GC28-0984-0
File No. S370-20

Systems

# OS/VS2 MVS Overview

**First Edition (June, 1978)**

This edition applies to release 3.7 of OS/VS2 MVS, and to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters. Changes may be made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest **IBM System/370 Bibliography**, GC20-0001, for the editions that are applicable and current.

Publications are not stocked at the address given below; requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Department D58, Building 706-2, PO Box 390, Poughkeepsie, New York 12602. Comments become the property of IBM.

# Preface

This book describes the main features of MVS. It explains each of these features and describes the flow of work through the major parts of the system. It does not, however, describe every feature of the system. The emphasis here is on what MVS does and how it accomplishes its objectives.

The book is intended for a general audience, but some knowledge of operating systems is necessary.

Chapter 1 is an introduction to the basic features of MVS. It shows how MVS accomplishes its main objective of doing more work. Those who require only a high-level overview of the system can obtain this from Chapter 1.

Chapters 2 – 10 provide detailed information on each of the concepts Chapter 1 introduces. Chapters 2 – 10 are, generally speaking, a chronological view of the system. That is, they take the reader from the concepts of virtual storage through initializing the system to entering, scheduling, and supervising work. The main topics they discuss are MVS, the System Resource Manager, Job Entry Subsystems, I/O, Error Recovery, and Multiprocessing.

There are no prerequisites to this publication. Related publications are:

*OS/VS2 System Programming Library: Initialization and Tuning Guide,* GC28-0681
*OS/VS2 MVS Release Guide,* GC28-0707
*OS/VS2 System Programming Library: System Generation Reference,* GC26-3792
*OS/VS2 System Programming Library: Supervisor,* GC28-0628
*OS/VS2 Supervisor Services and Macro Instructions,* GC28-0683
*OS/VS2 MVS Multiprocessing: An Introduction and Guide to Writing Operating and Recovery Procedures,* GC28-0952
*OS/VS2 Conversion Notebook,* GC28-0689
*OS/VS2 MVS Performance Notebook,* GC28-0886
*OS/VS1 to OS/VS2 Conversion Notebook,* GC28-0953
*OV/VS2 System Modification Program (SMP) System Programmer's Guide,* GC28-0673
*Operator's Library: OS/VS2 MVS System Commands,* GC28-0229
*OS/VS2 MVS JCL,* GC28-0692
*Operator's Library: OS/VS2 MVS JES2 Commands,* GC23-0007
*Operator's Library: OS/VS2 MVS JES3 Commands,* GC23-0008
*OS/VS2 MVS System Programming Library: JES2,* GC23-0002
*OS/VS2 System Programming Library: Job Management,* GC28-0627
*Introduction to JES3,* GC28-0607

# Contents

# Figures

# Chapter 1: Introduction

The basic difference between the IBM Operating System/Virtual Storage with Multiple Virtual Storage (MVS) and previous IBM operating systems is that MVS does more work. That is, MVS does things faster and does more things at the same time.

This ability to do more work benefits the user directly and indirectly:

- Directly, it provides greater support for a larger number of users, both interactive and batch. The user can have many more activities going on in the system simultaneously without loss of time.
- Indirectly, the ability to do more work allows the system to enhance its own capabilities by providing improved performance, improved security and integrity, and enhanced function.

What then allows MVS to do more work and what are these improvements to the basic abilities of any operating system?

## Direct Benefits

There are several basic MVS features that enable it to do more work. They are:

- Multiple virtual storage
- Increased multiprocessing capabilities
- Enhanced error recovery

These MVS features provide the most direct benefits to the user.

### *Multiple Virtual Storage*

Main storage is a scarce resource and even when it can be shared, the amount of space an installation's programs and data require far exceed the amount of main storage available. In previous systems, this was true not only on an installation basis, but on a program basis:

amount of storage available to a particular program = amount of storage available in the system – (system requirements + amount of storage already being used by other programs)

Furthermore, previous systems had to preallocate storage before the job executed, the preallocated storage had to belong to the job for the duration of the job, and the programmers had to plan complicated overlay structures to fit their programs into the available space. This caused three very expensive problems:

1. Some portions of storage may not be used at all.

*Example:* The system has one million bytes of main storage. Job A requests and receives 384,000 bytes; these bytes belong to Job A until it completes — they cannot be shared; Job B and Job C request and receive 300,000 bytes each. Now there is a **fragment** of 16,000 bytes that cannot be used at all unless the system starts a job that requires only that much. This is known as fragmentation — unused fragments, too small to start a normal job, exist throughout storage.

2. Though occupied, some locations did not contain active programs. These programs were waiting for some event to occur or they were waiting for another part of the program to be brought into storage to overlay the completed part. In any case, they tied up storage without being active.
3. To deal with the size limitations, users had to design complicated overlay structures. This took a great deal of programmer time. Also, the system had to wait while finding and bringing in the next part of the overlay structure.

In short, even though main storage was scarce, previous systems still wasted it. To help overcome this problem, IBM developed virtual storage and then multiple virtual storage systems. To understand how MVS overcomes these three problems, you must know a bit about addressing.

## Addressing in MVS

Generally speaking, an address is a group of characters that identify a physical location in main storage (called **real storage** in MVS). In MVS, an address has 24 positions (called bits). An addressing scheme based on 24-bit addressing allows up to 16,777,216 addresses (16 megabytes).

Of course, a normal system may not have this many real storage locations — and, even if it did, there would be other programs in real storage at the same time so that the 16 megabytes would have to be divided among them.

MVS allows each programmer to use all 16 million addresses, even though real storage includes only, for example, 4 million physical locations.

How?

The range of addresses in a program — from entry to completion, is called the program address space. When a programmer creates a program, he makes within it certain references to required pieces of information. These references are usually of a symbolic nature, such as: CALL UPDATE, where UPDATE begins a series of instructions. In previous systems, each of these program references had to be associated with a real storage location. Thus, specific real storage locations had to be preallocated to them leading, as we mentioned, to the problem of fragmentation.

In MVS, references in the program address space are **not** associated with a particular real storage location. They remain references to a particular piece of information. But where does virtual come in?

The references in the program are not references to real storage addresses but to pieces of information, they are called virtual addresses. They become real only when assigned to a physical location, and these real locations need not be assigned either contiguously or in a particular place. For example, the program might occupy 16,000 bytes in lower storage, 48,000 in the middle of storage, and another 64,000 bytes at the higher end of real storage. If that program had to be removed from real storage and later returned, it could be located or loaded anywhere in real storage: that is, it need not be in the same location as before.

When the program is ready to execute, the system, using a System/370 hardware feature called Dynamic Address Translation (or, more familiarly, the DAT feature), maps the virtual addresses in the program to the real storage addresses and resolves all references (for a more detailed description of this process see Chapter 2: Virtual Storage Management). By doing this, MVS can make the program address space larger than the number of physical locations available in real storage because each program can create references up to the theoretical limit of the addressing scheme: 16 megabytes. Thus, each program can operate as if it had access to all of storage.

In summary, then, there are three levels of addressing in MVS:

1. The theoretical limit, derived from the 24-bit addressing scheme: 16 megabytes. All users of MVS can program up to this limit, that is, there can be multiple virtual user address spaces.
2. Virtual addresses. These are the addresses within the program address space. They refer to a specific piece of information and not to a real storage location.
3. Real storage addresses. These are the addresses of the locations in the storage hardware unit.

When the program is ready to execute, the DAT feature translates the virtual addresses to real storage addresses. The real storage locations that the program occupies depend on which ones are available.

However, addressing is only part of the story. The second part is concerned with how the system makes use of it to do more work, how it allocates and shares real storage.

## Sharing Real Storage

MVS views real storage in 4K blocks called frames. When it allocates storage, that is, assigns storage areas to specific tasks, it allocates a certain number of frames. These frames may be contiguous, but they need not be. Because it allocates storage on a 4K -one page- basis, it minimizes the problem of fragmentation (if a fragment does exist, it will be smaller than 4K). If, for example, there are 10 frames available and they are scattered through storage, MVS can still allocate them as if they were contiguous.

What happens, though, if a hundred programs, each larger than the available real storage are ready to execute at the same time?

When MVS, is fully loaded, the only portion of a program allowed in real storage is one that is active, that is, one that is using the processor or being referenced. The remaining parts of the program remain on auxiliary storage (data storage other than real storage; for example, storage on direct access devices; space on auxiliary storage is called a **slot**; a slot is 4K) until they become active. (***Note:*** The user does not have to worry about any of this ... the system determines what should be in real storage and what remains on auxiliary storage). When a program in real storage must wait, it is moved from real storage to auxiliary storage, and another job or another part of the same program is brought in. (The process of moving a part of a program between real storage and auxiliary storage is called paging; a page is 4K. An access method — see the chapter "Satisfying I/O Requests" — moves the program from direct access storage to real storage and back.) When the program is again ready to execute, it is assigned whatever frames (MVS keeps track of the activity of each frame) are available — not necessarily the same ones it previously occupied. Thus, generally speaking, a program in MVS real storage is a working program, not a waiting one.

### Summary

These are the essential points to grasp about multiple virtual storage:

1. MVS does not waste very much storage. It does not preallocate storage thus significant fragments do not occur. In a fully loaded system, only active portions of programs occupy real storage locations.
2. MVS reduces program design time: the user does not have to worry about fitting his program into real storage.

Because of these factors, MVS can share the real storage resource among many more programs and start many more programs running. Thus, it can do more work.

## *Multiprocessing*

MVS supports many new hardware developments. Among them are:

- The IBM System/370 provides more capacity and speed than previous IBM systems, and at comparable prices. More real storage is available and the cost per byte has been significantly reduced. For example, the System/360 Model 50 had a maximum real storage size of 512K, while many System/370 models have more than 4 megabytes of real storage, more than eight times that of the Model 50.
- Complementing these real storage improvements are faster, more capable processors. For example, the processor cycle time on the System/360 Model 50 was 500 nanoseconds (one-thousand-millionth of a second); on the System/370 Model 158 it is only 115 nanoseconds, and on other models it is less.
- Block multiplexer channels (a multiplexer channel that interleaves -accesses two or more streams of data from distinct storage units simultaneously—blocks of data rather than bytes as in a byte multiplexer channel), not available on System/360, are standard on many System/370 models. While maintaining compatibility with the System/360 selector channels, block multiplexer channels can sustain much higher data rates.

Each of these hardware improvements contribute to MVS's ability to do more work. An even more direct influence is MVS's multiprocessing (MP) capability, which is incorporated into the MVS system control program. The optional MP and the Attached Processor (AP) System can increase the instruction processing capability of the installation; the AP system is discussed later in this chapter and in more detail in Chapter 10: Multiprocessing.)

## Tightly-Coupled & Loosely-Coupled Multiprocessing

Multiprocessing simply means executing two or more tasks simultaneously on two or more processors. It is a logical extension of multiprogramming, in which two or more tasks logically execute concurrently on a single processor.

When a single processor shares a common workload with other processors, but does not share storage, it becomes part of a loosely-coupled multiprocessing complex.

When a single processor shares real storage with another processor, and when both are controlled by a single system control program, they become part of a tightly-coupled multiprocessing complex. Both processors can run under the MVS system control program in multiprocessor (MP) mode. When a single processor is not sharing real storage, it can run under MVS in uniprocessor (UP) mode.

Our emphasis here is on tightly-coupled Model 158 or Model 168 multiprocessors, which have the following characteristics:

- The processors share access to all processor storage available to them.
- The processors communicate by storing data in shared storage and by direct processor-to-processor signals (both program-initiated and hardware-initiated).
- The processors operate under the control of a single operating system (MVS) that is resident in the shared processor storage. The operating system treats the processors as resources, assigning them to process tasks. Also, the operating system maintains one input queue and one task queue and can use either processor to process (although not concurrently) a single job, if necessary.

A component of MVS, called the Job Entry Subsystem (JES2 or JES3) assumes the role of coordinator and controls the flow of work through the system, that is JES controls the entry and exit of work to and from the system.

## Availability

Clearly, if you can now do two things where before you could only do one, you can now do more work. Multiprocessing also offers increased availability. Availability in data processing means the percent of scheduled time the system or an application is capable of processing. A system is available when both its hardware and programming system can process jobs. An application is available when it can perform processing for its end users.

The improved availability MVS offers derives from the ability to:

- Automatically switch from a failing unit to an alternate for it
- In MP, the system can switch work from a failing processor to the good one
- Reconfigure hardware components to fit an installation's needs
- Reconfigure hardware components to allow service personnel to perform concurrent maintenance

Thus, over a period of time, the system does more work because it loses less time due to failing hardware.

### Flexibility

You can divide a multiprocessor into two systems that operate in uniprocessor mode when necessary. For example, you might need a uniprocessor system for preventative maintenance, a test system for a system programmer, or a programming system other than MVS (VM/370, for example). The installation can divide the two systems so that only the hardware components actually required for the special system are allocated to one processor, leaving the balance of the hardware resources available for normal work on the other processor.

Thus, MP not only does more work in the sense of doing two things at one time, but also is available more responding to the different needs of an installation at different times.

### Attached Processor System

The attached processor (AP) consists of a System/370 Model 158 or Model 168 processor (host processor) combined with an attached processing unit to form a tightly-coupled processing system. The host processor provides instruction processing, I/O, and storage functions. The attached processor has a similar instruction processing capability, but no I/O or storage facilities of its own; the attached processor shares the storage facilities of the host processor. When joined in a tightly-coupled configuration to an Attached Processor system, the host and the attached processor provide significantly increased instruction processing power.

## *Error Recovery*

As mentioned, one way of doing more work is to ensure that the system is available when necessary. Multiprocessing is one means of increasing availability. Another is eliminating the need for unscheduled shutdowns.

When an error occurred in previous systems, the system could not do any work until the installation reinitialized the system. When an error occurs in the MVS system, the system attempts to continue operating. MVS attempts to retain availability through error recovery routines that:

- Isolate and record
- Clean up and repair
- Retry and reconfigure

Processing continues while the system carries out these tasks. Primarily, recovery management support and the recovery termination manager perform these functions (see Chapter 9 for more information on error recovery.)

## Recovery Management Support

One means of increasing availability is to reconfigure the system when there is a problem with a hardware component. In this way, the system can continue working. MVS provides this ability through RMS routines.

**Missing Interruption Handler:** The missing interruption handler (MIH) checks whether expected I/O interruptions occur within a specified time period. If the interruptions do not occur, the operator is notified so he can take steps to correct the situation before the system status is harmed.

The MIH checks for pending device ends, channel ends, DDR swaps, and MOUNT commands. When a pending condition is found, the condition is indicated in the UCB of the device. After a specified time elapses, another check is made for the pending condition. If the condition is still pending, a message is used informing the operator what condition is pending and what operator action is required.

**Dynamic Device Reconfiguration:** The operator may invoke dynamic device reconfiguration (DDR) when a device cannot be made ready, or the system may invoke it to bypass a permanent I/O failure. DDR makes it possible to move a demountable DASD or tape volume from one device to another. MVS processes DDR requests without shutting down the system and may eliminate the need for terminating a job.

**Channel Check Handler:** The channel check handler (CCH) receives control when a channel error is detected. CCH builds an error control block and records the error environment. When the CCH is entered due to an error affecting an entire channel, it invokes I/O restart routines to recover the I/O activity on the failing channel.

**Machine Check Handler:** The machine check handler (MCH) in MVS supports the expanded machine check hardware in the IBM System/370. A machine check is an interruption that a malfunction causes. Some machine checks can be corrected by hardware. Others require software recovery. The MCH records all machine checks and invokes software recovery routines when necessary. If the MCH determines that processing cannot continue on a processor, it terminates operations on that processor.

**Alternate CPU Recovery:** When running in MP mode, alternate CPU recovery (ACR) allows work in progress on a failing processor to be recovered on the good processor. The object is to retain system availability and continue system operation.

The ACR routine takes responsibility for all work in progress on the failing CPU, including I/O. If critical I/O devices are symmetrical (that is attached to both processors), or if channel reconfiguration hardware (CRH) is available, critical I/O can be recovered. ACR will attempt to restore resources to an operable state, recover from the failure, and continue operation. (The operator must also take actions such as reducing the workload or reconfiguring hardware if the system is to continue running efficiently.)

ACR is available only in MP and AP mode, and it can provide significant added availability.

## Recovery Termination Management

Recovery termination management (RTM) cleans up system resources when a task or address space terminates. Specifically, RTM performs normal and abnormal task termination, normal and abnormal address space termination, writes dumps, records errors, provides for recovery of supervisory routines via routing control to functional recovery routines, and recovers the system when a processor in a tightly-coupled multiprocessing environment fails. RTM provides these functions for both system and problem program routines.

