user's advantage. Programs use only the real storage that they require during execution. More programs might be in virtual storage than could possibly fit into real storage with little system degradation. In fact, as a special precaution, some virtual storage systems have the ability to monitor and control the paging rate. We will return to the monitoring techniques in the following topic.

A Single Virtual Storage Operating System

We have examined translation hardware, tables, registers and techniques used in a virtual storage system. Let's look at a conceptual virtual storage operating system. We will present, briefly, the structure of a virtual storage operating system with a single virtual storage, the system's address space.

Figure 54 shows a schematic view of a single virtual storage operating system. Although external page storage is not shown in Figure 54, enough would be required to back the virtual storage size being used. We show the system in a state where several jobs are active. Take a moment, examine Figure 54, and we will discuss the structure of the system in its virtual storage.

A single virtual storage operating system is structured in virtual storage (the system's address space) just as a static relocation system like OS/MFT or OS/MVT is structured in real storage. In MFT and MVT, the nucleus and the problem program regions or partitions are structured or layed out in real storage during operation. In a virtual storage system, the nucleus, problem program areas – whether they are called regions, partitions or whatever – and all other *resident* functions are structured in virtual storage. Real storage is dynamically shared and managed by the virtual storage operating system. As you will see, some resident portions of virtual storage may also be fixed in real storage. However, in a single virtual storage system, the logical representation of the entire structured system exists only in its virtual storage, and it is exactly analogus to the representation of a conventional system like MFT or MVT in real storage.

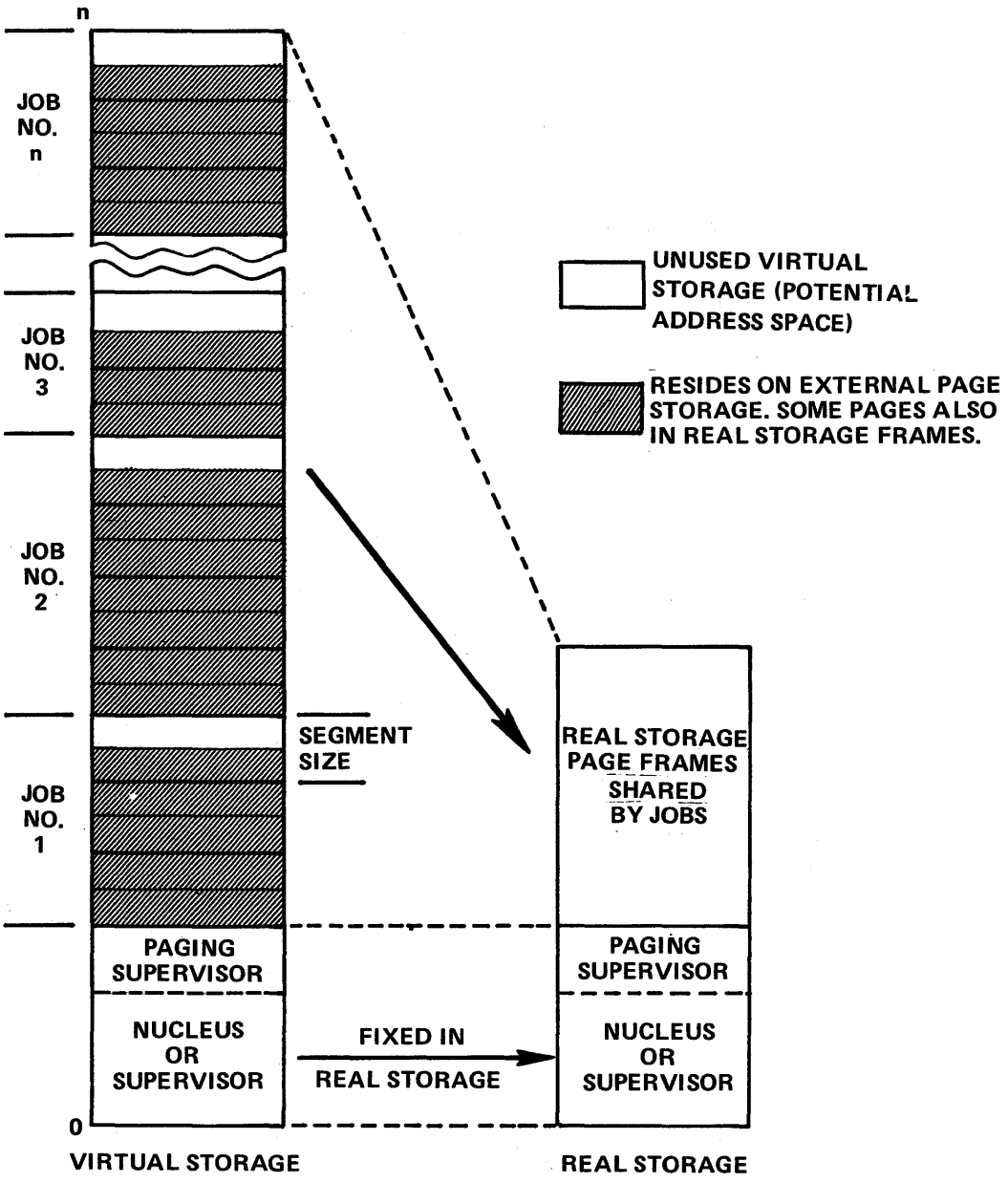


Figure 54

In Figure 54, our conceptual virtual storage system contains a *nucleus,* sometimes called a control program, that controls the use and allocation of all system resources. The nucleus is mapped into virtual storage beginning at the origin (location zero) of virtual storage – in its first segment. Pages of the nucleus are loaded into the corresponding real storage page frames and they remain there while the system is running. These pages are *fixed* in real storage; they can't be paged. This is an example of a resident part of virtual storage that is also fixed in real storage. Page fixing isn't necessary for the entire supervisor. It is usually done for those parts required for basic control of the system and quick response to service jobs in the system. In many systems, the remaining control program functions that are resident in virtual storage are paged.

In virtual storage systems both hardware and software play an important role. Virtual addresses are automatically translated by dynamic address translation hardware. Software maintains the tables used, controls the allocation of real, virtual and external page storage and controls paging between these two levels of storage. These software functions are contained in what is usually called the *paging supervisor.* The paging supervisor is a component of our conceptual system's nucleus. It is used to control the system's virtual and real storage. Because it is part of the nucleus, our paging supervisor is also fixed in real storage.

Our system's control program might contain several other subsystems – such as an input/output control system,

a job scheduling system and so forth – that are not indicated in Figure 54. These subsystems would be paged whenever possible.

Real storage, then, may be split into fixed and shared page frames. The shared page frames are shared by active jobs in the system using demand paging. As indicated in Figure 54, jobs are allocated virtual storage in segment size increments. Thus, our system may control virtual storage allocation through its segment table. Figure 54 shows several active jobs and some segments may be only partly loaded. The pages not used in these segments are potential virtual storage or address space for the job that owns the segment. If never used, these unused pages never use any external page storage. All unallocated segments are potential virtual storage or address space controlled by the system. If the system schedules a new job it will allocate segments from this potential address space. If all segments are allocated we are out of virtual storage. In this situation, no new jobs can be scheduled until segments become free when one or more jobs terminate. Segments allocated to a job must be contiguous. If a job needs three segments the system can't assign SEGMENT 12, SEGMENT 14 and SEGMENT 15. In this example, the system would have to assign SEGMENT 12, SEGMENT 13 and SEGMENT 14.

Program Loading in a Single Virtual Storage System

How is a job loaded into virtual storage? We will describe one technique. In our conceptual system, user programs are stored in a library in auxiliary storage. The address space of each program relates to a zero origin. The address space format need not be in a segment-page structure. In our example we assume that programs stored in program libraries use a different format. Perhaps they are stored in partitioned data sets.

The most important objective during program loading is to relocate the program's addresses to relate to their origin in a system's storage. In a system like OS, during loading the system relocates a program's addresses to a starting point in real storage. In our single virtual storage system, the loader program relocates a program's addresses to an origin in virtual storage. In both cases, a static relocation operation occurs. In our example, program loading will occur before execution begins. Prior to execution, we will statically relocate the program to a virtual storage origin address, and this origin address will be the beginning of a segment. Remember that during execution the DAT feature will dynamically translate virtual storage addresses that are in real storage locations, and the operating system will share real storage among all active programs using demand paging.

What happens during program loading? A system program, that we will call the loader program, is executing. See Figure 55. The program to be loaded is in our program library in auxiliary storage. We will call it Program B. The loader program will read Program B from its library as input data. The loader program will relocate Program B's addresses to their origin in virtual storage. During loading, Program B will be formatted into pages. Most likely, during loading most of Program B's pages will be paged out to external page storage as shown in Figure 55. Program B's page tables and external page tables will be built during the

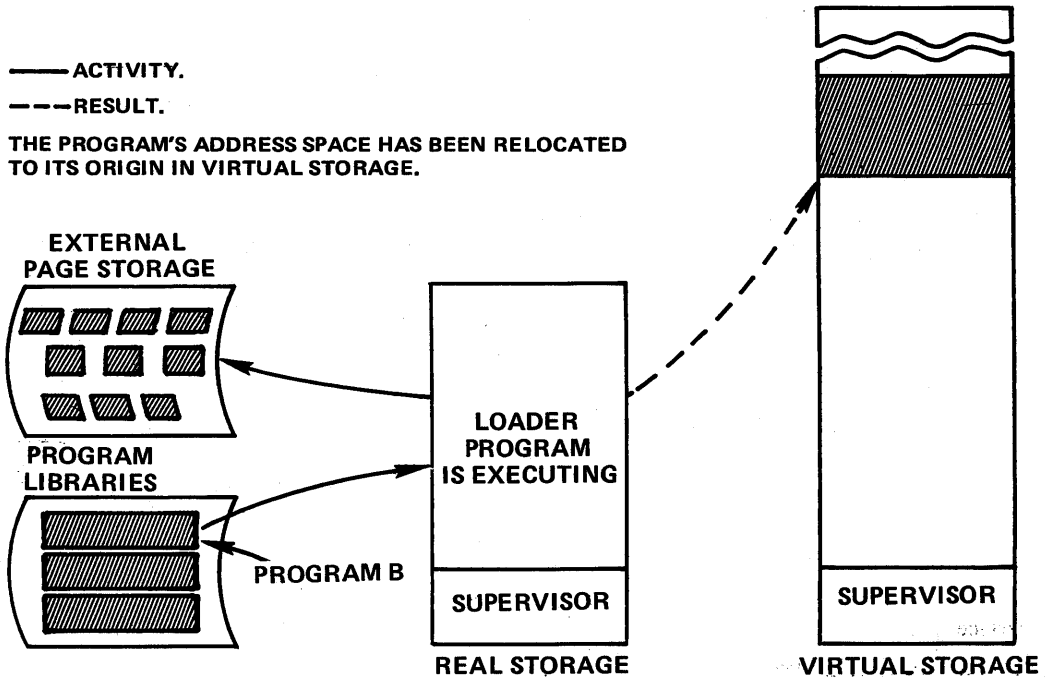


Figure 55

loading process. At the end of program loading, the loader program will transfer control to Program B and it will begin to execute. Through demand paging Program B's working set will eventually become resident in real storage. Thus, when a job is scheduled in our virtual storage system, and its programs loaded, a loading procedure like the one just described will occur in the system.

Through demand paging, the system shares real storage among several jobs. Because programs require only a working set of pages in real storage for satisfactory execution, allocated virtual storage may be larger than the system's real storage. It may be possible to multiprogram more jobs in the system and the system's effective throughput might improve.

Thrashing Monitors

We have just described the ideal situation. Active programs in virtual storage are larger than real storage; but there are enough frames of real storage to satisfy the working set requirements of the active programs. There are, however, two key questions:

1. How much paging activity can you have in the system?
2. Can you control paging activity in the system?

We have already answered the first question. Paging activity can increase so long as there are enough real storage page frames to satisfy the total working set requirements of all active programs. As soon as the system passes this point, the paging rate begins to increase and performance becomes degraded. If the paging rate becomes too high, the system won't accomplish any useful work. It will spend all of its time paging. This condition is called *thrashing.* Even though we have been discussing single virtual storage systems, thrashing can also occur in multiple virtual storage systems. The operating system can't predict the degree of paging activity that will occur, but it can control paging activity and prevent thrashing. This answers our second key question. Because the paging supervisor transfers pages between real and external page storage, it can monitor the paging rate. If the paging rate becomes too high, the system can:

1. interrupt and temporarily halt an active job.
2. free the real storage frames being used by the job.

The active jobs will then, through demand paging, begin to use these frames and the paging rate will decline. If the paging rate is still too high, another job could be interrupted and temporarily halted. As system activity declines an interrupted job is reactivated and its execution resumes. In this way the system can dynamically adjust its allocation of resources among users and assure satisfactory performance.

Even though virtual storage is not unlimited, a virtual storage system gives better service to its users. Through dynamic allocation, dynamic address translation and dynamic adjustments it produces better system resource management.

Multiple Virtual Storage Systems

System Structure

To go from a single virtual storage system to a system that supports multiple virtual storages requires only a small increase in software technology beyond what has already been presented. A single virtual storage requires one segment table with page tables to map its allocated segments. These tables reside in real storage. To create another address space, or virtual storage, simply requires another segment table in real storage along with page tables to map any of its allocated segments. In this manner, multiple address spaces, or virtual storages, can be created within a system, each virtual storage being mapped by a segment table. Rather than have all users in a system – batch jobs, TP applications, timesharing users and so forth – share one address space using a region or partition strategy, a multiple virtual storage system gives each user his own address space.

With this approach, a user may execute only after the system's supervisor or nucleus directs the Dynamic Address Translation (DAT) facility to the user's address space. This is accomplished with the Segment Table Origin Register (STOR register described in Lesson 3). Before a user executes, the supervisor loads the STOR register with the real storage origin of the user's segment table. During execution, the DAT facility will translate the virtual addresses in the user's address space using the segment and page tables that map it.

Mapping multiple address spaces and directing the DAT feature to the appropriate address space (through the STOR register) is the essential difference in the operation of a multiple virtual storage system. Demand paging remains the same. Page faults are signaled through the invalid bit in page table entries. The operating system must then select the "best" page frame for a page-in operation. External page storage is used in the same manner. Allocated pages must be backed by slots of external page storage. Page locations in external page storage are identified in external page tables, one for each page table.

Considering the small change in operation required to implement a multiple virtual storage system, the major difference between single and multiple virtual storage systems is the resources used by the system. Most likely, with multiple virtual storages more real storage will be used for

segment and page tables and more external page storage will be required to back each virtual storage. Before a system can create a new address space, it must be able to back the requirements of the address space with external page storage. It will be shown later that the additional system requirements in a multiple virtual storage system are more than compensated for by the functions that result, especially in a timesharing, teleprocessing environment.

Our discussion of multiple virtual storage systems thus far has extended the technique of segmentation and paging. The advantages that result from this combination also extend into a multiple virtual storage environment:

- Paging remains an efficient way to manage real storage.
- Segmentation allows convenient control and allocation of virtual storage (now, within each user's address space).

When a system supports multiple address spaces a new important benefit also results from segmentation: segments can be *shared* among two or more virtual storages. We will describe a technique for *segment sharing* later. Figure 56 is a schematic view of a multiple virtual storage system as we have described it until now showing three virtual storages. 24-bit addressing is assumed. As a result, each address space is shown to be the maximum size, 16 megabytes. Depending on a system's virtual storage implementation, users may have different size address spaces. Each address space would, however, usually be a multiple of segments. As shown in Figure 56 we continue to assume that segments

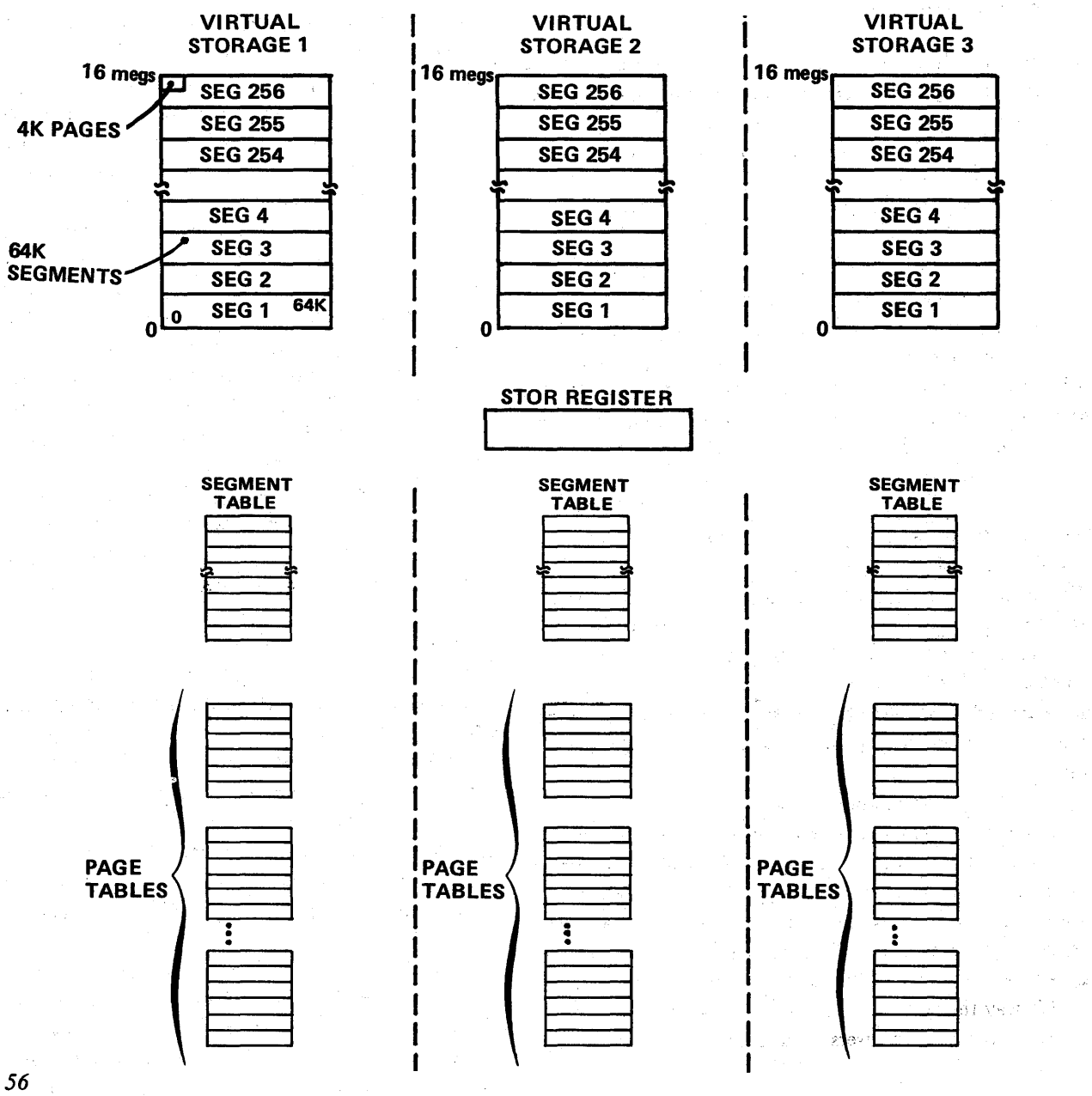


Figure 56

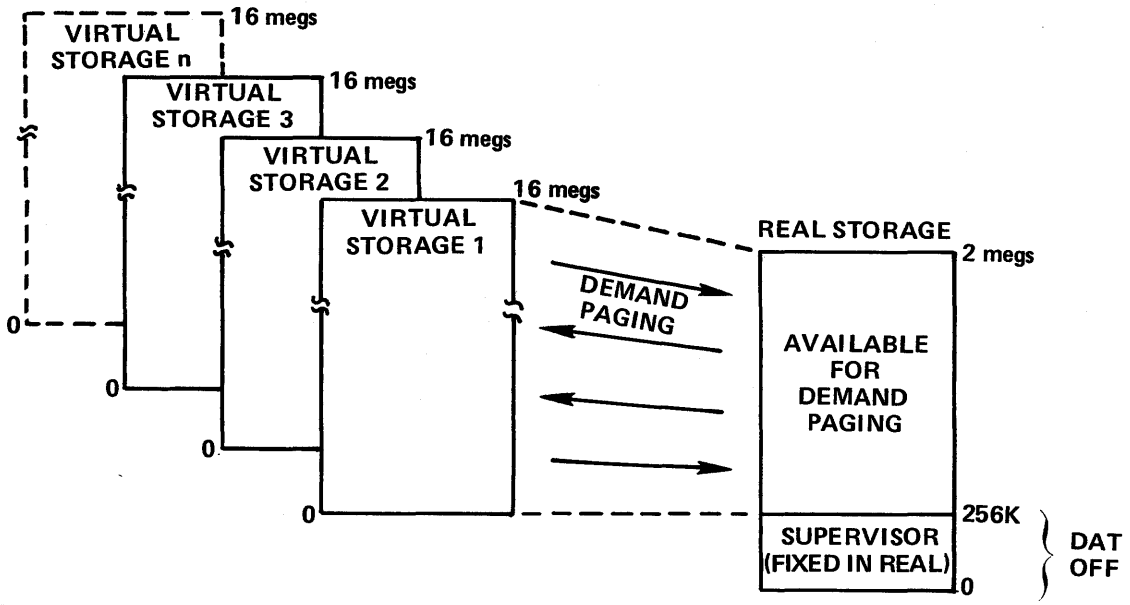


Figure 57

are 64K and pages 4K in size. This results in 256 segments in 16 megabytes of virtual storage, with 16 pages in a segment. The segment and page tables that map each address space are also represented in Figure 56. During system operation, these tables reside in real storage. The STOR register points to the segment table of the user that is currently executing.

There are various techniques to implement multiple virtual storages in an operating system. We will discuss two variations:

- In the first, the system's supervisor (or nucleus) is fixed in real storage during operation, but is not mapped in each user's address space.
- In the second variation, the system's supervisor is fixed in real storage during operation and is also mapped in each user's address space.

The first approach is shown in Figure 57. The supervisor is fixed in real storage and the remaining real storage is controlled and allocated through demand paging. External page storage, though not shown in Figure 57, must be available to back the allocated pages from each virtual storage. Notice in Figure 57 that the DAT feature is "turned off" during supervisor execution. This is because the supervisor is not mapped in any user's address space. No page tables map it. When references are made within the supervisor, the effective address generated is used by the CPU to reference real storage. Thus, there is no dynamic address translation during supervisor execution.

When control is transferred to a user and his address space, the DAT feature must be "turned on". Addresses referenced within a user's address space are virtual addresses. They must be translated (by the DAT feature) through the segment and page tables that map the address space. A system that uses this implementation technique must have some mechanism then to turn the DAT feature "on and off". When supervisor services are requested by a user, DAT is "turned off". When the supervisor passes control to a user, DAT is "turned on".

With this type of scheme, each user is allocated space within his address space starting with the first segment (location zero). Each user, therefore, whether a batch job, TP job or timesharing user, has a few more segments available than he would have in a system that maps the supervisor in each virtual storage. However, the system must have a technique to turn the DAT feature on and off as required.

In our second variation of a multiple virtual storage system the DAT feature is always turned on. This results when the system's supervisor is mapped in all virtual storages as illustrated in Figure 58. To understand how this works, consider for a moment the following example.

A program within a user's address space is executing. All addresses referenced, therefore, are translated by the DAT feature. A page fault occurs. Control will now be passed to the supervisor so that a page-in operation may be initiated. Because the supervisor is mapped in each address space, the DAT feature will translate all supervisor program references using the segment and page tables that map the current address space, that is, the segment table currently identified by the STOR register. During supervisor execution, no page faults will occur since the supervisor is fixed in real storage. After a page-in operation is started the supervisor may pass control to another user. When it does, the supervisor will first load the segment table origin of the new user's address space (which also maps the supervisor) into the STOR register. Thus, anytime control is passed to the supervisor, it will be mapped by the segment table of the current address space. The DAT feature will always be "turned on".

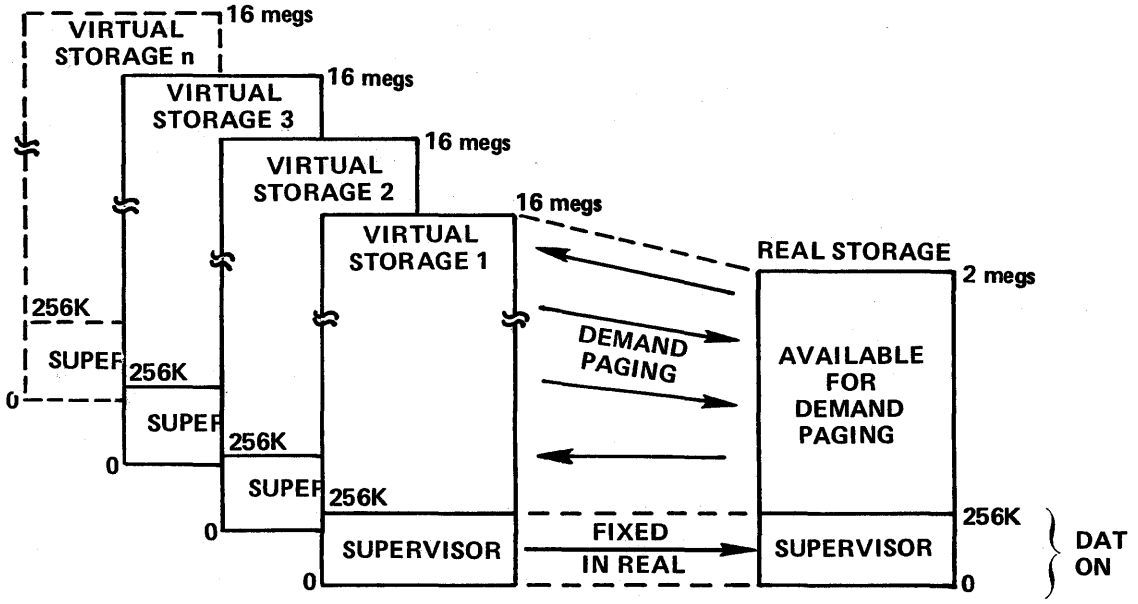


Figure 58

Even though the supervisor is mapped in each user's virtual storage, there is only one copy of the supervisor that is fixed in real storage. This is made possible through segment sharing. Remember, for every allocated segment, the segment table entry points to the real storage origin of the segment's page table. This page table identifies all of the pages within the segment, indicating whether the page is in real storage. Assume for a moment that a multiple virtual storage system has been initialized; the supervisor has been fixed in real storage; and one virtual storage (that maps the supervisor) has been created to process user jobs. If the supervisor is 256K, the first four segments of the initialized address space are used to map the supervisor. The segment table entries of these segments identify the page tables that map the page frame locations of each page in the supervisor. If a second virtual storage is created, it also must map the supervisor. However, to do so its first four segment table entries need only point to the existing page tables that already map the supervisor. The first four segment table entires of any subsequent virtual storage created also need only point to these same page tables to map the supervisor. Thus, the supervisor is shared among all virtual storages in the system with one copy of the supervisor and one set of *common page tables* that map the supervisor. Each virtual storage identifies these common page tables in its initial segment table entries.

If the supervisor can be shared in this manner, then so can other system programs, as long as they are reentrant. For example, a system's data management access methods can be loaded into specific segments of the initial virtual storage and then shared by all subsequent users simply by having the appropriate segment table entries identify the common page tables that map the access methods. The access methods can then be demand paged and shared by all users in the system. The only qualification is that any shared system programs be reentrant. In a similar manner, user-written programs might be shared among all users in the system and, if allowed by the system, among two or more users in a system.

Segment sharing is a powerful tool in a multiple virtual storage system. It must be remembered, however, that all programs shared on a system-wide basis reduce the amount of address space available to each user in his own virtual storage. For this reason, only programs applicable or useful to all or most users in a system should be shared using segment sharing.

System Operation

The implementation of a multiple virtual storage system through multiple sets of segment and page tables and demand paging is not too difficult to see. How such a system operates or function, however, is sometimes more difficult to envision. Several alternatives exist. We will describe one possibility using a conceptual system and demonstrate its operation. We assume that the supervisor is mapped in each virtual storage that is created.

To begin operation, an operator must initialize the system. An initial program loader function (started by the operator) must exist that creates the first virtual storage in the system, loads and fixes the supervisor in real storage, and maps the supervisor in the initial virtual storage. When loading is complete, the operator must then have some means of further communicating with the system to start batch job operations, time sharing operations and so forth.

A communications program could be within the supervisor but this would be fixed in real storage. If the remainder of the initial virtual storage is used for a communications program, then the communications function will be demand paged.

Once the communications function is active, the operator may begin batch job processing by starting a job scheduler. The result would be the creation of another address space that would map the supervisor in its lower segments. The supervisor can then load a scheduler in the new address space to select jobs for processing based on the system's job scheduling algorithm. When a job is scheduled, it will overlay the scheduler within the address space and the job will control the address space during job processing. At job termination, a fresh copy of the scheduler would be loaded into the address space to schedule the next job. With one scheduler active, only one job stream is processed. However, with each new scheduler started by the operator a new virtual storage is created and the degree of multiprogramming is increased. Protection is inherent in the design of this type of system. Each job scheduled executes in its own address space. There is no way that a job can alter the address space of another user.

If timesharing is implemented in our multiple virtual storage system, each user to log on will also receive his own address space. The timesharing user's address space will also map the supervisor, leaving the remainder of virtual storage under control of the user. Likewise, a teleprocessing application would also control its own address space. In all cases, the supervisor will be mapped into the lower segments of each virtual storage. If any system programs are shared system-wide through a segment sharing technique, they also will be mapped into common segments of each virtual storage.