**Functional Recovery Routines:** FRRs are provided for critical system components — those that have high availability requirements, such as the interruption handlers, the lock manager, and the dispatcher. Upon entry, a functional component establishes an FRR by issuing the SETFRR, a macro instruction. FRR's are placed in LIFO - last in, first out order in an FRR stack maintained by the RTM. Each FRR stack represents the functions being performed in a single path through the system control program. When an error occurs in a path, the RTM passes control to the most recent FRR placed in the appropriate stack. That FRR will attempt to contain the error, record it, repair it, and either request retry or termination. If retry is requested, RTM will reenter the function at a specified location. If termination is requested, the error is passed to the next FRR in the stack to attempt recovery; this process is called percolation.

**Task Recovery:** Task recovery routines may be written for critical units of user or subsystem work. Task recovery routines should be written for those critical user or subsystem tasks that have a high availability requirement. If they are not, the availability of critical subsystems, or critical user jobs may be unnecessarily reduced.

An MVS facility called the extended subtask abend exit (ESTAE) supports task recovery. With this facility, users can write and establish recovery routines in the form of user exits that will receive control at appropriate times during abnormal termination of the task. A recovery exit may be set up when a task is created or it may be established at any time by issuing an ESTAE macro instruction. Each ESTAE routine is placed in LIFO order on a chain established for that task. When RTM is entered, it routes control to the last ESTAE routine in a task's chain. That task recovery routine attempts to contain the error, record it, and repair it if possible. It will then request either retry or termination of the task. If retry is requested, RTM reenters the failing task or subtask at a specified location.

If you want your own exit routine to receive control for certain exceptions, you can issue the specify program interruption exit (SPIE) macro instruction. Any problem program being executed in performance of a task can issue SPIE. When the task is active, your exit routine receives control for all interruptions resulting from exceptions the SPIE macro instruction specifies unless the current routine for the task is operating in supervisor mode. For other program interruptions, control is given to the control program exit routine. Each succeeding SPIE macro instruction completely overrides specifications in the previous macro instruction.

**Percolation:** If an FRR or ESTAE routine is requesting or continuing termination, percolation occurs. The recovery termination manager passes the error to the next recovery routine in the FRR stack or in the ESTAE chain. This represents the previous or the next higher level of control. Hence, the term, percolation. This process continues until a retry results in recovery or until the FRR stack or ESTAE chain has been exhausted.

### *Summary of Direct Benefits*

MVS can do more work because:

1. MVS makes more effective use of real storage, in effect increasing the space available for installation programs.
2. MVS provides more throughput by extensive use of multiprogramming. Through MP and AP it can do two things simultaneously.
3. MVS has higher availability more of the time over the long term through enhanced error recovery function.

## Indirect Benefits

The ability of MVS to do more work also allowed IBM to improve the basic functions of the operating system itself. These indirect benefits lead to:

- Greater support for interactive users
- Improved performance
- Improved security and integrity
- Enhanced functions

### *Greater Support for Interactive Users*

The Time Sharing Option (TSO) is an integral part of MVS. IBM has enhanced TSO as follows:

- Each TSO user is assigned a private address space, and so has more space for processing and is protected from other users.
- TSO users may allocate a greater variety of data sets and devices.
- TSO command processors and service routines may be in pageable storage.
- TSO driver and swapping functions have been integrated into MVS.

TSO makes the operating system available to both local and remote terminal users. A TSO user, identified by a unique userid, can initiate a TSO session by issuing a LOGON command. Each TSO user can develop, test, and execute programs interactively without experiencing the usual delays associated with batch job processing.

#### Sessions and Transactions

MVS allocates data sets and I/O devices to a user at the beginning of a TSO session. In this respect, a TSO session is like a batch job. Interaction with a terminal user involves a terminal read, the appropriate processing, and a terminal write. Each such interaction is called a TSO transaction.

A user may be entering a line of input or compiling a program; both are transactions. Additional resources may be allocated during transaction processing. In this respect, a TSO transaction is somewhat like a batch job step.

Some TSO transactions are trivial and some are not. For example, TSO provides an EDIT facility to create and modify user data sets. When a data set is being created, EDIT prompts the user for a new line of input by displaying a line number. A line of data is entered, stored by EDIT, and a new line number is displayed. This is a trivial transaction because line number prompting requires very little processing and not much I/O.

By contrast, the user may enter a transaction that invokes a COBOL compiler. The response can be a full source listing with compiler diagnostics. This is a non-trivial transaction.

### Terminal I/O

All terminal I/O for TSO is controlled by the telecommunications access method (TCAM) or the virtual telecommunications access method (VTAM). (For information on TCAM and VTAM see *OS/VS TCAM Concepts and Applications*, GC30-2049 and *Introduction to VTAM*, GC27-6987.) A TSO address space is frequently in the wait state since terminal I/O is slow compared to internal processor speeds and terminal users tend to require "think time." During this time, processing is suspended and the user can be swapped out.

### Swapping

Swapping means moving address spaces in and out of real storage. When an address space is swapped out, the virtual storage pages associated with that user are moved from real storage frames to auxiliary storage. Other users who have processing to do can then use the frames. When the swapped-out user is again ready to run, the appropriate virtual storage pages can be swapped in and processing can be resumed.

MVS uses swapping to manage the workload and control the job mix. Swapping takes place for almost all TSO and batch users. An new MVS function, the system resources manager (SRM), makes swapping decisions to meet performance objectives and to balance the use of resources. (For more information on swapping, see Chapter 2.)

## *Improved Performance*

Improved performance derives from control of system resources and a reduction in bottlenecks.

### Control of Performance

As discussed, MVS allows more users (address spaces) to be active concurrently in the system. More users mean more competition for available system resources — processor time, I/O resources, and real storage. An address space has access to these resources only when it is in real storage. The system resources manager (SRM) is the component in MVS that decides which address spaces to swap in or out and when to swap them in or out; therefore, it is the component that controls access to system resources.

The SRM has two objectives:

- Objective One: Meet installation-specified performance guidelines, which reflect the installation's response and turnaround time requirements
- Objective Two: Achieve the optimal use of processor time, real storage, and I/O resources, from the viewpoint of system throughput.

SRM makes decisions that represent trade-offs between these two conflicting objectives.

## Overview of the SRM

The installation specifies its requirements for the first SRM objective in a member of the parameter library (SYS1.PARMLIB) called the installation performance specification (IPS). Through IPS, the installation divides its types of work into distinct groups, assigns relative importance to each group, and specifies the desired performance characteristics for each address space within each group.

A secondary means of specifying requirements to the SRM is through the OPT, member of PARMLIB. (The OPT member contains parameters that affect swapping decisions by the SRM.) Through a combination of IPS and OPT parameters, an installation can exercise a degree of control over system throughput characteristics (objective two). That is, an installation can specify whether, and under what circumstances, throughput considerations are more important than response and turnaround requirements when the need arises to make tradeoffs between objectives one and two.

The SRM attempts to ensure optimal use of system resources by monitoring and balancing resource utilization. If resources are under-utilized, the SRM attempts to increase the system load. If resources are over-utilized, the SRM attempts to alleviate this by reducing the system load or by shifting commitments to under-utilized resources. Examples of such resources are the processor, logical channels, auxiliary storage, and pageable real storage.

For more information on the SRM see Chapter 8. For information on performance analysis see *OS/VS2 MVS Performance Notebook*.

## Reduction in Bottlenecks

A bottleneck is an obstruction, something that slows down work. While specific bottlenecks differ from installation to installation, there are some general ones. MVS design has attempted to reduce the impact of these and improve performance by:

- Reducing path lengths
- Increasing parallelism
- Reducing contention for system resources

These concepts are defined and illustrated in the following descriptions of:

- The Scheduler Work Area
- Device Allocation
- Virtual Input/Output
- Service Request Blocks
- Multiple Locks

**Scheduler Work Area:** In MVS, the scheduler work area (SWA) contains much of the same job control information that the System Job Queue (SYSJOBQE) did in previous systems. SYSJOBQE was a major source of contention in MVT and SVS because almost every component of the job scheduler (and every job in execution) required concurrent access to it.

SWA is, in effect, a local job queue for each MVS user and it resides in the user's private address space. All control information that applies to a single job, such as data set and device allocation information, is placed in SWA when a job is selected. It is available to the job scheduler and the user while he is executing. It improves performance, therefore, by eliminating SYSJOBQE.

**Device Allocation:** The process used to allocate I/O resources is called device allocation. Data sets, volumes, and devices are allocated to a batch user when a job step is initiated and to a TSO user when a session begins. They may also be allocated dynamically. In MVT and SVS, allocation requests are processed one at a time. This serialization eliminates potential conflicts and possible deadlocks. However, in a fully loaded MVT or SVS system, device allocation can be a serious bottleneck.

MVS eliminates this bottleneck by processing requests in parallel. The process may be summarized as follows:

- Associate a user data set with a volume
- Associate the volume with a device
- Allocate the device to the user

Significant performance improvement has been realized through this redesign of device allocation.

**Virtual Input/Output:** In MVS, temporary data sets can be handled by a new facility called virtual input/output (VIO). Data sets for which VIO has been specified reside in paging space on auxiliary storage. However, to a user or to one of the access methods, the data appears to reside in a real data set on a DASD volume. A VIO specification exists only for the duration of the job.

During system generation, one or more unit names can be defined as VIO and associated with a real DASD device type, such as a 3350. These unit names are then specified on the job control statements requesting device allocation. These requests are processed in parallel and no device is allocated for the VIO request.

After the job has gone through the device allocation process, and as data is being stored in 4K blocks on a VIO data set, real storage frames and auxiliary storage slots are assigned as required. These frames and slots may not be contiguous and the data may be dispersed in real storage and on auxiliary storage. When a user accesses a VIO data set, the desired data is paged in and out of real storage as required. The auxiliary storage slots are released when the data set is deleted or the job ends and are immediately available for paging. VIO offers these performance advantages:

- Elimination of some device allocation and data management overhead
- Generally more efficient use of DASD space
- Use of the I/O load balancing capability of the auxiliary storage manager (ASM)

**Service Requests:** Service requests, a new facility in MVS, improve performance and make MVS a more responsive system. The system, a privileged (authorized) user, or subsystem may issue them.

The requester builds a service request block (SRB) and issues the SCHEDULE macro instruction. The SRB represents work to be done and the SCHEDULE macro instruction places the SRB on one of the service manager queues. An SRB for a particular address space is given control before any tasks associated with that address space.

An SRB is an efficient way to communicate between address spaces. SRBs also make it possible to handle multiple events in parallel.

**Multiple Locks:** A lock is a means of serialization. MVS has implemented multiple system locks to improve and standardize serialization techniques. There are two different categories of locks. A global lock protects a serially reusable resource that relates to the whole system — for example, there is a global lock for each unit control block (UCB) associated with each device in the system. A local lock serializes address space related storage areas. Implementation of these locks offers the MVS user these performance improvements:

- A standard for path serialization techniques
- Less disabled processor time and a more responsive system
- More parallelism and less contention

## Improved Security and Integrity

Increased security and integrity are major design objectives of MVS:

- Security is the ability to protect resources from unauthorized access, alteration, or destruction.
- Integrity is the inability of any program not authorized by a mechanism under the customers control to:

  1. Circumvent or disable store or fetch protection

  2. Access a password-protected or a RACF-protected resource (RACF is the Resource Access Control Facility program product)

  3. Obtain control in an authorized state, that is, in supervisor state, with a protection key less than eight, or protected by the authorized program facility

A goal of MVS is to build integrity into the base system so that if an installation wishes, it can add a security system to it.

### *Isolate and Protect*

In MVS, virtual storage consists of a system area, a common area, and a private area. Every MVS user can address one private area. MVS isolates each user from every other user in a private address space — thereby preventing him from violating another user's address space. MVS uses multiple storage protect keys to protect the system and subsystems from unauthorized users.

### *Validate and Authorize*

Before MVS performs services on behalf of the users, it takes steps to validate any protected resources that are to be used and to authorize the use of any restricted functions. This is done to prevent possible security violations through the use of invalid control blocks or the execution of unauthorized code and to avoid user-induced system failures due to improperly specified requests.

### *User Responsibility*

To avoid compromising MVS security, each installation must assume responsibility for:

- The integrity of user written authorized programs
- Password protection of critical system libraries
- Access to the system by programmers and operators
- The physical security of the computing systems

Increased security and integrity costs some processor time and real storage space. However, every effort has been made to employ efficient programming techniques that do not significantly impact performance.

## Enhanced Function

There is an overall enhancement of function in MVS. MVS function has been enhanced by integrating into the system many functions that previously were only available as add-on support and by extending these functions to include multiple virtual storage. In particular this enhancement applies to:

- JES2 and JES3
- System generation and initialization
- The virtual storage access method (VSAM)

### *Job Entry Subsystem*

Job management has been enhanced by the implementation of JES2 and JES3. Either JES2 or JES3 may be specified as the primary job entry subsystem. Job management in MVS is handled by the job entry subsystems

(JES2 and JES3). They control the entry of jobs and perform job scheduling functions upon request. MVS interfaces with these job entry subsystems via a new component, the subsystem interface (SSI). For further information on JES, see Chapter 5.

### JES2

JES2 is the MVS replacement for HASP II (Houston automatic spooling program). Most of the functions performed by HASP II have been integrated along with many functions formerly performed by the job scheduler in MVT and SVS. These are some of the functions performed by JES2:

- Reading jobs and SYSIN data, both local and remote
- Spooling jobs and input data to direct access storage
- Scheduling, initiating, and monitoring jobs
- Reading SYSIN data and writing SYSOUT data for active jobs
- Writing jobs and SYSOUT data, both local and remote

An extensive set of JES2 operator commands is provided. Job accounting, journaling, and restarting capabilities have been integrated into the subsystem; and the scheduling of TSO sessions and the control of batch output for TSO users is done by JES2.

### JES3

JES3 functions, integrated into MVS, are generally the equivalent of those in ASP (asymmetric multiprocessing system) Version 3. Multiple processors in a variety of loosely-coupled combinations are supported.

When JES3 is used to manage a loosely-coupled multiprocessing complex, it controls job scheduling and device allocation for the entire complex. The controlling processor is called a global processor and the others are called local processors or ASP mains. A local processor with access to the necessary I/O devices and connected to all other processors can assume global functions if the global processor fails. JES3 provides even more extensive job management functions than those listed for JES2. In addition to increasing availability, JES3 permits more efficient use of system resources by providing:

- Automatic scheduling of jobs to multiple processors
- Controlled allocation of all I/O devices in the complex
- Mounting and verifying of private volumes before scheduling a job
- Deadline scheduling

### Subsystem Interface

Both JES2 and JES3 use MVS functions and service MVS requests. Each is considered a subsystem and communicates with MVS via a component, called the subsystem interface (SSI). SSI makes it easier to add subsystems to MVS, including those written by users.

## *System Generation and Initialization*

During system generation and system initialization, an installation can select options and specify parameters that tailor an operating system to meet specific needs. In MVS, the number of SYSGEN options that must be

specified have been minimized and initialization flexibility has been increased. Operating procedures have been simplified and dependence upon the system operator has been reduced, while the control of system resources has become more automated during system initialization. Preset initialization options may be stored in the parameter library and invoked by specifying the parmlib member at initial program load (IPL).

### System Generation

Macro instructions are used during system generation to select options from IBM Distribution Libraries (DLIBs). This process has been simplified for MVS in the following ways:

- Many previous options are now standard
- Several macro instructions have been eliminated, consolidated, or clarified
- Multiple jobs can be run to speed up the SYSGEN process

See Chapter 3 for more information on system generation.

### System Initialization

The installation can use the console to select parameter lists from PARMLIB or to specify additional parameters during system initialization. In MVS, changes have been made to the initialization process that provide greater flexibility in specifying parameters, and that simplify the process by reducing the amount of operator intervention required. These changes include:

- Fewer operator messages and fewer replies
- Multiple parameter lists and selective merging of parameters

See Chapter 4 for more information on system initialization.

### System Operation

MVS depends less upon the system operator than any of its predecessors. Operator commands are used to request system and user status and to initiate, alter, or terminate system functions. Many functions that previously depended upon operator commands are now performed by JES2 or JES3. In some cases, the system may not wait for operator intervention when devices being allocated are offline or not ready. The operator is usually not required to make job scheduling and storage configuration decisions.

## *Virtual Storage Access Method (VSAM)*

The virtual storage access method (VSAM) is a high performance access method for direct access storage. It is designed to run in virtual storage and uses virtual storage to buffer input and output operations. VSAM provides support for batch users, online transactions and data base applications. Through a master catalog, VSAM controls the allocation of data space on VSAM volumes and the location and use of VSAM data sets. In MVS, the VSAM master catalog is also the system catalog. (See Chapter 8 for more information on VSAM.)

## Summary

Through better management of real storage, increased multiprocessing and instruction processing capability, and enhanced error recovery MVS can do more work than previous systems. This has improved the system's basic operating capabilities, especially in the areas of resource management, integrity, and function.

MVS integrates many items, such as TSO and tightly-coupled multiprocessing, into the overall system that has been special purpose options.

Some of the major new features that MVS includes are recovery facilities, VSAM, virtual I/O, and multiple virtual address spaces.

MVS offers more space to more users, greater throughput, high availability, and more control of the system. In short, it does more work than previous systems.