In a theoretical sense, the degree of multiprogramming or the number of timesharing users is almost unlimited in a multiple virtual storage system. However, there are practical limits. There must be enough real storage to support the paging activity in the system. External page storage is required to back the active pages of all the virtual storages that have been created in the system. The ideal situation exists when the system paging rate does not overcome the system's ability to perform useful work. If this occurs then the system is thrashing as described for a single virtual storage system. Thrashing may be prevented in a multiple virtual storage system by deactivating a user (or users) until a satisfactory paging rate is attained. When a user is deactivated, the pages in the user's virtual storage that are resident in real storage (and that have been changed since page in) are transferred to external page storage. When the user is later reactivated, these same pages will be transferred to real storage and the user can resume execution. This transfer of multiple pages is called *block paging* (as compared to demand paging which results in the transfer of one page at a time).

Block paging can be used to prevent thrashing and it can also be used effectively as a timesharing technique in a multiple virtual storage system. Rather than allow a system's paging capacity to become saturated as more and more timesharing users log on (and then become forced to deactivate users), a multiple virtual storage system can schedule the deactivation and reactivation of timesharing users using the block paging technique. When activated, a user can contend for the CPU with all other active users for a certain time period. During that period, when a user gets the CPU his program(s) will execute under the control of demand paging. At the end of his time period the user would be block paged out. When again scheduled (or reactivated) the same set of pages would be block paged in. After a period of execution, the set of pages being block paged in and out would tend to become the user's working set. Both timesharing and batch job users can be controlled in this manner. This block paging technique may also be thought of as swapping users between real and external page storage on a scheduled basis.

A multiple virtual storage system makes even more distinct the two levels of control – the system level and the user level – within an operating system. On the system level, the supervisor must control real storage, auxiliary storage (including external page storage), the system's CPU, address space creation, and so forth. The system level of control may be thought of as *global control.* The supervisor is responsible for all resources that affect the overall performance of the system. The user level of control in a multiple virtual storage system relates to the user's *local control* of the address space. The user may allocate space to programs and the user may not reference outside of his address space. These two distinct levels of control improve the reliability of a multiple virtual storage system.

Virtual Machines

With multiple virtual storages, each user is given a large address space that has the characteristics of real storage. This is possible because of the DAT function, a mapping scheme to describe each user's address space, and a technique to control the allocation of the real storage resources such as demand paging.

A user can control his address space in a manner somewhat analogous to the way an operating system controls address space allocation in a single virtual storage system. A user may start multiple independent tasks, allocating the virtual storage space required for each task. Dispatching or

execution priority can be assigned by the user to each task as it is started. During execution, a user can continue to control his tasks and change dispatching priority if necessary. This, however, is where the analogy to an operating system ends.

Besides allocating the system's address space, an operating system controls the use of a system's CPU, real storage, and channels and the allocation of its I/O devices. An operating system performs user services, such as I/O operations, usually through some type of interrupt scheme and it uses the system's console for operator communication. An operating system, then, controls all of a machine's resources, not just the system's address space.

A common technique used to implement operating system control of a system's resources is through a privileged instruction set. A set of machine instructions – such as START I/O – is reserved for use of the operating system's supervisor only. For a user to perform I/O, for example, a service request is made to the supervisor to start the I/O operation. This gives the supervisor complete control of channel operations. Through privileged instructions, the supervisor can also control the CPU and other key system hardware resources. If a user (or non-privileged program) tries to execute a privileged instruction, the program will be interrupted and normally the supervisor will terminate its execution.

If it is possible for a control program or supervisor to give each user his own address space (virtual storage) while still controlling the real storage resource, it seems possible to give each user a virtual CPU, virtual channels, virtual I/O devices, in effect a virtual machine, while the control program still controls all real system resources. This in fact is what is done in the *Virtual Machine Facility/370* (VM/370) system. With VM/370, each user is assigned his own virtual machine, while VM/370 controls all of the System/370's real resources, its CPU, real storage, channels, I/O devices and so forth. At this point we will briefly depart from our presentation of concepts only and discuss the VM/370 system.

VM/370

VM/370 is basically a timesharing system. However, unlike other timesharing systems, when a VM/370 user logs on, the user has a virtual machine at his disposal. The configuration of the virtual machine is usually determined when a user signs up for VM/370 service. A basic virtual machine configuration would consist of a CPU (the same model as the real System/370 that is used for VM/370), a virtual storage, where virtual storage size is usually some multiple of 64K, a virtual card reader, card punch and printer, a virtual console and several virtual disk storage units.

These various components in the virtual machine configuration are "virtual" in different aspects:

1. The user's terminal becomes his virtual machine console. The console may be used to IPL, to display and/or alter the contents of virtual storage and as a means of operator communication if an operating system is executing in the virtual machine.
2. Because card readers, card punches and printers are sequential in nature, direct access storage is used to represent these devices.
3. A user's disk storage drives are "logical" disk drives whose physical addresses can be different than those assigned to the virtual machine. Therefore, several users may have disk drives assigned as 190. The control program identifies the user's logical address with its physical location. Also, disk drives need not be full capacity. In VM/370, it is possible to allocate *minidisks* to virtual machines. For example, a minidisk may be 10 cylinders in capacity with all the remaining characteristics of a 2314.
4. Each user's virtual storage is an address space that is usually some multiple of 64K in size. The VM/370 control program maps each address space with a segment table, page tables and external page tables. During system operation, real storage is shared through demand paging.
5. Each virtual CPU is the same model System/370 that is under the control of VM/370. VM/370 shares the real CPU among all the active virtual machines to give each the appearance of having its own CPU.

After a VM/370 user logs on, he then *IPL's* the system to be used in his virtual machine. The Initial Program Load (IPL) Function is a special hardware function on a real System/370. VM/370 simulates this function (after the IPL command is issued from the user's terminal) to load the system requested by the user into his virtual machine. If a user requires interactive computing and program development, he can load VM/370's *Conversational Monitor System (CMS)* into his virtual machine. CMS provides services such as programming languages like FORTRAN and COBOL. However, because each user has a virtual machine, any program or operating system that executes on a real System/370 may also be executed in the user's virtual System/370 (provided the virtual machine has the required configuration). Thus, a VM/370 user can IPL the DOS or OS system and control its operation from the terminal. In this case the user's terminal becomes the operating system's console for operator communication. This is possible even though the operating system will require the System/370 privileged instructions.

All of the virtual machine functions described so far are possible because of the VM/370 control program. The VM/370 control program manages System/370 real system resources – the CPU, real storage, channels, I/O devices and so forth – in such a way that each virtual machine functions as a real System/370. The control program shares the real CPU among all virtual CPU's using a time-slicing technique. When a virtual machine has the real CPU it may schedule it using its own dispatching (or scheduling) algorithm. Each virtual machine has its own address space. The VM/370 control program maps each address space and shares real storage among all virtual machines using demand paging. In an analogous manner, VM/370 maps I/O devices to virtual machines. All I/O operations are controlled by VM/370, even though an operating system executing in a virtual machine may "think" that it is controlling its own I/O. This is accomplished because only VM/370's control program may execute the System/370 privileged instructions. Only VM/370's control program processes real interrupts. When an interrupt occurs in a virtual machine, it is a virtual interrupt. VM/370 has a mechanism to direct the system so that virtual interrupts will be processed by the proper interrupt handlers within the virtual machine.

When a virtual machine attempts to execute a privileged instruction, VM/370 traps the instruction, performs the required function, and then returns control to the virtual machine. Thus, operating systems executing in a virtual machine are in a *pseudo-privileged state.* The VM/370 control program has the proper mechanisms to allow virtual machines to operate as though they are in the privileged state. Only the VM/37C control program executes in the privileged mode. Using this approach, then, only the VM/370 control program manages the real system resources.

This introduction to virtual machines was included to show you how a virtual machine system extends the functions of a multiple virtual storage system. We introduced VM/370 to show you an example of a virtual machine system, but not to describe in detail any of the functions in VM/370. For more information on VM/370, you may refer to *IBM Virtual Machine Facility/370 (VM/370): Introduction,* Form GC20-1800.

System/370 Virtual Storage Support

Until now our presentation has been primarily on the concepts of virtual storage using examples of conceptual software systems. We have described hardware such as the 24-bit address structure, the DAT function, the segment table origin register, associative array registers, change bits and reference bits and we have used all of this hardware in our conceptual virtual storage operating system. We have mentioned the IBM System/370 only in examples. Most models of System/370 have a DAT function and the other hardware necessary to support a virtual storage operating system. The DAT function supports virtual storage using a combination of segmentation and paging. Thus, virtual storage can be managed and protected through segmentation; and real storage is managed through paging. System/370 has a segment table origin register (STOR register). It points to the real storage origin of the segment table. Smaller models of System/370 with the DAT function use eight associative array registers for fast virtual address translation. Larger System/370 models have a different device called the *translation look aside buffer* that performs a function similar to associative array registers for fast translation. Reference and change bits are in the storage protect key associated with each 2K real storage block in System/370. The hardware automatically turns on the reference and change bits when the contents of a page frame are referenced or altered. System/370 has a 24-bit address structure. Thus, maximum virtual storage size is 16,777,216 (16,384K) bytes (16 megabytes).

In System/370, the DAT function can be turned on or off. When the DAT function is off, no address translation occurs during program execution. Systems like OS/MVT or OS/MFT will run just as though no DAT function were in the System/370. With the DAT function off, during execution System/370 will generate effective addresses (from base displacement addresses) that reference real storage locations.

What happens that is different in System/370 when the DAT function is on? During program execution the system's CPU takes a base displacement address and generates an effective address. This effective address (24 bits long) is the virtual address that is translated by the DAT function. The DAT function uses the first 8 bits as the segment number, the next 4 bits as page number and the last 12 bits for displacement when the system uses 4K pages. This results in 64K segments and 4K pages with 16 pages in each segment. If 2K pages are used, the first 8 bits of the virtual address – effective address – identify segment number, the next 5 bits page number and the last 11 bits displacement. This results in segments of 64K and pages of 2K, with 32 pages in each segment. Thus, it is the way that the DAT function "looks at" virtual addresses that determines the structure of virtual storage. A special purpose control register is used to specify whether the system is using 2K or 4K pages. This register also controls segment size. In System/370 segments may be 64K or 1024K. During address translation the DAT function references segment and page tables. The tables must be in real storage. Fast translation is performed by associ-

ative array registers or the translation look aside buffer.

Because System/370 starts with a base displacement address, generates an effective address (virtual address) and then translates the virtual address, there is no change in how a programmer will code programs. When coding in assembler language a programmer doesn't need to do anything different than for a conventional System/370. Likewise, compilers for high-level languages can generate object code just as they do for conventional System/370s. Programs may execute in an area of virtual storage well beyond the end of real storage, but System/370 will translate the effective addresses (virtual addresses) that reference these locations. For these reasons we say that dynamic address translation is "transparent" to the user.

In this section we have described the hardware in System/370 used to implement virtual storage. In PART II of this text we will describe the software – operating systems – that support virtual storage in System/370.

Lesson 6 Test Questions

1. A virtual storage system senses a page fault during address translation through a page table entry's
   A. fault bit
   B. invalid bit
   C. reference bit
   D. change bit

2. After a page fault occurs a virtual storage system must perform a (an)
   A. page in
   B. page out
   C. external page
   D. page name

3. Before a page-in operation, a virtual storage system locates the page that will be paged in through its
   A. page table
   B. external page table
   C. segment table
   D. look aside buffer

4. In a virtual storage system, each page table has a corresponding
   A. segment table
   B. associative array register
   C. external page table
   D. translation look aside buffer

5. The external page table identifies the location of a page in
   A. auxiliary page storage
   B. real storage
   C. buffer page storage
   D. external page storage

6. The technique of loading a page into real storage only when referenced is called
   A. demand paging
   B. reference loading
   C. reference paging
   D. transfer paging

7. The page replacement rule considered as the best that can be used in virtual storage systems is
   A. LRU
   B. FIFO
   C. LIFO
   D. FM

8. During a page replacement operation a page out is first required only if the
   A. invalid bit is on
   B. reference bit is on
   C. change bit is on
   D. protection bit is on

9. In the conceptual virtual storage system described in this lesson programs are loaded into virtual storage using
   A. simple program loading
   B. static relocation
   C. dynamic relocation

10. System/370 with a 24-bit address structure supports a maximum virtual storage size of
    A. 8 million bytes
    B. 12 million bytes
    C. 16 million bytes
    D. 32 million bytes

11. The System/370 DAT function supports a virtual storage with
    A. paging only
    B. segmentation only
    C. segmentation and paging

12. In System/370 the DAT function may be turned on or off. (True/False)

13. System/370 supports a virtual storage with a page size of
    A. 1K or 3K
    B. 2K or 4K
    C. 2K or 6K
    D. 4K or 8K

14. System/370 has special devices for fast address translation. On small System/370 models they are called associative array registers; on large System/370 models they are called the translation look aside buffer. (True/False)

We have presented the basic concepts and techniques used to implement virtual storage systems. We have described the concepts, one piece at a time, until the presentation of a virtual storage operating system in the preceding lesson. In this lesson we will summarize the requirements, in hardware and software, needed to implement a virtual storage system. We will then compare these requirements to the results that are derived from implementing the virtual storage concept.

Hardware and Software Requirements for Virtual Storage

A variety of new hardware devices have been introduced for virtual storage systems. The dynamic address translation feature (the DAT feature) automatically translates virtual addresses. Translation begins with the Segment Table Origin Register. To reduce translation time through segment and page tables, associative array registers make the difference in instruction execution 'timings' negligible when compared to translation through the tables.

In addition to these new devices, some conventional system resources are used in a virtual storage system. The segment and page tables reside in real storage. This, in effect, expands the size of the supervisor. In a single virtual storage system the system has only one segment table and each allocated segment one page table. This results in a small additional requirement of real storage. In a multiple virtual storage system this requirement increases somewhat. In either case, it's a variable quantity that depends on the amount of virtual storage in use. Some of the system's auxiliary storage is used for external page storage. The amount of external page storage also depends on the amount of virtual storage in the system. A channel's or at least part of a channel's time will be used for page transfers between external page storage and real storage (paging).

Some new software is required in a virtual storage system. This primarily takes the form of the paging supervisor. Paging, table maintenance and functions such as the page replacement strategy are built into the paging supervisor.

Benefits of Virtual Storage

The benefits and potential that result from these hardware and software requirements are numerous. In fact, they make the virtual storage concept a profitable investment in almost any application environment.

A virtual storage system's *real* storage is used only on demand, when referenced. This results in many benefits:

1. Programs only require their working set for effective execution. This may enable the system to initiate and execute more jobs.
2. Real storage fragmentation is almost eliminated.
3. It's possible to design and implement an application the size of which greatly exceeds real storage size. This would be impossible in a conventional system without the use of overlays, or a multi-step job.
4. Long running applications will use only the amount of real storage required at any point in time. Thus, a dedicated teleprocessing network will use very little real storage when idle.
5. Virtual storage systems might be usable as back-up for systems that have a larger real storage size.
6. In a system with multiple virtual storage support, each user has his own address space.

From the programmer's point of view, virtual storage systems eliminate limitations that exist in static relocation systems. Because virtual storage is much larger than real storage the programmer has much more space to design problem solutions. This greatly reduces, if not completely eliminates, the need to overlay programs. In overlayed programs, the programmer must plan for storage requirements, storage content and branching conventions. Revising a program or application that uses overlays can be a complicated job. In virtual storage systems the *system*, through demand paging, manages real storage. In effect, it "automatically overlays" programs too large for available real storage. The programmer develops his program for execution in a large contiguous space in virtual storage. This saves time and eliminates complicated procedures. It lets the programmer concentrate on the solution to his application.

Because the operating system itself (its supervisor and subsystems) is structured in virtual storage and because large portions are paged, the number of resident functions may be expanded to suit an installation's needs. They will only need real storage when used, but they are always available to be shared by the system and its users. In static relocation systems the number of resident functions used is limited because they require full-time residence in a system's real storage. In general the benefits of a virtual storage system relate to two factors:

1. There is less dependence on the system's configuration and resources, especially real storage size.
2. There are less restrictions on the size and functions of the operating system that controls the system's resources, and on the people – programmers, operators and so forth – who use the system's resources.

Lesson 8. Programming Style in a Virtual Storage System

Methods and techniques for programming in static relocation systems are normally used for any of three reasons:

1. To conserve real storage.
2. To improve execution speed.
3. For programming convenience.

Virtual storage systems operate very differently than static relocation systems. Real storage use is *dynamic,* and a program's working set requirements are largely unpredictable during execution. Although a program's reference pattern is not always predictable, some programming techniques can be used to improve *locality of reference* during execution, to reduce paging and to reduce real storage requirements. These techniques are not complicated. They don't require a lot of time. They are new, however, and somewhat different.

Before discussing any specific techniques let's briefly consider one item. How much thought and time should you give to programming style before you begin coding a program? The answer depends on the expected usage of the program. Will it be used once a day or once a year? When used, will it run for five minutes or five hours? The frequency and length of use are the two important factors in deciding how much time to devote to programming techniques. Programs used in a daily application deserve more attention than a program run once a year. What follows is an introduction to some of the considerations that you should make if programming for a virtual storage system.

The key objective of programming techniques in a virtual storage system is the reduction of page faults. The techniques involve two approaches:

1. Considerations that should be made while coding a program or group of modules that will form a program.
2. Considerations that should be made when you load a program; specifically, where you place modules in virtual storage.

Three general rules will help to keep page faults to a minimum. The first relates to *locality of reference* – keep things used with each other near each other. The second rule relates to *minimum real storage* – keep as small as possible the amount of real storage required for a program to execute without a large amount of degradation caused by paging. The third relates to *validity of reference.* Don't retrieve useless data when you reference storage. Some examples of these rules follow.

To achieve good *locality of reference*, processing should be sequential for both code and data. Computer programs normally execute sequentially but in certain situations this is not the case:

1. Error handling or unusual-situation routines should be separated from the main section of a program. Preferably they should be separate subprograms.
2. If a short subprogram is used (or called) only once or twice (and it is not an unusual-situation routine), then its code should be included in the calling program. An example of this would be a routine to calculate square roots. Include its code wherever you would normally call it. You would use more virtual storage, which is very large, but get less page faults and use less real storage.
3. Subprograms should be loaded near the programs that use them. An understanding of program flow between routines can help the user package them to reduce page faults.

Whenever possible, program data should be arranged so that it is accessed sequentially in memory:

1. Initialize data as close as possible to its first use. This will improve the probability that the page containing the data will still be in real storage when again referenced.
2. When a new data item is defined place it as close as possible to the items that will use it.
3. Reference arrays (or other data structures) in the order in which they are stored in storage, for example, columnwise versus rowwise when using FORTRAN; or define arrays in the order that they will be referenced.

To achieve *minimum real storage* keep as low as possible the amount of storage that a program references in a short time period. In other words, use some techniques to attempt to get small working sets. Use separate subroutines whenever the flow of control in your program suggests that execution will not always be sequential. Load modules in some optimum order. This will greatly reduce the number of page faults. For example, load all frequently used subroutines near each other in virtual storage. By the nature of their frequent usage, these routines will tend to stay in real storage. As we mentioned earlier, load unusual-situation routines far away from the main stream of code. Write highly modular code. Then package the code according to the frequency of reference.

Have few storage references that retrieve useless data; consider the *validity of reference.* One should try to avoid long searches for data. Expend virtual address space, which is very large, to reduce the need for real storage. Use data

structures that can be directly addressed, like arrays, rather than structures that must be searched, such as chains. Indirect addressing is bad, and so is any method that simulates indirect addressing.

In a virtual storage system you can begin to consider approaches to problem solutions that are not very feasible in other systems. This is due to the size of virtual storage. For example, use virtual storage for what, in a non-virtual storage system, would have been scratch files or spilled tables. When coding new applications, don't use overlays; they don't reduce paging. System I/O (paging) will work faster than user-initiated I/O, and paging will be done anyway.

Even though this lesson introduced several new techniques, they shouldn't add time to coding programs. Keep these factors in mind and code problem solutions in a natural way. This will most likely result in the best use of a virtual storage system.

Lesson 2 Answer Key

1. A

2. C

3. time

4. static relocation, dynamic relocation

5. C

6. B

7. C

8. A

9. B

10. False

11. True

Lesson 3 Answer Key

1. C

2. A

3. C

4. Dynamic Address Translation

5. False

6. B

7. Segment Table Origin Register

8. A

9. D

10. B

11. A. 247230
    B. 128032

12. D

Lesson 4 Answer Key

1. A

2. B

3. False

4. True

5. True

6. B

7. False

8A. 45024

8B. This address can't be translated through the associative array registers. None of the registers contain its segment and page number. Therefore, translation would occur through the segment and page tables.

9. Real storage and auxiliary storage

10. True

Lesson 5 Answer Key

1. C

2. B

3. virtual address structure

4. C

5. True

6. False

Part II.....

7. D

8. A

9. True

10. True

11. False

Lesson 6 Answer Key

1. B

2. A

3. B

4. C

5. D

6. A

7. A

8. C

9. B

10. C

11. C

12. True

13. B

14. True

Lesson 9. Introduction to Part II

We have presented a conceptual virtual storage operating system in PART I, discussing the hardware and software needed for its implementation. Also in PART I, we described the System/370 DAT feature and other hardware used to support virtual storage in several models of System/370. However, we only presented a *conceptual* virtual storage operating system. Several operating systems support virtual storage in System/370. In PART II we will describe these systems and how they use the new hardware features in System/370. We will begin with a discussion of the Operating System/Virtual Storage (OS/VS). There are two versions of OS/VS:

1. OS/VS1, which is an extension of OS/MFT.
2. OS/VS2, which is an extension of OS/MVT.

In addition, OS/VS2 has two releases that are significantly different. Release 1 (Rel 1) of OS/VS2 supports a single virtual storage. Release 2 (Rel 2) of OS/VS2 implements multiple virtual storage support. In PART II we will first present features common to both options of the OS/VS system; we will then present the features of OS/VS1, and then the features of OS/VS2 both Rel 1 and Rel 2. The last topic in PART II will be a description of virtual storage support in the Disk Operating System (DOS/VS) when used with System/370 models that have the DAT feature.

There will be no additional description of VM/370 in PART II. If you have further interest in VM/370 you can reference "Introduction to VM/370," Form Number GC20-1800.

You should not begin to study PART II until you study PART I of this text. Our presentation in PART II assumes that you have read and understand PART I.

Lesson 10. Introduction to Operating System/Virtual Storage

Operating System/Virtual Storage (OS/VS) is an extension of OS on System/360. It brings virtual storage to the OS user who has a System/370 model with the DAT feature. Our presentation in PART II will focus on how OS/VS implements virtual storage.

Dynamic address translation, segmentation, paging and virtual storage are all supported in the OS/VS system. The result for the user is a *virtual storage*, 16,777,216 bytes (16 megabytes) in size. This results in several benefits:

1. Programmers have more space for problem solutions. With OS/VS, programmers will have less need for overlay or multi-step design. OS/VS will "automatically overlay" with demand paging.
2. OS/VS gives you dynamic system management. All active jobs share real storage through demand paging. If a job has little activity – for example, a teleprocessing application getting few transactions – the job will use little real storage.
3. The system has more space for resident system functions that can be shared by all users. OS/VS is structured in virtual storage, not in real storage like OS on System/360. You can make more system functions like SVC's or access methods resident in virtual storage. They won't use real storage unless referenced. When used, they might be shared by more than one active job.
4. Real storage fragmentation is reduced. OS/VS uses paging to manage real storage. This eliminates a problem of OS on System/360 – real storage fragments between regions and real storage fragments within regions or partitions. In OS/VS fragments may exist, but they will exist only in virtual storage. This has little or no impact on the real storage resource in your System/370.
5. With OS/VS, you can experiment with applications whose storage requirements are larger than your System/370's real storage resource. In OS/VS, programs are loaded in virtual storage and demand paged in real storage.
6. Your System/370's real storage size will no longer be such a critical factor when configuring the structure of your OS/VS operating system – nucleus size, selection of resident functions, number and size of regions or partitions, and so forth. Your OS/VS system is structured in virtual storage. In an emergency, you could run on a back up System/370 with a smaller real storage than your normal system. If you add more real storage to your system, OS/VS will automatically take advantage of the additional real storage because of the paging concept.

We will now describe how the OS/VS system implements virtual storage to give you these advantages and others that can be derived from the use of a virtual storage system. In this topic we will describe features common to both options of the OS/VS system.

Virtual Storage in OS/VS

The OS/VS system has single and multiple virtual storage implementations. OS/VS1 and OS/VS2 Rel 1 implements a single virtual storage, the system's address space. OS/VS2 Rel 2 implements multiple virtual storages; in OS/VS2 Rel 2 each user has his own address space. Virtual storage size is 16 megabytes. OS/VS1 has a user option to make virtual storage smaller. We will discuss this in the section on OS/VS1. An important concept that you must remember is that the OS/VS system *is structured in its virtual storage* (the system's address space) – just as OS/360 is structured in real storage. Parts of OS/VS that are resident in virtual storage include its nucleus, any user-selected optional system functions such as access methods and SVC's, and active user problem programs. In the single virtual storage implementation, active users execute in regions or partitions that are structured in virtual storage. With the multiple virtual storage implementation, each user executes in his own address space. OS/VS manages real storage with demand paging. Libraries in auxiliary storage contain system functions and user programs just like OS on System/360. However, because OS/VS is structured in virtual storage more system functions like SVC's and access methods can be made resident in virtual storage. This is simply because virtual storage can be larger than real storage. Also, even though an SVC or access method is resident in OS/VS's virtual storage, it won't use real storage unless referenced. We will describe the structure of virtual storage in a moment and we will describe how each OS/VS system looks in virtual storage later. For now, remember that OS/VS is structured in its virtual storage, the OS/VS system's address space; real storage is a system resource controlled through demand paging.

Virtual Storage Structure

In the OS/VS System, virtual storage has a segment-page

structure. Segments are fixed in size (64K). There are 256 segments in 16 megabytes of virtual storage. Page size is different for each option of OS/VS. OS/VS1 has 2K pages; therefore, there are 32 pages in each segment. OS/VS2 has 4K pages. In OS/VS2, there are 16 pages in each segment. Because virtual storage in OS/VS has a segment-page structure, it needs a segment table to map virtual storage. The System/370 DAT feature uses the segment table for dynamic address translation. OS/VS uses the segment table for virtual storage allocation. It allocates virtual storage to system programs and user problem programs in segment size increments. Thus, OS/VS takes advantage of segmentation as a good technique for managing virtual storage. Each segment of virtual storage has a page table. The page tables are also used by the System/370 DAT feature during dynamic address translation.

Virtual Address Translation

Virtual addresses in OS/VS are translated using the technique that we described in PART I. The System/370 Segment Table Origin register (STOR register) points to the real storage location of the segment table. The segment number in the virtual address is used as an index into the segment table. The segment table entry points to its page table. The page number in the virtual address is used as an index into the page table. The page table entry supplies the required page frame location, or, if the page is not in real storage, a page fault occurs. If the page is in real storage, the page frame number is linked or concatenated to the virtual address displacement and translation is complete. Translation is performed by the System/370 DAT feature. Also remember that fast translation occurs in parallel using associative array registers or the translation lookaside buffer (TLB). OS/VS builds and maintains the segment and page tables. The System/370 DAT feature performs virtual address translation. The segment and page tables must reside in real storage. However, OS/VS doesn't build page tables for unallocated segments. When a segment is allocated its page table is built. This saves some real storage because page tables and the segment table must be in real storage during address translation.

Levels of Storage in OS/VS

External Page Storage

There are two storage levels used for paging in OS/VS. Pages are transferred between *page frames of real storage* and *slots of external page storage.* External page storage is part of auxiliary storage in OS/VS. It uses direct access devices. These may be disk devices like the 3330, 2319 and 2314, or a fixed-head file like the 2305. The amount of external page storage needed for an OS/VS system relates to the size and use of virtual storage. Another term that you will read or hear in discussions of external page storage is *paging data sets.* They are simply data sets that store pages in external page storage.

Real Storage

In the OS/VS system, real storage is divided into page frames. Page frames in OS/VS1 are 2K. Page frames in OS/VS2 are 4K. OS/VS has a page frame table that indicates the status (available or in-use) of each page frame. With demand paging, OS/VS loads pages into real storage from external page storage as they are referenced during program execution. If all page frames are used, page replacement is required. OS/VS tries to replace the least recently used (LRU) page in real storage – no matter to whom the page belongs. OS/VS does this by using the reference bits associated with page frames and by manipulating the page frame table. During page replacement, if the page being replaced has changed – as indicated by its page frame's change bit – a page-out operation is also required. By using the LRU rule, OS/VS tries to keep the most active pages in real storage. In effect, OS/VS tries to keep each program's working set in real storage.

The Paging Supervisor

The OS/VS system's nucleus, its major control program, is an extension of the nucleus in OS on System/360 (in OS/VS, another name for the nucleus is the resident supervisor). In addition to controlling and allocating system resources, the OS/VS nucleus has a *paging supervisor.* The paging supervisor controls all of the paging activity in OS/VS. Some of the paging supervisor's functions are to:

1. Maintain the system's segment table, page tables and the page frame table.
2. Initiate page-in and page-out operations.
3. Control page replacement using a page replacement algorithm like LRU.
4. Service all page faults.
5. Monitor the system's paging rate to prevent thrashing.

In effect, the paging supervisor manages real and external page storage to implement virtual storage in OS/VS. The paging supervisor is the software support in OS/VS for demand paging. The System/370 DAT feature is OS/VS's hardware support.

Page Fixing

As we said earlier, the OS/VS system is structured in virtual storage. Real storage is a system resource controlled by the paging supervisor. However, pages from certain areas of virtual storage are fixed in real storage for long or short time periods. What pages of the virtual storage are fixed is determined by the nature of their function.

Some pages are always fixed during OS/VS operation. For example, the nucleus is crucial for system operation. In OS/VS, the nucleus is resident in virtual storage and all of its pages are fixed in real storage during system operation. Pages used for the system queue area are also permanently fixed in real storage as they are allocated.

Some pages are fixed for the duration of a job or job step. For example, pages that hold certain control blocks related to a job are fixed for the duration of the job. These examples so far all relate to pages that have a *long term fix.*

Pages may be given a *short term fix.* For example, pages used as I/O areas for input/output operations are fixed only for the duration of the I/O operation. They get a short term fix in their real storage page frames (the next lesson will present how I/O functions in the OS/VS system).

Page fixing is controlled by the OS/VS paging supervisor. Fixed pages are identified in the OS/VS page frame table. Who can fix pages? Only the OS/VS nucleus and supervisor subsystems and functions are authorized to selectively fix some or all of their pages. User programs cannot selectively fix pages of a program in real storage. However, the OS/VS system might selectively fix pages for you. During I/O operations OS/VS will fix pages used for I/O areas. When your programs perform I/O, OS/VS will give your I/O area page(s) a short term fix. At the end of each I/O operation, OS/VS frees these pages. They are again available for replacement. We will describe why OS/VS fixes pages during I/O operations in the next lesson.

Why are pages fixed in a demand paging system? OS/VS uses page fixing for one of three reasons:

1. Some functions are used by everyone in the system, and some functions must be fixed in real storage for the system to function at all. The nucleus serves everyone in the system. If it were paged everyone in the system would wait. The paging supervisor – contained in the OS/VS nucleus – controls paging. If it isn't fixed in real storage, OS/VS can't page other system and user programs.
2. Some pages are fixed so that their real storage page frame locations won't change during an operation. I/O areas are fixed for this reason during I/O operations.
3. Some devices are highly time dependent. For example, a bank reader/sorter device requires rapid response because of its high time dependence. All MICR devices have a high timing dependence. Programs that service these devices shouldn't be paged. If they are, transactions might occur during a paging operation and be lost. Pages of highly time dependent programs should be fixed during execution. However, this represents a very small class of devices and programs.

Even though OS/VS may fix pages for the conditions we have just described, the system, in general, is very conservative about page fixing. When a page is fixed its page frame isn't available for paging.

Virtual Equals Real Option

We have just seen that the OS/VS system can fix pages and why it fixes pages. An OS/VS user can fix pages, but not selectively. You can fix pages by labeling a job step *virtual equals real (V=R).* We will describe how each OS/VS system implements the V=R option later. Its effect is to give a long term fix to all pages in programs used by a V=R job step during program execution. Therefore, a V=R job step is not paged. V=R job step allocation reduces the number of page frames that can be shared by other active jobs in the system. For this reason, you should be careful about using the V=R option. In general, there are only two reasons why you would execute a job step as V=R.

1. It services a MICR device.
2. The job step dynamically modifies channel programs during I/O operations.

We will explain the second reason in the next lesson on channel program translation.

Virtual Storage Allocation

In OS/VS, virtual storage is allocated to system components and active user jobs in segment size increments. Segments are fixed in size. They are 64K. With a single virtual storage, if your job needs 96K, it will get two segments of virtual storage (128K), even though it will only use 96K. With multiple virtual storages, your job would have access to all segments in the private area of your address space (this will be explained later). Unused virtual storage doesn't waste any real storage. It represents unused or potential address space. The system allocates virtual storage in segments because it is convenient and easy to control. OS/VS only needs to examine the segment table. This tells the system what segments are in use and how much virtual storage is available.

When we described page fixing we said that pages used for I/O areas must be fixed in real storage during I/O operations. In the OS/VS system these pages get a short term fix. Before an I/O operation begins, any page(s) used as an I/O area is fixed in its page frame. At the end of the I/O operation, OS/VS frees the page(s). It is again available for replacement through demand paging.

Short term page fixing during I/O is controlled by the OS/VS Input/Output Supervisor (IOS). Short term page fixing, plus some other special operations provide an OS/VS function called *Channel Program Translation.* It is a function of the OS/VS Input/Output Supervisor (IOS) that is needed in a virtual storage system. Why does OS/VS use Channel Program Translation? How does Channel Program Translation work? We will begin with a brief discussion of channels in System/370 and proceed to answer these questions.

System/370 Channels

System/370 uses channels to process I/O operations. Thus, channels control the transfer of data between real storage and I/O devices. A channel is actually like a small, special purpose computer. Special commands or instructions control its operation. These commands are called *Channel Command Words* (CCW's). When combined to control an input or output operation, a string or group of CCW's is called a channel program. A channel program, because it is executed by the channel, may execute concurrently with the System/370 CPU. This is called I/O overlap.

Let's assume that you write a program. Your program gets input records from tape, gathers some statistics and prints these statistics as output. You use a language like COBOL or PL/I. When your compiler builds machine executable code, the code will contain two channel programs. One channel program will control the operation that reads tape records. The other channel program will control printing the output. As a programmer, you don't need to worry about how these channel programs are built, when they execute, or how they execute. The system's software supplies these functions. To see why channel programs get special consideration in a virtual storage system, let's further investigate what happens in our example.

The commands or instructions in a channel program are called Channel Command Words (CCW's). In our example, one group or string of CCW's – the first channel program – would control the tape input; the other group of CCW's – the second channel program – would control printing the output. In the case of tape input, the channel program would control reading tape records and placing them into an input area in real storage. Thus, the channel program's CCW's contain instructions to read the tape records and they contain the real storage location or address of the input area. Therefore, CCW's reference real storage. To control printing output, the second channel program must reference the output area in real storage. One or more of its CCW's must have the real storage location or address of the output area. In System/370 then, channels reference real storage locations just as the CPU does. In a virtual storage system like OS/VS, the System/370 CPU uses the DAT feature to translate virtual addresses into real storage addresses. However, System/370 channels do not have an address translation feature. Channel programs in OS/VS have virtual addresses. The virtual addresses in CCW's must be translated into real storage addresses by software in the OS/VS system. This software function of OS/VS is called *Channel Program Translation.* In the OS/VS system, the System/370 CPU performs address translation using special hardware, its channels use software translation.

The Channel Program Translation Function

Let's name our tape-to-printer program. We'll call it Program A. Figure 59 shows Program A residing in a single virtual storage system such as OS/VS. We have indicated the tape input area and the printer output area. For convenience we made each area 4K. We also assume that page size is 4K. Both channel programs reside within Program A. The channel program for tape input contains the location of the tape input area in one of its CCW's. In Figure 59 this is virtual address 336K. The channel program for printer output contains the location of the printer output area in one of its CCW's. In Figure 59 this is virtual address 324K. Program A resides in virtual storage. It is executed using the demand paging technique.

What happens when I/O occurs? Figure 60 shows Program A executing. Several pages of Program A are in real storage. Noted in particular in Figure 60 are the page frames that contain the tape input area and the printer output area. They are located at real storage locations 84K and 96K, respectively. If the channel program for tape input executes, it will try to place a record into virtual storage location 336K. The channel program for printing output would try to get data beginning at virtual storage

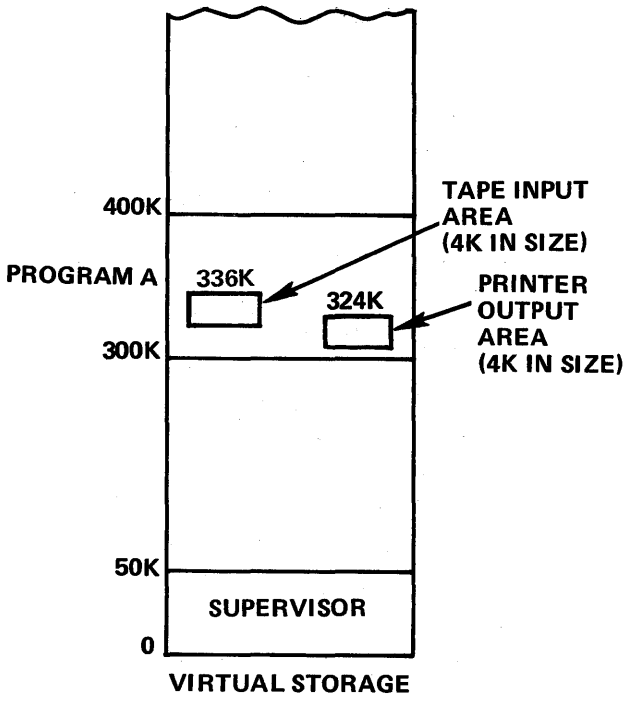


Figure 59

location 324K. This is because channels have no hardware DAT feature to translate the virtual addresses in channel program CCW's. To compensate, the Input/Output Supervisor (IOS) must translate these CCW virtual addresses into their current real storage locations before each I/O operation begins.

In our example in Figure 60, before a tape input operation can begin IOS will translate the CCW address of 336K to 84K. The input operation could then proceed. If printer output is also required at this time, what is the result of IOS channel program translation? The CCW in the printer output channel program that has virtual address 324K will be translated into real storage location 96K. The output operation could then proceed. When IOS translates the virtual addresses in channel program CCW's it does not replace them with the resulting real storage addresses. If it did the next channel program translation would be invalid. During translation, IOS takes a CCW virtual address, uses the segment and page tables to translate it and places the real storage location into a duplicate CCW string. In effect, IOS builds its own "real" channel program before I/O begins. The channel executes I/O using this "real" channel program. IOS translates the channel program virtual addresses each time an I/O operation is started. Translation occurs for every I/O operation – unless a program executes as Virtual = Real – so that the system may demand page I/O areas.

I/O areas may be paged between I/O operations. However, an I/O area may not be replaced during an I/O operation. Channel program translation occurs before I/O begins. If all or part of an I/O area were replaced during an I/O operation, the channel would not be given any indication. The channel would continue to execute the I/O operation. The page that replaced all or part of the I/O area would either be changed with input and become invalid, or written out as data that is invalid. To prevent this, the system issues a short term fix for all pages that contain I/O areas related to an I/O operation before it begins. IOS actually issues the fix. At the end of an I/O operation, IOS frees its I/O area page(s).

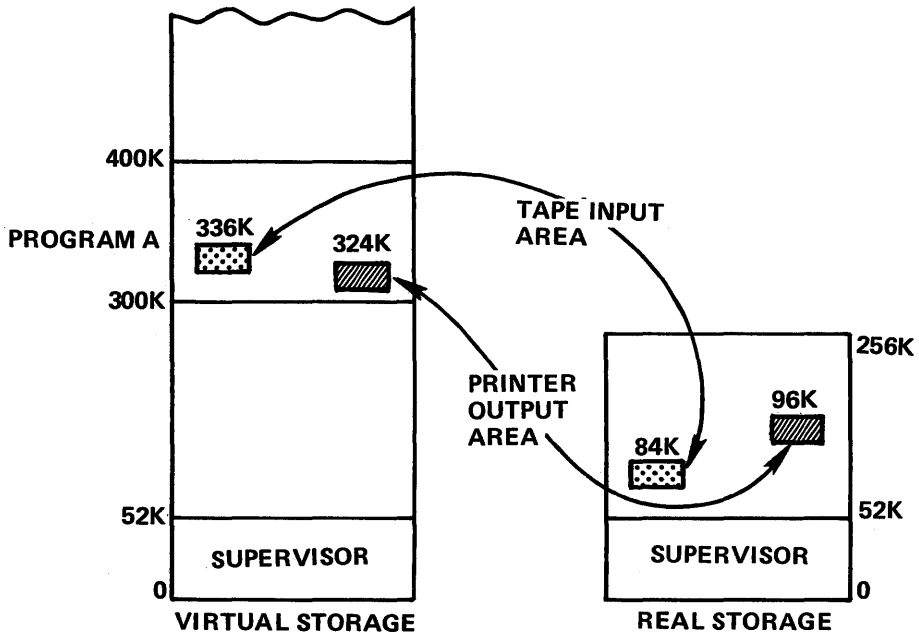


Figure 60

Channel Program Translation allows OS/VS to demand page most programs that you will execute in OS/VS. However, one class of programs will not run under paging. Their channel programs are dynamically modified during I/O operations. A dynamically modified channel program is one in which the CCW address of an input or output area is changed during an I/O operation. Because CCW address translation occurs before I/O begins, OS/VS won't be notified during an I/O operation of any change in the CCW addresses. In the event of a change, input or output data would be transferred to or from the wrong real storage location. Programs that use dynamically modified channel programs must be executed in the virtual equals real area of virtual storage. When a program is run as virtual equals real in OS/VS, channel program translation is not required because the virtual addresses in channel programs equal their real storage locations. Not many programs use dynamically modified channel programs. As a result, there is no large impact on OS/VS.

The Channel Program Translation feature in OS/VS involves four steps:

1. Virtual addresses contained in a channel program's CCW's are translated to indicate the real storage locations of their related I/O areas.
2. Page(s) that contain the I/O areas are given a short-term fix in their page frame(s).
3. The I/O operation is executed.
4. At the end of I/O, the page(s) used for I/O areas are freed; they are again pageable.

The Channel Program Translation feature is a part of the Input/Output Supervisor (IOS) in OS/VS. It is a type of software translation and an automatic function in the system. Channel Program Translation requires no programmer action. It is internal to OS/VS. Channel Program Translation allows OS/VS to better share its real storage among all system users.

OS/VS1 (or VS1) is an option of the OS/VS system. OS/VS1 is a functional extension of OS/MFT. In OS/VS1, problem programs execute in fixed-size partitions in a single virtual storage. Maximum virtual storage size is 16 megabytes. A user may specify a smaller virtual storage at system generation and/or IPL time. The number of user partitions is specified during OS/VS1 system generation. You may redefine the number of partitions, their class, and their size during system operation. However, you may not exceed the number of partitions specified during system generation. A user may specify up to 15 problem program partitions.

OS/VS1 uses the System/370 24 bit-address. Its virtual address structure is shown in Figure 61. In VS1, segments are 64K and pages are 2K. There are, therefore, 32 pages in each segment.

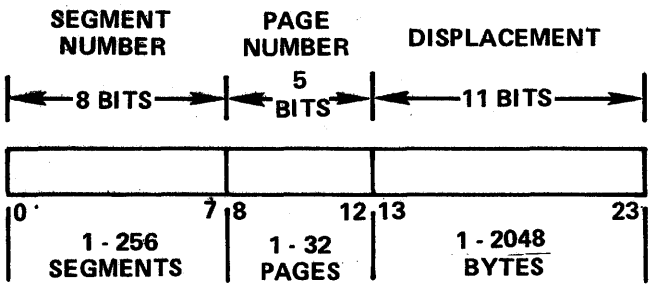


Figure 61

VS1 is structured in virtual storage just as MFT is structured in real storage. Thus the nucleus, all resident supervisor functions and all defined user partitions are in virtual storage. When VS1 schedules a job, it loads a program into a partition in virtual storage. When execution begins, the system loads pages into real storage as they are referenced through demand paging. We will make a more detailed presentation on program loading in VS1 in a later topic.

VS1's Structure in Virtual Storage

How is VS1 structured in its virtual storage, VS1's address space? Not much differently than MFT is structured in real storage. Examine Figure 62. In Figure 62, we cut virtual storage into two areas, the non-pageable area and the pageable area. The major difference between each area is how VS1 allocates virtual storage within the area. In the non-pageable area, VS1 allocates virtual storage in page size increments (2K). This is done because all pages in the non-pageable area, when allocated, get a long term fix in real storage:

1. All pages in the nucleus are fixed during system operation.
2. Any pages allocated for *S*ystem *Q*ueue *A*rea (SQA) expansion get a long term fix as they are allocated.
3. Pages allocated to a virtual equals real job step are fixed for job-step duration.

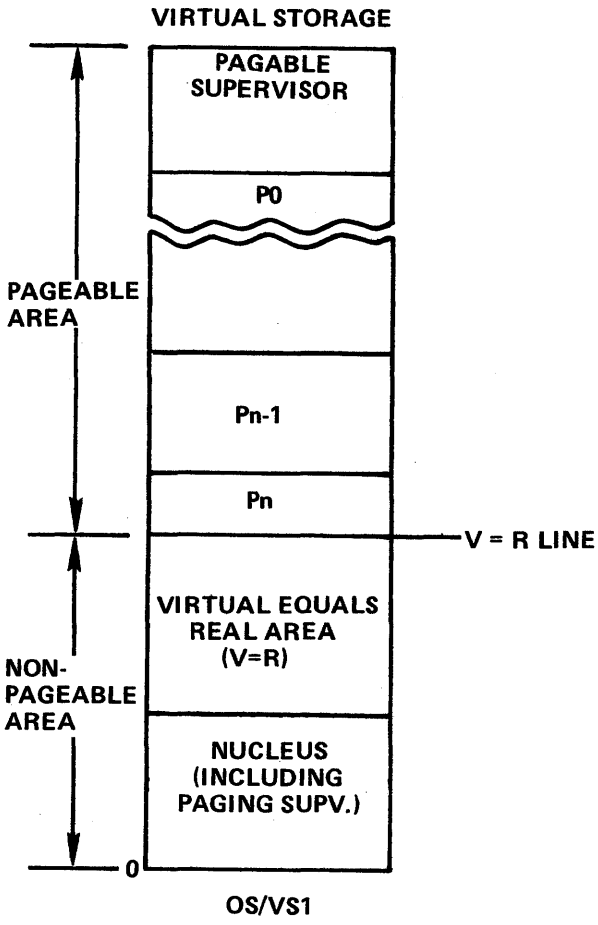


Figure 62

In the pageable area, VS1 allocates virtual storage in segment size increments (64K). Partitions may be one or more segments in size. The pageable supervisor will use one or more segments. Since VS1 manages the pageable area of virtual storage using segmentation, the number of segments in your virtual storage becomes an important consideration.

The size of the non-pageable area in virtual storage is defined by where the Virtual Equals Real (V=R) area ends.

In VS1, the V=R area in virtual storage ends at the V=R line. This is indicated in Figure 62. In VS1, the V=R line in virtual storage corresponds to the end of real storage in your System/370 – up to 768K. If your System/370 has more than 768K real storage, the V=R line in virtual storage is defined at 768K. Thus, if your System/370 has 512K, the non-pageable area of virtual storage ends at 512K. If your System/370 has one megabyte of real storage, the non-pageable area ends at 768K in virtual storage.

The size of the pageable area depends on the size of your virtual storage. Simply take virtual storage size and subtract the size of the non-pageable area; the result is the size of the pageable area. Because the pageable area is allocated in multiples of segments, if you divide its size by 64K this tells you how many segments may be allocated to partitions and the pageable supervisor.

We will now describe the characteristics of each part of the VS1 system structure in virtual storage: the nucleus, the V=R area, the pageable supervisor and partitions. Continue to reference Figure 62.

The Nucleus

VS1's *nucleus* begins at the origin of virtual storage, location zero. It contains the essential control functions of VS1 including the paging supervisor and the segment table. Minimum nucleus size is about 60K. As you add options at system generation; nucleus size will increase. Another name for the nucleus in VS1 is the *resident supervisor.* During system operation, all pages of the nucleus are fixed in corresponding page frames of real storage.

One special part of the nucleus is called the System Queue Area (SQA). SQA is used as a system workspace or scratch pad to hold control blocks and queue entries. SQA is a dynamic area of OS/VS1 that expands and contracts in size based on system activity. Its initial size (within the nucleus) is specified during system generation. If this initial SQA becomes filled during system operation, VS1 will allocate pages from the V=R area to SQA, one page at a time. All SQA pages get a long-term fix in real storage until deallocated.

V=R Area

The virtual equals real (V=R) area in VS1's virtual storage begins above the Nucleus and ends at the V=R line. It is used to execute job steps whose virtual addresses must be the same as real storage addresses. When a V=R job step is executed, VS1 must load it into a contiguous section of the V=R area. V=R space is allocated in page size increments. These V=R pages are then loaded into corresponding page frames in real storage. In effect, there is a one-to-one correspondence between V=R pages and their real storage page frames. These pages are then given a long term fix in real storage until the end of the V=R job step. Even though a V=R job step is executed in the V=R area, it is initiated from a partition in upper virtual storage. The partition used will service any scheduling needs of the V=R job during its execution. Thus, in addition to using part of your V=R area and fixing a significant portion of your real storage resource, a V=R job step will also tie up the partition that schedules it.

As we said before, the V=R feature is in VS1 to service a small class of jobs that can't run in a paging environment. For example:

1. Jobs that use dynamically modified channel programs.
2. Jobs that service time dependent devices, like MICR devices.

A V=R job step executes using the DAT feature; its addresses are translated even though its virtual addresses equal their real storage locations. However, no paging occurs and channel program translation isn't used. Therefore, the above types of jobs will execute properly in the V=R area. Don't execute your other job steps as V=R. Let VS1 manage its real storage resource through demand paging.

One important characteristic to remember about the V=R area is that it is a part of VS1's virtual storage; its pages in no way reserve or occupy real storage until used by SQA or a V=R job step. With no V=R job steps executing, real storage is shared by the pageable area of virtual storage under the control of demand paging. We will show the relationship of virtual storage to real storage more explicitly in a later section.

Pageable Supervisor

The pageable supervisor is composed of VS1 system control functions that are paged. The pageable supervisor is resident in virtual storage beginning at the top of the pageable area. It has several parts, some standard, some optional. As standard, it contains:

1. Several supervisor functions formerly in the OS/MFT nucleus such as the Communications Task and the SVC transient area.
2. The new VS1 Job Entry Subsystem (JES).

JES is an integrated subsystem of VS1 that controls job input and writes job output using a SPOOLING technique. Because JES is a part of the pageable supervisor it is paged during execution.

As an option, the VS1 user may load Supervisor Calls (SVC's) and/or access methods into the pageable supervisor during IPL. Thus, in VS1 resident access methods (RAM) and resident SVC's are resident in virtual storage in the

pageable supervisor:

1. Access methods, like BSAM, BDAM and so forth, may be made resident in virtual storage. They are reentrant, they are paged, and they may be shared by all user partitions. The user selects which *R*esident *A*ccess *M*ethods (RAM), if any, will be in his system at IPL time.
2. Supervisor Calls (SVC's) may be made resident in virtual storage. They are reentrant, paged, and may be shared by all user partitions. Resident SVC's are not loaded from a system library into the pageable supervisor's transient area. They are resident in virtual storage and demand paged when referenced.

Because all of these functions are pageable, they use real storage only when referenced. With VS1, you will have the opportunity to structure a system in which all of the functions that you require are resident in virtual storage. Yet you won't need to worry about committing real storage to SVC's or access methods that are seldom used or used only periodically. These functions reside in virtual storage. They use real storage only when referenced through the demand paging technique.

Partitions

After allocating VS1's virtual storage to the nucleus, and the V=R area in page size increments, and after allocating virtual storage segments to VS1's pageable supervisor, the virtual storage that remains can be divided into partitions. For example, let's assume that we have a VS1 system with one megabyte (1024K) of virtual storage. Further assume that the non-pageable area of virtual storage is 256K and that the pageable supervisor is allocated 128K, or two segments. 640K, or 10 segments of virtual storage, remain for partition allocation. Figure 63 shows this example. In OS/VS1, the smallest partition size is 64K, or one segment. The largest partition size depends on how much virtual storage is available for partition allocation. In our example (Figure 63) we could have one 640K partition. We could also have four partitions as shown in Figure 64.

Notice that partitions must be allocated in multiples of segments. In VS1, therefore, partitions are multiples of 64K. Also, in VS1, job initiation requires 64K. Because the smallest partition size is 64K, every partition can initiate jobs. There is no longer a small partition scheduling problem as there is in OS/MFT.

It isn't necessary to use all of the partitions that you specify during system generation or to use all of virtual storage during system operation. Any unused virtual storage remains potential address space. You might reserve a partition and its virtual storage for "hot job" scheduling. You might use a large area of virtual storage to develop and test

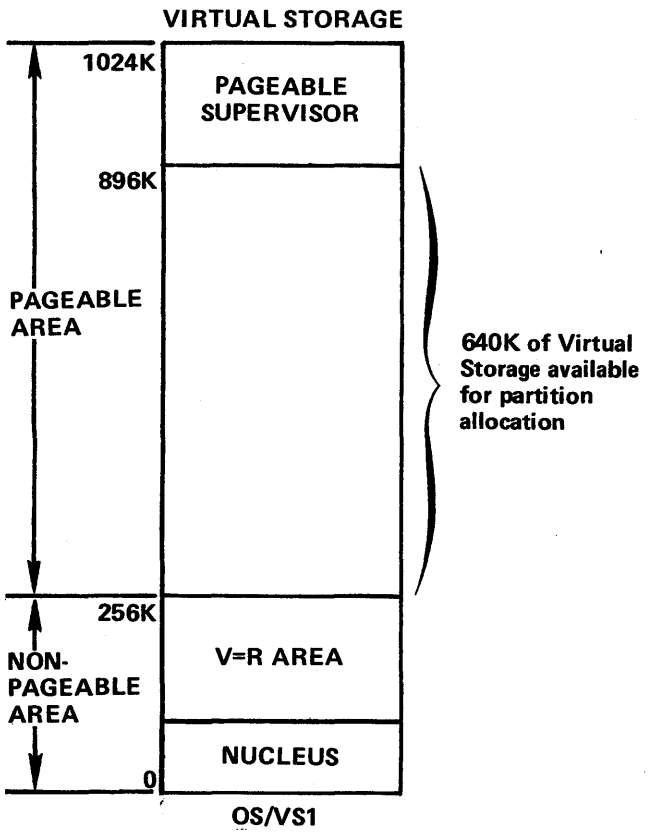


Figure 63

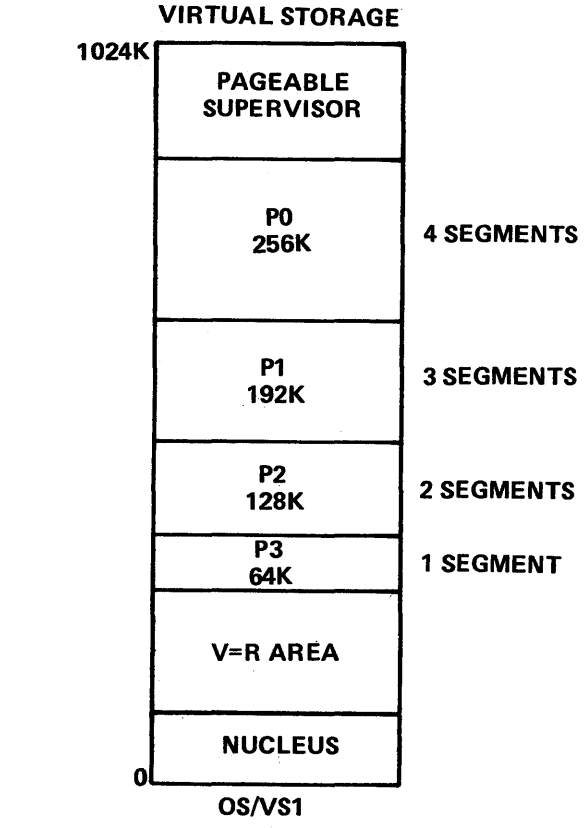


Figure 64

a large teleprocessing application and add more real storage to your system when you install the application. You might also define all partitions used for production jobs as the same size, selecting a size large enough to hold your largest job. Job scheduling would then become much easier. You could concentrate on assigning I/O-bound jobs to high priority partitions and CPU-bound jobs to low priority partitions to balance the CPU load. Fragmentation would occur, but within virtual storage not within real storage.

In our example in Figure 64 we assumed a virtual storage of one megabyte and then determined how much space was available for partitions. When you select a virtual storage size, it will depend on the number of partitions you need and their size, the size of the pageable supervisor, the size of the non-pageable area and any extra virtual storage or potential address space that you may reserve for special situations like scheduling hot jobs or application development. Remember that you need adequate real storage and external page storage to support your VS1 virtual storage. Enough real storage should be in the system to support its activity at any point in time. In the next topic we will describe the relationship between virtual storage and real storage in the VS1 system.

Real Storage in OS/VS1

Let's assume that your System/370 has 256K of real storage and you are operating with a virtual storage size of 768K. Your VS1 system might look like the one in Figure 65. We further assume a nucleus size of 80K shown in the non-pageable area of virtual storage. Figure 65 also includes a picture of real storage. Notice that the entire nucleus is fixed in real storage during system operation. The remaining real storage – 176K or 88 page frames – is available for paging. It is shared among the active user partitions and the pageable supervisor in the pageable area of virtual storage. If SQA expands, its newly allocated page is selected from the V=R area and given a long term fix in a corresponding page frame of real storage.

Notice also in Figure 65 that the V=R area in virtual storage ends at 256K, the location of the V=R line. Remember that the V=R line corresponds to the end of real storage up to a real storage size of 768K. No real storage is reserved for the V=R area under normal system operation. Real Storage is shared dynamically among all active jobs and the pageable supervisor in the pageable area of virtual storage, as indicated by the arrow in Figure 65. The V=R area simply reserves address space for V=R job steps in the event that they occur.