# Chapter 2: Virtual Storage in MVS

Storage in an MVS system — or any computing system, for that matter — consists of a number of locations available for programs and data. In a system without virtual storage, the range of addresses (the number of storage locations, each having a unique address) is equal to the number of addressable physical locations in the main storage installed. In a system with virtual storage, however, the range of addresses available for programs and data is equal to the theoretical limit of the addressing scheme. In MVS, this theoretical limit — the size of the virtual storage available to the programmer — is 16 megabytes, the maximum number of addresses allowed by the 24-bit addressing scheme that MVS uses. Virtual storage is larger than main storage (called real storage in MVS); how much larger depends on the size of real storage installed. Therefore, the use of virtual storage increases the number of storage locations available to hold programs and data.

In most computing systems, a program cannot execute unless there is a single block of storage big enough to hold it, and the block of storage is allocated to the program until it has finished. However, when a program executes in virtual storage under MVS, only the parts of the program that are currently active need be in real storage at any particular time. The inactive parts of any executing program are held in auxiliary storage, in special data sets that most probably reside on a high-speed direct access device. Thus, the programmer is freed from the problem of designing a program to fit a predetermined limit of real storage. Additionally, more programs can occupy real storage concurrently because only the active parts of each program are in real storage at any particular time; thus, the system can start more jobs.

## Pages, Frames, and Slots

To enable the movement of the parts of a program executing in virtual storage between real storage and auxiliary storage, the MVS system breaks real storage, virtual storage, and auxiliary storage into blocks:

- A block of real storage is a frame.
- A block of virtual storage is a page.
- A block of auxiliary storage is a slot.

A page, a frame, and a slot are all the same size; each is 4K bytes. An active virtual storage page resides in a real storage frame; an inactive virtual storage page resides in an auxiliary storage slot. Moving pages between real storage frames and auxiliary storage slots is called paging.

Figure 2.1 shows how paging is performed for a program running in virtual storage. Parts A, B, and C of a three-page program are in virtual storage. Page A is active and executing in a real storage frame, while pages B and C reside in auxiliary storage slots. At point (1) page B is required; the system brings B in from auxiliary storage and puts it in an available real storage frame. At point (2) page C is required; the system brings C in from auxiliary storage and puts it in an available real storage frame. If page A became inactive and the system needed its frame in real storage, page A would be moved to an auxiliary storage slot, as shown at point (3).

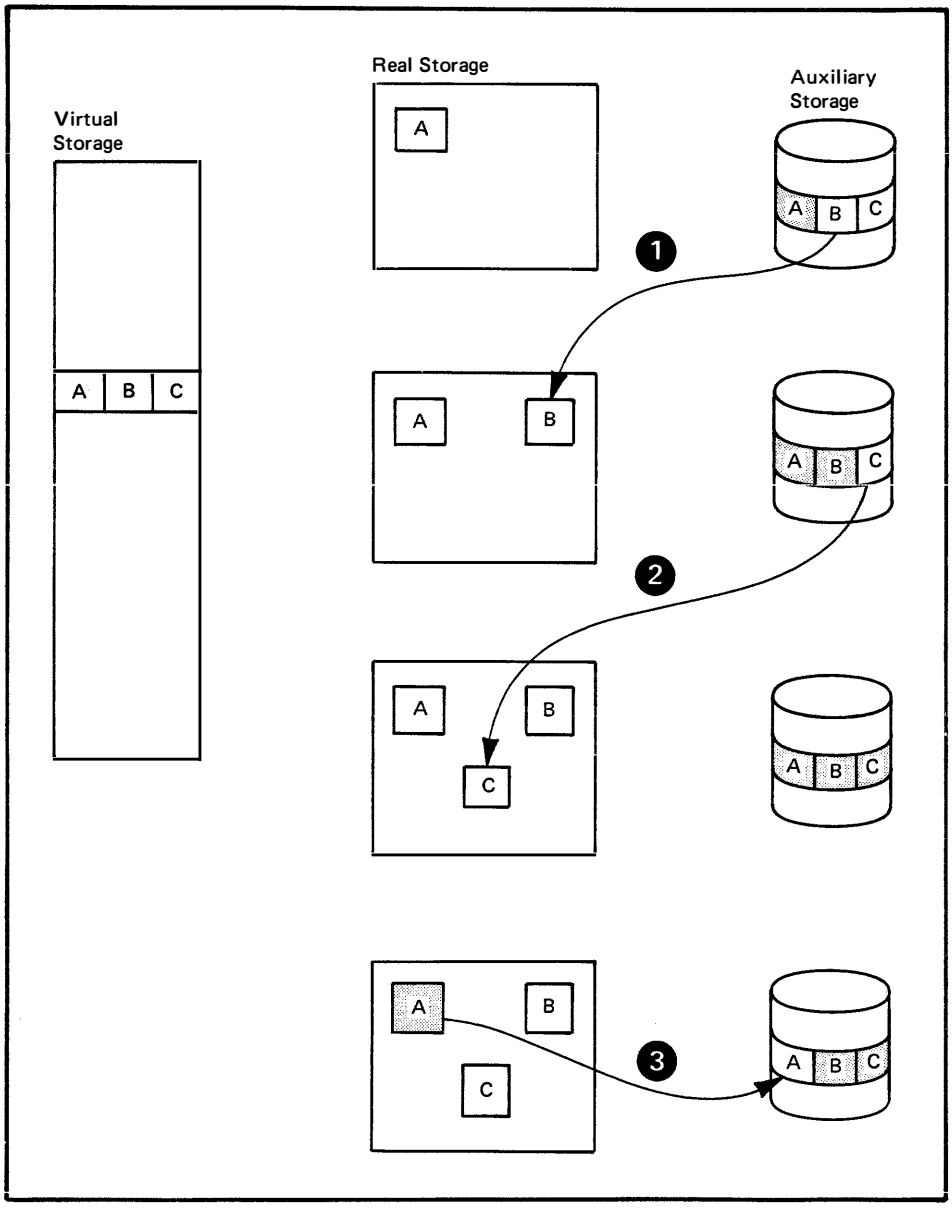



**Figure 2.1. Basic Virtual Storage Concepts**

Thus, the entire program resides in virtual storage; the system moves pages of the program between real storage frames and auxiliary storage slots to ensure that the pages that are currently active are in real storage when they are required. Note also that both the frames and the slots allocated to a program need not be contiguous; thus, a page could occupy several different frames and several different slots during the execution of a program. That is, if page A in the example become active again, MVS could move it to any available frame.

## Integrity

Figure 2.1 showed how virtual storage works for one program; in reality, of course, many programs or users would be competing for use of the system. MVS implements two techniques to preserve the integrity of each user's work: (1) a private address space for each user and (2) multiple storage protect keys. Each of these techniques is described in the following text.

### *Storage Protect Keys*

Under MVS, the information in real storage is protected from unauthorized use by means of multiple storage protect keys. A control field in storage called a key is associated with each 2K block of real storage. This field or key, sometimes called a "storage bump," is not part of addressable storage.

The key in storage contains the protect key of the owner and a fetch protect bit (as well as the reference and change bits maintained by the hardware and used by the software to make paging decisions, as described later in this chapter under "Paging.") The protect key protects the block of storage from unauthorized modification. The fetch protect bit protects the block of storage from an unauthorized attempt to read or fetch its contents. Figure 2.2 shows the format of the key in storage.

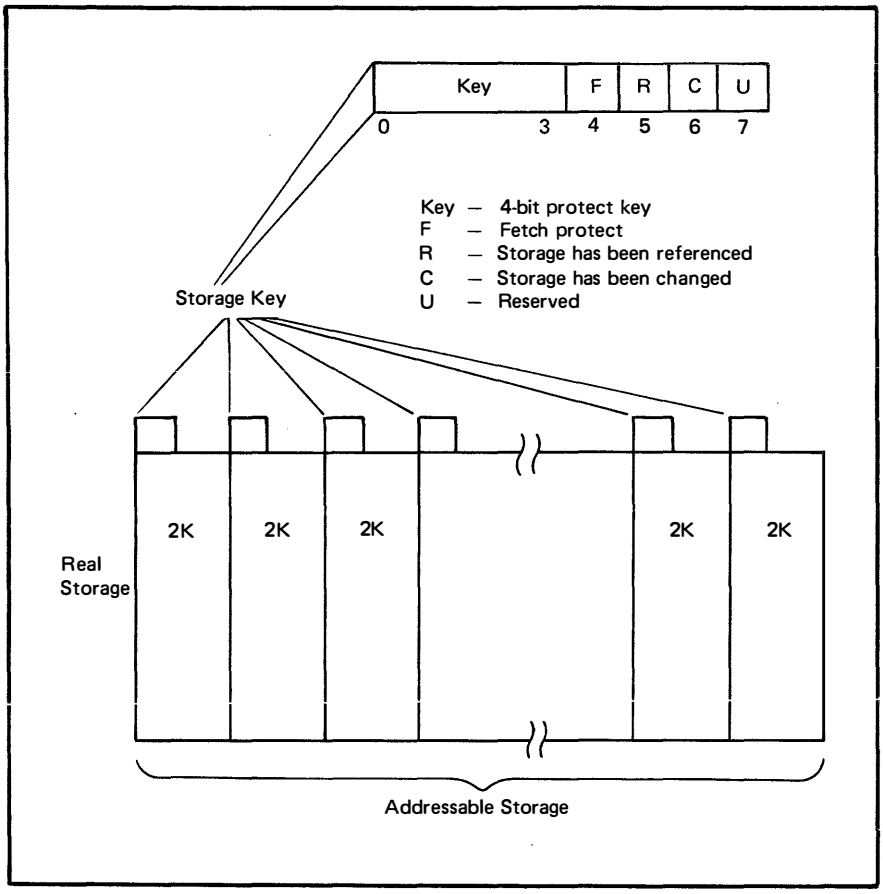



**Figure 2.2. The Key in Storage**

When a request is made to modify the contents of a real storage location, the key in storage is compared to the storage protection key associated with the request, which appears in the current program status word (PSW). (See "The Role of Program Status Words" in Chapter 6 for more information about the PSW.) If the keys match, the request is satisfied. If the key associated with the request does not match the key in storage, the system rejects the request and issues a program exception interruption.

When a request is made to access (read or fetch) the contents of a real storage location, the request is automatically satisfied unless the fetch protect bit is on. When the fetch protect bit is on, the block of storage is fetch-protected. When a request is made to access the contents of a fetch-protected real storage location, the key in storage is compared to the key associated with the request. If the keys match, the request is satisfied. If the keys do not match, the system rejects the request and issues a program exception interruption.

There are sixteen possible storage protect keys available. A specific key is assigned according to the type of work being performed. Figure 2.3 summarizes the assignment of storage protect keys.

Storage protect keys 0 through 7 are reserved for the MVS system control program and various subsystems. Storage protect key 0 is the master key. When a storage protect key of 0 is associated with a request to access or modify the contents of a real storage location, the request is automatically satisfied. Thus, the use of key 0 is restricted to those parts of the MVS system control program that require unlimited store and fetch capabilities.

Storage protect keys 8 through 15 are assigned to users. Because all users are isolated in private address spaces, most users — those whose programs run in a virtual region — can use the same storage protect key. These users are assigned a key of 8. Some users, however, must run in a real region. These users require individual storage protect keys, which are assigned from the range of 9 through 15. Descriptions of a virtual region and a real region appear later in this chapter under "Virtual (V=V) User Region" and "Real (V=R) User Region."

| Key | Use |
|---|---|
| 0 | MVS system control program |
| 1 | Job scheduler and job entry subsystems (JES2 or JES3) |
| 2-4 | Reserved |
| 5 | Data management |
| 6 | TCAM and VTAM |
| 7 | IMS |
| 8 | V=V users |
| 9-15 | V=R users |

**Figure 2.3. Storage Protect Key Assignment**

Frequently, a user program requests a service from a system (or subsystem) program; with the request the program passes the address of an area in storage to be modified by the system program. This area should belong to the user. However, if an error occurs and the area really belongs to the system instead of the user, the system could be destroyed. Thus, the system program does a key switch before performing the service for the user. A key switch means that the system program uses the storage protect key of the user rather than its own storage protect key while performing the requested service. The key switch is thus another mechanism MVS uses to provide protection from possible destruction.

### *Address Space*

MVS assigns each user his own map of virtual storage. The 16-megabyte virtual storage available to each user is called an address space. A 16-megabyte address space is available to each job, TSO user, or system task. Each address space competes with all other active address spaces for the use of real storage and other system resources, and the work being performed in each address space is paged between real and auxiliary storage.

In order for this paging activity to take place quickly and efficiently, the system must be able to translate a virtual address (the address of a specific instruction or data item in virtual storage) into a real address (the address of the corresponding location in real storage). The solution is dynamic address translation.

## Dynamic Address Translation

Dynamic address translation (DAT) is a System/370 hardware feature that makes virtual storage possible. The DAT feature hardware works in conjunction with MVS system software to translate a virtual address into a real address.

### *Virtual Address*

In order to obtain a virtual address, MVS breaks the 16 megabytes of virtual storage into 256 segments, numbered 0 through 255. Each segment consists of 64K bytes. The 64K bytes in each segment are further broken down into 16 pages, numbered 0 through 15. Each page, as stated earlier, consists of 4K bytes. Within each page, a specific location is addressed by its byte displacement, that is, the number of bytes between the page origin and the specific location.

A virtual address consists of the segment number, the page number within that segment, and the byte displacement within that page. Figure 2.4 shows how virtual storage is broken down to provide a virtual address that consists of a segment number, a page number, and a byte displacement.

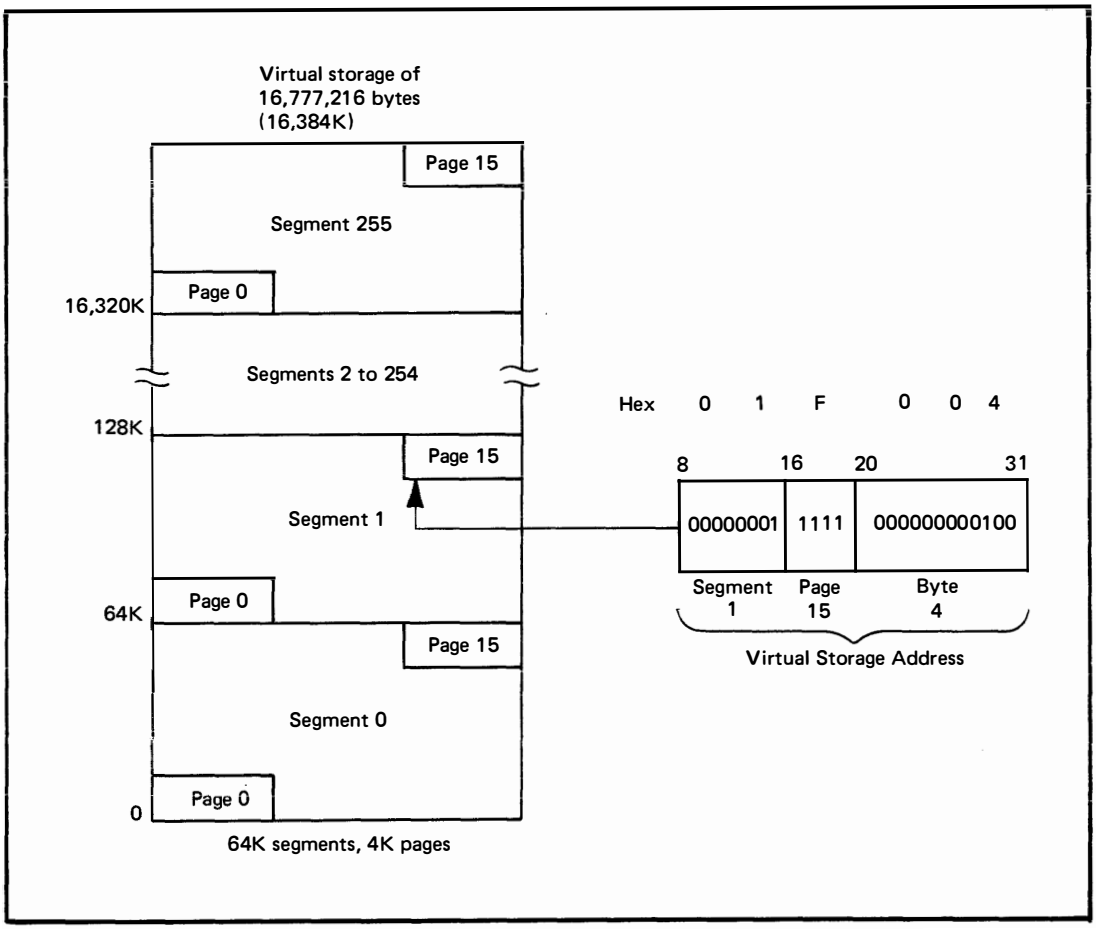



**Figure 2.4. Virtual Storage Address**

## *Segment and Page Tables*

To translate a virtual address into a 24-bit real address, the DAT feature requires tables that describe each address space. These tables are the private segment table and the private page tables. The segment table has one entry for each of the 256 segments in the address space; each entry contains a pointer to the page table for that particular segment. The page table for each segment has one entry for each of the 16 pages in the segment. If a page is currently in a real storage frame, the entry consists of the real storage address of that page. If a page is not currently in real storage, the entry in invalid; that is, the system must move the page from auxiliary storage to real storage and update the page table before the virtual address can be successfully translated. Figure 2.5 shows the relationship between the segment table, the page tables, and the pages in virtual storage.