What happens if a V=R job step is scheduled? Let's assume that a V=R job step called VRSTEP is scheduled in P1. The scheduling activity is indicated by the shading in P1 in Figure 66. A job step is specified as V=R in the step's JCL card. Also specified is the size of the job step. Let's assume VRSTEP needs 62K. Before VRSTEP's execution, VS1 must allocate 62K of contiguous space from the V=R area. VS1 must also reserve the page frames in real storage

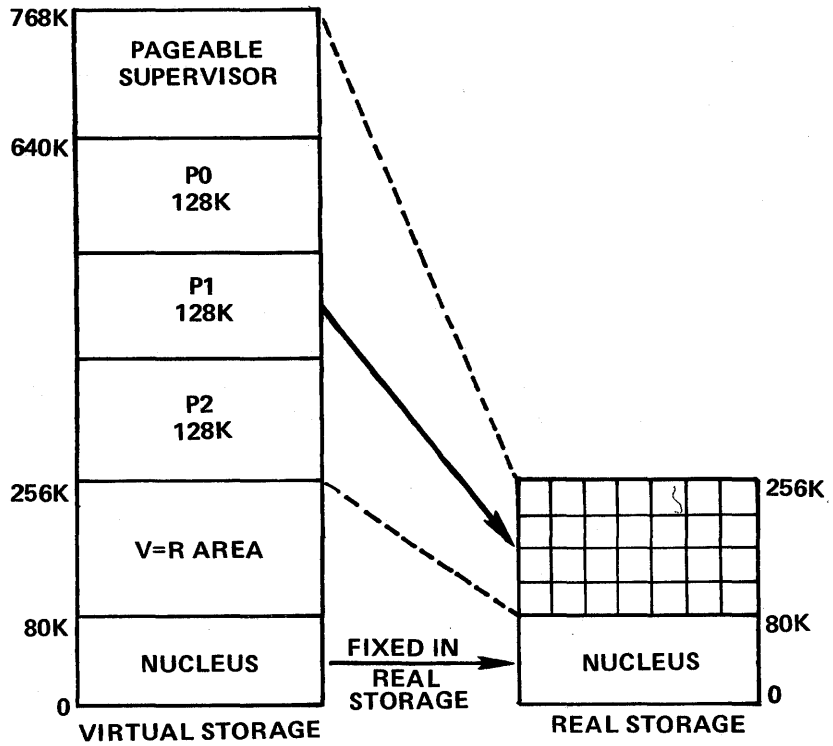


Figure 65

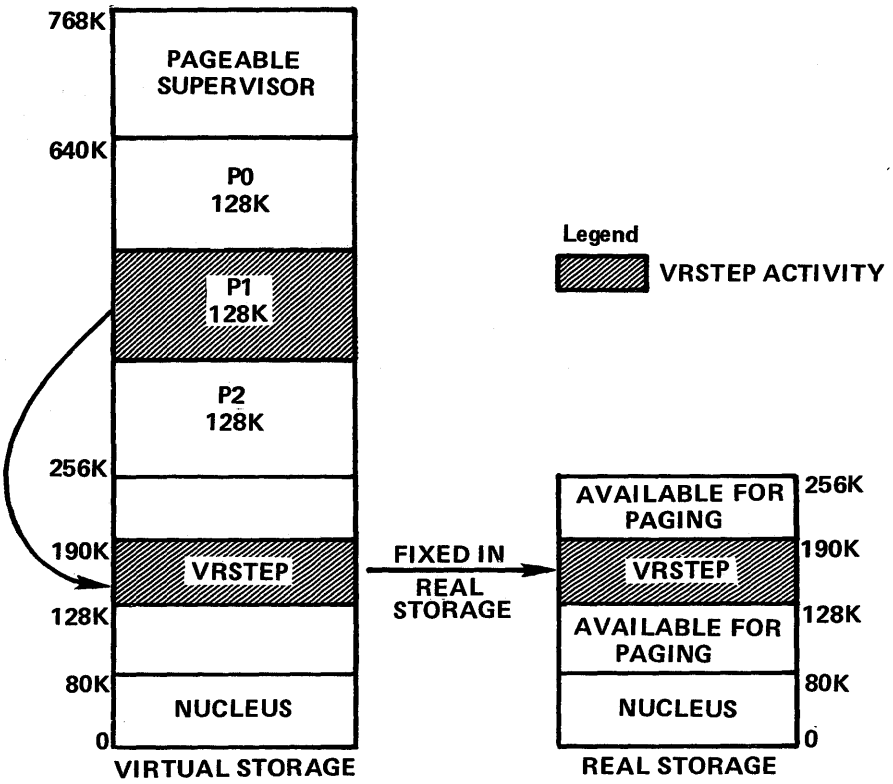


Figure 66

that correspond to this 62K of virtual storage. VRSTEP can then be loaded and executed as shown in Figure 63. Notice that the remaining page frames in real storage are still available to the pageable area of virtual storage through demand paging.

We have just described how a V=R job step is loaded and executed in virtual storage. In a later topic we will describe how jobs are loaded into the pageable area of virtual storage.

External Page Storage in OS/VS1

You have just seen how VS1 uses real storage. Now let's examine external page storage in VS1. Disk devices like the 2314, 2319 and 3330 or a fixed-head file like the 2305-2 may be used for external page storage. Because OS/VS1 has a single virtual storage, the system uses a simple but effective approach to control external page storage. All of the VS1 system's virtual storage above the V=R area resides on external page storage. Thus, for each page of virtual storage in the pageable area there is a corresponding slot of external page storage. There is a one-to-one correspondence between the pageable area of virtual storage and external page storage. The non-pageable area of virtual storage – the nucleus, and the V=R area – requires no external page storage. When used, pages from the non-pageable area are fixed in real storage. They aren't paged; therefore, they require no external page storage. Figure 67 shows the relationship between virtual storage and external page storage. For each page in the pageable area of virtual storage there is a corresponding slot in external page storage. In Figure 68 we show the relationship between virtual storage, real storage and external page storage. The nucleus is fixed in real storage. Any of the V=R area, when allocated to a V=R job step, is fixed in real storage until the job step ends. The non-pageable area of virtual storage is never paged. No external page storage is needed. Paging occurs between external page storage – where partitions and the pageable supervisor reside – and real storage. When a page fault occurs VS1 must page in the referenced page from its slot in external page storage. It may be placed in *any available page frame.* If this page is paged out at some later time it must return to the *same slot in external page storage.* This preserves the one-to-one correspondence between VS1's pageable area in virtual storage and its external page storage. Because pages always return to the same slot in external page storage there is no need for external page tables in OS/VS1. In VS1, the pageable area of virtual storage physically exists in the form of external page storage. Using this design the pageable supervisor and active user partitions share real storage in OS/VS1. Fragmentation

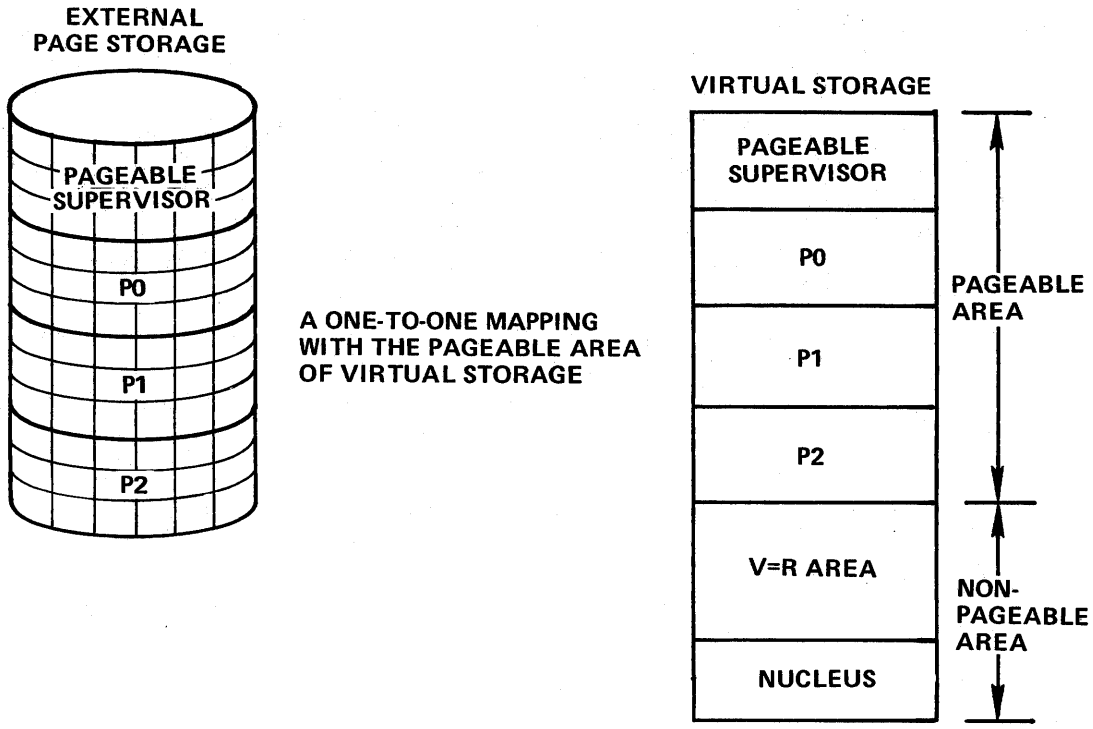


Figure 67

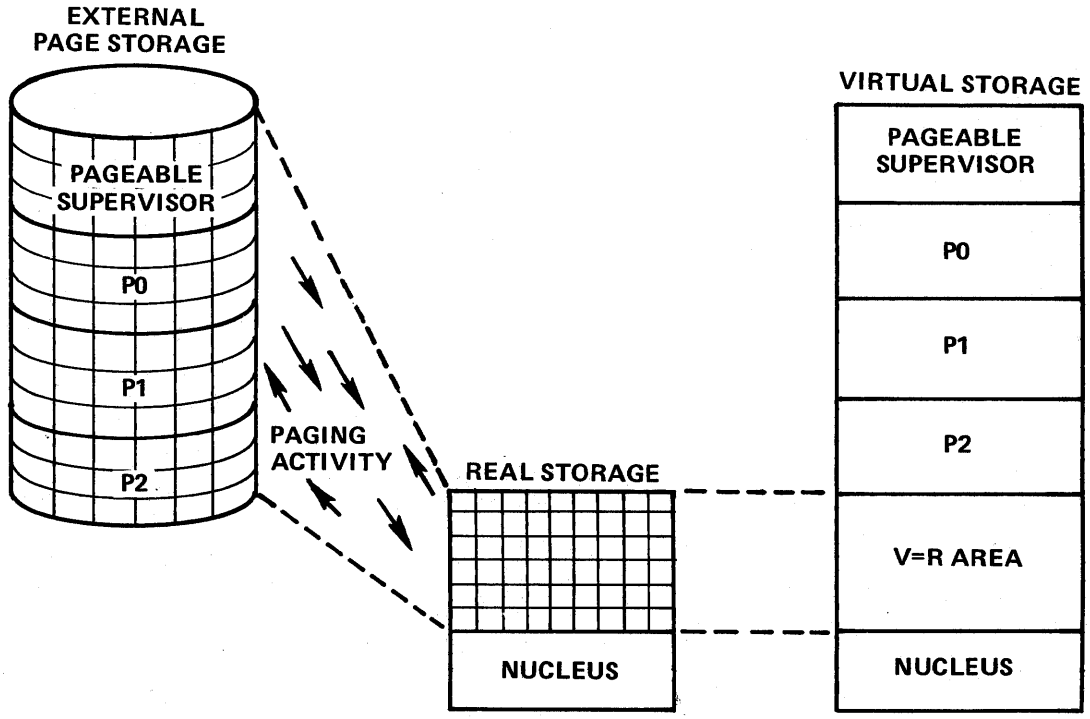


Figure 68

is removed from real storage. VS1's virtual storage may be fragmented but this only results in some waste of external page storage, not the system's real storage.

Program Loading in OS/VS1

As we said earlier, in VS1 virtual storage size is a user option. It cannot exceed 16 megabytes, but it might be less. You might specify 1024K, 2048K and so forth at IPL time.

When you IPL VS1, the system will ask you several questions. For example, it will ask you what size virtual storage you want for this run. Let's say that you generated a VS1 system with a 2048K virtual storage. You could reply 2048K, 1024K, or even 4096K. This lets you experiment, and determine a virtual storage size most suitable for your installation. Another question that VS1 asks during IPL is the number of partitions that you will use and their size. You may redefine your standard partition sizes and their classes at this time. Partition size is specified in multiples of segments (64K). The number of partitions cannot exceed the maximum that you specify during SYSGEN.

After IPL, VS1 begins to process jobs. All jobs waiting for service are kept in a system job queue. Each active partition may schedule jobs. As one job ends in a partition, another may be scheduled from the system job queue. What happens after a job has been scheduled into a partition? How is the program for each job step loaded into virtual storage? Loading a program into a virtual storage partition in VS1 has the same requirements as loading a program into a real storage partition in OS/MFT. It is done in the same way, except that virtual storage, not real storage, is loaded.

In VS1, after compilation or assembly, user programs are stored in a system library. Their addresses are relative to a zero origin. They don't need to be in a segment-page format. In fact, they're not. In VS1, programs are stored in partitioned data sets. The essence of program loading is to relocate a program's relative addresses to the origin of the partition in virtual storage in which it will execute. Each partition begins on a segment boundary. At load time the program's relative addresses are relocated to a range of addresses in virtual storage. This is a form of static relocation. It is performed by the loader program in VS1 called program fetch. During the loading process the loader program is executing. It reads the program to be loaded from the system library into real storage as data, relocates its addresses to a partition's segment boundary, and arranges the program in a page format. As loading proceeds, the program being loaded is paged out to external page storage. When the program is completely loaded the system transfers control to it; it begins to execute; its pages are loaded into real storage as referenced through demand paging.

Now that you have studied this lesson, you should be able to describe how virtual storage is implemented in OS/VS1. The VS1 system effectively implements a single virtual storage, the system's address space, using the IBM System/370 DAT feature.

OS/VS2 (or VS2) is a functional extension of OS/MVT. VS2 brings virtual storage to the MVT user. It also supports TSO in its virtual storage environment. VS2 has had two major releases, *Release 1* (Rel 1) which implements a single virtual storage, and the recently announced OS/VS2 Release 2 (VS2/2) system. VS2/2 is a multiprocessing system that implements multiple virtual storages. In this lesson we will describe how Release 1 of OS/VS2 implements its single virtual storage. Throughout this lesson all references to OS/VS2 and VS2 refer to Release 1 unless noted otherwise. Lesson 14 will then describe the Release 2 version of OS/VS2.

OS/VS2 has a single virtual storage. You can run multiple jobs in a variable number of regions in your virtual storage. Virtual storage size is 16 megabytes. During system generation the system will automatically specify a virtual storage size of 16 megabytes. Although you might not use all of virtual storage during operation, it is always there as potential address space.

VS2 uses the System/370 24-bit address. Therefore, its virtual address is 24 bits long. This determines the size of VS2's virtual storage, 16 megabytes. The structure of VS2's virtual address is slightly different than OS/VS1's. Examine Figure 69. Pages are 4K as they were in our examples in PART I. Sixteen pages are in a segment and segments are 64K. Eight bits identify SEGMENT NUMBER. This makes 256 the number of segments that VS2 may allocate in its virtual storage. VS2 allocates virtual storage to jobs in segment size increments. The number of segments in its virtual storage is an important consideration, perhaps even more important than virtual storage size itself. We will expand this consideration during our presentation of OS/VS2.

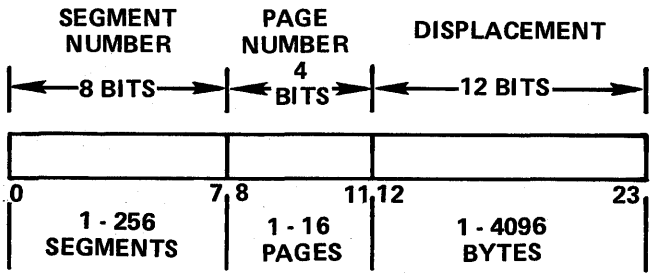


Figure 69

During operation, VS2 is structured in virtual storage, just as OS/MVT is structured in real storage. With VS2, real storage is a system resource allocated to active jobs through demand paging. System libraries and system data sets are stored in auxiliary storage just as in OS/MVT. Only active jobs reside in regions of virtual storage during their execution. And only the most recently referenced pages of an active job are in real storage. When each new job is scheduled, VS2 loads the program for the first job step into a virtual storage region from a system library. Once the program for the first job step is loaded, execution begins using demand paging. We will return to the subject of program loading and execution later. Right now we'll present how VS2 is structured in its virtual storage.

OS/VS2 Structure in Virtual Storage

We divided virtual storage into two areas – the pageable area and the non-pageable area – just as with VS1. Look at Figure 70. There are two attributes that distinguish the non-pageable area from the pageable area. Is it paged? How is it allocated?

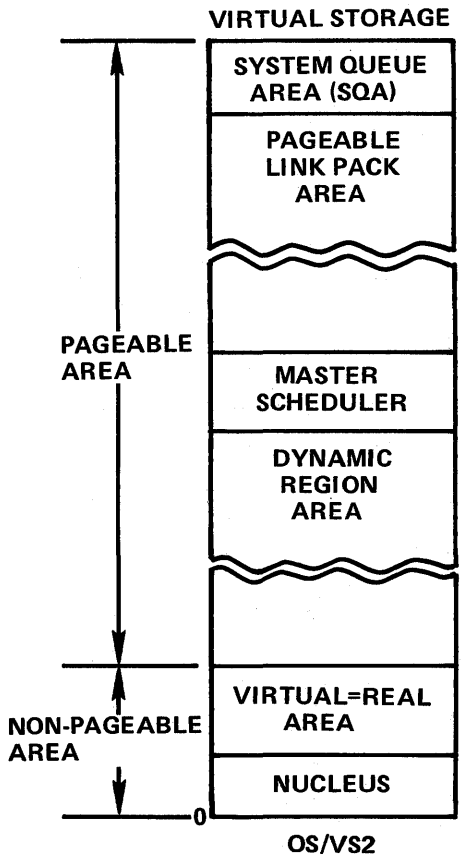


Figure 70

The non-pageable area, when allocated, is fixed in real storage. It is not paged. The non-pageable area of virtual storage is allocated in 4K increments, the size of a page in VS2. It is allocated in page size increments because its pages are fixed in real storage page frames when allocated. In this way VS2 tries not to fix any more page frames than necessary.

OS/VS2 allocates the pageable area of virtual storage in segments (64K) that are paged with one exception. The System Queue Area is allocated one or more segments of virtual storage. SQA pages are allocated dynamically one page at a time. However, when allocated SQA pages aren't paged. They are fixed in real storage. All unused SQA pages represent potential SQA address space, available for SQA expansion during system operation. Jobs in the pageable area, whether the system's or a user's, get their virtual storage in multiples of segments. During execution, they use real storage in page size increments through demand paging. Thus, OS/VS2 uses segmentation to manage virtual storage and paging to manage real storage.

OS/VS2 Components

Nucleus

We will now describe the structure and function of VS2's system components as they appear in virtual storage. Figure 70 shows their location in virtual storage. The nucleus begins at the bottom of virtual storage. It controls the use of all key resources like the CPU and real storage. Minimum nucleus size is 128K. The paging supervisor is a part of VS2's nucleus.

Virtual Equals Real Area

The Virtual Equals Real (V=R) area serves the same function as in VS1. It lets you execute a job step with virtual addresses equal to a contiguous range of real storage addresses. When a job step executes as V=R, all of its pages are fixed in real storage. During execution, addresses are still translated. There is no paging. There is no channel program translation. The V=R area is in the system for jobs with dynamically modified channel programs and for jobs that use time dependent devices like MICR devices. Don't use V=R except in these situations. The V=R area can be allocated to a variable number of job steps. You specify its size at SYSGEN and you can alter its size at IPL time. The upper boundary of the V=R area can never exceed real storage size.

System Queue Area

Let's examine the top of virtual storage. See Figure 70. The System Queue Area (SQA) resides there. You must allocate a minimum of one segment (64K) to SQA. You may allocate additional segments depending on your system's needs. During system definition, you allocate virtual storage to SQA in segment size increments. During operation, VS2 gives real storage to SQA in page frame increments. Pages of SQA are allocated and released dynamically during VS2 operation. Active pages of SQA have a longterm fix in real storage. VS2 uses SQA to hold queues and control blocks that it references during operation. The system's segment table is in SQA.

Pageable Link Pack Area

OS/VS2 has no transient areas for SVC's or Error Recovery Programs. Any system or user routines needed to service your VS2 operation will reside in the Pageable Link Pack Area in virtual storage. These functions are sharable and they are paged. OS/MVT has a resident link pack area, but you must commit real storage to each resident function. When not used, this real storage is wasted. As a result MVT users usually have small link pack areas. Other service routines are executed through transient areas. OS/VS2 has its pageable link pack area. You commit virtual storage to each resident function. When not used, its functions use no real storage. When used they are made available with demand paging. As a result, with VS2 you will have a complete link pack area, in virtual storage, tailored to your needs. There are no transient areas. There is no need for them. The pageable link pack area needs a minimum of 15 segments (960K) of virtual storage. This includes space for directories and code. It will become larger if you include optional functions such as the access methods used in your installation.

Master Scheduler

The Master Scheduler, sometimes called the Master Region, is the communications link between OS/VS2 and the system's operator. It is located below the Link Pack Area in virtual storage. See Figure 70. VS2 needs a minimum of two segments (128K) of virtual storage for the master scheduler. During operation it is paged just like all other system components in the pageable area of virtual storage.

Dynamic Region Area

After the system allocates virtual storage for the VS2 components, what remains belongs to the Dynamic Region

Area, also shown in Figure 70. User jobs execute in this area. VS2 will execute a variable number of jobs in regions in the dynamic region area. Virtual storage is allocated to regions in multiples of segments. If you need 100K for a job, VS2 will give you a region of three segments. Two segments (128K) will be used for your programs. One segment will be used for the *Local System Queue Area (LSQA).* Imagine that you have a job – JOB ONE. It needs 180K of virtual storage to execute its programs. VS2 will give you a region with four segments, three segments (192K) for JOB ONE's programs and one segment for LSQA. This is shown as the shaded area in Figure 71. Notice that LSQA segments are allocated from the top of the Dynamic Region Area. Segments allocated to your job's programs are taken from the bottom of the Dynamic Region Area.

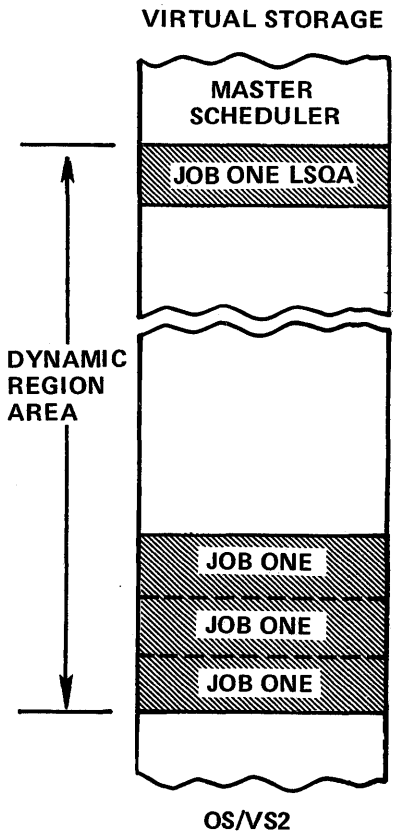


Figure 71

We have used the term Local System Queue Area (LSQA) but we haven't described its function. A job uses LSQA just like VS2 uses SQA, to hold control blocks and queues related to the job. They are referenced and updated by VS2 during job execution. A region's page tables are located in its LSQA. One or more virtual storage segments are allocated to each region's LSQA. Giving each active region its own LSQA makes it somewhat independent in the system during scheduling and execution.

Let's start another job – JOB TWO. It needs 60K of storage to execute its programs. Figure 72 shows JOB TWO loaded in the Dynamic Region Area. JOB TWO's region has two segments. One segment has been allocated for JOB TWO's programs and one segment for LSQA.

You can see from Figure 72 that minimum region size in VS2 is two segments of virtual storage – one for the job and one for LSQA.

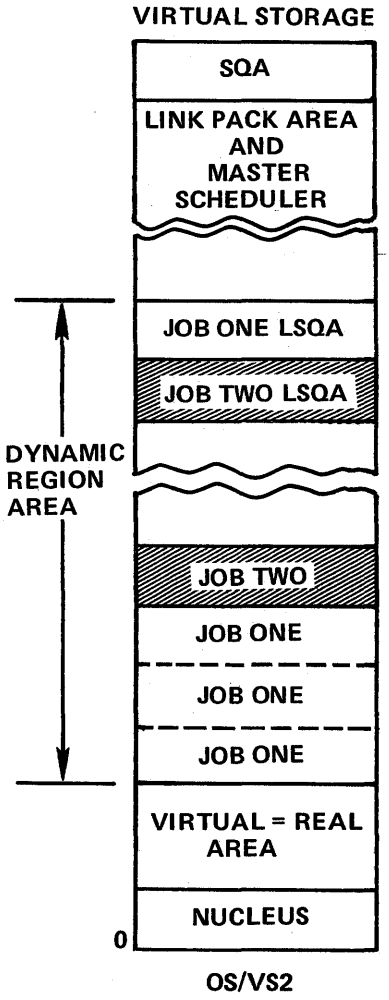


Figure 72

VS2 allocates segments to regions dynamically. The system can schedule jobs until it runs out of virtual storage or until it runs out of regions. The VS2 user controls the number of active regions in the system. You do this simply by controlling the number of initiators that are active in your system. An initiator schedules, initiates and terminates jobs in a VS2 region. If you are executing one initiator you have one active region, two initiators, two regions and so forth. The number of active regions coupled with your jobs' storage requirements determines the amount of virtual storage being used in the Dynamic Region Area at any point in time. If you have three active regions, the first

region with a job that uses 5 segments, the second region with a job that uses 3 segments and the third region with a job that uses 4 segments, 12 segments (768K) of the Dynamic Region Area are allocated. No new jobs can be scheduled – even though virtual storage is available – until one of your active jobs terminates or unless you start another initiator. Let's say that the job in the second region terminates. The system selects a new job from the system job queue and initiates it in the second region. We assume that this new job needs 3 segments. The Dynamic Region Area again has 12 segments (768K) of its virtual storage allocated. Thus, by controlling the number of active regions, you can indirectly control the amount of virtual storage being used in your system. How much virtual storage is actually being used depends on the size of the jobs running in your regions at any point in time.

You can use all of virtual storage if you are using a large number of regions or if you execute jobs with extremely large virtual storage requirements. However, you shouldn't use virtual storage unwisely. You need enough real storage and external page storage in your system to support your active virtual storage. If the working sets of active programs in your VS2 system largely exceed the real storage available for paging, the system will begin to thrash. Excessive paging activity will occur. VS2 monitors its paging rate and will prevent thrashing. But you also can indirectly control system paging activity simply through the number of regions that you use. In fact, through experience you can determine how many regions you can use and still achieve good system performance. This is one advantage of a system like VS2.

OS/VS2 has another advantage for its users. Virtual storage that is not allocated, or allocated but not used, doesn't waste any system resources – real storage or external page storage. Unallocated segments of virtual storage are simply potential address space. Unless allocated for a region segments don't use any system resource, not even external page storage. This is also true for segments that are only partially used. This can happen for several reasons:

1. In a segment allocated to LSQA, the system is only using the first four pages (16K).
2. In a segment allocated to a job, only the first five pages (20K) are actually used. Let's assume that your job has a program that needs 84K. VS2 would allocate two segments (128K) to a region for this job. Only five pages (20K) in the second segment would be used.

These examples show how fragmentation of virtual storage can occur within a VS2 region. No real storage is wasted. No external page storage is wasted. We shall present how VS2 controls external page storage later. At that time we shall show you why fragments, or unused parts of segments, within a virtual storage region cause no waste of external page storage.

Let's take another look at our VS2 example. Figure 72 shows the previous state of the system. We assume that JOB ONE ends, and the system schedules JOB THREE in the available region. It needs 100K to execute programs (two segments) and one segment for LSQA. Figure 73 shows these two changes in our system. Part A of Figure 73 shows the system after JOB ONE ends; PART B of Figure 73 shows the system when JOB THREE begins. We present this example to give you a "slow motion" view of virtual storage allocation to regions in VS2. It should also help you see the advantage of segmentation for managing virtual storage.

The number of segments in your VS2 virtual storage is an important consideration. We have indicated the virtual storage requirements for each major part of VS2. Figure 74 shows virtual storage and it indicates the number of segments allocated to each system component. In Figure 74 we let the nucleus equal 128K and we gave the V=R area 128K. This makes the non-pageable area four segments in size. In the pageable area we have indicated the minimum segment requirements for each component. If you total each system component's segments in the pageable area this results in a total of 19 segments. Add this to the non-pageable area and you have 23 segments or 1472K of virtual storage. There are 256 segments in VS2's virtual storage. In this example, the Dynamic Region Area has 233 segments, about 14 megabytes. Let's assume that you expand the Link Pack Area. You include some access methods and LPA now requires 20 segments. There are still 228 segments in the Dynamic Region Area. You can execute batch jobs in your regions. You can also define TSO regions in virtual storage. TSO users will now share real storage through the paging technique. A larger number of users will be able to share real storage in VS2 than in OS/MVT. They will also be sharing more system programs with the Link Pack area resident in virtual storage. Remember, however, that you must have enough real storage and external page storage to support the amount of virtual storage that you use in your system. We will present how VS2 uses real storage and external page storage in a later topic. Right now, we will present *segment protection* in the VS2 system.