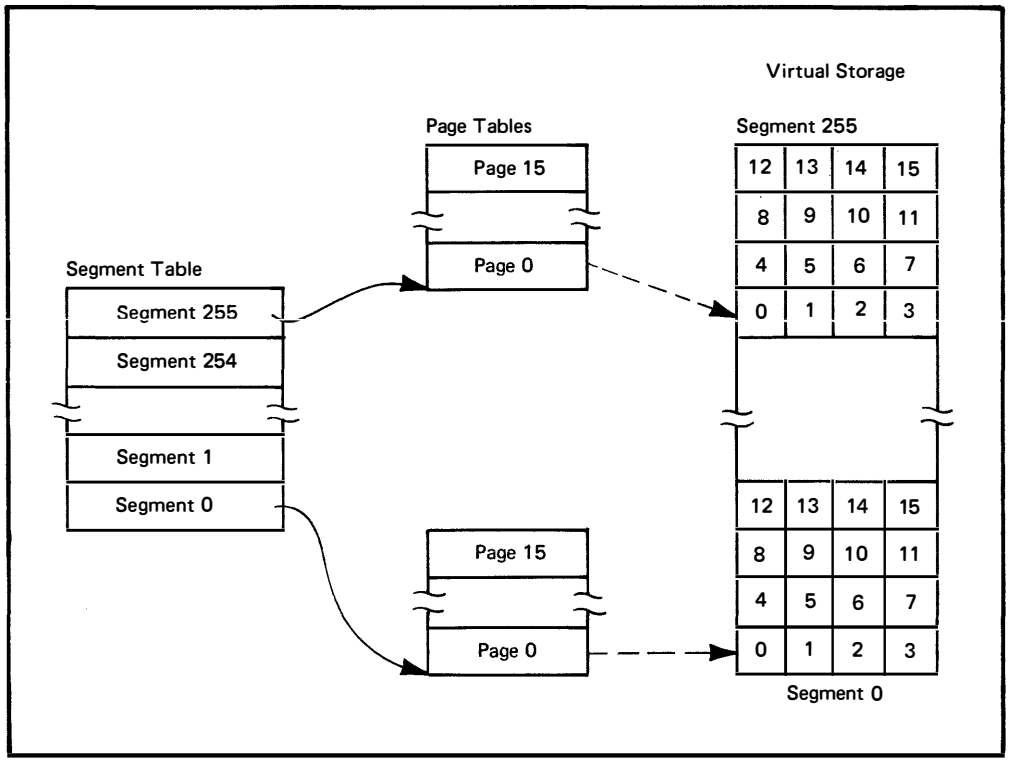



**Figure 2.5. Segment Table and Page Tables**

### *Two–Level Table Lookup*

To translate a virtual address into a real address, DAT uses a two.level table lookup. Figure 2.6 illustrates this process. The first table lookup (1) uses the segment table origin in the segment table origin register (STOR) and the segment number in the virtual address (multiplied times 4, the length of each segment table entry) to locate the origin of the page table for that segment. The second table lookup (2) uses the page table origin from the segment table entry and the page number in the virtual address (multiplied times 2, the length of each page table entry) to locate the required entry in the page table. Unless the entry is invalid, the page table entry contains the address of the real storage frame that holds the page specified in the virtual address. The final step (3) in dynamic address translation adds the address of the real storage frame to the byte displacement in the virtual address to compute the 24-bit real address. This value is loaded into a hardware storage address register (SAR).

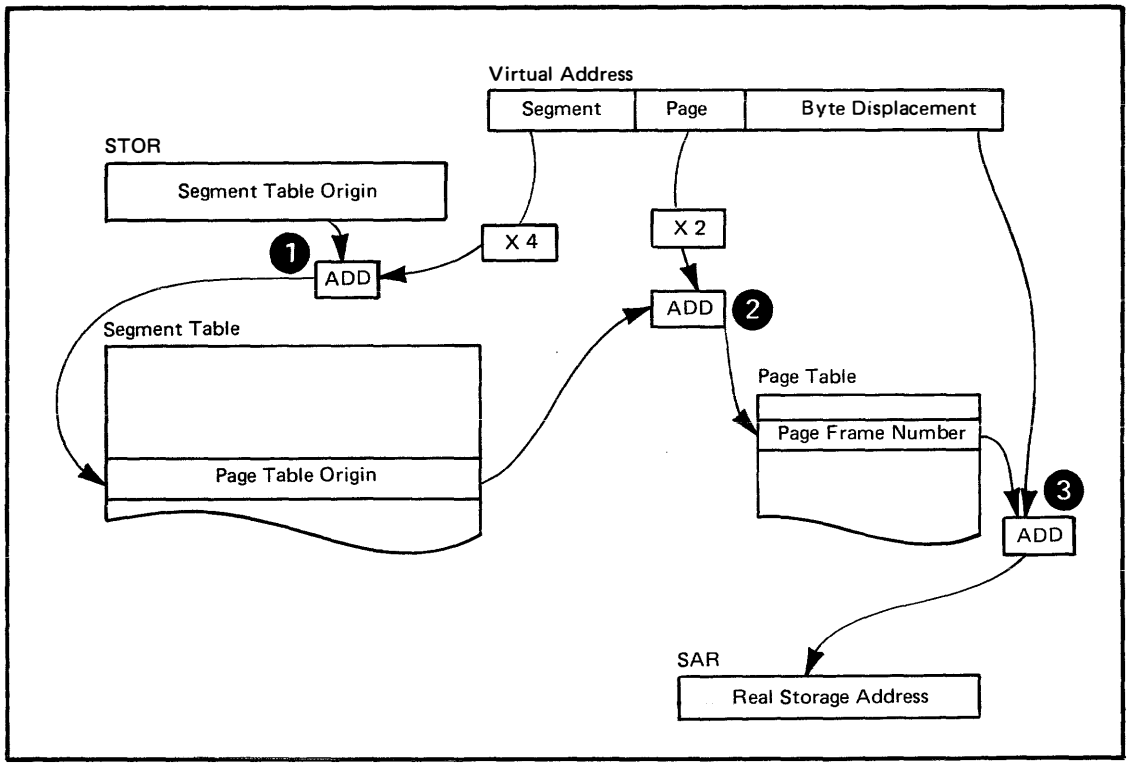



**Figure 2.6. Dynamic Address Translation**

Each time a virtual address is successfully translated into a real address, the system saves the address of the real storage frame in a special hardware buffer called the translation lookaside buffer (TLB). The TLB contains the segment number and page number from the virtual address and the corresponding real storage address for the most active virtual pages. The DAT hardware checks the TLB before beginning the process of address translation, and, because a very high percentage of addresses can be found in the TLB, address translation time is significantly reduced by bypassing the two–level table lookup process.

When the second step of the table lookup process encounters an invalid page table entry, the required page is not in real storage. The DAT hardware thus cannot translate the virtual address, and a page translation exception, known as a page fault, occurs. Paging — the movement of pages between auxiliary storage and real storage — is required to bring the page into real storage.

## Paging

Paging is the movement of pages between real storage and auxiliary storage to ensure that currently active pages are in real storage. In addition to the DAT hardware and the segment and page tables required for address translation, paging activity involves a number of system components to perform the movement of pages and several additional tables to keep track of where each page is at any particular time.

### *Demand Paging*

To understand how paging works, assume that DAT encounters an invalid page table entry during address translation, indicating that a page is required that is not in a real storage frame. To resolve this **page fault**, the system must locate an available real storage frame. If there is no available frame, an assigned frame must be freed. To free a frame, the system moves its contents to auxiliary storage. This movement is called a page-out. The system performs a page-out only when the contents of the frame have been changed since the page was brought into real storage.

Once a frame is located for the required page, the contents of the page are moved from auxiliary storage to real storage. This movement is called a page-in. The process of bringing a page from auxiliary storage to real storage in response to a page fault is called **demand paging**.

MVS tries to avoid the time-consuming process of demand paging by keeping an adequate supply of available real storage frames constantly on hand. Swapping is one means of ensuring this adequate supply. Page stealing is another.

### *Swapping*

Swapping is the movement of an entire address space between virtual storage and auxiliary storage. It is one of several methods MVS employs to balance system workload, as well as to ensure that an adequate supply of available real storage frames is maintained. Address spaces that are swapped in are active, currently executing in virtual storage with pages in real storage frames and pages in auxiliary storage slots. Address spaces that are swapped out are inactive; the address space resides on auxiliary storage and cannot execute until it is swapped in. Swapping is performed in response to recommendations from the system resources manager (SRM), described later in this book in "Chapter 7: Managing System Resources."

### *Page Stealing*

In addition to swapping, the system uses page stealing to ensure an adequate supply of available real storage frames. Page stealing occurs when the system takes a frame assigned to an active user and makes it available for other work. The decision to steal a particular page is based on two factors: (1) the size of the working set for an address space and (2) the activity history of each page currently residing in a real storage frame. The **working set** is the number of virtual pages that should reside in real storage frames in order for work in an address space to run effectively. Each user — that is, each address space — has a working set, and the system does not steal pages from the working set under normal operating conditions.

### Page Frame Table

Any active pages that exceed the working set, however, are candidates for page stealing. To determine the pages that are to be stolen, MVS examines the activity history of the pages that are currently in storage. This information is held in the page frame table. There is one page frame table for the entire system, and it has an entry for each frame of real storage. Each entry includes the address space identifier and the segment and page number within the address space for the virtual page that is currently using the frame.

Other information in the entry describes the activity history of the page. The status field indicates whether the frame is currently in use; if the status field is set to zero, the frame is available. Two additional bits associated with the entry, the reference bit and the change bit, are relevant when the frame is in use. (***Note:*** These bits are actually part of a control field associated with each 2K block of storage. They are maintained by the hardware and used by the software to make paging decisions; they are therefore described here as if they were physically part of the page frame table.)

The reference bit is set on whenever the page is referenced. At regular intervals, the system sets the reference bits back to zero. Thus, the reference bit is an indication of how recently the page has been used. A page in storage with the reference bit set off has not been referenced recently; it is a candidate for page stealing.

The change bit is set to zero when a page is initially brought into a real storage frame. When the contents of the page are changed during execution of work in the address space, the change bit is set on. Setting the change bit on tells the system that it must move the contents of the frame to auxiliary storage before making the frame available for other work. Checking the change bit ensures that no changes made during program execution are lost during the paging process.

Figure 2.7 shows how the page frame table entries are set up and how the status, reference, and change information is used to determine which pages will be stolen. All of the pages in the table are active; the status field is set to one. The system checks the reference bits and finds two pages that have not been referenced recently and are, therefore, temporarily inactive. These two pages will be stolen. The first page (1) has not been changed since it was brought in from auxiliary storage; therefore, no physical page-out is required to save its contents because the copy of the page in real storage is the same as the copy of the page in auxiliary storage. The second page (2) has been changed; therefore the system performs a page-out before it steals the page, and the contents of the page are written to auxiliary storage. The system is thus able to steal two pages, only one of which requires a page-out. To save the time required to perform a page-out, the system, whenever possible, steals pages that have not been changed.

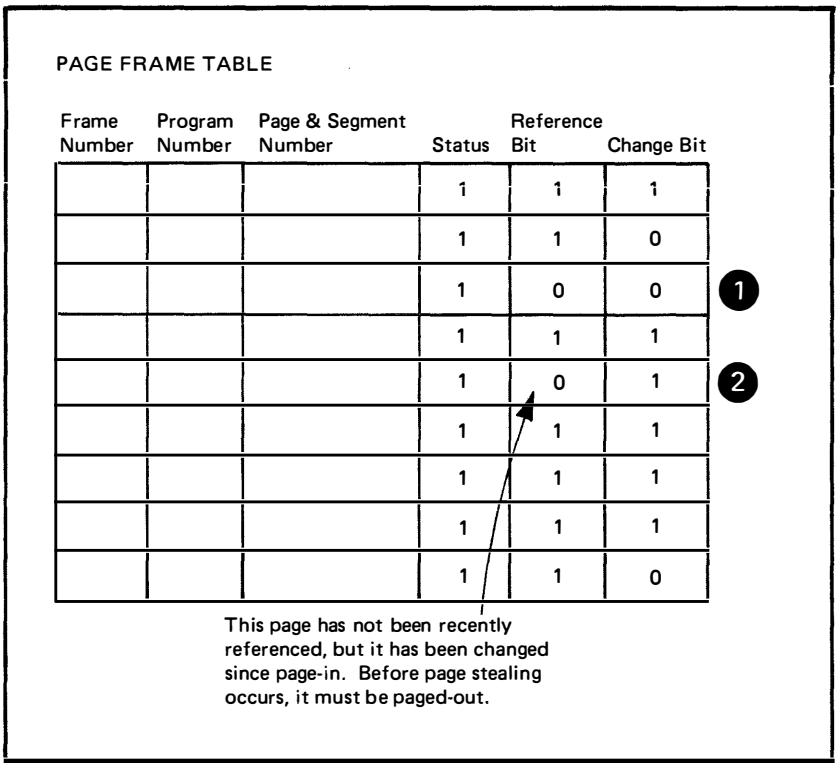



**Figure 2.7. Page Frame Table**

## *System Components*

Through swapping, page stealing, and, when required, demand paging, MVS ensures that the most active pages of each address space are in real storage when required and keeps track of the exact location of each page. This complex paging process is transparent to the user; each program runs in its own address space as if it were the only program executing at any particular time and as if it had all of virtual storage at its disposal. The paging process is managed by several components of MVS. The three major ones are the real storage manager, the auxiliary storage manager, and the virtual storage manager.

### Real Storage Manager (RSM)

The real storage manager (RSM) checks and maintains the entries in the page frame table. It determines which pages are to be moved out of real storage in response to a request for swapping an entire address space out of storage or in response to a need for page stealing or demand paging.

The real storage manager also verifies the storage protect keys. The use of storage protect keys is described earlier in this chapter under "Storage Protect Keys."

### Auxiliary Storage Manager (ASM)

The auxiliary storage manager (ASM) to keeps track of the contents of the page data sets and swap data sets. Page data sets contain virtual pages that are not currently occupying a real storage frame. Swap data sets contain the virtual pages for address spaces that have been swapped out.

The ASM also maintains a table called the external page table. Entries in the external page table enable ASM to determine the location of a page residing in an auxiliary storage slot. When a page-in is required, the RSM locates an available frame, and the ASM uses the external page table to find the required page on auxiliary storage and bring it into real storage. When a page-out is required, ASM locates a slot on auxiliary storage, moves the page from real storage to auxiliary storage, and updates the external page table.

### Virtual Storage Manager (VSM)

The virtual storage manager (VSM) provides the map of virtual storage for each address space. VSM works with RSM to handle subpool management, requests to obtain and free storage, and storage allocations for programs that must run in real storage rather than virtual storage.

Figure 2.8 summarizes the paging process, showing how pages move between real and auxiliary storage in response to a page fault or to fill the need for an adequate supply of real storage frames.