Segment Protection

Segmentation is used very effectively in OS/VS2 to manage virtual storage. Because VS2 allocates virtual storage in segments, it can also provide storage protection on a segment basis. In VS2, when a program is executing it may

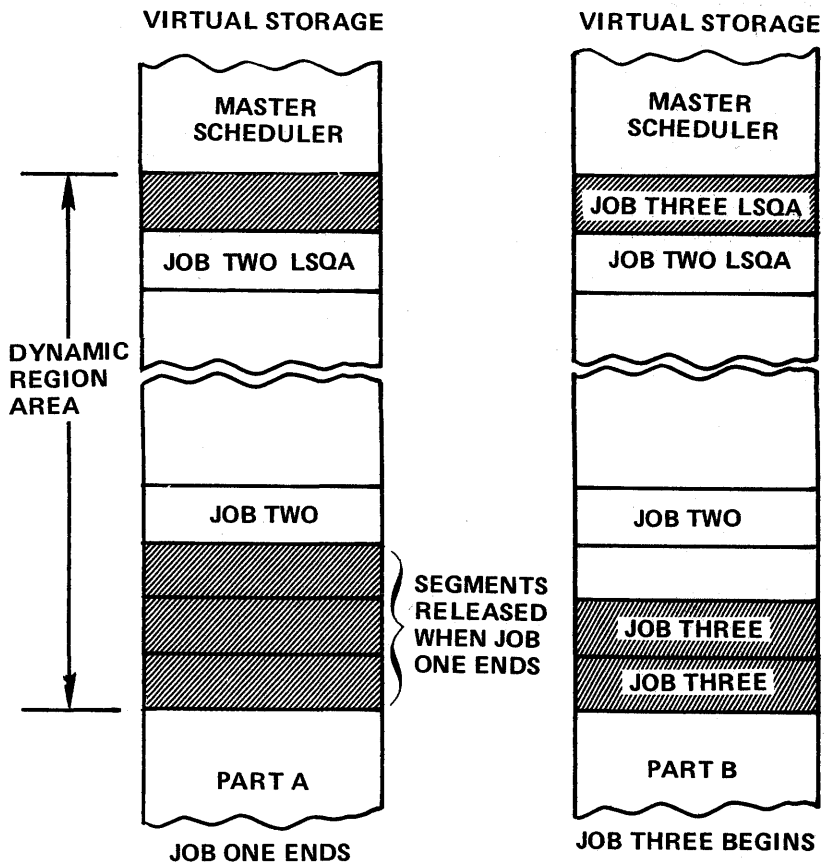


Figure 73

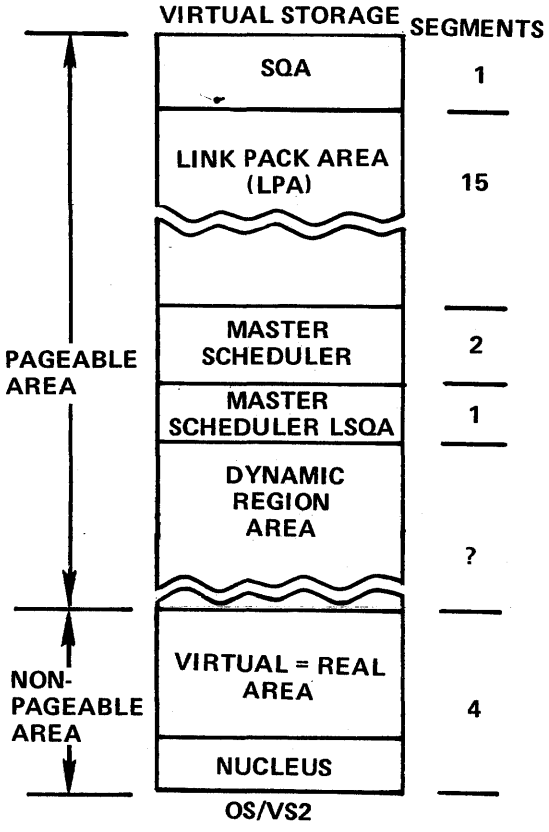


Figure 74

only reference virtual addresses within its segments and the sharable system segments. All other user segments and system segments are protected. Segment protection adds a new level of reliability and security to your system. One user can't inadvertently or knowingly access or alter another user's job or a system component unless authorized by VS2.

VS2 uses its segment table to implement segment protection. Each segment table entry contains a protection indicator called the invalid bit. The invalid bit specifies whether reference to that segment is valid. Virtual address translation occurs through VS2's segment table and page tables. If a virtual address references a segment whose invalid bit is off, translation will proceed. If the invalid bit is on, translation is interrupted and the system reports a protection violation. Let's consider an example. You are going to execute a program called PROGRAM ONE. PROGRAM ONE is loaded starting at virtual storage location 448K – the beginning of SEGMENT 7. Its code needs 160K of storage so VS2 will allocate three segments to PROGRAM ONE – SEGMENT 7, SEGMENT 8 and SEGMENT 9. It would also get a segment for LSQA. When PROGRAM ONE begins to execute, VS2 will turn off the invalid bits in the segment table entries for SEGMENT 7, SEGMENT 8, SEGMENT 9, and the LSQA segment. The invalid bits for

all other segments will be turned on. If PROGRAM ONE references the virtual addresses shown in Figure 75 translation will proceed.

SEGMENT NUMBER	PAGE NUMBER	DISPLACEMENT
8	3	1024
9	2	2048

Figure 75

Segment 8 and Segment 9 have been allocated to PROGRAM ONE. If PROGRAM ONE tries to reference the virtual address in Figure 76 address translation will stop and an interrupt will result.

SEGMENT NUMBER	PAGE NUMBER	DISPLACEMENT
11	6	1024

Figure 76

The invalid bit for SEGMENT 11 is turned on. PROGRAM ONE cannot reference it. What happens if PROGRAM ONE itself is interrupted and another program begins to execute? VS2 will turn on the invalid bits for PROGRAM ONE, turn off the invalid bits for the new program and allow the new program to begin execution. To fully control segment protection VS2 uses two segment tables. When the system is in the supervisor state it uses the first segment table for virtual address translation. All of the invalid bits are turned off in this segment table. This lets supervisor code – the nucleus and so forth – reference all of virtual storage in VS2. When the system is in the problem program state it uses the second segment table for virtual address translation. Only the invalid bits related to the executing problem program and any sharable system programs are turned off in the second segment table. This implements the method that we just described using our example of PROGRAM ONE. Thus, an executing problem program may only reference its own segments or sharable system programs in the Link Pack Area. Segment Protection gives VS2 an excellent hierarchy of protection to the system and its users.

Until now, we have presented the VS2 system as it is structured in its virtual storage, VS2's address space. We will now present how VS2 uses two of its key resources – real storage and external page storage. We will begin with real storage.

Real Storage in OS/VS2

In Figure 77, VS2 is structured in its virtual storage and mapped onto System/370 real storage. With demand paging, page frames of real storage are shared by pages of active system and user jobs in the pageable area of storage. Notice that the VS2 nucleus is fixed in real storage. The nucleus is fixed all during system operation. This assures rapid system response to service system and user jobs. If a job step is run as "virtual equals real" all of its pages, allo-

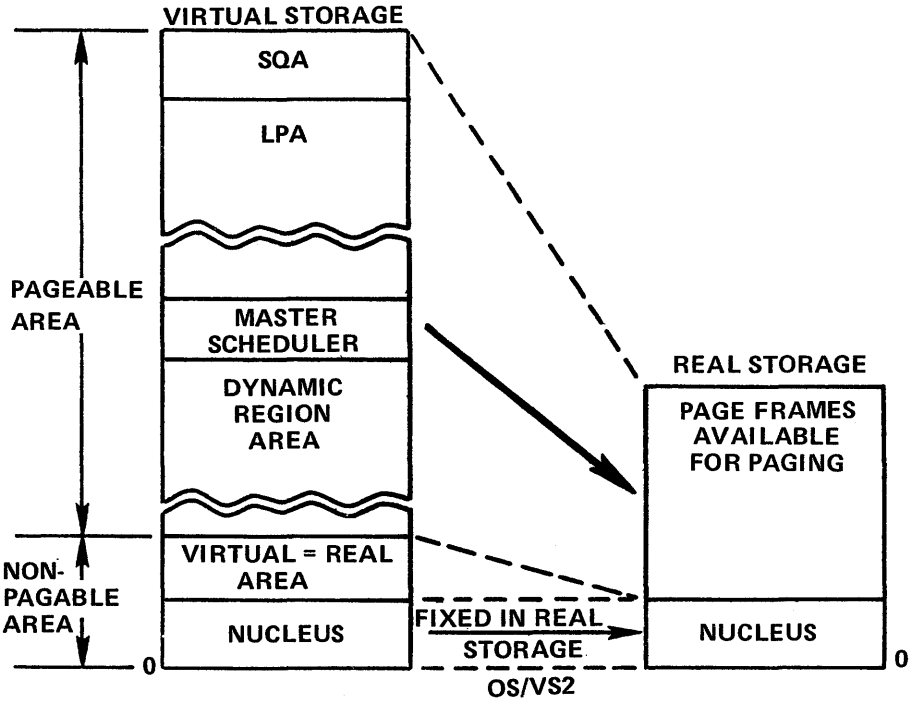


Figure 77

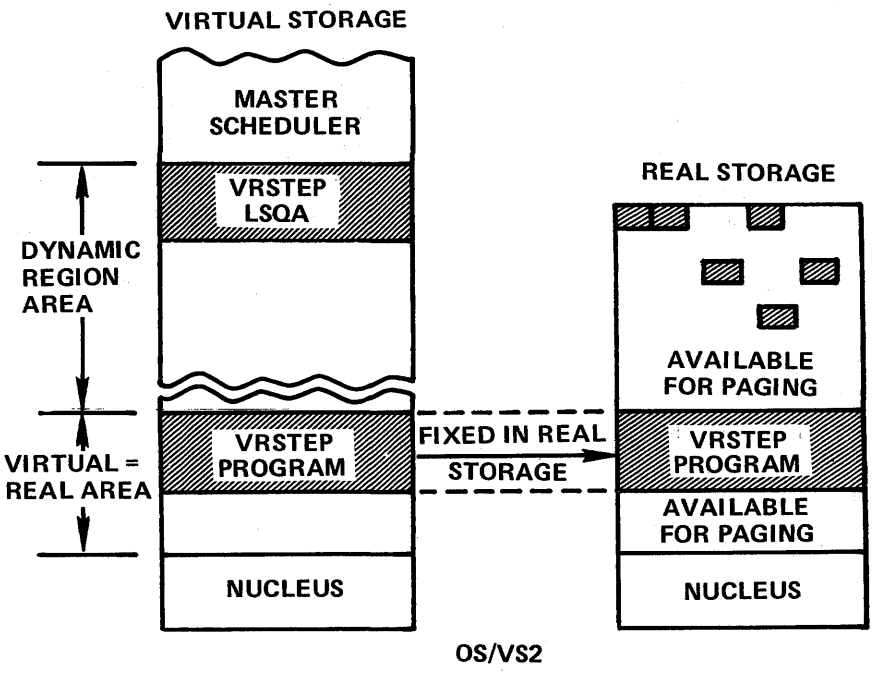


Figure 78

cated in 4K increments, are fixed in corresponding page frames of real storage during execution. They also must be contiguous. The effect of a virtual=real job step is shown in Figure 78.

The V=R job step's pages are fixed in corresponding real storage page frames and the job step's virtual addresses equal its real addresses. This is shown as the shaded area in real storage and virtual storage called VRSTEP program. Even though a job step is V=R, it has a segment of virtual storage for LSQA. Active pages from LSQA are indicated as the small shaded blocks in real storage page frames in Figure 78. A V=R job step must be scheduled, initiated and, after execution, terminated from the Dynamic Region Area. Even though a V=R job step fixes a set of page frames during its execution, the remaining page frames of real storage are available for demand paging among other active jobs in the pageable area of virtual storage.

External Page Storage

If an active page is not in real storage, where is it? In VS2, it's in a slot of external page storage. Where in external page storage? VS2 knows where by referencing an external page table that is located in real storage. What about unallocated segments from your Dynamic Region Area of virtual storage? Where are they? They don't use any external page storage in your VS2 system. Unallocated segments are potential address space. They don't use any physical system resource. Unlike VS1, VS2 uses virtual storage and external page storage as we described it in PART I. Virtual storage is not mapped one-to-one with external page storage. Instead, external page tables are used to map the slot locations of pages in external page storage. In this section we will discuss external page storage in VS2.

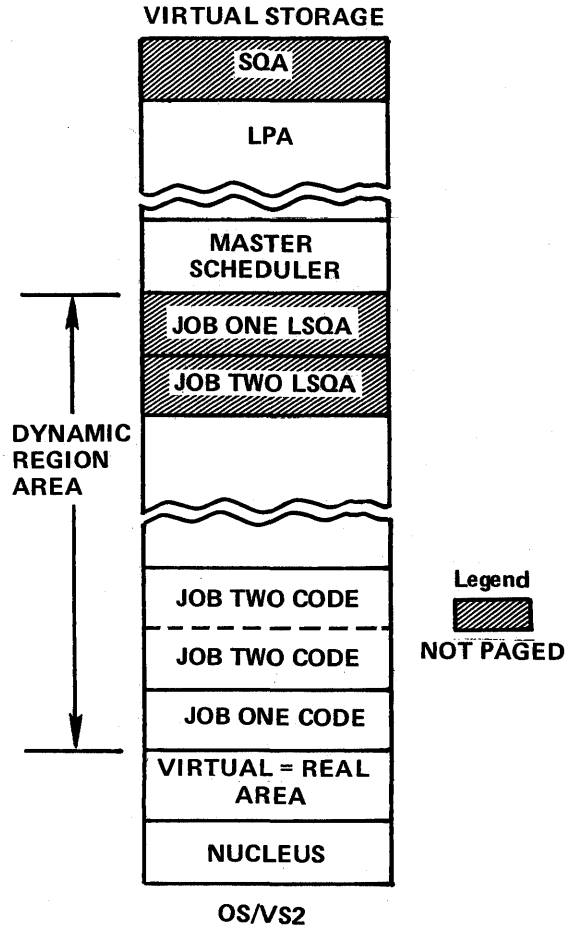


Figure 79

We defined paging as the transfer of pages between real storage and external page storage. During a page-in operation, a page is moved from external page storage into real storage. During a page-out operation, a page is moved from real storage to external page storage. If part of virtual storage is not paged, no external page storage is needed to back it. In VS2, several parts of virtual storage are not paged. Examine Figure 79. The shaded areas of virtual storage indicate parts of OS/VS2 that are not paged. The nucleus is always fixed in real storage during operation. If a job step is run as V=R, its pages are fixed in real storage during execution. Pages allocated in SQA segments and some pages allocated in LSQA segments get a long-term fix. Because these parts of VS2 are not paged, no external page storage is needed to back them.

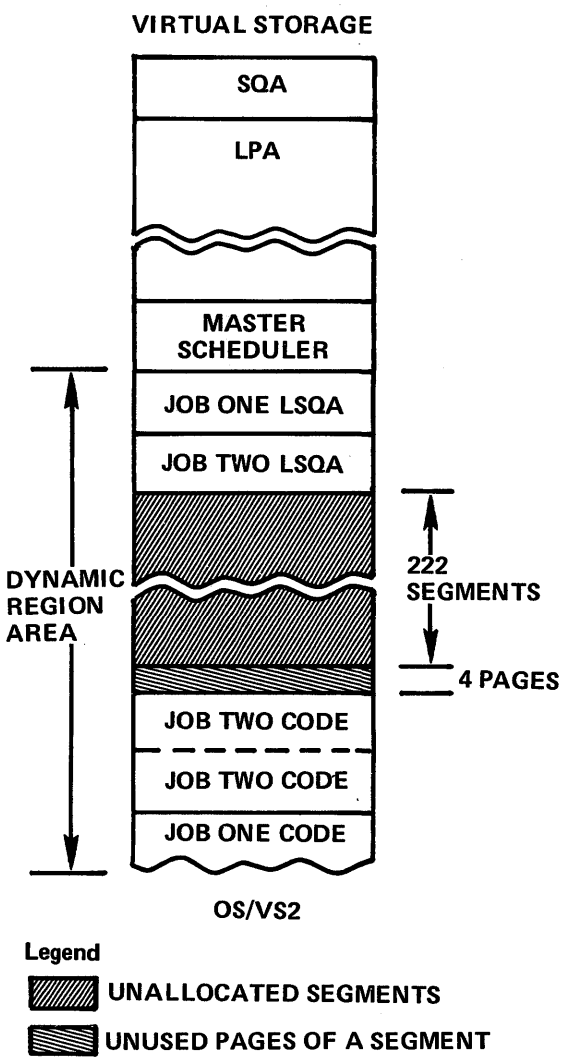


Figure 80

What about those components of VS2 that are paged? This represents the greater part of the system that is structured in the pageable area of virtual storage. System components – the Link Pack Area, and the Master Scheduler – are paged. They are initialized during IPL. During operation they will use external page storage. The Dynamic Region Area is paged. All active regions will use external page storage for paging. However, unallocated segments of the Dynamic Region Area will not use external page storage. Also, any unused pages in a segment will not use external page storage. Unallocated segments and unused pages within segments represent "potential" address space to VS2. Figure 80 is a picture of the pageable area of virtual storage. JOB ONE and JOB TWO are executing in regions in the Dynamic Region Area. The 222 unallocated segments don't use any external page storage. They represent unused virtual storage, or potential address space. If a new job is scheduled in a newly allocated region – let's say three segments for the job's code, and one segment for LSQA – potential virtual storage is reduced to 218 segments. The three segments used for the job's programs will begin to use external page storage as paging occurs. Also notice, in Figure 80, the shaded area within the last segment allocated to JOB TWO's code. We assume that JOB TWO's code needs 112K. Therefore, 16K (or 4 pages) of the second segment allocated to JOB TWO won't be used during execution. If these 4 pages are not used, they will not use any external page storage during JOB TWO's execution. VS2 uses external page storage dynamically during system operation. You must allocate enough space to external page storage to support the amount of virtual storage that you expect to use. However, at any point in time during system operation, external page storage may be using much less space than what it has available.

External Page Tables

In PART I of this text we introduced the concept of an external page table. An external page table maps the locations of pages in slots of external page storage, while a page table maps the location of pages in page frames of real storage. Each page table has an associated external page table. If a page fault occurs during virtual address translation, the system uses this corresponding external page table to locate the referenced page in external page storage. It may then be paged in. If a page replacement operation occurs and the replaced page has changed, it must be paged out. VS2 can page out to any available slot in external page storage. It will try to select a slot that requires the shortest time to transfer the page from real storage to external page storage. When the page out occurs, VS2 will mark the appropriate page table entry invalid and place the slot location in the corresponding external page table entry. Thus, VS2 uses external page tables to control its external page storage.

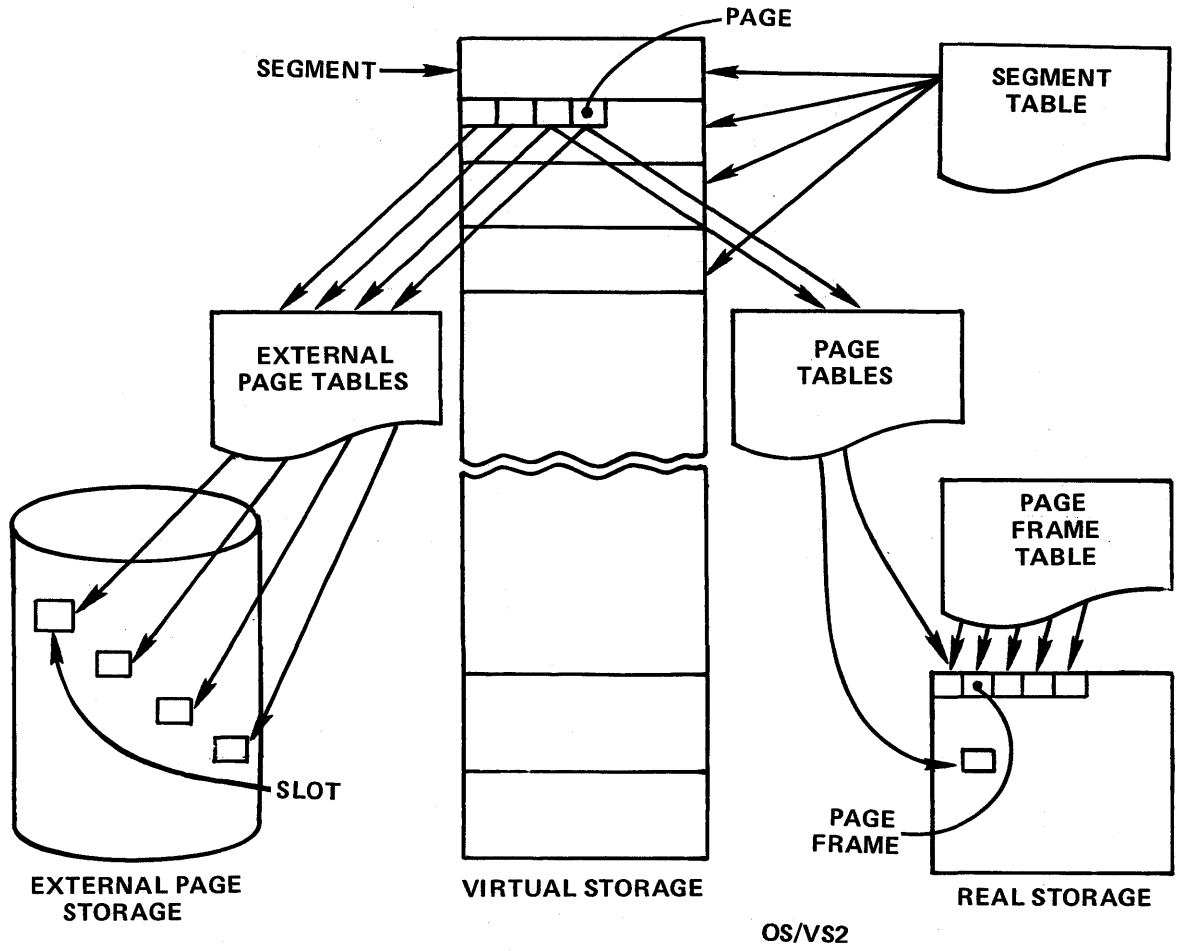


Figure 81

VS2 uses the DAT feature and various tables to implement virtual storage. Figure 81 shows how these tables relate to each type of storage in the system. The segment table and page tables are used for dynamic address translation. In addition to address translation, the segment table is used to manage virtual storage allocation. Page tables and their corresponding external page tables are used to control the transfer of pages between real storage and external page storage – paging. The page frame table is used to manage real storage allocation. Through its tables and the System/370 DAT feature, OS/VS2 implements virtual storage.

Program Loading in OS/VS2

What happens after a job is scheduled into a region of OS/VS2? How is the program for the first job step, as well as programs for subsequent job steps and jobs, loaded into its virtual storage region? Program loading in VS2 is very similar to program loading in OS/MVT. The big difference is this: in VS2, programs are *loaded into virtual storage.*

In OS/VS2, programs are stored in libraries in the same way as in OS/MVT. Let's assume that we are going to load a program called Program One into Region 1. Program One is stored in a VS2 system library. It was put there, after compilation or assembly and a link edit operation with relative addresses. Its addresses are related to a zero origin. During loading, the VS2 program loader will relocate Program One's addresses from their zero origin – as stored in the system library – to the starting location of Region 1 in virtual storage. This is a form of static relocation. Figure 82 shows this process conceptually. During program loading, Program One acquires a segment – page format. It is loaded beginning at a segment boundary. During loading, most of Program One will be paged out to external page storage. Even though Program One's addresses have been relocated at load time, they are still relative. Now they are virtual addresses related to virtual storage's segment-page structure. When program loading is complete, PROGRAM ONE will begin to execute. The System/370 DAT feature translates PROGRAM ONE's virtual addresses dynamically and automatically during execution. VS2 loads PROGRAM ONE's pages into real storage, as referenced during execution, with

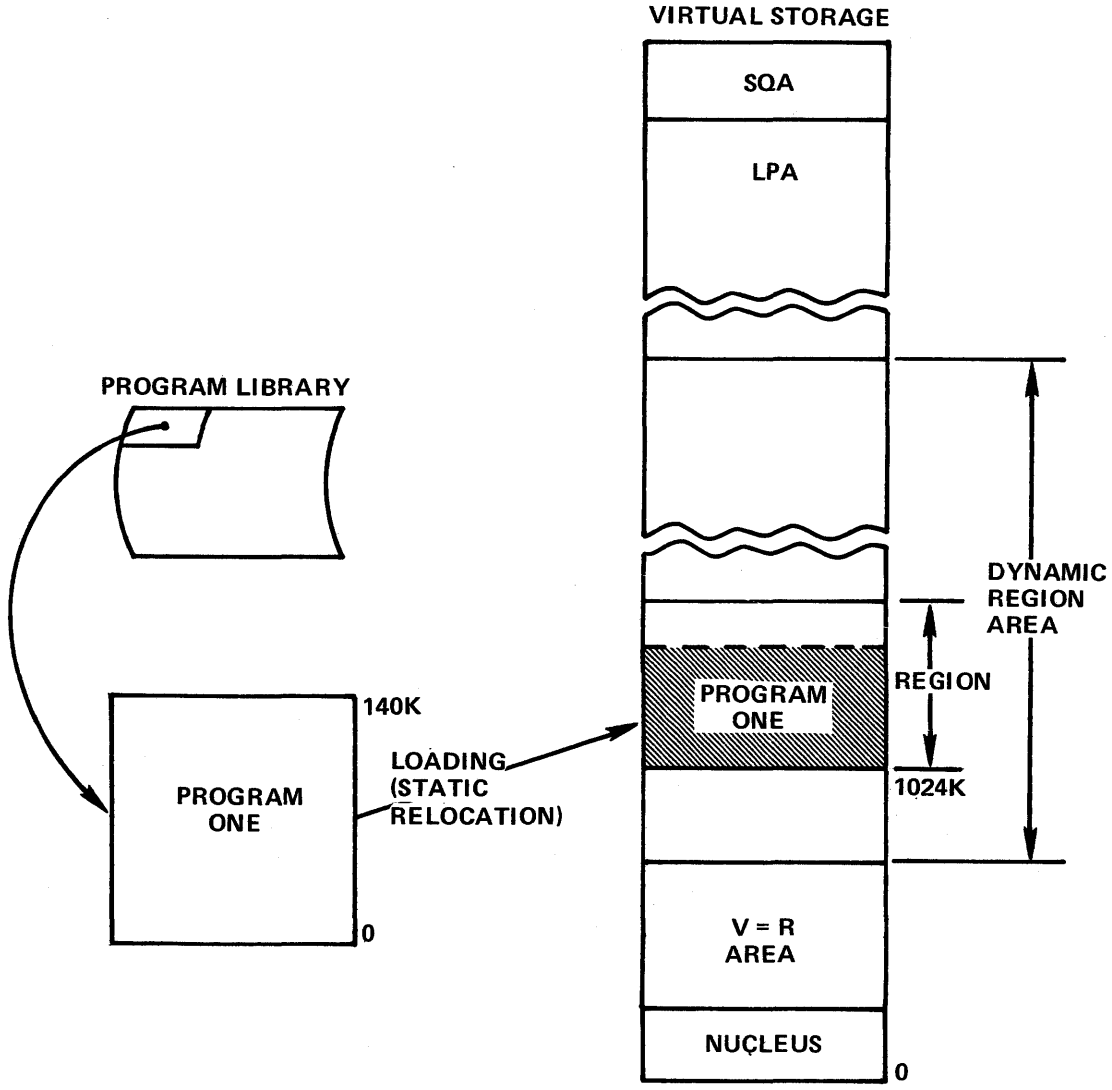


Figure 82

demand paging. Thus, VS2 uses two types of relocation and demand paging. Programs are loaded into virtual storage using a type of static relocation. During execution, System/370 dynamically translates a program's virtual addresses with its DAT feature. VS2 shares real storage among all active problem programs with the demand paging technique.

In Figure 82, we presented the conceptual process of program loading in VS2. We will now describe briefly some of the events that happen during program loading. We will continue to use PROGRAM ONE as an example.

At load time PROGRAM ONE is in a system library. VS2 has a system program, called PROGRAM FETCH, that will load PROGRAM ONE into its virtual storage region. During the load process, PROGRAM FETCH is executing, PROGRAM ONE is its data. PROGRAM FETCH reads PROGRAM ONE as input data (from the system library) and relocates PROGRAM ONE's address to their starting point in virtual storage. PROGRAM FETCH operates just as it does in OS/MVT. However, in VS2, PROGRAM FETCH relocates a program's addresses to virtual storage locations; in OS/MVT, PROGRAM FETCH relocates a program's addresses to real storage locations. PROGRAM ONE gets its segment-page format during the loading process. Because it is loaded into a virtual storage area that has a starting location on a segment boundary (all VS2 regions begin at a segment boundary) it will begin at a segment origin and use one or more segments of virtual storage. As PROGRAM ONE is loaded, PROGRAM FETCH reads it into page frames of real storage. Through demand paging, PROGRAM ONE will be paged out to external page storage. Thus, PROGRAM ONE receives its segment-page format. It is during the loading process that PROGRAM ONE's page tables and external page tables are built.

Figure 83 depicts the loading process for PROGRAM ONE using a static relocation technique. All programs in

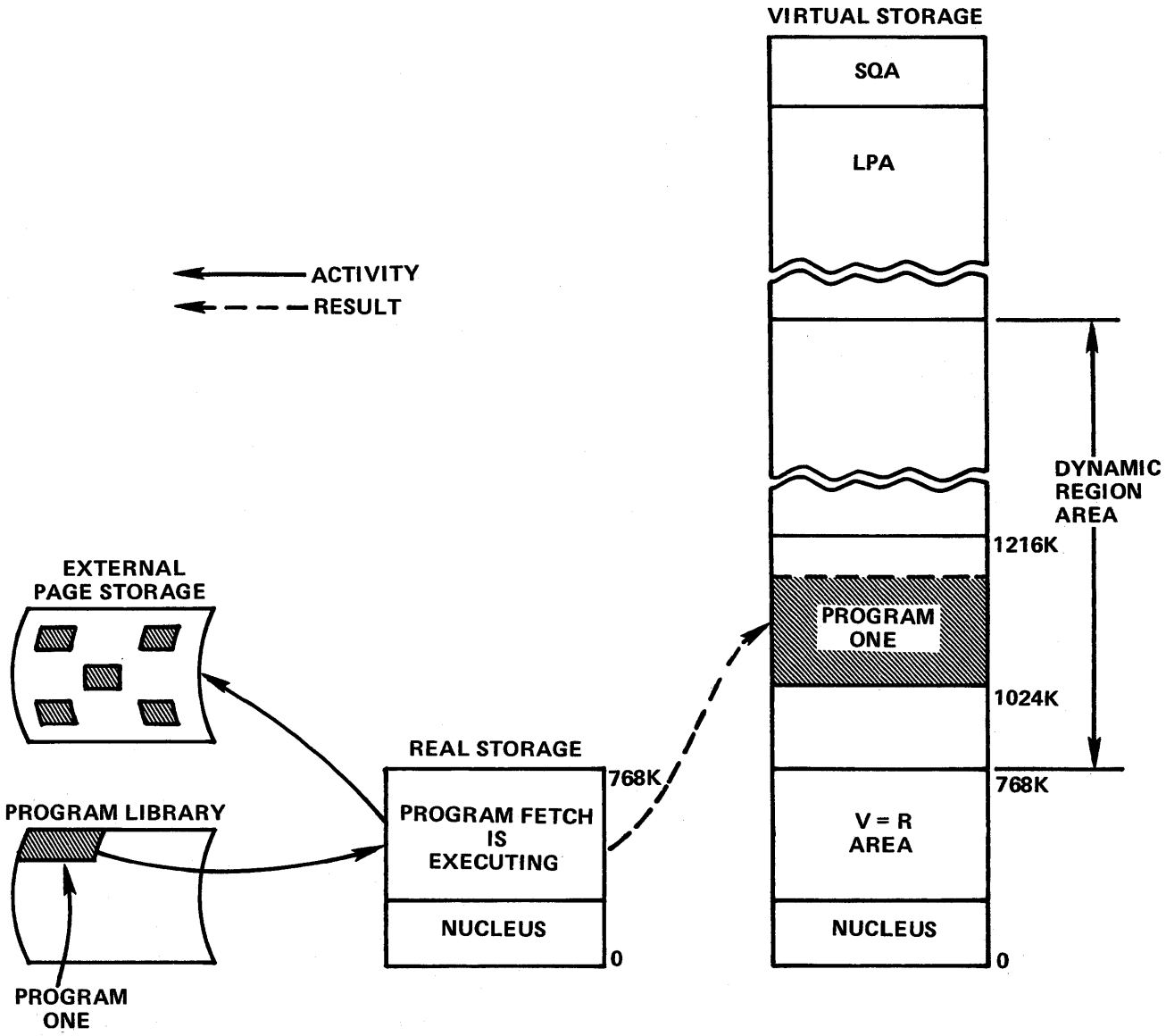


Figure 83

VS2 are loaded in this way. When the loading process is finished, PROGRAM ONE will begin to execute using dynamic address translation and demand paging.

In Lesson 10 we introduced the OS/VS system. We stated several new benefits for users of OS/VS. In this lesson we have described Release 1 of OS/VS2, its structure, and how it implements a single virtual storage to achieve these benefits. The key to the VS2 system is its dynamic response to user needs. Release 2 of OS/VS2 surpasses Release 1 by incorporating multiprocessing and multiple virtual storage support into the VS2 system. Lesson 14 presents how multiple virtual storages or multiple address spaces are implemented in Release 2 of OS/VS2.

Release 2 of OS/VS2 (VS2/2) is a multiprocessing system that supports multiple virtual storages. As such, VS2/2 extends the virtual storage capability of Rel 1 to give each user his own address space. This is implemented in a manner similar to the technique described in PART I of this text.

VS2/2 completely integrates TSO into its architecture, and multiprocessing support is standard in the system. Multiprocessing support is symmetrical, that is two or more CPU's of the same model share the same real storage resource. Another way to describe this type of multiprocessing support is to say that the CPU's are tightly coupled. Although VS2/2 supports multiprocessing as standard, it can operate with one CPU as a uniprocessor. In the remainder of this lesson, we will assume a uniprocessor system and describe how VS2/2 implements multiple virtual storages.

VS2/2 Structure

In VS2/2, each user has his own address space or virtual storage. By user, we mean a batch or TP job that has been scheduled by an initiator, a logged on TSO user, and certain system components that have their own address space. For example, VS2/2's Master Scheduler resides in its own address space. For each job initiator that is started, the system creates an address space. The initiator can then schedule jobs in the address space. When each new TSO user logs on, VS2/2 creates a new address space for that user. The VS2/2 nucleus controls all system resources – the CPU(s), real storage, channels, I/O devices and so forth – and provides user services such as address space creation and CPU dispatching.

The structure of each address space in VS2/2 is the same as the single address space in VS2 Rel 1. Each virtual storage is 16 megabytes in size and divided into 256 64K segments. Page size is 4K. Each virtual storage in the system is mapped by a segment table, with page tables for all allocated segments. These segment and page tables reside in real storage. To pass control to a particular user, the nucleus must first load the System/370 STOR register with the real storage origin of the segment table that maps the user's address space.

Although each VS2/2 user has his own address space, he doesn't control the entire address space. Each virtual storage is divided among a *private area*, a *common area* and the nucleus. This is shown in Figure 84.

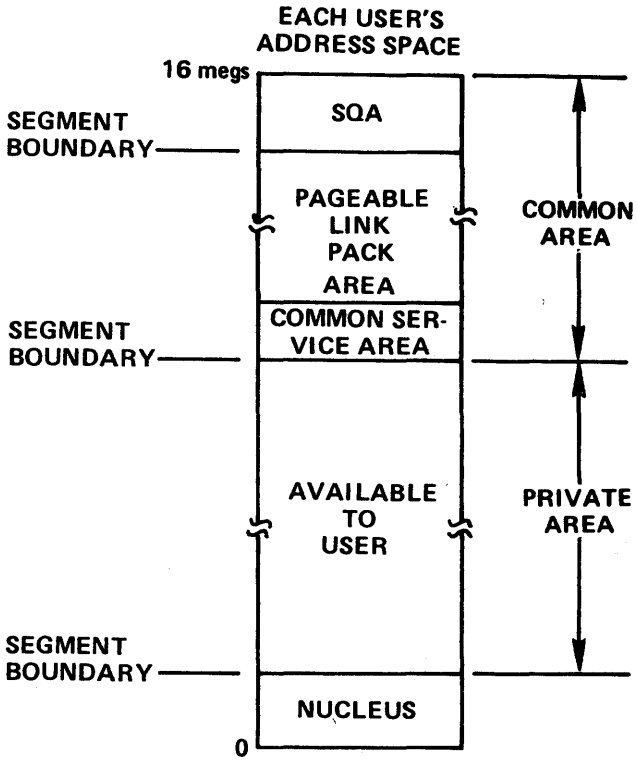


Figure 84

The nucleus is fixed in real storage and then mapped in each virtual storage on a one-to-one basis. Nucleus size is a multiple of segments. Thus, the lower segments of each address space map the nucleus. This is done by using the segment sharing technique described in PART I of this text. The segment table entries that map the nucleus in each virtual storage point to a set of common page tables. These page tables map the nucleus page locations in real storage. Thus, there is only one copy of the nucleus in the system and it is fixed in real storage for performance purposes.

The *common area* contains the System Queue Area (SQA), the Pageable Link Pack Area (PLPA), and something new in VS2/2 called the Common Service Area (CSA). As in VS2 Rel 1, SQA contains tables and queues related to the entire system. It is used by routines resident in the nucleus and other system components as a working storage area. For example, SQA contains a table which identifies each address space that has been created in the system. SQA also contains the page tables that map the nucleus and the common area itself. Minimum SQA size is two segments, 128K. An installation specifies SQA size (if more than two segments are required) as a multiple of segments. SQA space is used dynamically during system

operation, one page at a time. As each new page of SQA is used, it is fixed in real storage automatically by the system. The SQA segments specified by an installation are mapped in each address space in the same manner as the nucleus.

The Pageable Link Pack Area (PLPA) in VS2/2 performs the same function as in VS2 Rel 1. In VS2/2, the PLPA is mapped in each address space through segment sharing. Thus SVC's, access methods, and other *READ-ONLY* system programs, and any *READ-ONLY* user programs, which may be selected by an installation, are placed in the PLPA. These programs can be shared among all users in the system, but, because they are demand paged, they only use real storage as referenced. To enhance VS2/2 reliability PLPA is duplexed in external page storage. Thus, there are two copies; the primary copy is normally used for demand paging PLPA. The secondary copy is a backup, only used if a read error occurs during page-in from the primary PLPA. Because programs in PLPA must be READ-ONLY, there will be no page-out operations to PLPA and this duplexing approach is an effective reliability technique. Although mapped in each address space, only the two duplexed copies of PLPA programs are in the system because of the segment sharing technique used. As a result, external page storage is only required to back two copies of these programs. If a copy of PLPA were in every user's address space the external page storage required would be a multiple of the number of address spaces active in the system. More real storage would also be used. For example, if two or more users were using the same access method, multiple copies of the same page(s) could reside in real storage. Segment sharing then results in many system efficiencies in VS2/2.

Minimum PLPA size (which is specified as a multiple of pages) is about two megabytes. Its size will vary from installation to installation depending on system program options or user-written programs selected to reside in PLPA. Remember that programs placed in PLPA must be READ-ONLY and should be applicable to multiple users, since each program placed in PLPA will expand the common area size and reduce the size of the private area in each user's address space.

The Common Service Area (CSA), also shown in Figure 84 is new in VS2/2. Because users may only reference within their own address space, VS2/2 must provide some way for one user to communicate with one or more other users. CSA is common to all address spaces. It can hold information for inter-user communication in VS2/2. This function of CSA replaces the inter-region communication function in VS2 Rel 1. Allocated CSA pages are demand paged by VS2/2 during system operation. CSA space may also be used by the VS2/2 nucleus for control blocks that don't require a long term fix in real storage.

The private area in each VS2/2 address space replaces the concept of the region in VS2 Rel 1. Each user's private area contains all of the segments that remain after the common area and the nucleus are mapped. Thus, all executing users, whether batch, TP, or timesharing, have the same amount of address space. The JCL REGION parameter, although still used, no longer has the same interpretation (unless a job step is executed V=R). Rather than bind the address space assigned to a user, the REGION parameter in VS2/2 only controls the amount of address space that a user may obtain in his private area using a variable GETMAIN.

Figure 85 shows the structure of the private area in each address space. Space is assigned to user programs from the bottom up. Space is dynamically assigned to the *Local System Queue Area (LSQA)* and something new in VS2/2 called the *Scheduler Work Area (SWA)* from the top down. Space is assigned to LSQA and SWA in page size increments (4K blocks).

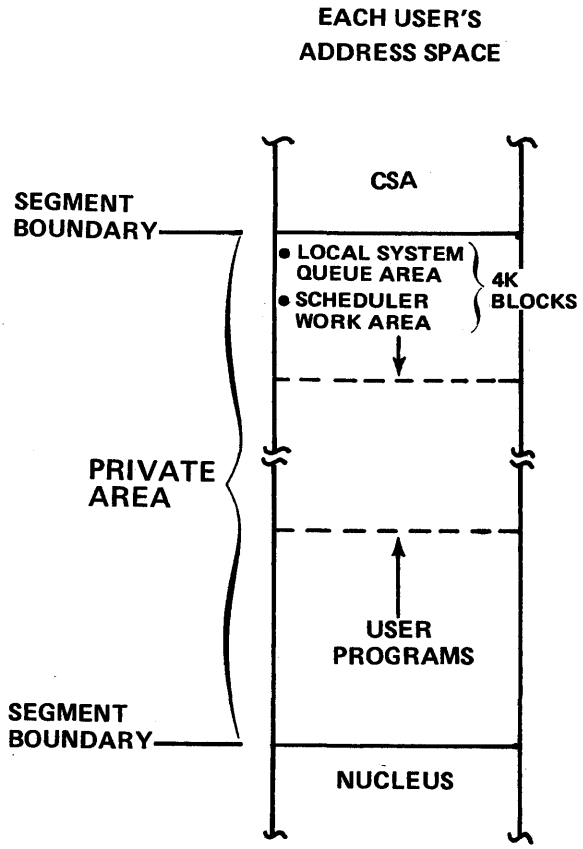


Figure 85

As in VS2 Rel 1, LSQA in VS2/2 is used for tables and queues associated with a user's job and address space. For example, the segment table for an address space and page tables for all allocated segments within the private area are stored in LSQA. As LSQA pages are allocated they are

fixed in real storage. They remain fixed unless a user is "swapped out" or they are released by the system. We will describe "swapping" in VS2/2 later. In addition to LSQA, a VS2/2 user can allocate space in page size increments from special storage subpools at the top end of the private area. This space may be used for control blocks that can be demand paged during operation.

In MVT and VS2 Rel 1, the control blocks and tables used by the initiator during job step scheduling are contained on the system job queue device. If four initiators are scheduling jobs, contention can result when two or more initiators require scheduling information from this single device. In VS2/2, the Scheduler Work Area (SWA) reduces this contention for the system job queue by storing the control blocks and tables used by the initiator during job step scheduling within virtual storage. Each address space dynamically allocates SWA space (in page size increments) as required. The allocated SWA space is used to store the tables and control blocks created during JCL interpretation (which occurs in VS2/2 after a job is selected, not when it is read into the system). Each initiator then has its own scheduling work area within the private area of its address space. During job step initiation or termination, the initiator obtains job control information from SWA. Pages allocated to SWA are demand paged and only come into real storage when they contain scheduling data referenced by the initiator.

As we said earlier, space is dynamically allocated to LSQA and SWA in page size increments starting at the top of each user's private area. The remainder of each private area is available to its user, with space being allocated from the bottom up. All user jobs are demand paged unless a job step is executing as V=R (which will be described later). Until now we have described the characteristics of virtual storage in VS2/2. For you to better appreciate how VS2/2 implements multiple virtual storages we will now describe the operation of the system.

VS2/2 Operation

System Initialization

When the operator IPL's VS2/2 several events occur:

- The nucleus is fixed in real storage.
- Three address spaces are created and the nucleus is mapped in each.
- SQA is allocated in the first address space and mapped in the second, third and all subsequent address spaces.
- PLPA is loaded (if this is a cold start) in the first address space and mapped in the second, third and all subsequent address spaces.
- The Master Scheduler is loaded in the private area of the first address space.
- The Auxiliary Storage Manager (ASM) is loaded in the private area of the second address space. ASM in VS2/2 controls all paging I/O and manages the allocation and use of external page storage.
- The VS2/2 Job Entry System (JES) is started in the private area of the third address space. JES will be further discussed later.

During IPL, the system communicates with the operator to establish various operation parameters. When IPL is complete, VS2/2 appears as shown in Figure 86.

Referring to Figure 86, several observations may be made about the system. SQA pages are fixed in real storage as they are allocated. SQA contains the common page tables that map the nucleus, PLPA, CSA and SQA itself. Since SQA pages receive a long term fix in real storage, they don't need to be backed by external page storage.

PLPA, however, is backed by external page storage and is demand paged. Remember, for reliability there are two copies of PLPA in external page storage. They are mapped by the common page tables in SQA. The appropriate segment table entries of each address space created point to these common page tables in SQA. This is also true of SQA, CSA and the nucleus. The nucleus, like allocated SQA pages, is fixed in real storage, and therefore needn't be backed by external page storage.

The Master Scheduler, also shown in Figure 86, contains the VS2/2 communications task. Once initialized, the communications task allows the operator to start one or more initiators and thus create additional address spaces for batch job processing. With the Master Scheduler loaded, the operator may also start TSO operations. The Master Scheduler then forms the main communication link between VS2/2 and the system operator.

The third address space in Figure 86 shows the VS2/2 *Job Entry System* (JES) loaded in the private area. One prime function of the Job Entry System (JES2) is the integration of the VS2 Rel 1 HASP option into VS2/2. With this function JES2 spools batch jobs from local and remote work stations. Once JES2 is active, jobs may be submitted for processing from these work stations. The VS2/2 system is ready to begin batch operations. JES3, not shown in Figure 86, incorporates the features of ASP into VS2/2.

VS2/2 – Batch Job Processing

For VS2/2 to process batch jobs, the oprerator can start one or more initiators. Each initiator will then schedule jobs from the system job queue based on job class and

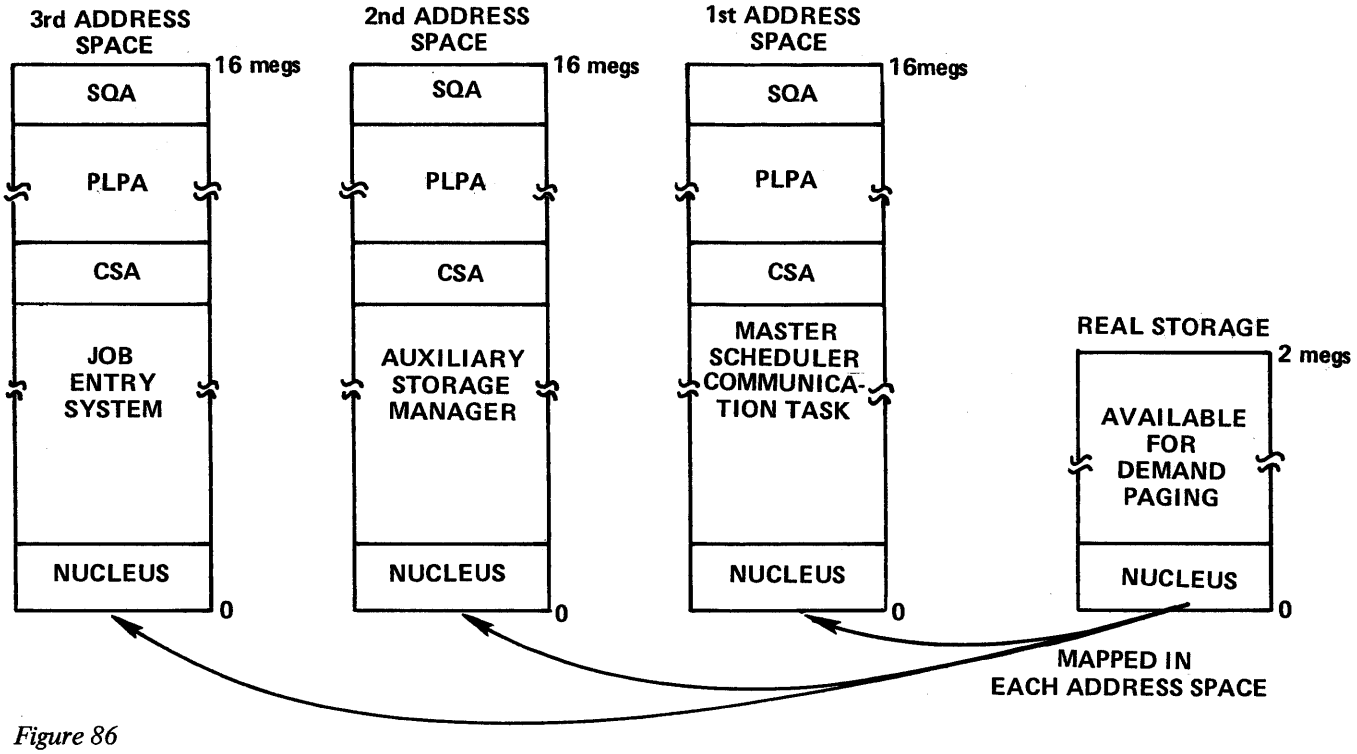


Figure 86

priority assignment. When the START command is issued by the operator to start an initiator, VS2/2 creates a new address space. The nucleus and the common area are mapped through the segment table of this address space and the START command is then processed within it. The START command loads a copy of the initiator into the new address space and it is then ready to schedule jobs. When the first job is selected, its first job step is scheduled and its programs are loaded into the private area of new address space. During execution, the job step controls the private area of the address space and it is demand paged, sharing real storage with other jobs and system components. When each job step ends, the initiator is reloaded to schedule the next job or job step for its address space. While the initiator is executing it also is demand paged.

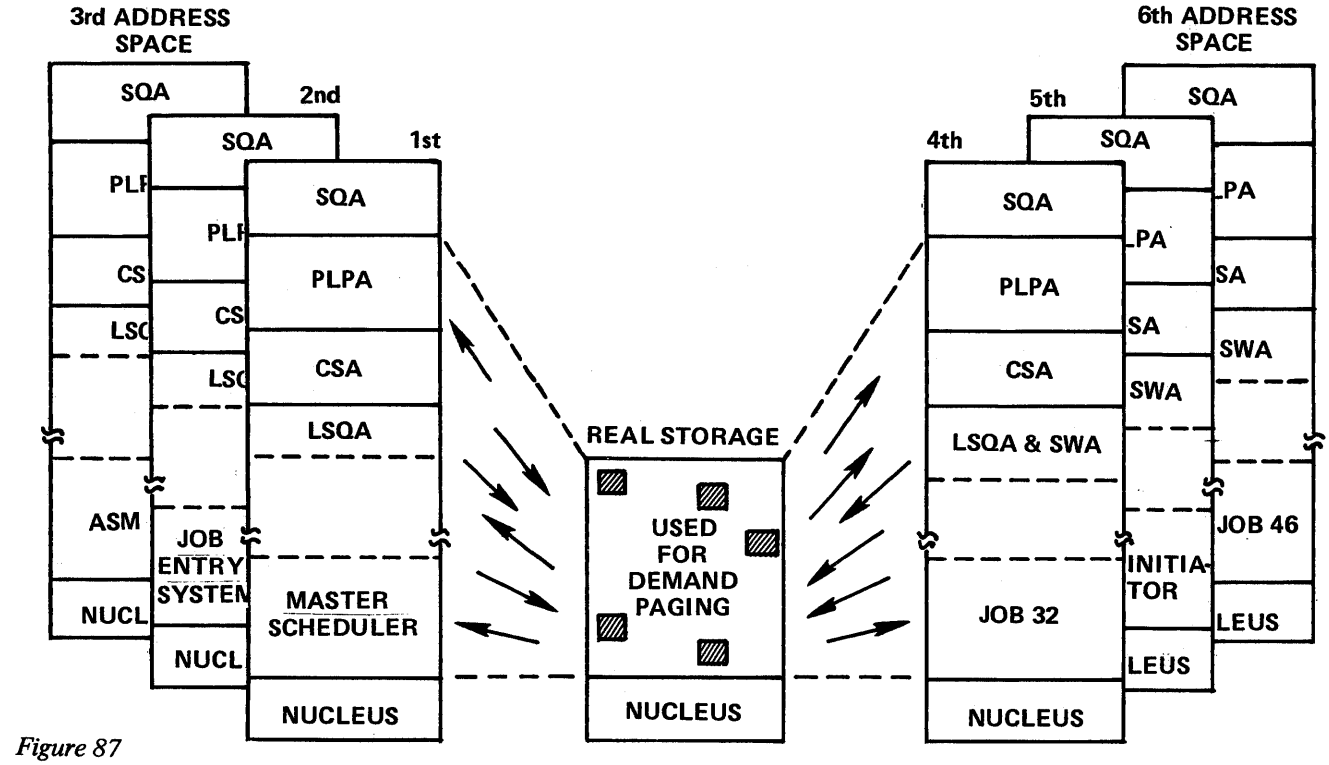


Figure 87

The number of initiators active in VS2/2 determines the degree of multiprogramming within the system and the number of address spaces available for batch job processing. With one initiator started, only one job stream may be processed. To increase the degree of multiprogramming another initiator (and, thus, address space) must be started. Figure 87 shows an example in which three initiators have been started. Job steps are executing in two of the address spaces. The initiator in the third address space is scheduling a new job.

Demand paging controls the real storage allocation for active programs in the private area of each address space. External page storage (not shown in Figure 87) must be large enough to back the requirements of each user's address space and the common area that is mapped in each address space. As in VS2 Rel 1, there is an external page table associated with each page table in VS2/2. These external page tables map active pages, identifying their slot locations in external page storage.

V=R Job Steps

Demand paging is the normal mode of operation for user programs and most system programs that execute in VS2/2. However, as in Rel 1 there are certain types of programs that, due to time dependencies or dynamic channel program modification, must be executed as Virtual = Real (V=R). As a result, VS2/2 has an option for V=R job steps. At IPL time, the system operator may define a V=R area for the execution of V=R job steps. If selected, the size of the V=R area is specified as a multiple of 4K. Control and allocation of the V=R area is maintained by VS2/2's nucleus. When the V=R option is selected, the V=R area is available in real storage beginning 4K above the nucleus. This is indicated in Figure 88 which shows a one megabyte real storage with a 256K nucleus and a 256K V=R area specified beginning at 260K.

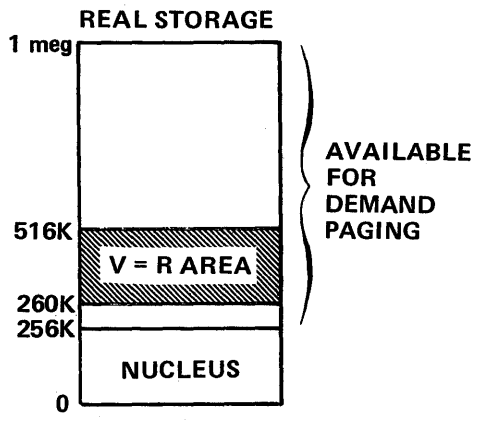


Figure 88

Notice that all of real storage above the nucleus is available for demand paging. The V=R area, although specified, does not reserve real storage. It simply indicates the area of real storage that may be used for V=R job steps. If no V=R job steps are being executed, VS2/2 uses the real storage for demand paging. However, VS2/2 does try to avoid long term page fixes within the V=R area to prevent fragmenting it.

When a job step is specified V=R, this is detected by the initiator during job step scheduling within an address space. The job step's execute statement contains the V=R parameter and the region size required. The initiator then passes control to the nucleus so that it may attempt to satisfy the V=R space request. The nucleus tries to find a contiguous space within the V=R area large enough to satisfy the region request. If the V=R area is fragmented by pages with a long term fix or by another V=R job step so that the required space is not available, the operator is notified and the requesting job step must wait. If the space is available, the V=R job step may be scheduled. Figure 89 shows an example of V=R job step scheduling and execution with active initiators in two address spaces.

In Figure 89 ADDRESS SPACE ONE has scheduled the V=R job step. ADDRESS SPACE TWO is executing a batch job that is being demand paged. Other system components are not shown for clarity. Notice that after the V=R job step begins execution, the initiator in ADDRESS SPACE ONE cannot schedule another job or job step until V=R job step completion. The V=R job step is fixed in real storage during execution. There is no demand paging for the job step. Its addresses are still translated by the DAT feature but channel programs are not translated. Also, observe that any unused V=R space is available for demand paging. JOB56 is executing in ADDRESS SPACE TWO. It is loaded at the beginning of the private area because it is mapped by a different set of segment and page tables, and JOB56 is demand paged.