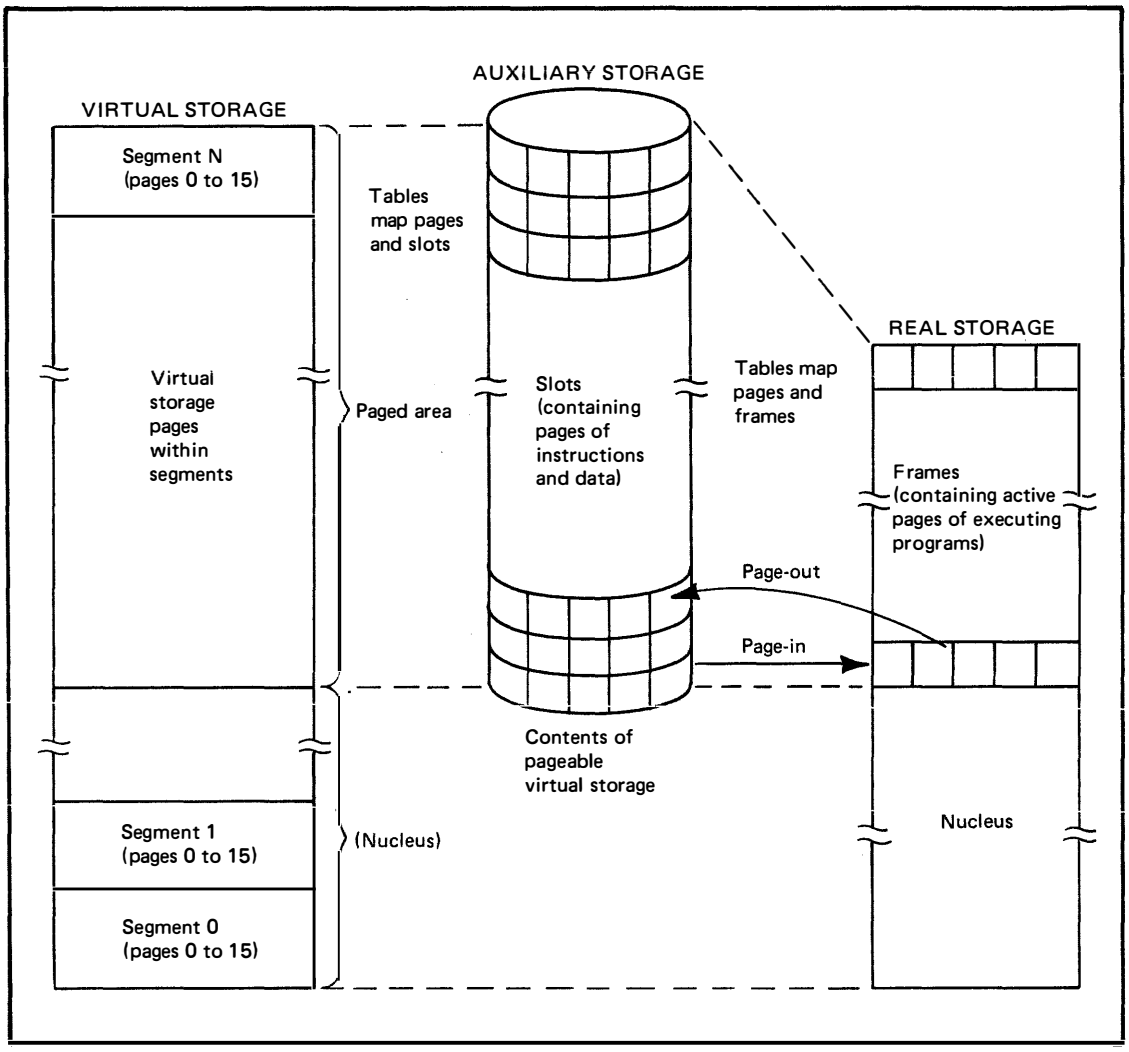



**Figure 2.8. Page-out and Page-in**

## *Program Loading*

Paging also takes place when the program loader initially loads a program into virtual storage. The program loader brings an entire program into virtual storage from the library on which the program resides. Virtual storage is obtained for the user program. Each page in the program is brought into real storage; that is, a real storage frame is allocated to each page and an entry, including reference and change bits, is built in the page frame table. Each page is then active and subject to the normal paging activity; that is, the most active pages are retained in real storage while the pages not currently active are paged out to auxiliary storage. Figure 2.9 summarizes the program loading process.

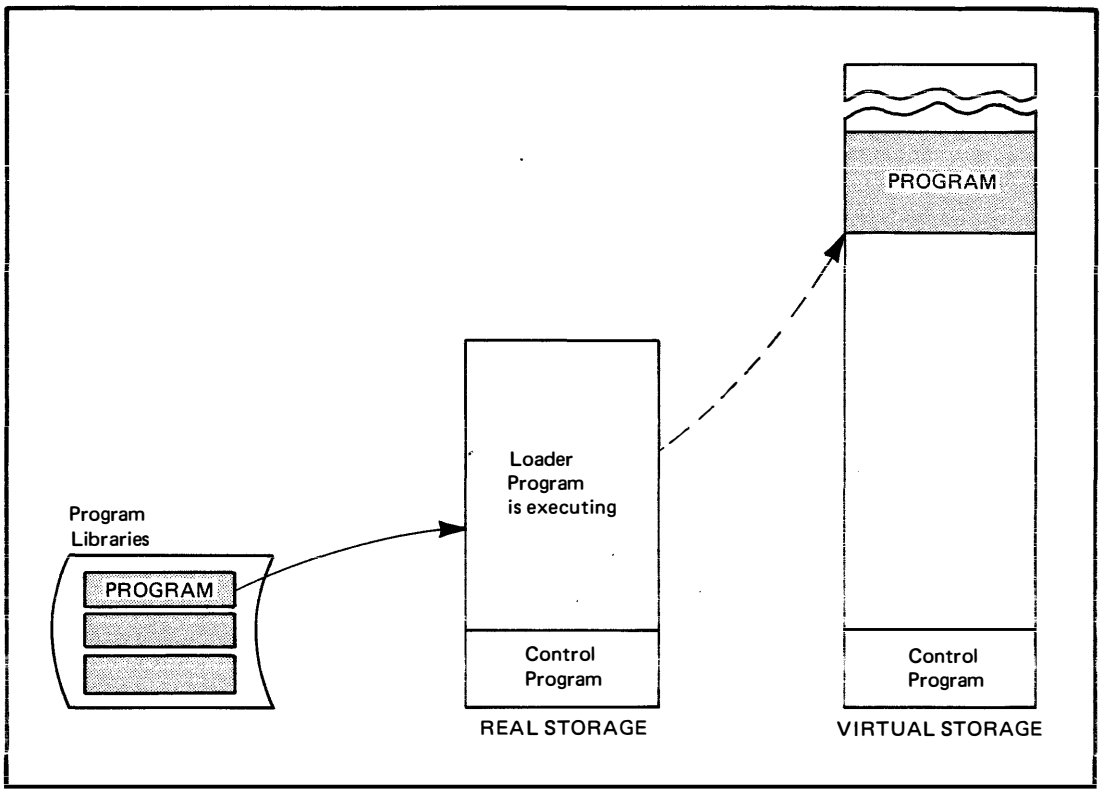



**Figure 2.9. Program Loading**

Up to this point, virtual storage has been described as if the entire 16-megabyte address space is available for user programs and as if all of real storage is available for paging. As Figure 2.8 and 2.9 show, however, some virtual storage and a corresponding amount of real storage are taken up by the control program, also called the nucleus. In most systems, an area of approximately eight to ten megabytes is available for user programs in an address space. The map of virtual storage for each address space includes both the areas used by the control program and the area available for a user program. The remainder of this chapter describes the map of virtual storage in more detail to show how storage is organized in MVS to make effective use of real storage, an important system resource.

## Virtual Storage Areas

Each virtual storage area consists of a system area, a private area, and a common area. The address space each user controls enables him to address all three areas. However, private segment and page tables and storage keys isolate one address space from all other address spaces and protect the system from destruction.

Figure 2.10 shows the major parts of virtual storage. The system area (1) and the common area (2) contain the system control program and various routines and data areas that pertain to the entire system. The private area (3) is the area available for user programs. As the figure shows, both the common area and the private area contain several separate parts. The contents of the system area, the common area, and the private area are described in the following text.

In addition to the basic storage layout shown in Figure 2.10, the system area and the common area can be extended or changed, depending on the configuration or options a particular installation selects. These additions to the storage layout are described later in this chapter under "Extensions and Options."

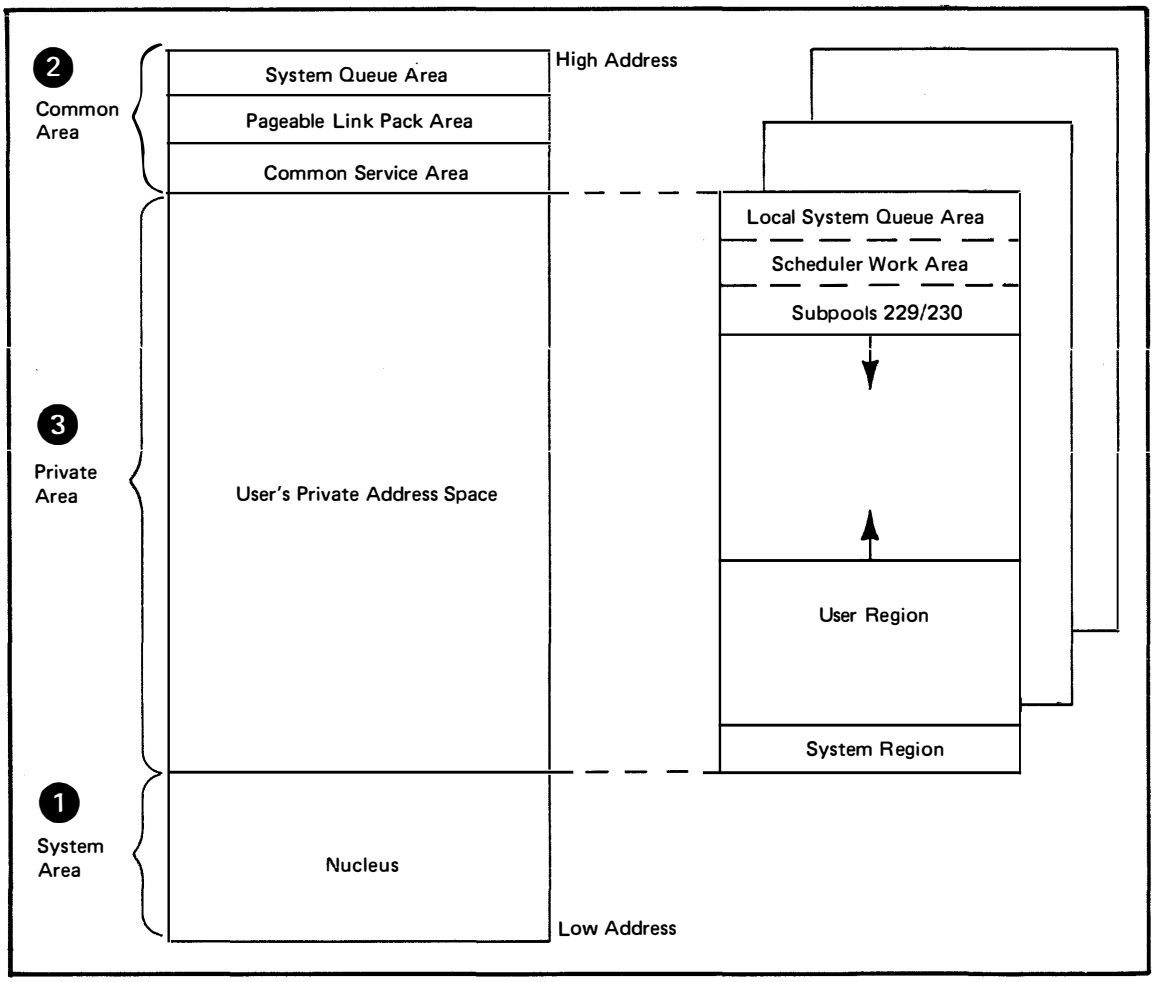



**Figure 2.10. Virtual Storage Layout**

## *System Area*

The system area is allocated from the bottom of virtual storage during system initialization. It contains the nucleus load module and any extensions to the nucleus, the page frame table entries, DEBs (data extent blocks) for the system libraries, recovery management support routines, and unit control blocks. The nucleus and the other contents of the system area make up the resident part of the MVS system control program.

The system area is initialized after initial program load (IPL) by the nucleus initialization program (NIP). The system area is fixed; that is, it is non-pageable and non-swappable. Its contents are mapped one for one into real storage frames at initialization time and remain fixed for the duration of the IPL. While the size of the system area varies depending on the system configuration and the extensions and options an installation chooses, the size of the system area does not change once it is initialized.

## *Common Area*

The common area is allocated from the top of virtual storage. It contains parts of the system control program, control blocks, tables, and data areas. The basic parts of the common area are:

- The system queue area (SQA), which contains tables and queues that are used by the entire system
- The pageable link pack area (PLPA), which contains system programs, such as SVC routines and access methods, and selected reentrant user programs
- The common service area (CSA), which contains system and user data areas

### System Queue Area (SQA)

The system queue area (SQA) contains tables and queues relating to the entire system. For example, the page tables that define the system area and the common area are held in SQA. The contents of SQA depend on an installation's configuration and job requirements.

SQA is allocated from the top of virtual storage in 64K segments; a minimum of three segments are allocated during system initialization. Within the virtual segments, SQA space is allocated as long-term fixed frames when it is required. Because it consists of long-term fixed frames, allocated SQA space is both non-swappable and non-pageable.

### Pageable Link Pack Area (PLPA)

The pageable link pack area (PLPA) contains SVC routines, access methods, other system programs, and selected user programs. As its name implies, PLPA is pageable; however, no physical page-outs are performed. Because any changes made to a module would be lost and because the modules in PLPA are shared by all users, all program modules in PLPA must be reentrant and read-only.

PLPA space is allocated in 4K blocks directly below SQA. The size of PLPA is determined by the number of modules included, and, once the size is set, PLPA does not expand.

### Common Service Area (CSA)

The common service area (CSA) contains pageable system and user data areas. It is addressable by all active virtual storage address spaces and is shared by all swapped-in users. Data associated with an individual address space can be isolated by a storage protect key.

Virtual storage for CSA is allocated in 4K pages directly below PLPA. The amount of storage allocated is determined by the value specified for the CSA parameter during system initialization. CSA is paged in and out of storage as required.

## *Private Area*

As stated earlier, each address space can access the contents of the system area and the common area. In addition, each address space has its own private area. Virtual storage for the private area is allocated from the top of the system area up, and from the bottom of the common area down.

In most installations, the size of the private area ranges from eight to ten megabytes. Even when there are significant extensions to the nucleus, SQA, CSA, and PLPA, more than five megabytes should be available to each user. The private area is made up of the local system queue area (LSQA), the scheduler work area (SWA), subpools 229/230, and a system region, in addition to the user region.

The user region is the space within the private area that is available for running the user's problem programs. There are two types of user regions: virtual (V=V) and real (V=R). The two types are mutually exclusive; that is, a user region can be V=V or V=R, but it cannot be both.

The two types of user regions, as well as the other areas within the private area, are described in the following text.

### Local System Queue Area (LSQA)

The local system queue area (LSQA) contains tables and queues that are unique to a particular address space. For example, LSQA includes the user's private segment table and private page tables. LSQA also contains all the control blocks required by the region control task (RCT). The region control task is the highest level task in each address space; it plays a key role when an address space must be swapped in or out.

LSQA is allocated downward from the top of the private area, intermixed with the scheduler work area (SWA) and subpools 229/230. LSQA for each address space that is swapped in is fixed in real storage frames.

### Scheduler Work Area (SWA)

The scheduler work area (SWA) contains the control blocks that exist from task initiation to task termination. It is, in effect, a local job queue, and the information it contains eliminates contention for a system job queue. The information in SWA is created when a job is interpreted and used during job initiation and execution. "Chapter 5: Entering and Scheduling Work" describes how MVS processes a job.

SWA is allocated from the top of each private area, intermixed with LSQA and subpools 229/230. It is pageable and swappable.

### Subpools 229/230

A subpool is a logical group of storage blocks that share some common characteristics; each type of subpool has a unique identifying number. Subpools 229 and 230 are both protected by the user's storage key. In addition, subpool 229 is fetch-protected, which means that its contents cannot even be read unless the key in storage matches the key in the PSW.

Subpools 229/230 contain user control blocks that can be used only by programs with the appropriate storage protect key. Protected user resources, such as the data extent block (DEB) that describes a user data set, reside in these subpools.

Space for subpools 229/230 is allocated from the top of each private area, intermixed with LSQA and SWA.

### System Region

The system region within the private area is used by system functions performing work for an address space. These system functions run under the region control task (RCT) and obtain the storage they need from the system region by issuing GETMAIN macro instructions.

The system region consists of four virtual pages (16K) allocated from the bottom of the private area. It is pageable and exists for the life of the address space.

### Virtual (V = V) User Region

A virtual (V=V) user region can be any size up to the size of the private area minus the size of LSQA, SWA, subpools 229/230, and the system region. Its size can be limited by the REGION parameter on the user's JOB or EXEC statement.

V=V user regions are pageable and swappable. Only enough real storage frames are allocated at any particular time to hold the active (paged-in) parts of the problem program. A V=V region, as shown earlier in Figure 2.10, begins at the top of the system region and is allocated upward to the bottom of LSQA, SWA, and subpools 229/230.

## Real (V = R) User Region

A real (V=R) user region is assigned a virtual space within the private area that maps one for one with real storage; that is, each virtual address in the region always corresponds to the same real address. Figure 2.11 illustrates V=R storage mapping; the shaded areas in Figure 2.11 indicate unallocated storage. Real storage for the entire region is allocated and fixed when the real region is created. An installation must use the ADDRSPC=REAL parameter at system generation time to reserve sufficient storage for all V=R regions that might exist at any particular time. When no V=R jobs are running, the system uses the storage reserved for V=R jobs for normal paging activity. Particularly when system activity is high, a V=R job might not be started immediately; it must wait until the system can free the storage required for the real region.