Timesharing in VS2/2

Until now, we have described the batch, multiprogramming functions as they are implemented in VS2 Rel 2. In the introduction to this lesson we said that the Time Sharing Option (TSO) is fully integrated into VS2/2's design. As with batch job initiators, when a TSO user logs on, the system creates a new address space for the TSO job's use. When the system operator starts TSO, an address space is created for the teleprocessing method (TCAM is used) that services all TSO user lines. As each new TSO user logs on, a new address space is created for his use. All address spaces created for TSO operation appear the same as those

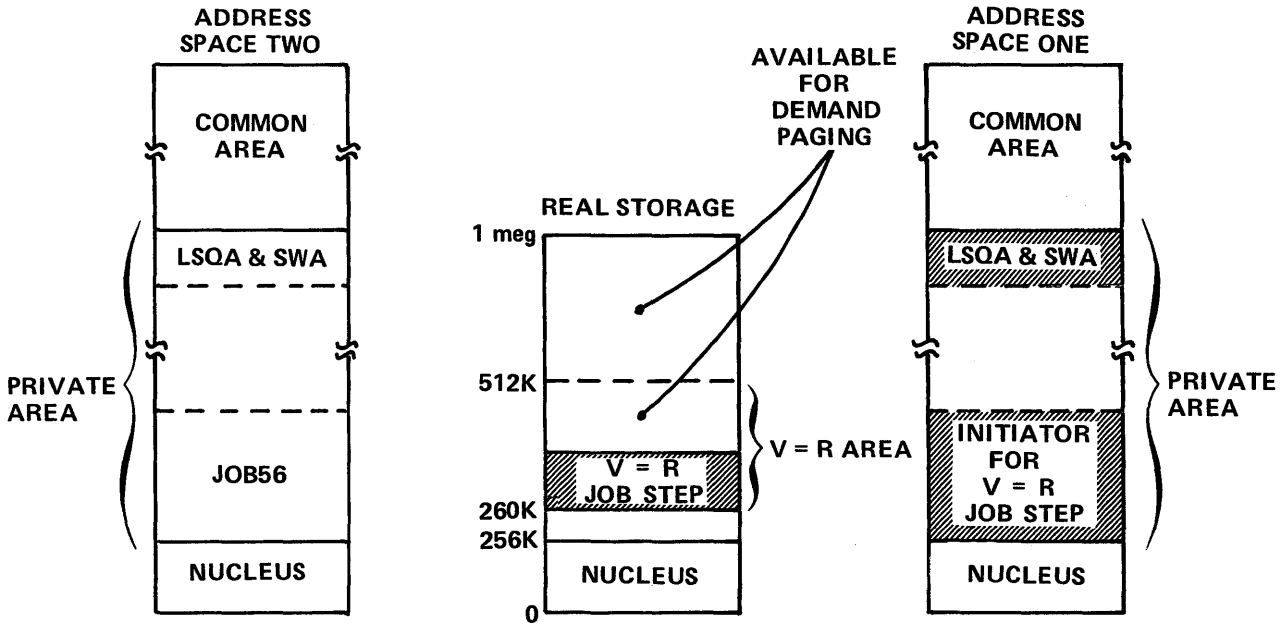


Figure 89

for batch jobs. They map the nucleus and the common area through segment sharing. Each TSO user then has the entire private area for his own use. Figure 90 shows an example of a TSO operation with three logged-on users and no batch operations.

Because VS2/2 creates an address space or virtual storage for each user in the system, an important question arises: is there any limit to the number of address spaces that VS2/2 can create? Theoretically, there is no limit. However, practical limits are imposed by the size of a system's real and external page storage. VS2/2 has a thrashing monitor to prevent excessive paging and it also implements a "swapping" algorithm. Remember, TSO is fully integrated into the design of VS2/2. As discussed in PART I of this text, in a timesharing environment it is more effective to allocate real storage to multiple users through a combination of demand paging and block paging (or swapping). This will assure that all users, both batch and timesharing, have an opportunity to execute over a reasonable time period. In a system with a thrashing monitor only, some users might be deactivated for a rather long time period. This would be intolerable if such a user were at a terminal waiting for a response.

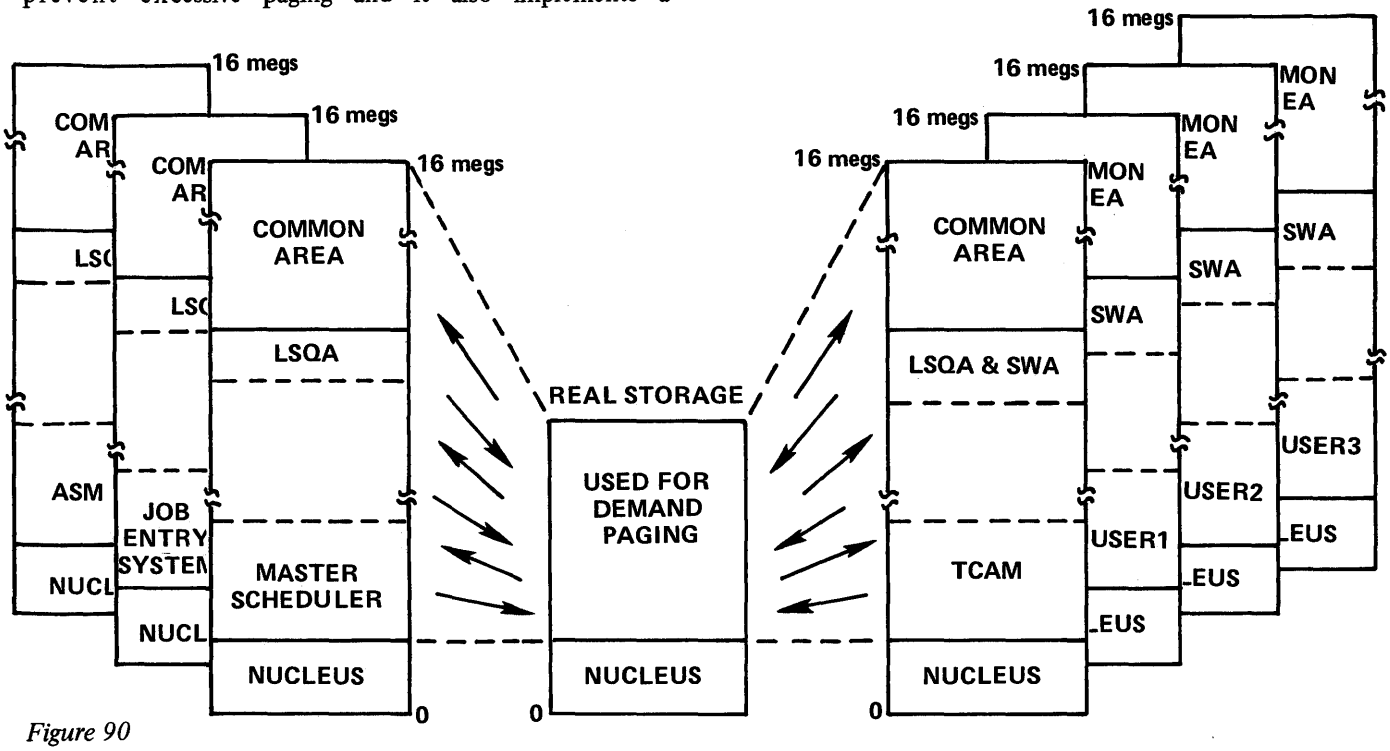


Figure 90

Levels of Control VS2/2

To prevent this problem, VS2/2 has three levels of control. The first level involves how many address spaces or virtual storages may exist in the system. Will the system allow the operator to start another initiator for batch or TP jobs? Will VS2/2 allow another TSO user to log on? This first level of control is implemented so that VS2/2 can prevent external page storage and real storage from being overrun.

The second level of control involves all the address spaces that currently exist in the system (for both TSO users and batch or TP jobs). If enough users exist to cause trashing, VS2/2 will schedule these users to decide which ones may contend for real storage through demand paging (and thus have an opportunity to execute). Of all the users in the system then, some are active and some are quiesced. The VS2/2 scheduling rule attempts to give all users an opportunity to execute. When VS2/2 quiesces a user (to activate another user) it block-pages all of the changed pages in the user's private area to external page storage. This will include the user's segment and page tables and any other control blocks required for reactivation. Only changed pages need be block-paged out because copies of any active unchanged pages already exist on external page storage. When a user is reactivated, the same set of changed pages is block-paged in to real storage. When execution resumes, the user is again under the control of demand paging.

With this second level of control then, the VS2/2 scheduling rule "swaps" virtual storages to assure that all users have an opportunity to execute. An installation may assign higher preference to TP jobs or time sharing users so that the scheduling rule will give such users a higher level of service.

The third level of control in VS2/2 involves dispatching (or scheduling) the system's CPU(s) among the active users. Dispatching in VS2/2 has been enhanced over Rel 1 to service a system with multiple virtual storages (this will be discussed in more detail later).

The VS2/2 Systems Resources Manager

The three levels of control in VS2/2 all involve system resources, whether external page storage, real storage, the number of address spaces or the CPU(s). Each level of control is affected by a new component of the VS2 nucleus called the *systems resources manager.* The resources manager is an extension of and a replacement for the TSO driver in VS2 Rel 1. The resources manager affects the use of all system resources – the CPU(s), real storage, virtual storage creation, virtual storage swapping, external page storage and so forth. The resources manager is a collection of algorithms that are supplied with VS2/2. An installation may specify parameters to tailor these algorithms to its particular needs or use default parameters provided with the VS2/2 system. In fact, an installation may replace an algorithm with one more suitable to its needs. The overall objective of the resources manager is to control system resources in such a way that VS2/2 achieves good performance objectives, whether the objectives be throughput, good response to timesharing users, some combination of these two, or some other set of objectives desired by an installation.

One area with which the resources manager interacts is CPU dispatching. Unlike VS2 Rel 1 which supports multitasking in a single address space, VS2/2 supports multitasking in multiple address spaces. Before dispatching tasks, the dispatcher must first select an address space (or user) for execution. The highest priority address space in the ready state is always selected. When an address space is selected for execution it may contain multiple tasks. The highest priority task in the ready state is then selected for execution. Once a task begins to execute, it will continue until one of the following events occurs:

- The task is interrupted by a higher priority task within the address space.
- The task is interrupted and another address space is dispatched.
- The task goes into a wait state. The next highest priority task would then receive CPU control. If no other tasks are in the ready state another address space will be dispatched.

The VS2/2 resources manager assigns address space priority and tries to assure that each address space (or user) receives a certain degree of service (CPU time and time in the active state between swapping). A VS2/2 installation can favor certain users by requesting a high degree of service for that user. For example, a TP application can be assigned a high degree of service to assure good response. Thus there are two levels of dispatching in VS2/2, the *global level* where the system decides which address space (or user) to dispatch, and the *local level* where the dispatcher selects a task for execution from within an address space.

This idea of global and local services extends beyond dispatching in VS2/2. On the global level VS2/2 controls the CPU, real storage, external page storage, I/O device assignment, address space creation and so forth. SQA is used for control blocks and queues that pertain to global control of the system. The Pageable Link Pack Area contains system and user programs that are available to all users in the system.

On the local level, resources are used to service a particular address space. LSQA contains control blocks and

tables that pertain to the address space, for example, the segment and page tables that map the address space. SWA is used for control blocks for job scheduling within the address space. While a user is executing within an address space, the user may attach tasks and allocate space to them from within the address space. The user may also create a Job Pack Area (JPA), analogous to the Pageable Link Pack Area, but only for use within the address space. If at any time a user inadvertently destroys data within the address space the remainder of the system and its users will not be affected because of the inherent protection in a multiple virtual storage system.

VS2/2 is not just a system that supports multiple virtual storages. It is an operating system that services batch jobs from both local and remote work stations. It is an operating system that fully integrates the TSO timesharing service into its structure. It is an operating system that supports uniprocessing for a single System/370 CPU or multiprocessing on two or more tightly coupled System/370 CPU's. VS2/2 makes a major stride in the evolution of the OS/VS operating system.

Lesson 15. DOS/VS

The Disk Operating System/Virtual Storage (DOS/VS) adds major functional enhancements to DOS. Five user partitions, an enhanced POWER facility, variable partition priority, a relocating loader, cataloged procedures and virtual storage implementation are the new functions in DOS/VS. In this lesson we shall describe briefly the five user partitions, variable partition priority and the relocatable loader. We shall then present the virtual storage implementation in DOS/VS. DOS/VS implements a single virtual storage as a standard feature of the system. DOS/VS executes only on a System/370 with the DAT feature.

Five User Partitions

With DOS/VS, you may execute job streams in one to five batch user partitions. Single Program Initiation (SPI) is not required and thus not supported in DOS/VS. Standard partition sizes are specified during system generation. At IPL time you may change the standard size of any partition. Minimum partition size is 64K. This allows Job Control to execute in all partitions and eliminates the need for SPI support. Figure 91 contrasts the former DOS system structure with DOS/VS showing the maximum number of user partitions in each system.

You can run batch jobs in the two new DOS/VS partitions F3 (Foreground 3) and F4 (Foreground 4). Standard partition dispatching priority, which determines what partition gets the system's CPU next, is F1, F2, F3, F4 and BG in that order. A job executing in the F1 partition has the highest priority; jobs that execute in BG have the lowest priority. To properly balance the use of the CPU and channels among the jobs executing in your system, you must consider the priority of the partition where a job will execute. Assign jobs to partitions in a way that will balance the use of your CPU and channels using the DOS/VS standard dispatching priorities. For example, an I/O bound job should have a higher priority than a CPU bound job. This will tend to produce overlap between channel and CPU operation.

Variable Partition Priority

DOS/VS also has a new feature that will let you change standard partition dispatching priorities during system operation. It is called *variable partition priority.* In a situation where, for example, a job executing in the BG partition (normally the lowest priority partition) needs to be rushed, your operator could give the BG partition a higher priority. Under normal circumstances you should plan for system operation with standard dispatching priorities. Use the variable partition priority feature for exception situations.

Relocating Loader

Early in the text we described static relocation and dynamic relocation. System/360 versions of DOS, with their

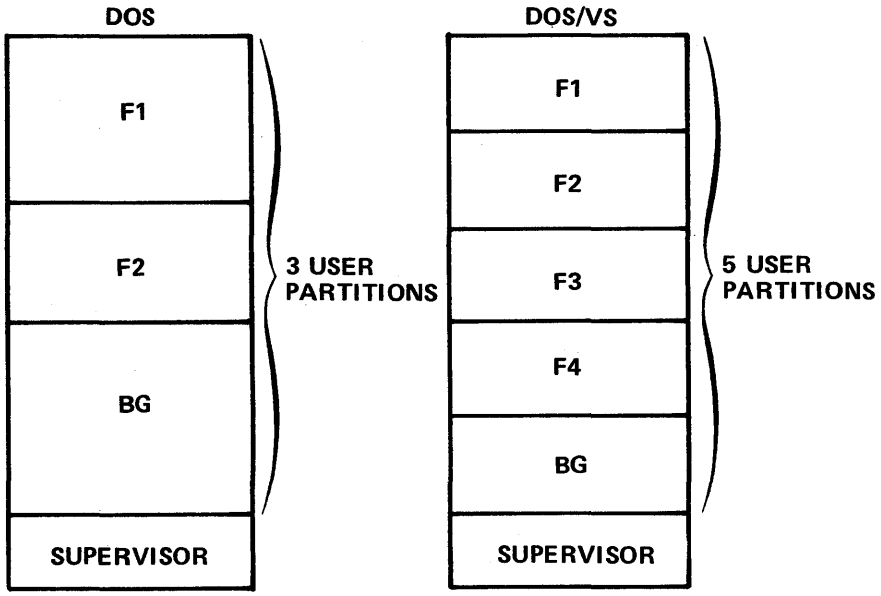


Figure 91

link edit process, don't implement either type of relocation. With DOS on System/360, programs, in general, are bound to their real storage locations at link edit time. In effect, programs will execute in the same real storage locations every time, unless you relink edit to another area of real storage. In other words, a program is bound to a partition, F1, F2, or BG, at link edit time. This resulted in several considerations for the System/360 DOS user:

1. A partition's size and location in real storage had to be planned in advance.
2. Programs were usually bound to one partition. If there was a need to execute a program in multiple partitions, this usually required multiple copies of the program, each copy link edited to its partition's location.
3. It was difficult to change the real storage boundaries of a partition. This last item is very significant for users. Consider the situation shown in Figure 92.

Users periodically generate a new system to add new functions. In the process, the new supervisor usually becomes larger as indicated in Figure 92. The larger supervisor offsets the boundary of the BG partition. You might also have to change the real storage locations of the F1 and F2 partitions. Since programs are bound to partitions and their real storage locations, a System/360 DOS installation in this situation would have to relink edit all programs that execute in BG to BG's new real storage locations. If you change the boundaries of the F1 and F2 partitions, you must also relink edit their programs.

DOS/VS users won't have this kind of problem if they use the DOS/VS relocating loader option. The relocating loader is a software feature that translates or relocates a program's addresses to a partition's boundary. The relocation occurs at program load time. Relocation occurs every time that you load a program, not just once at link edit time as in DOS. Since relocation occurs at program load time, programs don't become, in effect, bound to partitions. You may load a program into any partition. Thus, partition locations aren't as rigid. The situation that we described for DOS with a linkage editor only, as shown in Figure 92, is no longer a problem.

The DOS/VS relocating loader is a type of static relocation. It gives the DOS/VS user an assist for effective multiprogramming in five partitions and for effective use of virtual storage in DOS/VS. We shall describe the relocating loader – virtual storage relationship in a later topic.

Virtual Storage in DOS/VS

The DOS/VS system has a single virtual storage. In many ways the DOS/VS virtual storage implementation is similar to OS/VS1. DOS/VS is structured in its virtual storage, while real storage is a system resource shared by all user partitions through the demand paging technique.

In DOS/VS, virtual storage size may not exceed 16 megabytes (16 megs). It may be smaller. Virtual storage size is specified by a user at system generation time. Figure 93 shows a virtual storage of 16 megs.

Notice in Figure 93 that virtual storage is divided into segments. Each segment is 64K in size. In 16 megs, there are 256 segments numbered from 0 through 255. In DOS/VS, segments are useful when considering virtual storage size. A 512K virtual storage would be 8 segments in size.

Also notice in Figure 93 the structure of each segment.

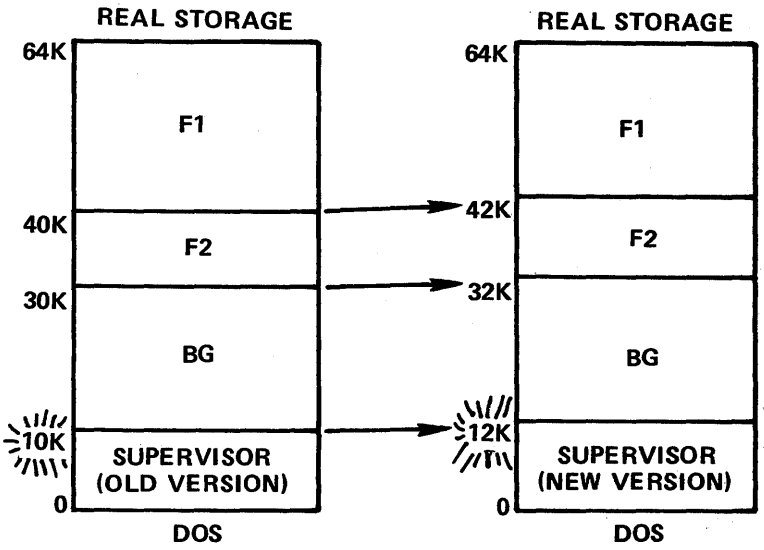


Figure 92

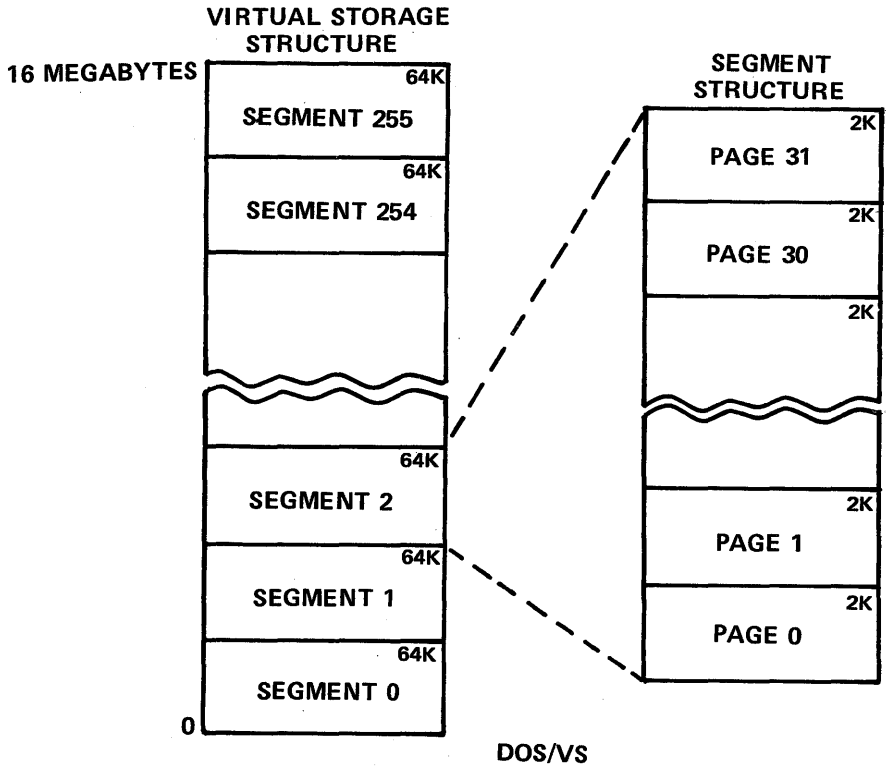


Figure 93

Each segment has 32 pages, numbered from 0 through 31. Each page is 2K in size. In DOS/VS, the page is the primary building block. Virtual storage is allocated to the supervisor and partitions in page size increments.

The DOS/VS Structure in Virtual Storage

The DOS/VS system is structured in its virtual storage. Figure 94 shows an example of a DOS/VS system structured in virtual storage. Virtual storage is divided into two major areas, the virtual address area and the real address area. The DOS/VS virtual address area of virtual storage is analogous to the pageable area of virtual storage in VS1 and VS2. The DOS/VS real address area of virtual storage is analogous to the non-pageable area of virtual storage in VS1 and VS2. In DOS/VS, during operation active partitions in the virtual address area of virtual storage dynamically share real storage through demand paging. The DOS/VS supervisor and the V=R area are defined in the real address area of virtual storage. The real address area is not controlled by demand paging. Its pages, when allocated, are fixed in corresponding page frames of real storage. We will return to the relationship of virtual storage to real storage later. Next we will present a short description of the DOS/VS supervisor, the V=R area and user partitions.

DOS/VS Supervisor

The supervisor is the primary control program of the system. It controls the allocation of all system resources – the CPU, I/O devices, and so forth. Minimum supervisor size for DOS/VS is 26K. Supervisor size expands in increments of 2K, the size of a page, depending on the options selected for your installation. In addition to normal DOS control functions, the DOS/VS supervisor contains a paging supervisor. The paging supervisor controls real storage using the demand paging technique. The DOS/VS supervisor also contains the segment table and page tables used during dynamic address translation. The DOS/VS supervisor is loaded at the origin of virtual storage (the DOS/VS system's address space) as shown in Figure 94. If the supervisor were 32K, it would use the first 32K of virtual storage.

The Virtual Equals Real Area

The Virtual Equals Real (V=R) area is the part of virtual storage (see Figure 94) used for V=R job steps. When a job step is executed as V=R, it is not paged and its channel programs are not translated. A job step is executed as V=R for two primary reasons:

1. The job cannot tolerate time delays caused by paging. This would be the case for a MICR job.
2. The job dynamically modifies channel program addresses during I/O operations (see *Lesson 11, Channel*

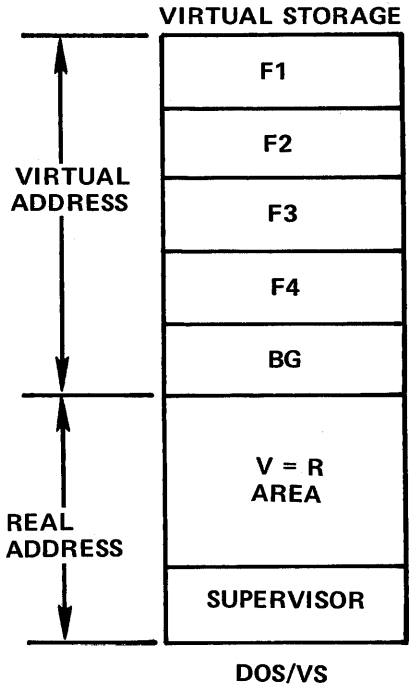


Figure 94

Program Translation for additional explanation). Job Control language is used to specify a job step as V=R. How V=R space is allocated and how V=R job steps are loaded will be presented later in this lesson.

DOS/VS Partitions

Earlier in this lesson we described the five partition feature in DOS/VS. Figure 94 shows all five partitions in virtual storage. However, the number of partitions in your DOS/VS system is a system generation or user option. If you were to select three partitions at system generation time you would have F1, F2 and BG in your DOS/VS system. Partitions are defined in the virtual address area of virtual storage. A size for each partition is specified during system generation as the system standard. You can change the size of any partition during system operation. Partition size is always specified as a multiple of 2K, the size of a page. Minimum partition size is 64K, the size of Job Control. Thus, in DOS/VS, all partitons are batch partitions.

Virtual Storage – Real Storage Relationship in DOS/VS

We have thus far described the major parts of DOS/VS in virtual storage. Figure 95 shows the relationship of virtual storage to real storage in the DOS/VS system. Real storage is divided into fixed size page frames. Page frames are 2K in size, the same size as a page. Whenever the real address area of virtual storage is used, its pages are fixed in corresponding real storage page frames. Thus, all supervisor pages are fixed in real storage, as shown in Figure 95, during

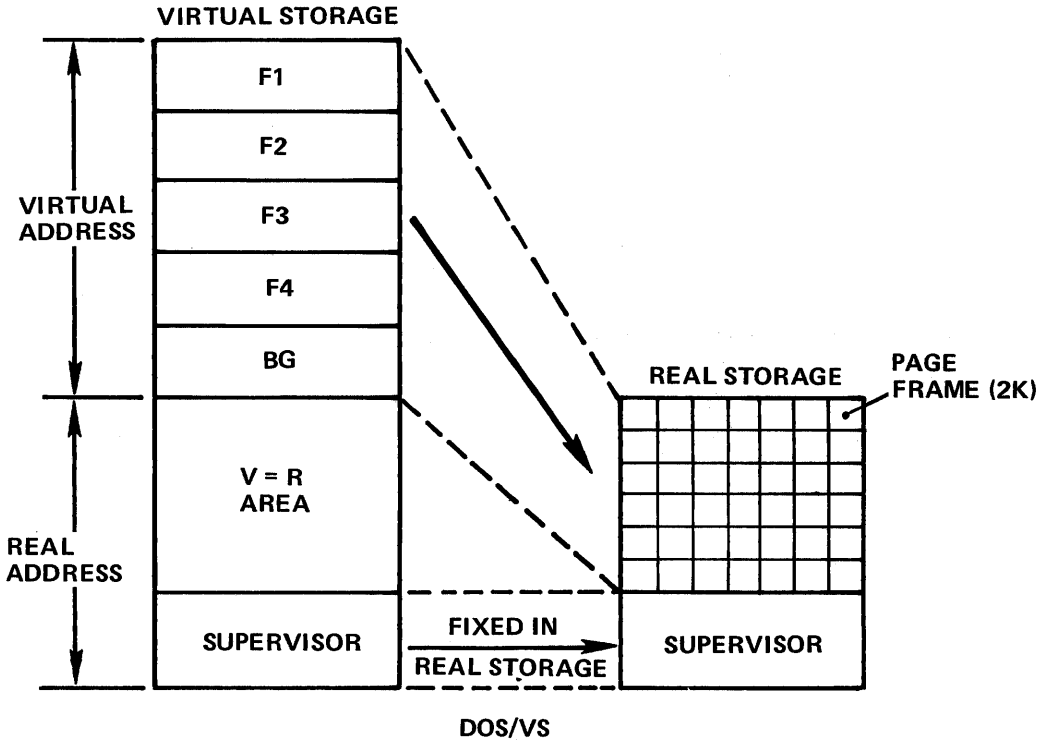


Figure 95

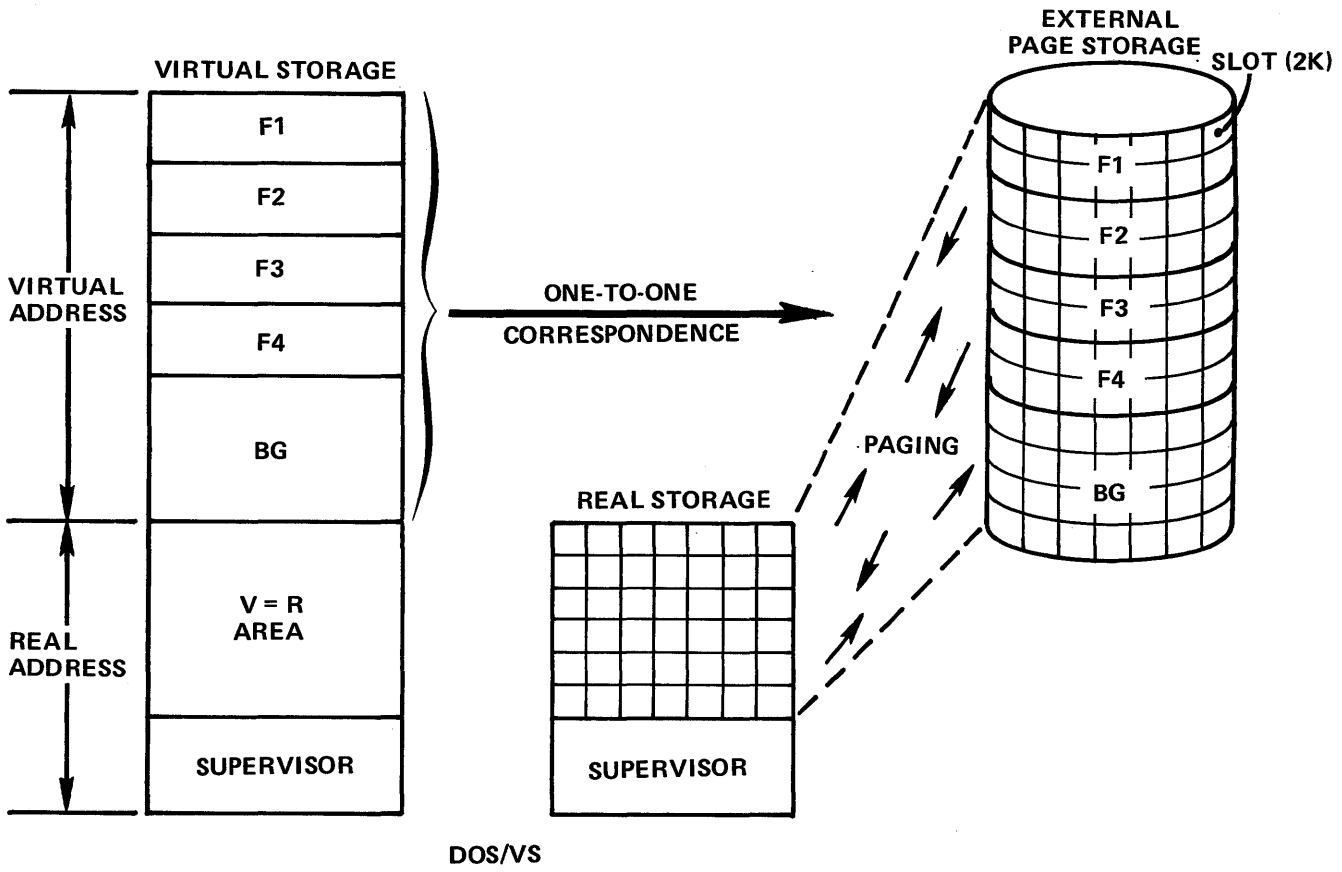


Figure 96

system operation. Pages allocated for a V=R job step (from the V=R area) would also be fixed in corresponding page frames of real storage.