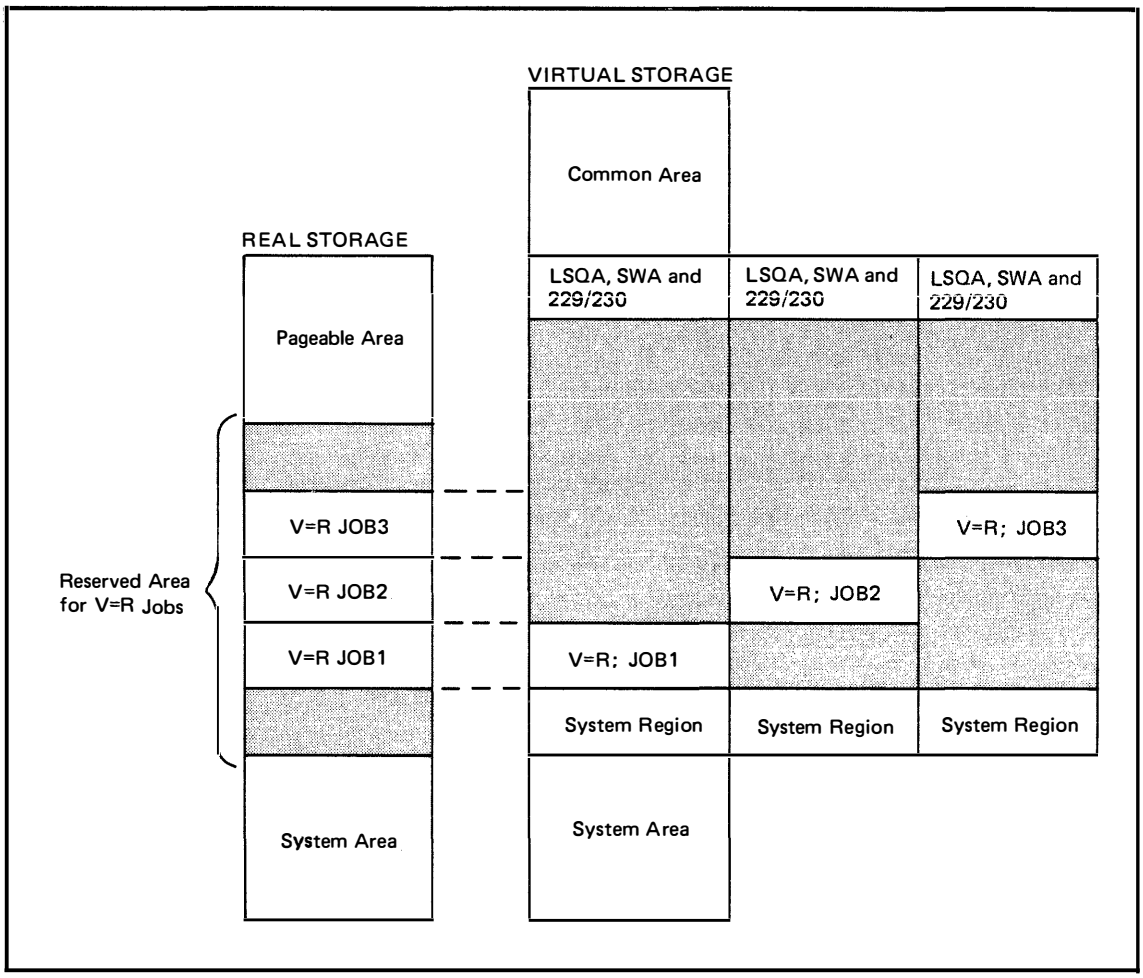



Figure 2.11. V=R Storage Mapping

Real regions should be used only for jobs with time-dependent functions (that is, jobs that cannot wait for paging activity to take place) or for jobs that cannot run in the virtual environment, such as jobs with channel programs that use the program control interruption (PCI) to dynamically modify themselves. See "Chapter 8: Satisfying I/O Requests" later in this book for more information about channel programs.

V=R region size is controlled by the VRREGN parameter specified at IPL or by the REGION parameter in a user JOB or EXEC statement.

### *Extensions and Options*

Both the system area and the common area can be extended, depending on the configuration of the system or options an installation selects. Figure 2.12 shows all possible extensions, in addition to the storage areas described earlier (which are shaded in the figure).

Two of the extensions, the RMS (recovery management support) nucleus extension and the prefixed storage area (PSA), depend on your system configuration.

The RMS nucleus extension contains the recovery management support routines that increase the availability of the MVS system. The size of this extension depends on the particular configuration at an installation, but it is always present in the system area.

The prefixed storage area (PSA) is only present for a multiprocessor system. Its use is described more fully in "Chapter 10: Multiprocessing" later in this book. When present, the PSA occupies 4K of virtual storage and is allocated in the common area just above the CSA.

Other extensions are optional; you choose them at either system generation time or IPL time. These extensions are:

- The fixed link pack area (FLPA)
- The modified link pack area (MLPA)
- The BLDL list, which can be either fixed or pageable

Each of these optional areas is described in the following text.

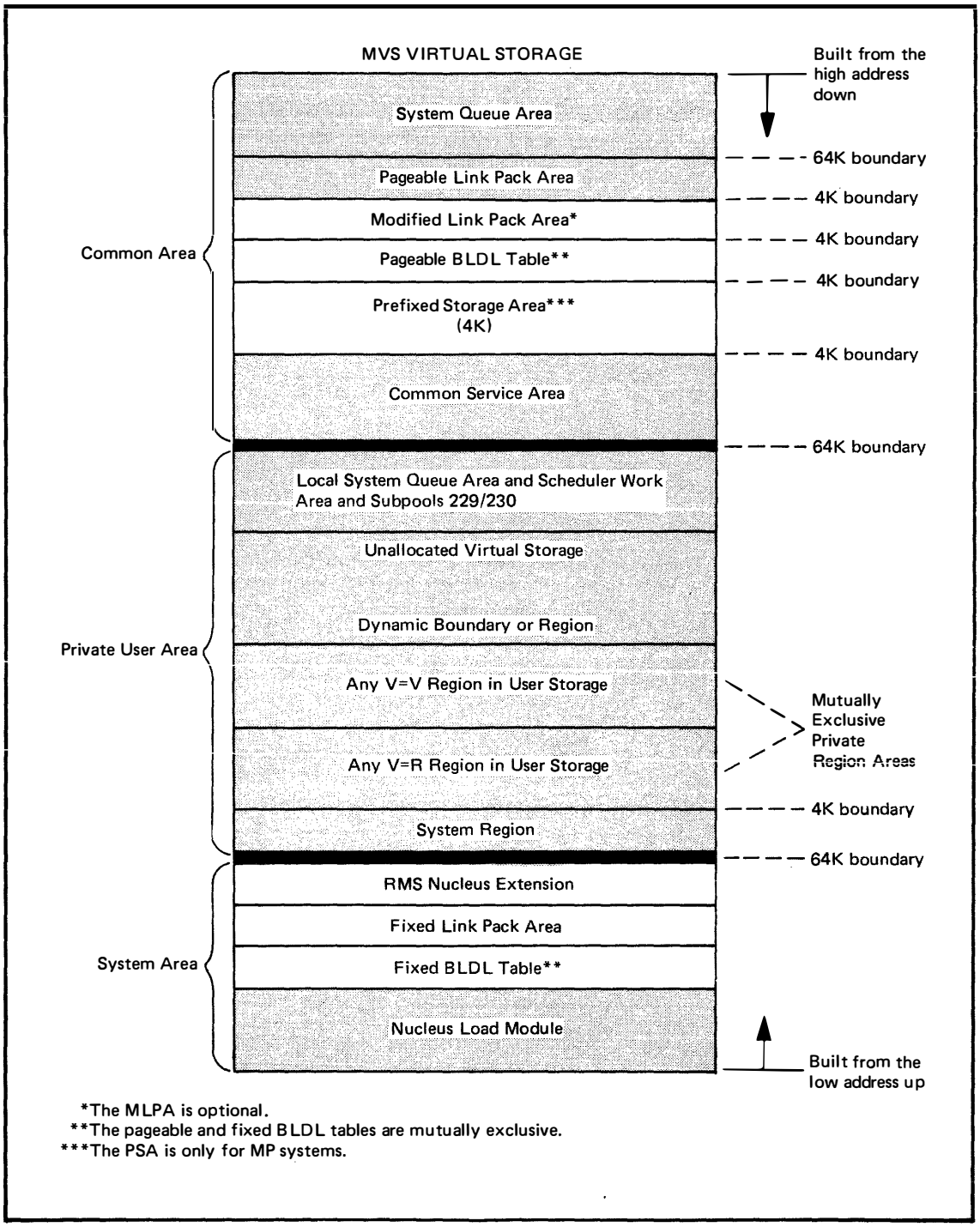



**Figure 2.12. Extensions and Options**

## Fixed Link Pack Area (FLPA)

The fixed link pack area is an extension to the system area that an installation defines at system generation time. It contains reentrant, read-only modules similar to those loaded in PLPA.

Because FLPA is fixed — mapped one for one against real storage — it reduces the amount of storage available for running installation programs. Thus, the modules selected for FLPA should be chosen with care. The paging algorithm MVS uses tends to keep a heavily-used PLPA module in real storage. Therefore, the most likely candidates for FLPA are modules that significantly improve system performance when they are fixed rather than paged, such as a module that is infrequently used but that requires rapid response when it is needed.

## Modified Link Pack Area (MLPA)

The modified link pack area (MLPA) can be used for reentrant modules from selected system or user libraries; it acts as an extension to PLPA, but it exists only for the duration of the current IPL. That is, the MLPA is not saved from IPL to IPL as the PLPA is.

MLPA modules do not have to be read-only, and they can be modified. One effective use of MLPA is to modify and test modules before adding them to PLPA.

When MLPA is specified during system initialization, it is allocated just below PLPA in the common area. It exists for the life of the IPL, and it is pageable.

## BLDL Lists

A BLDL list is a list of directory entries for modules residing on a system library. Specifying a BLDL list can improve system performance because the system does not have to perform a library search to locate a required module. Each entry in a BLDL list contains the information the system requires to locate the module. The type of module that can be most effectively included in a BLDL list would be a heavily-used module that cannot be loaded in FLPA or PLPA because either it is too large or it is not reentrant.

A BLDL list can be either fixed or pageable, but not both. An installation can choose either a fixed or a pageable BLDL list during system initialization.

**Fixed BLDL:** If you choose a fixed BLDL list, the BLDL is allocated in the system area directly above the nucleus. As part of the system area, it is not pageable. Fixed BLDL removes a relatively small amount of real storage from use by installation programs. However, fixed BLDL can reduce the number of page faults that occur during system execution and should be considered when fast processing by the modules in the list is critical.

**Pageable BLDL:** If you choose a pageable list, the BLDL is allocated in the common area below PLPA, or below MLPA, if present.

# Chapter 3: Installing and Servicing the System

This chapter contains information on installing and servicing an MVS system. Among the items discussed are: installation planning; system generation; an alternative to system generation called the MVS System Installation Productivity Option (MVS System IPO); and the System Modification Program (SMP) used to service the system.

## Installing the System

The installation of OS/VS2 MVS involves the creation of an MVS system tailored to the needs of a specific installation and to a particular set of user requirements. The installation can choose to perform a full system generation, use the IBM-provided installation productivity option (MVS System IPO), or use combinations of these to assist in the tailoring process.

### *Preliminary Considerations*

For many locations, installing MVS includes converting existing OS/MVT functions, SVS functions, or OS/VS1 functions to comparable MVS functions and adding certain new OS/VS2 MVS features and enhancements. Such an effort requires a good deal of preliminary thought prior to system generation in the areas of migration planning, conversion planning, and installation planning. Those installations who are migrating or converting from MVT or SVS should refer to *OS/VS2 Conversion Notebook*, for information on migration and conversion planning. Those installations who are migrating or converting from VS1 should also refer to *OS/VS1 to OS/VS2 Conversion Notebook* for information on migration and conversion planning. This section focuses on installation planning, system generation, and the MVS System IPO.

### *The Installation Plan*

Installation planning is a key step to successfully installing OS/VS2 MVS. A well thought out, managed, documented, and executed plan takes into consideration everyone who uses or supports the system. The installation should prepare a planning document that includes:

- A guide that indicates the appropriate tasks to be performed and identifies who should perform these tasks
- Appropriate checkpoints, interdependencies, and deadlines
- User goals and performance expectations
- Staffing and assignment of personnel

**Installation Tasks:** Installation tasks can be categorized in five phases, as shown in Figure 3.1: overall installation planning, generating the system, integrating and testing the various components, testing the production system, and stabilizing the production system. These phases are basically the same as those provided in the MVS System IPO installation plan discussed later in this chapter. Refer to that discussion for details on how each of the planning phases should be handled if the MVS System IPO is going to be used.

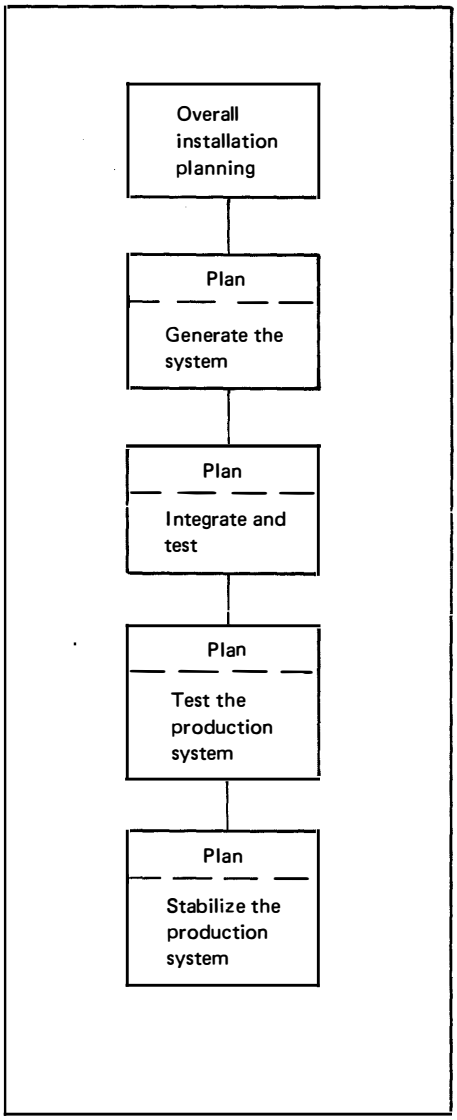



**Figure 3.1. Installation Planning Phases**

**Checkpoints and Interdependencies:** Checkpoints should be established for each of the tasks within a given phase. Interdependencies of tasks, identification of tasks that can be run in parallel, and other related planning information can be established and documented during the overall installation planning phase.

**Performance:** In order to migrate or convert to an MVS system from an existing system, the installation must understand the performance of the current system and the desired performance of the new system. Performance expectations should be documented in the installation plan and should include such items as:

- Turnaround time for all classes of batch jobs
- Response time for online transactions
- Elapsed time for long-running jobs

In addition, the installation should create a workload profile to document the expected volume of transactions and storage requirements. It may also be possible to estimate processor use, channel use, and system paging rates. Several IBM facilities are available to help the installation perform this task. These include the Generalized Trace Facility (GTF), System Activity Measurement Facility (MF/1), and the Resource Measurement Facility (RMF), an IBM program product. Once performance expectations are understood and system growth is projected, the proper hardware and software configuration can be designed and generated. The *OS/VS2 MVS Performance Notebook*, includes information on defining performance objectives.

**Staffing and Personnel:** Ideally, the installation plan will be carried out by the current system programming staff. As an example, a typical programming staff for installing MVS might include:

- Two people for MVS with JES2/JES3 experience
- One person for TSO with TCAM/VTAM experience
- One person for IMS/CICS (IBM program products)

This staff would be responsible for system generation, problem diagnosis, monitoring and tuning, and other operation support activities. Each participant should be fully educated, either in a classroom or self-study environment, on how to handle each of the installation planning tasks to which he is assigned. This education time should not be compromised.

## *System Generation*

System generation is the process of selecting modules, options, and parameters from IBM distribution libraries (DLIBS) and using them to tailor the installation's MVS system. As shown in Figure 3.2, the system generation procedure uses an MVS starter system (or a previously-working MVS system), a set of IBM distribution libraries, and a set of installation-specified JCL and macro instructions (user specifications) to produce the new MVS system.