Pages in the virtual address area of virtual storage share real storage page frames dynamically through demand paging. This is indicated by the arrow in Figure 95. Thus, it is possible for you to define partitions whose total size exceeds real storage size. Your active partitions will share real storage under the control of the DOS/VS paging supervisor. The System/370 DAT feature will translate all virtual addresses using the segment table and the page tables in the DOS/VS supervisor. With DOS/VS, you will be able to define a system that is less dependent on real storage size than past versions of DOS. The DOS/VS system is structured in virtual storage. Real storage is a system resource controlled by the paging supervisor.

Virtual Storage – External Page Storage Relationship in DOS/VS

Since virtual storage in DOS/VS is the system's address space, the virtual address area of virtual storage must have some physical resource within the system to back it. DOS/VS must have some type of External Page Storage as described in PART I of this text. Figure 96 shows external page storage in DOS/VS. External Page Storage may reside on a 2314, 2319 or 3330 device. Its record size is 2K, the same size as a page and page frame. External page storage records are called slots. As shown in Figure 96, there is a one-to-one correspondence between pages in the virtual address area of virtual storage and slots of external page storage. That is, for every page in the virtual address area there is an assigned slot in external page storage. Paging then physically occurs between external page storage and real storage. Pages are moved between slots of external page storage and page frames of real storage. This is indicated in Figure 96. During a page-in operation, a page is moved from its slot in external page storage to any available page frame in real storage. During a page-out operation a page is moved from the frame it occupies in real storage to its assigned slot in external page storage. That is, in DOS/VS a page can occupy any page frame but it always returns to the same slot.

Figure 96 shows the complete relationship of virtual storage to real storage and external page storage. The real address area, whenever used, uses corresponding page frames of real storage. The supervisor is always fixed in real storage during system operation. Pages in the V=R area, when used, are always fixed in corresponding page frames

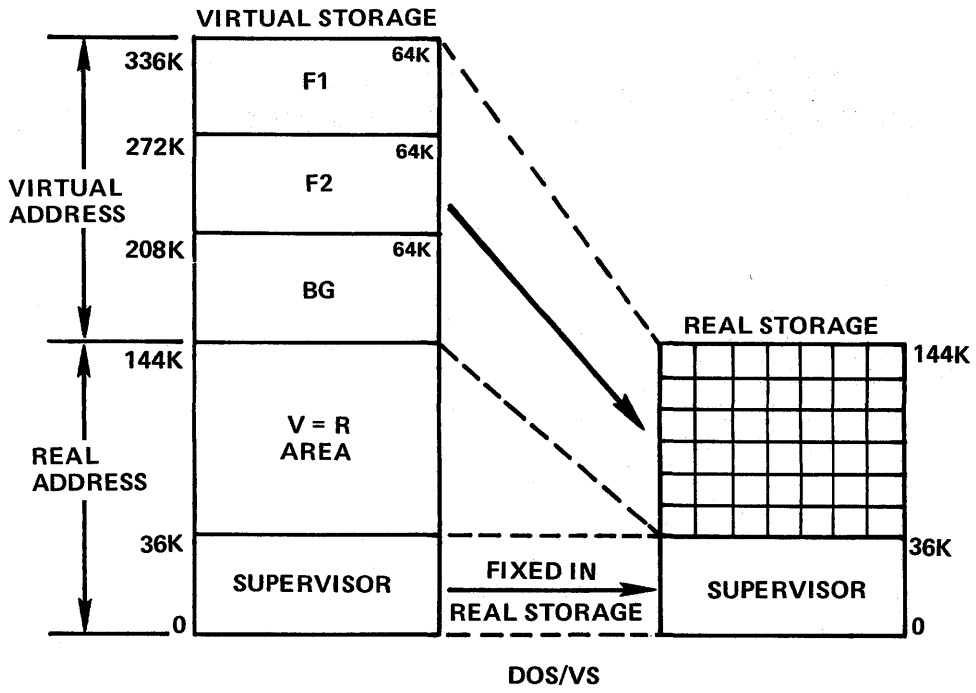


Figure 97

of real storage. The virtual address area of virtual storage has a one-to-one correspondence with external page storage. Paging physically takes place between external page storage and real storage.

DOS/VS System Definition and Operation

In this next topic we will describe the DOS/VS system using a specific example. This section will include items such as system generation considerations; loading a program into a partition in virtual storage using the relocating loader; and loading and executing V=R job steps. Our sample system configuration assumes that we have a System/370 Model 135 with the DAT feature and 144K of real storage. We will define a system with 3 partitions, F1, F2 and BG, the relocating loader function, and a virtual storage (or address space) size of 336K. Figure 97 shows this DOS/VS system structured in virtual storage next to its real storage resource.

At system generation time, three major decisions were necessary to build the sample system shown in Figure 97:

1. Virtual storage size
2. The number of partitions (from one to five)
3. Whether or not to use the DOS/VS relocating loader

There are many other system generation considerations. We only mention those most pertinent to some of the new DOS/VS features. With these options we assume a supervisor size of 36K. Remember, supervisor size is always a multiple of 2K, the size of a page.

Standard partition sizes are also selected at system generation time. Remember, partition sizes may be changed during system operation, usually at IPL time. In our example, Figure 97, all partitions are the same size. This has a nice scheduling advantage. All jobs 64K and under can be executed in any of the partitions. You can assign jobs to partitions, concentrating on effective use of the CPU. For example, you can execute I/O-bound jobs in high priority partitions and CPU-bound jobs in lower priority partitions. If, for example, you execute a 10K I/O-bound job in F1, 54K of virtual storage (not real storage) is wasted. This is a result of the demand paging technique implemented by the DOS/VS paging supervisor. With DOS/VS, fragmentation may occur, but it occurs in virtual storage not in real storage as in former DOS systems. This is an example of how you can make good use of the size of virtual storage. With virtual storage you might also try full-function processors like the PL/I Optimizer or full ANS COBOL. Or you might try large applications, too large to attempt with former versions of DOS because of real storage limitations. Be careful, however, not to assume that virtual storage is a substitute for real storage. If you recall the working set concept from PART I, some programs may require much less real storage than their actual size; some programs may need the same real storage as their size. Some of the advantages in DOS/VS are:

1. Virtual storage enables flexible operations, such as our example of equal size partitions.
2. Virtual storage allows experimentation with programs too large for your real storage resource. This enables easier growth into new applications. You can try an application before you install additional real storage.
3. In some cases, virtual storage will allow you to exe-

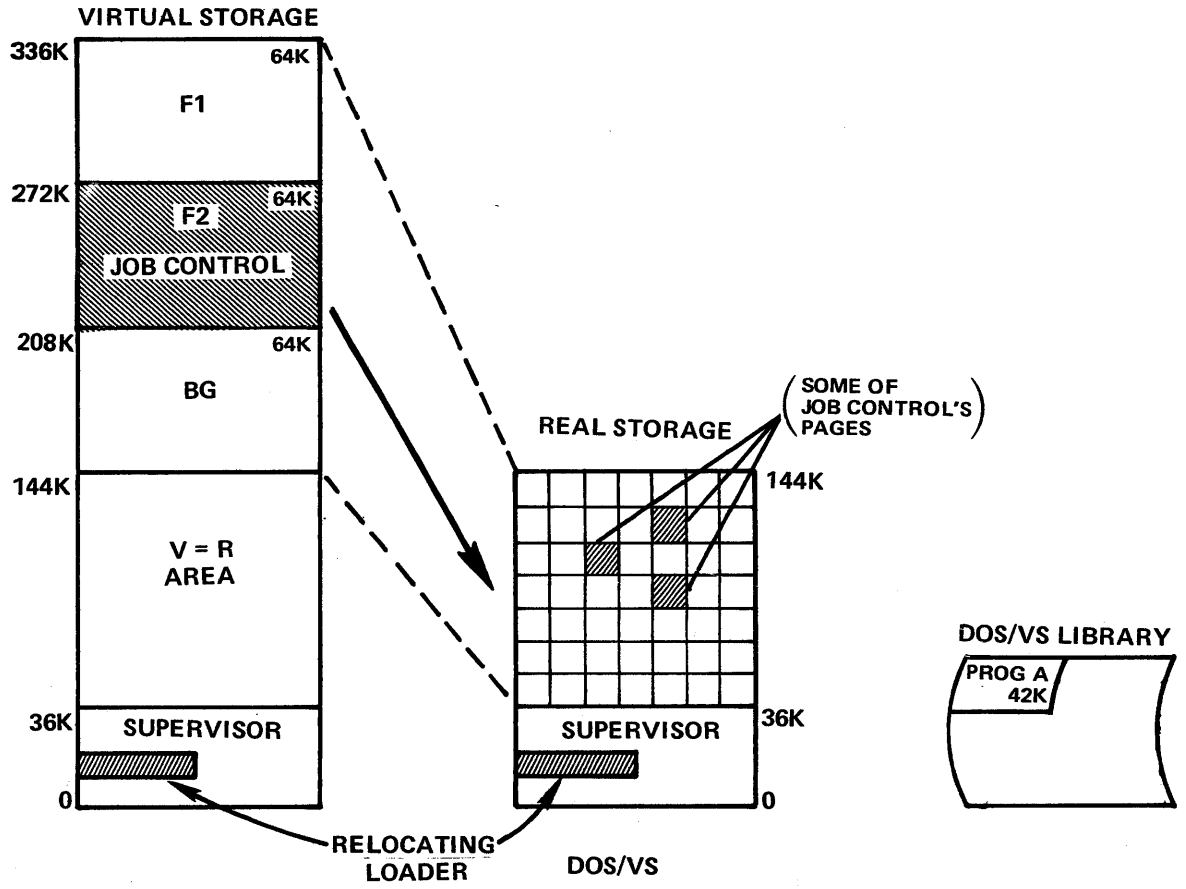


Figure 98

cute a set of jobs too large for your real storage because of the working set concept.

4. Virtual storage should make programming easier. You can use large partitions instead of an overlay or multi-step approach when designing an application.
5. The relocating loader option will make operations and the installation of new system releases easier.
6. Design and development of teleprocessing and data base applications should be easier with a large virtual storage.

Although we have suggested how to schedule jobs in DOS/VS, we haven't described how programs are loaded. We said before that we included the relocating loader option in our DOS/VS sample system. We will now describe how programs are loaded into DOS/VS partitions.

Let's assume that we have a job that we want to execute in F2. To schedule the job, Job Control is loaded into F2. It uses 64K. Job Control is paged, just like all other programs that execute in the virtual address area of virtual storage. Our sample job will execute a program called PROGRAMA. PROGRAMA is in a DOS/VS library (the Core Image library) in the format required by the relocating loader. The Relocating Loader, which is part of the DOS/VS supervisor, can load PROGRAMA into any contiguous range of virtual addresses, therefore, into any partition. In our example, the range of virtual addresses is F2, from 208K to 272K. Let's assume that PROGRAMA is 42K in size. Figure 98 shows the situation that we have just described. Notice that we show some of Job Control's pages in real storage. This is the case because we assume that Job Control is executing at this time.

When Job Control is told to load PROGRAMA (through the JCL execute statement) it passes control to the Relocating Loader. The Relocating Loader reads PROGRAMA as data and translates PROGRAMA's address constants related to F2's origin in virtual storage (208K). After loading, PROGRAMA's addresses start at 208K and end at 272K. This translation of addresses at load time is the static relocation technique that was described in PART I. Thus far we have described the logical process performed by the relocating loader in DOS/VS. Physically, the Relocating Loader reads PROGRAMA into page frames and translates PROGRAMA's addresses. This process continues until all of PROGRAMA is translated (or loaded). In most cases, many of PROGRAMA's pages will be paged out during the load operation. Therefore, at the end of loading most of PROGRAMA will be in external page storage. When loading is complete, control passes to PROGRAMA and it begins to

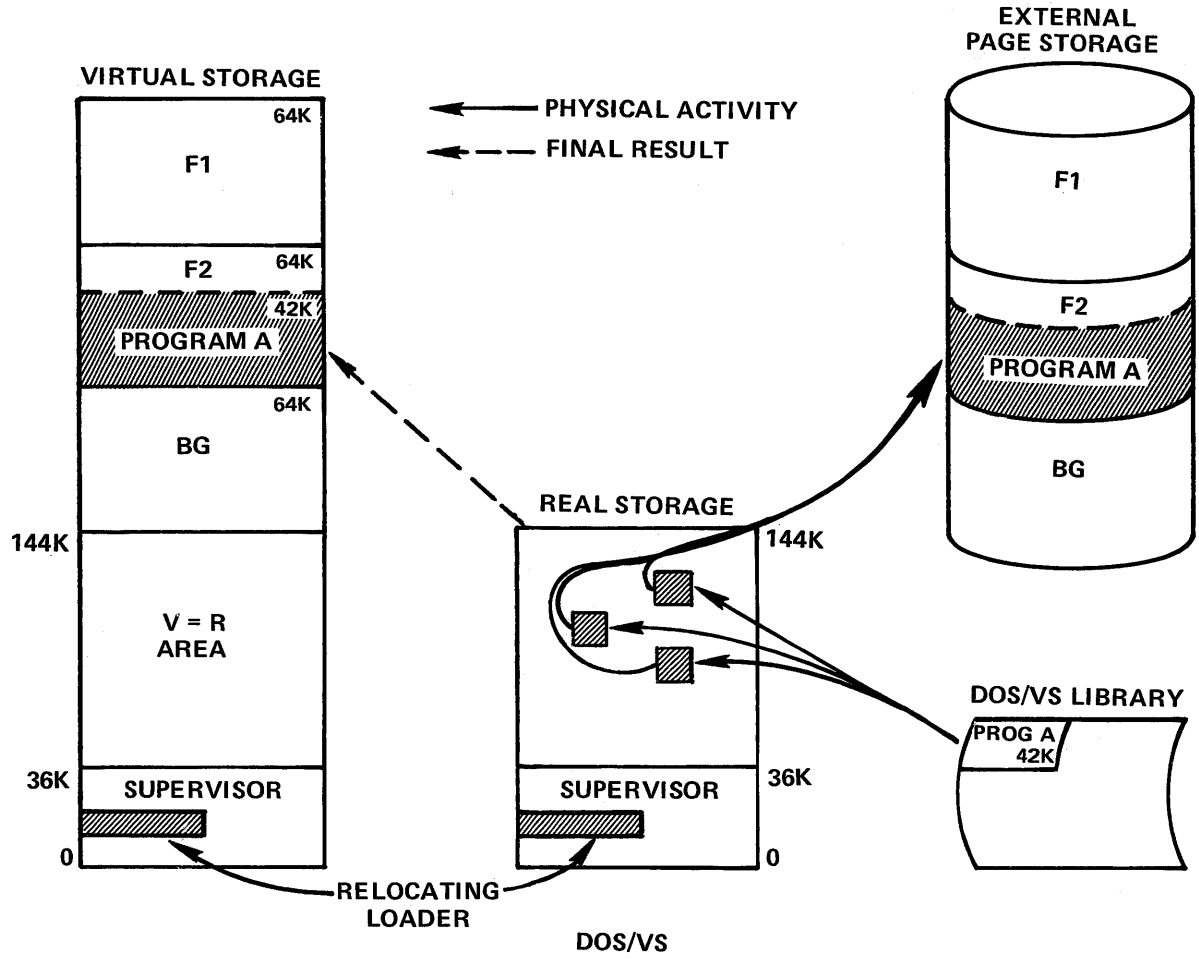


Figure 99

execute under the control of demand paging. Figure 99 shows both the physical and logical results of program loading using the DOS/VS Relocating Loader.

Because the Relocating Loader loads programs just before execution, we could have executed PROGRAMA in F1 or BG by scheduling the job through an F1 or BG system reader. There would be no requirement to relink-edit or to have multiple copies of PROGRAMA (one for each partition).

Executing V=R Job Steps

Normally, jobs will run in the virtual address area of virtual storage in DOS/VS. Job initiation is performed by Job Control. Program loading is controlled by the Relocating Loader (if you select that option). But what about V=R job steps? We have described the need for them – time-dependent jobs or programs that dynamically modify their channel programs. You have seen the V=R area of virtual storage in DOS/VS. However, we have yet to describe how V=R space is allocated and how V=R job steps are scheduled and executed.

V=R Space Allocation

We said before that pages from the real address area of virtual storage are fixed in real storage when they are used. The DOS/VS supervisor is fixed in real storage during system operation. The V=R area allows the DOS/VS user to fix programs (or V=R job steps) in real storage in a similar manner. This type of page fixing is called a long term fix. That is, a V=R job step's pages are fixed in real storage during the entire execution of the job step.

V=R space (or pages) can be allocated to any or all partitions during system operation. You can make a standard assignment of V=R space to a partition at system generation time and then change the allocation during system operation. Using our sample DOS/VS system, we have assigned V=R space to F1 and BG, 16K (or 8 pages) to F1 and 30K (or 15 pages) to BG. This is shown in Figure 100.

Notice that a user assigns V=R space starting at the bottom of the V=R area. We identify the allocated pages as BGR and F1R. BGR identifies the Background partition's V=R pages. F1R identifies the Foreground One partition's V=R pages. These pages are reserved for any V=R job steps that

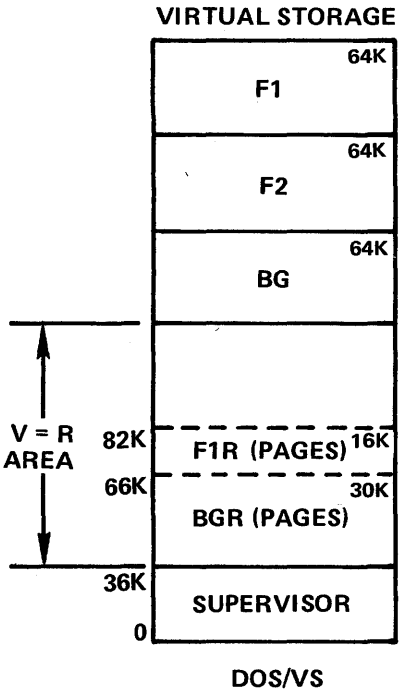


Figure 100

will execute in BG or F1. Because we have not assigned V=R space to F2, no V=R job steps can run in F2 (unless V=R space is assigned to F2 during system operation).

V=R space allocation does not affect real storage unless you use it for a V=R job step. Thus, in normal system operation, all real storage page frames (above the Supervisor) are available for demand paging. Let's assume a situation for our sample DOS/VS system in which jobs are executing in F1 and F2; V=R space is allocated as shown in Figure 100, and we are about to schedule a V=R job step in BG. Figure 101 shows this situation.

Notice how real storage page frames are shared among the three partitions. This includes some of Job Control's pages in BG. We will call the V=R job step VRSTEP. VRSTEP is specified as Virtual Equals Real (V=R) in its JCL execute statement. When Job Control reads this statement, it directs the Relocating Loader (assuming that you have installed the option in your DOS/VS supervisor) to load VRSTEP into BGR in the V=R Area of virtual storage. Let's assume that VRSTEP is 20K in size. It could not exceed 30K, the size of BGR, unless we reassigned space in the V=R Area to make BGR larger than 30K.

Before the Relocating Loader can load VRSTEP, the DOS/VS paging supervisor must move any active pages (from the virtual address area of virtual storage) that currently reside in page frames of real storage corresponding to BGR's pages in the V=R Area. These pages (from the pageable area of virtual storage) will be moved to available page frames or external page storage. When the area in real storage that corresponds to BGR is clear VRSTEP can be

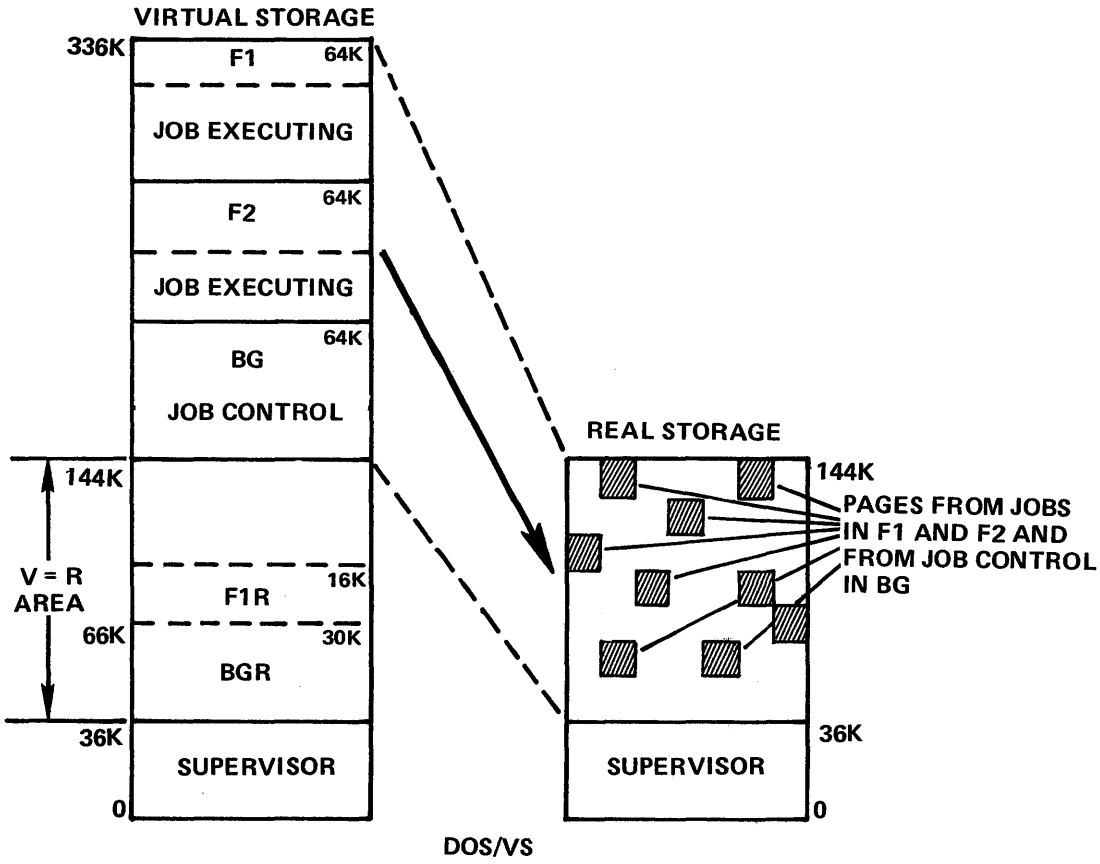


Figure 101

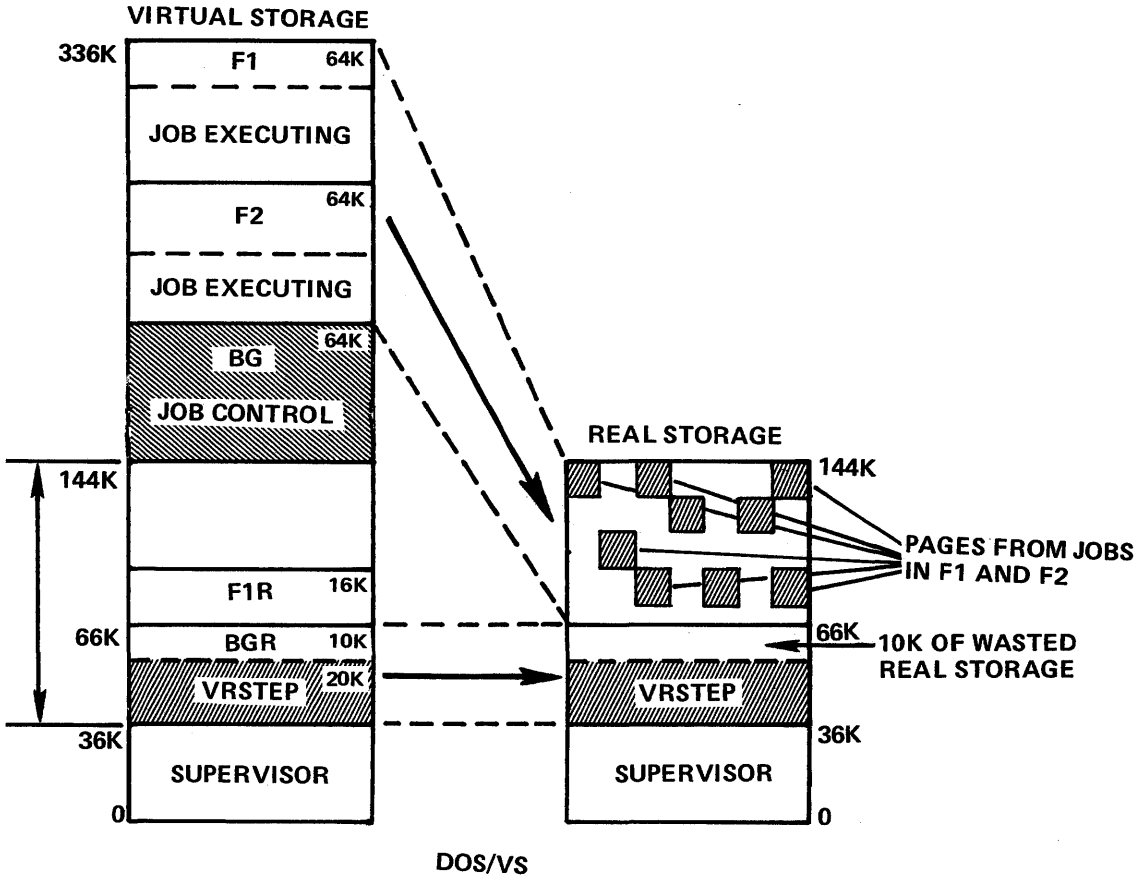


Figure 102

loaded. The Relocating Loader loads VRSTEP, relocating its addresses to an origin of 36K. All of VRSTEP's pages get a long term fix in real storage. VRSTEP then begins to execute, its pages fixed for the entire execution. VRSTEP's virtual addresses equal its real addresses. Therefore, no paging occurs. VRSTEP's pages are fixed in real storage. Also, no channel translation occurs. VRSTEP could be a program that dynamically modifies CCW's during I/O operations. Figure 102 shows the status of our sample DOS/VS system during VRSTEP's execution. No new jobs or job steps can be initiated in the BG partition until the VRSTEP program terminates. There is only one BG partition. When used for a V=R job step the program is loaded into BG's V=R space (BGR); BG's space in the virtual address area is unused until the V=R job step terminates. At that time, Job Control would be loaded into BG in the virtual address area and initiate the next job or job step for the BG partition. Figure 102 also shows the effect of a V=R job step on real storage. A part of real storage is dedicated to VRSTEP. This real storage can no longer be shared dynamically among all active jobs in the system until VRSTEP terminates. Also notice in Figure 102 that the unused 10K in BGR results in 10K of wasted real storage. This can be prevented if you specify the size of VRSTEP in its JCL execute statement. For example, if we had specified VRSTEP as 20K, only the first 20K of BGR's pages and the corresponding 20K in real storage would have been used. The 10K of wasted space shown in Figure 102 would be available to F1 and F2 under the control of the paging supervisor.

Plan to use the V=R area only when required. Let the DOS/VS paging supervisor control the allocation of real storage through demand paging.

The many new functions in DOS/VS add significant improvements to its multiprogramming, teleprocessing, data base and interactive computing capabilities. With the DOS/VS system structured in virtual storage, system operation and application development should be much improved. These improvements in DOS/VS truly make it a system of the '70's.

READER'S COMMENT FORM

Introduction to Virtual Storage in System/370 GR20-4260-1

Please comment on the usefulness and readability of this publication, suggest additions and deletions, and list specific errors and omissions (give page numbers). All comments and suggestions become the property of IBM.

COMMENTS

Reply Requested

Yes ☐

No ☐

Name _____

Job Title _____

Address _____

_____ Zip _____

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

FOLD ON TWO LINES, STAPLE AND MAIL

YOUR COMMENTS PLEASE

Your comments on the other side of this form will help us improve future editions of this publication. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material.

Please note that requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or the IBM branch office serving your locality.

FOLD FOLD

FIRST CLASS
PERMIT NO. 40
ARMONK, NEW YORK

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY:

IBM Education Center, Building 005
Department 78L, Publications Services
South Road
Poughkeepsie, New York 12602

FOLD FOLD

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

GR20-4260-1

Introduction to Virtual Storage in System/370 Printed in USA GR20-4260-1