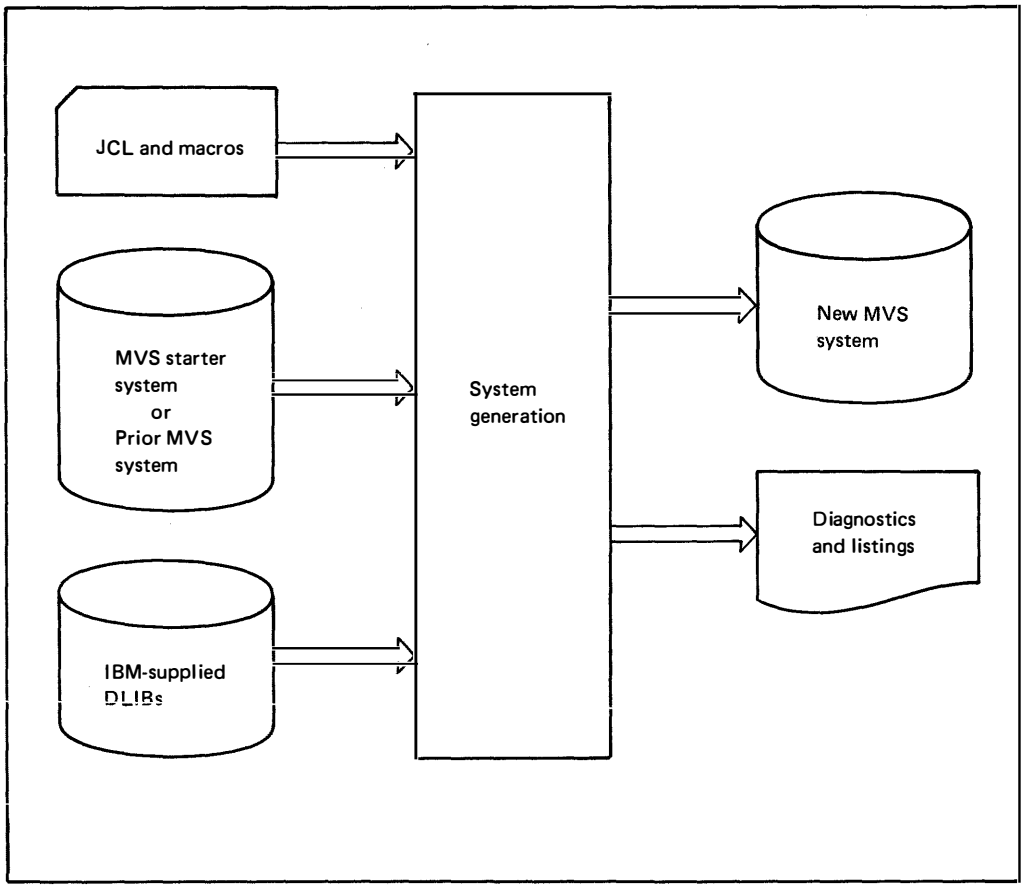



**Figure 3.2. Creating an MVS System with the System Generation Procedure**

When the MVS system is already generated but the installation wishes to change the machine configuration or certain other program configurations, an I/O device generation can be performed. Refer to the publication *OS/VS2 SPL: System Generation Reference*, for a detailed description of I/O device generation.

***Note:*** Distribution libraries can be modified prior to system generation to include specific IBM-supplied selectable units (a new way of packaging function). This enables the installation to reap the benefits of an improved MVS packaging and distribution process provided under the selectable unit (SU) concept. More is said on this new process under "Servicing the System."

## Planning and Preparing for the System Generation

To prepare for the system generation process, the installation must:

1. Order the MVS distribution libraries from IBM. Information on how to do this is in the latest edition of the *OS/VS2 Release Guide*.
2. Select the appropriate MVS system control program options from those available with MVS. Selected options, with the standard

features, comprise the installation's system. An explanation of all MVS-supported options is available through the local IBM branch office representative.

Note that in MVS the number of system generation options that must be specified has been reduced. Many of the previous options have been made standard under MVS. In addition, several macros (used to specify the selected options) have been eliminated, consolidated, or clarified.

3. Select and code the system generation macro instructions that specify the selected options, standard features, and the allocation or pre-allocation of data sets on the system. Instructions on defining system data sets and a list of system generation macros and their uses can be found in *OS/VS2 MVS SPL: System Generation Reference.*

   If program products, such as IMS or CICS, are included in the system, consult the local IBM Branch Office representative for the appropriate documentation.

4. Initialize the DASD volumes required for the system generation. Before the system can be generated, the DASD volumes that contain the MVS distribution libraries, the MVS starter system (or prior MVS system), and the MVS system-to-be must be initialized.

## Executing the System Generation

With MVS system generation, multiple jobs can be run in parallel to speed up the process. In addition, because many of the previous system options have been standardized, installation time is saved in coding applicable macro instructions for these options.

System generation is executed in two stages, as shown in Figure 3.3. In Stage I, the system generation macros are assembled and then expanded into job control statements, utility control statements, assembler statements, and linkage editor control statements. Together, these statements describe:

- The hardware configuration
- The system control program
- The access methods
- Installation routines that are to become part of the system
- Installation-selected program options that are to be included in the new system

In other words, the statements describe the new, tailored MVS system. (Additional tailoring can be done during subsequent initializations of the generated MVS system.)

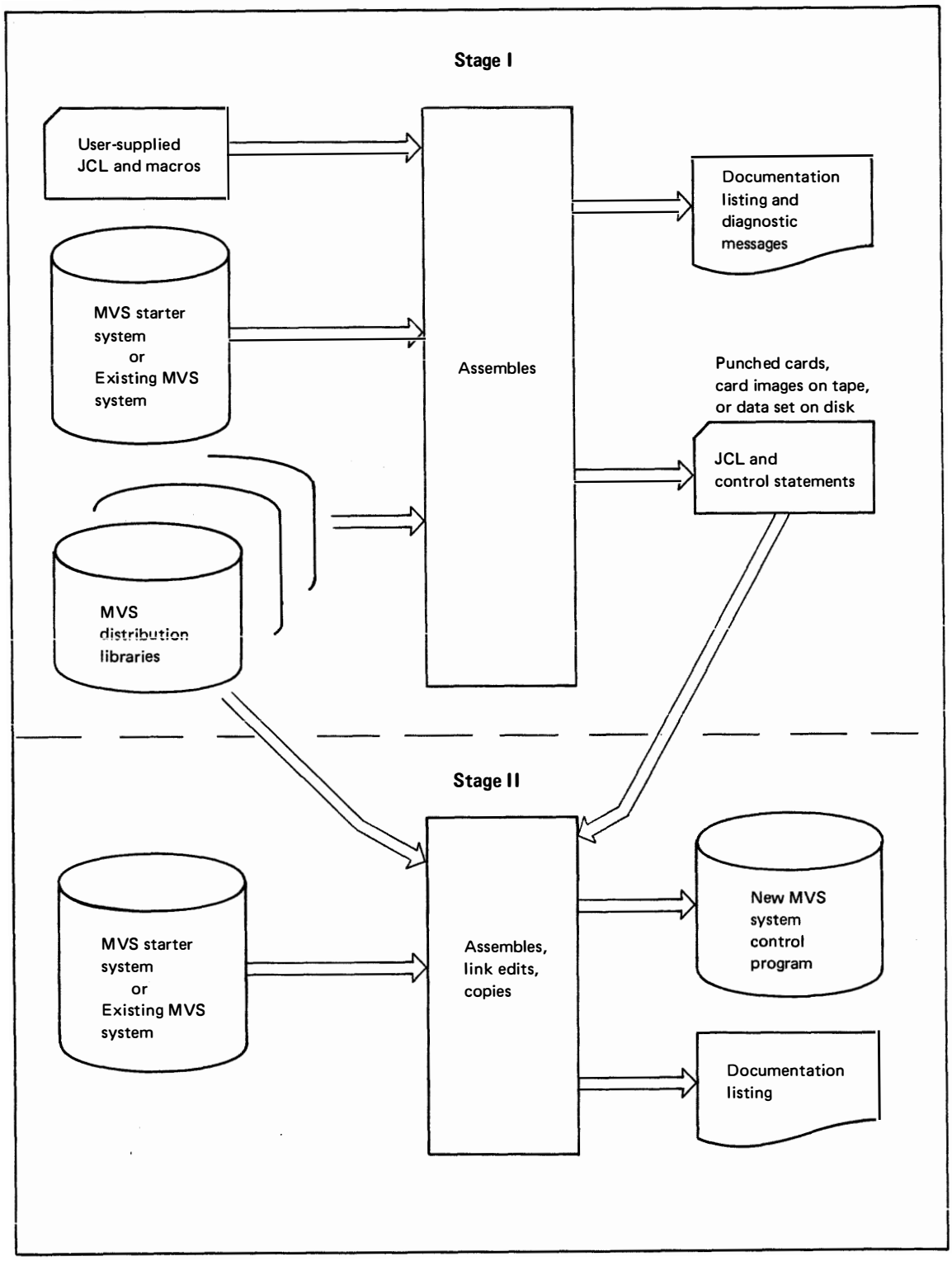



Figure 3.3. Executing the System Generation

The output of Stage I is input to Stage II. During Stage II, modules from the distribution libraries are assembled, link edited, and copied to the data sets that are allocated on the new system volumes.

For a full system generation, Stage II consists of six or seven jobs, depending on what the installation has pre-defined. For an I/O device generation, Stage II consists of only five jobs. In all cases, the sequence of execution is the same and is designed so that multiple jobs are executed in parallel; that is, it is a multiprogrammed job stream.

The output of Stage II is the installation's MVS system control program and a listing that documents Stage II execution.

### Verifying the System Generation

After the system generation process completes, an IBM-supplied installation verification procedure (IVP) should be performed to verify that the new system is operating properly on the specified hardware configuration. Optionally, the installation can perform an I/O device generation to alter or extend the I/O configuration of the MVS system. The Installation Productivity Option (MVS System IPO), to be discussed next, contains information on system integration and testing of the production system.

## *MVS System Installation Productivity Option (MVS System IPO)*

The MVS System IPO, an alternative to the full system generation process, is a new approach to packaging, distributing, installing, and servicing a system. It is a result of an MVS installation completed at an IBM internal location. As such, the MVS System IPO package provides the installation with the benefit of extensive installation experience. It should help to achieve full production status with fewer resources as well as to significantly reduce the time and effort required to plan, prepare, and execute the installation of the MVS system.

This section discusses the MVS System IPO, the MVS System IPO installation plan, and the documentation provided in support of the MVS System IPO.

### The MVS System IPO

MVS System IPO comes to the installation as a pre-generated extension of the MVS starter system, supporting batch and TSO operation. The standard version includes JES2, an expanded I/O configuration, TCAM or VTAM support for TSO or IMS (a separately orderable feature of the MVS System IPO is available for IMS/VS), and the most common MVS system options.

The system is a moderately tuned, two-volume MVS system that can be used as is or altered to meet the installation's requirements. It comes with a set of installed selectable units and programming temporary fixes (PTFs). (Though the MVS System IPO is not formally tested when the SUs and PTFs are applied, IBM uses the latest distribution level as a production system at the IBM installation producing the MVS System IPO package.)

To simplify the installation process, the MVS System IPO package includes examples of JCL usage and procedures to show how the installation can use certain functions, change them, or incorporate them into the MVS system. TSO userids, LOGON procedures, and a sample command processor are provided, as is information about operating a time-sharing system, including initializing, monitoring, and terminating TSO. In addition, examples of exit routines are provided.

The MVS System IPO can be used to educate the installation's system programmers, system operators, and users. With it, the installation can:

- Perform early testing without extensive tailoring or reconfiguration
- Minimize the number of installation decisions to be researched, implemented, and tested
- Reduce the stand-alone machine time required

Note, however, that the IBM internal location where the MVS System IPO package was constructed was limited by the specific hardware/software configuration at that location. Therefore, the installation should do an I/O device generation to match the configuration of the installation's system, as shown in Figure 3.4. Later, the system can be tailored and extended to meet installation and user requirements.

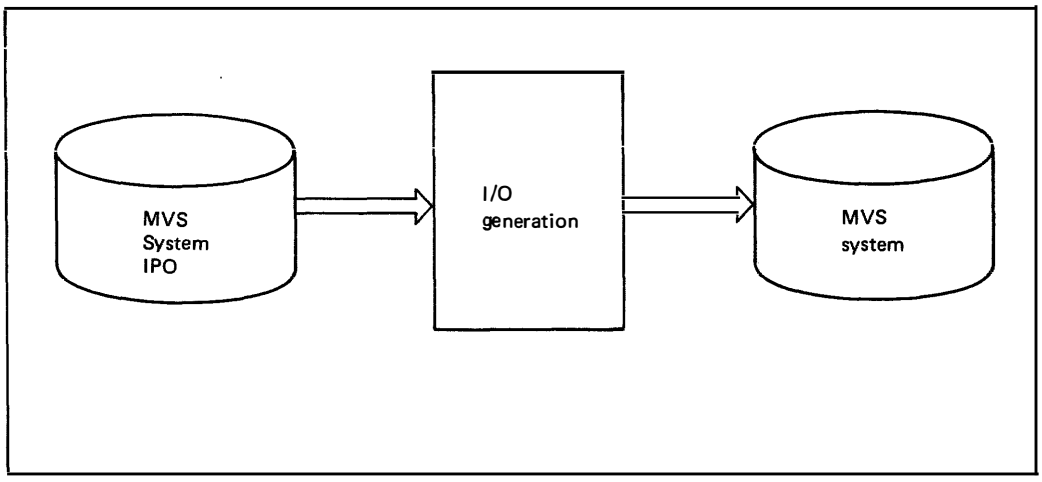



**Figure 3.4. I/O Device Generation**

The MVS System IPO package also contains supporting documentation and an installation plan. Discussions of each of these follow.

## MVS System IPO Documentation

The MVS System IPO package includes a comprehensive set of documents to assist the installation in using the MVS System IPO package. These documents, shown in Figure 3.5, explain how to use the MVS System IPO, describe how to build a production test system, and provide hints and techniques relating to the installation process.

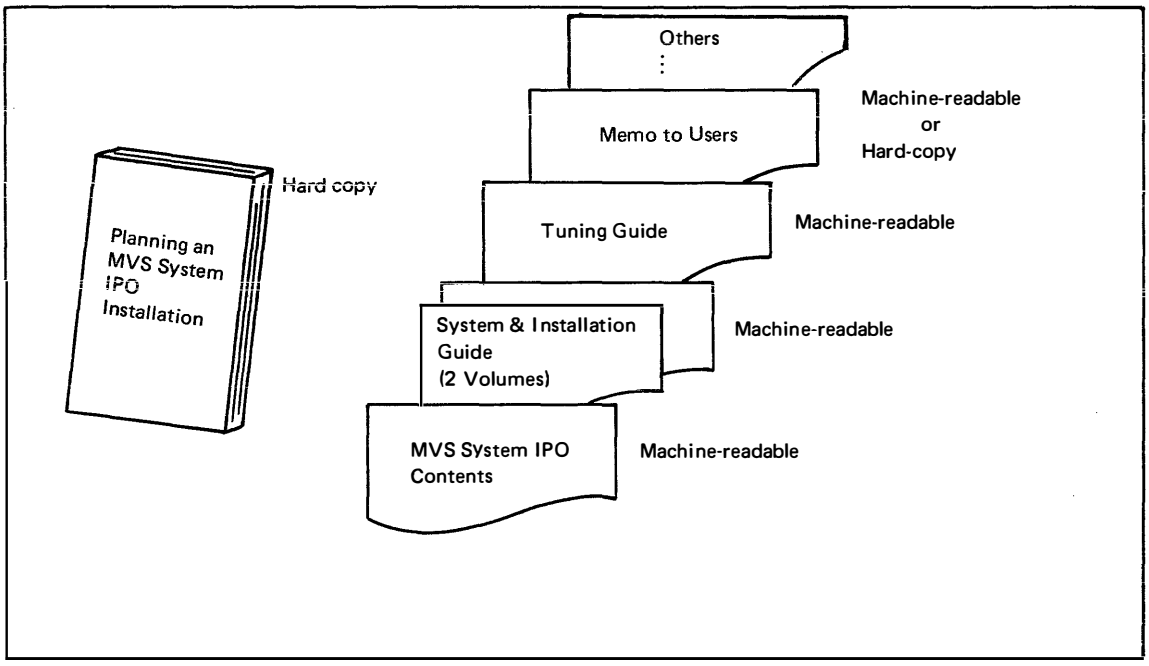



**Figure 3.5. MVS System IPO Documentation**

All MVS System IPO documents except the planning document are distributed in machine-readable form. Because of this, they reflect the latest experience and the most current MVS System IPO information. The machine-readable documents can be listed on a system printer or displayed on a TSO terminal. Their contents follow:

- *Memo to Users:* This document contains a general description of the MVS System IPO package. It includes the purpose and concept of the MVS System IPO, a description of the physical characteristics of the tapes on which it is distributed, and a brief summary of each MVS System IPO document.
- *Planning an MVS System IPO Installation:* This document contains general information about MVS System IPO. It is intended to assist those responsible for installation planning in evaluating the use of the MVS System IPO for their installation. It describes in detail a structured installation plan that makes maximum use of the MVS System IPO package.
- *MVS System IPO System Contents:* This document contains a physical description of the:
  - MVS System IPO distribution libraries and the MVS System IPO itself
  - Installed selectable units and applied programming temporary fixes

- I/O configuration and defined UNITNAMEs

- Contents of the MVS System IPO data sets, physical data set characteristics, and library members

- *System and Installation Guide, Volume I:* This document discusses the procedure for installing the MVS System IPO and the rationale behind the procedure. In addition to discussing the basic system set-up, it describes procedures for:
  - Printing the MVS System IPO documents and listings
  - Coding system generation macro instructions
  - Performing an I/O device generation
  - Verifying the initial system
  - Building a test production system
- *System and Installation Guide, Volume II:* This document discusses the techniques for tailoring the MVS System IPO. These techniques include the use of the System Modification Program (SMP), user SVC routines, user exits, and the program properties table. It also discusses password protection and provides catalog examples, hints about system back-up, and fall-back and recovery techniques.
- *Tuning Guide:* This document discusses IBM experience in measuring and tuning the MVS system along with experience in using certain programs and aids for tuning purposes. It provides a tuning methodology, discusses the tailoring of MVS System IPO, and offers general tuning advice.

There are various other MVS System IPO documents as well. For example:

- *MVS System IPO User's Guide*
- *MVS System IPO Communication and Interactive Guide*
- *MVS System IPO Operator's Guide*
- *Program Product Usage and Experience Guide*
- Various Conversion Guides

These are explained in more detail in the publication *Installation Productivity Option (IPO) for OS/VS2 Release 3.7 (MVS): Planning an MVS System IPO Installation*, GC20-1852-2.

## The MVS System IPO Installation Plan

The MVS System IPO package includes an installation plan that helps the installation's project leaders develop their own plans tailored to the needs of the installation. The MVS System IPO installation plan, which is divided into five phases, does the following:

- It defines the required tasks.
- It identifies those tasks that can be performed in parallel.
- It suggests a schedule for executing the various tasks.

As shown in Figure 3.6, each of the system installation phases following the initial planning effort is preceded by planning activity pertinent to that phase. Keep in mind while reading the discussions of each of these phases that the MVS System IPO installation plan formalizes some of the activities

that the installation should seriously consider doing whether or not the MVS System IPO, itself, is used.

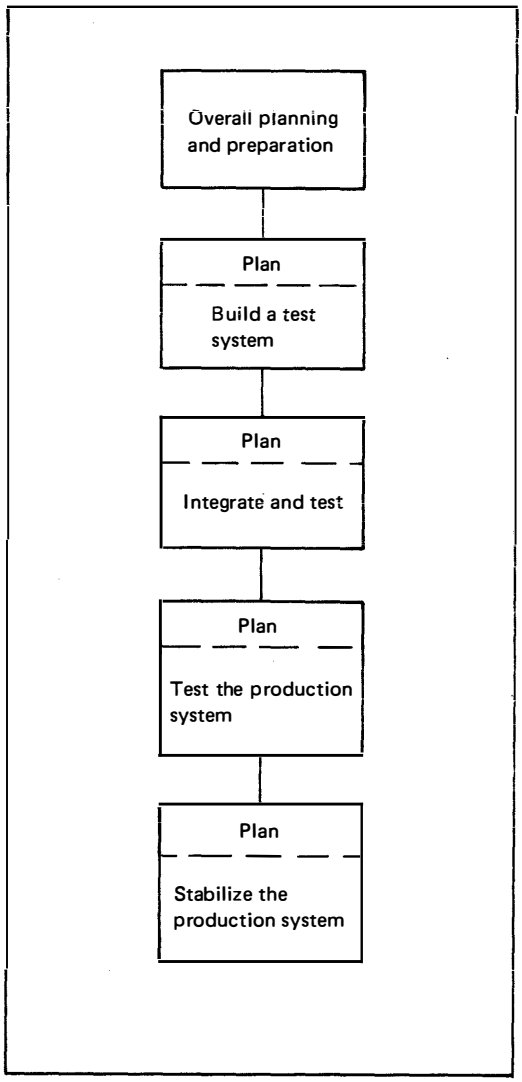



**Figure 3.6. The MVS System IPO Installation Phase Plan**

**Phase 1 -- Plan and Prepare:** During Phase 1, the MVS programming group will obtain the necessary MVS education and study the MVS publications. Then, after printing and reviewing the MVS System IPO documentation, detailed tasks can be incorporated into the installation plan. Note that similar tasks are performed in parallel by TSO and IMS programming groups, as well as operations and users. (This applies to the other phases, as well.)

To use the new operator and user facilities MVS offers, the installation may have to revise its standards and procedures. Those responsible for operations and user applications should evaluate this need.

When the installation has completed all other Phase 1 planning and preparation, the MVS System IPO and the distribution libraries should be moved from IBM tapes to installation DASD volumes in preparation for an I/O device generation.

**Phase 2 -- Build a Test System:** During this phase, an MVS system tailored to the installation's needs and suitable for subsequent production testing is built. Activities in this phase include:

- An I/O device generation
- Creating PARMLIB and PROCLIB members
- Entering user data sets in the catalog
- System verification
- Preparing the TSO component
- Component testing

The MVS System IPO documents and listings include detailed instructions for completing this phase.

**Phase 3 -- Integrating and Testing:** The objective of this phase is to ensure that the individual components, with system enhancements and extensions, work with one another to accomplish the various system functions. At the end of this phase, the system that the installation began building in Phase 2 is available for production testing. All functions and options are completely integrated and the structure of the MVS system is complete. (Note, however, that overall system tuning is not completed until the system stabilization phase is executed.)

To expedite this phase, there is much parallelism and overlapping that can be done in the testing of the various components. For this reason, it is important that the installation synchronize the various activities, and that the various TSO, IMS, operations, and user groups communicate with each other and with the MVS system programming group before and during the testing.

**Phase 4 -- Testing the Production System:** The objective of this phase is to test the entire system with simulated production. The MVS system programming group should control the testing, but all groups are involved. Several tests should be planned and executed early, including terminal simulations, if required. Many installations schedule at least one production test with live, on-line users prior to releasing the system for limited production. In any event, it should prove useful to introduce the MVS system to end users during this phase to familiarize them with new procedures, modified standards, and enhanced facilities. *The MVS System IPO Tuning Guide* provides excellent guidance for this phase.

Before proceeding into limited production (assuming that production testing has gone satisfactorily), fall-back procedures should also be tested. The *MVS System IPO Operator's Guide* includes recommended steps and procedures.

**Phase 5 -- Stabilizing the Production System:** The objective of this phase is to bring the MVS system to a point where it can move into full production status. Phase 5 is a continuous activity that includes releasing the system for limited production and for eventual full production. During limited production, the tuning process is continued to ensure that the system is adjusted to meet installation performance expectations. Full production is

achieved when performance expectations and all planned user requirements have been met. In addition to the *MVS System IPO Tuning Guide*, the installation will find the following publications useful in reaching full production status: *OS/VS2 MVS Performance Notebook*, and *OS/VS2 System Programming Library: Initialization and Tuning Guide*.

## Servicing the System

After full production status has been attained, the installation will want to control the application of service, including the installation of new selectable units (SUs), program temporary fixes (PTFs), and user modifications. System service may also involve ordering a more current release of the MVS System IPO and repeating some of the key installation tasks.

The System Modification Program (SMP) is the primary IBM-provided tool for servicing the MVS system.

### *The System Modification Program (SMP)*

The SMP controls the application of service at the installation. To do this, SMP creates a record of all modules and macro instructions in the target system (that is, the system to be serviced). As service for the system is received (in the form of new SUs, PTFs, or user modifications), SMP checks these records to see what modifications have been made. In this manner, a high degree of control of what is to be included in the system can be maintained.

SMP can also be used to modify and keep a record of modifications to permanent user libraries and the IBM distribution libraries. This section discusses the kinds of modifications that can be made, namely:

- Installing new selectable units
- Installing programming temporary fixes
- Installing user modifications

In addition, some information is included about the SMP functions used to carry out these modifications.

#### Installing Selectable Units (SUs)

Selectable units (SUs) represent a recent change to the MVS packaging and distribution process. By choosing appropriate selectable units, the installation can add enhanced or new functions to their MVS system whenever these functions are needed by the installation. This means installation on a more timely basis with fewer untimely disruptions to operations.

SUs are installed using a new MVS macro called the INSTALL macro. The parameters in this macro identify the SUs to be installed and indicate where the SUs are to be installed. SUs can be installed in the distribution library for a subsequent MVS system generation (called the SYSGEN option) or they can be installed from a distribution library into the target system itself (called the SMP option). The SMP program controls both methods.

**SYSGEN Option:** When the SYSGEN option is selected, the INSTALL macro creates a new set of distribution libraries from the IBM distribution library and the SU tape. Various SMP functions are performed during the installation process, as discussed under "SMP Control Functions." The resulting modified distribution libraries (see Figure 3.7) can be used to generate a new MVS system that will include the selected SUs.

Note that when the SYSGEN option is selected, the target system, itself is not affected.

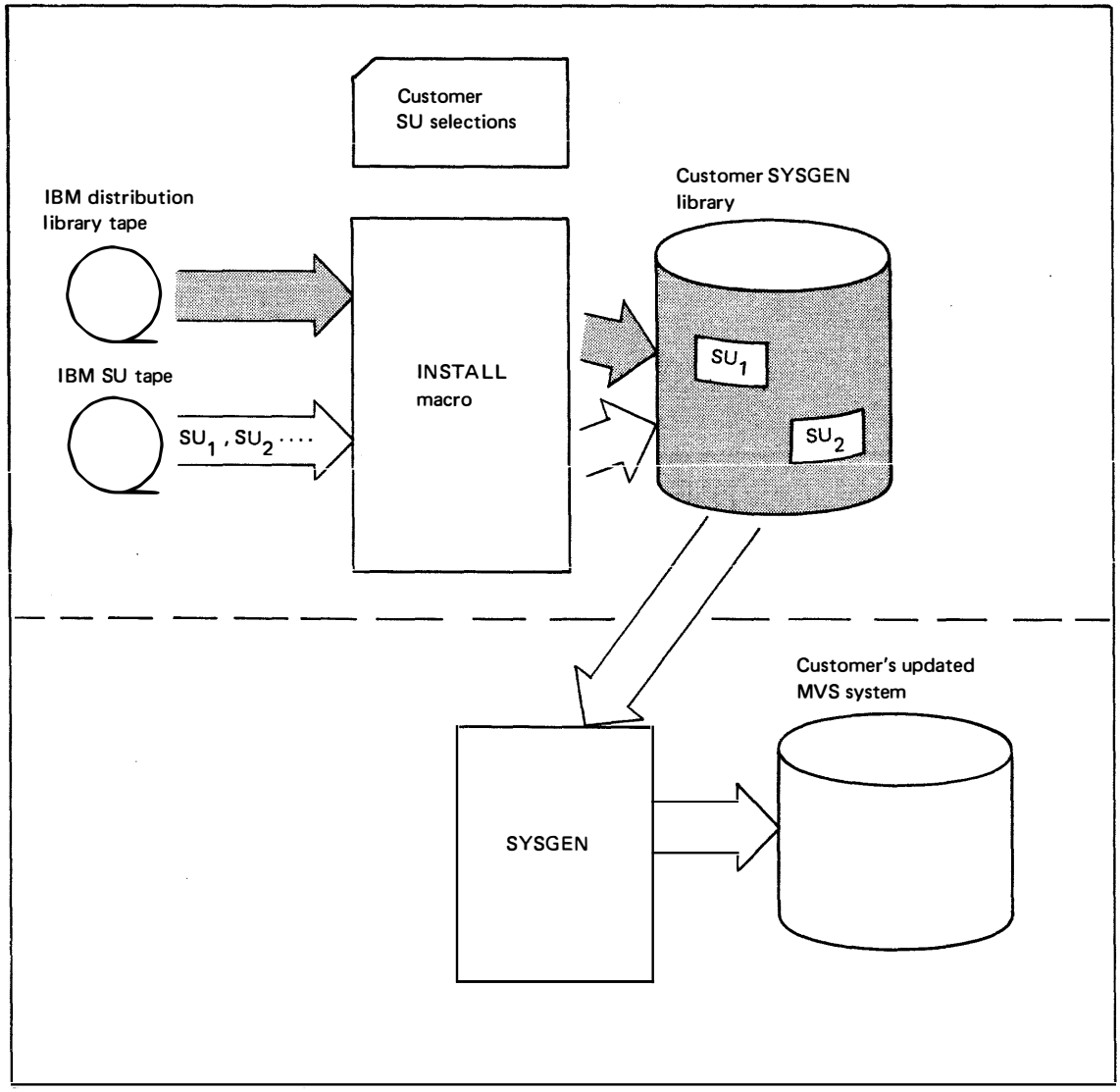



Figure 3.7. SYSGEN Install Option

## SMP Option

When the SMP option is selected, the INSTALL macro receives applicable SUs, applies them to the existing MVS system, and accepts them as modifications to the permanent user libraries or to the distribution libraries. This is carried out according to the SMP function control statements encountered by SMP. When the SMP option is selected, the target system is directly modified, as shown in Figure 3.8 -- no new system generation is required.

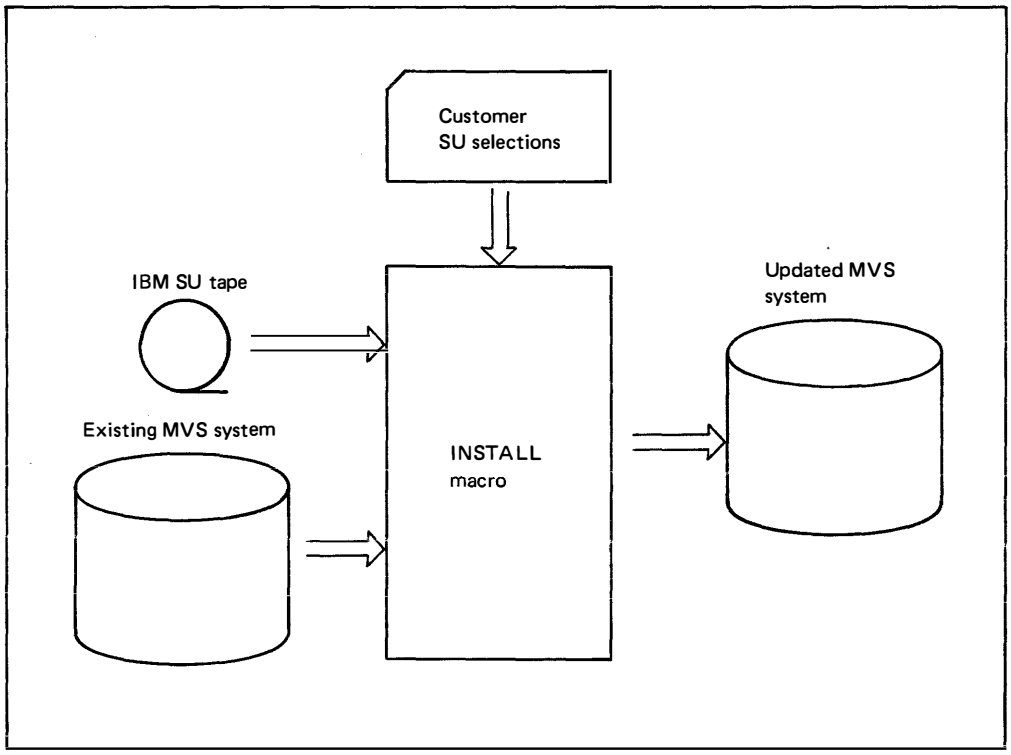



**Figure 3.8. SMP INSTALL Options**

## Installing Programming Temporary Fixes (PTFs)

A programming temporary fix (PTF) is an IBM-supplied correction to a defect in one of its programs. It is intended to fix or prevent problems. Unless the defect is removed in a later release, the PTF becomes a permanent part of the system. IBM distributes these corrections on a PTF tape. IBM also distributes program update tapes (PUT) to reduce the effort required to perform service. The tapes contain selected PTFs organized and arranged to facilitate easy application.

Each PTF contains a series of SMP function control statements and one or more changes. The control statements:

- Identify the change
- Verify that the change applies to the installation's system
- Specify prerequisite additions to or deletions from the system for this particular PTF. (In some cases, a PTF cannot be applied unless one or more prior PTFs are first added, or unless a PTF added earlier is first removed.)
- Indicate whether the change is to macro instructions, source modules, object modules, or load modules
- Indicate whether the change is an update or a replacement

## Installing User Modifications

Once your system is installed, you may want to develop and code your own changes. These changes may be new or replacement macros or source, load, or object modules. Changes can be assembled and link edited, if that is required, or SUPERZAP statements can be used. Each change should have an identifying number.

SMP can be used to apply user modifications. It provides the same control capabilities and benefits for user modifications as it does for applying IBM PTFs. To install user modifications with SMP, you write SMP function control statements to specify the changes you want to make and to verify the correct base level of the system. The SMP statements should also be used to check prerequisite changes or changes in the system that might preclude the present change.

## SMP Control Functions

SMP can process several changes at once and can accept input in the form of SUPERZAP statements, module replacements, and in PTF form. It controls application of changes through the use of SMP function control statements. Figure 3.9 illustrates the function provided by the SMP control statements. Additional details can be found in the publication *OS/VS System Modification Program (SMP) System Programmer's Guide.*

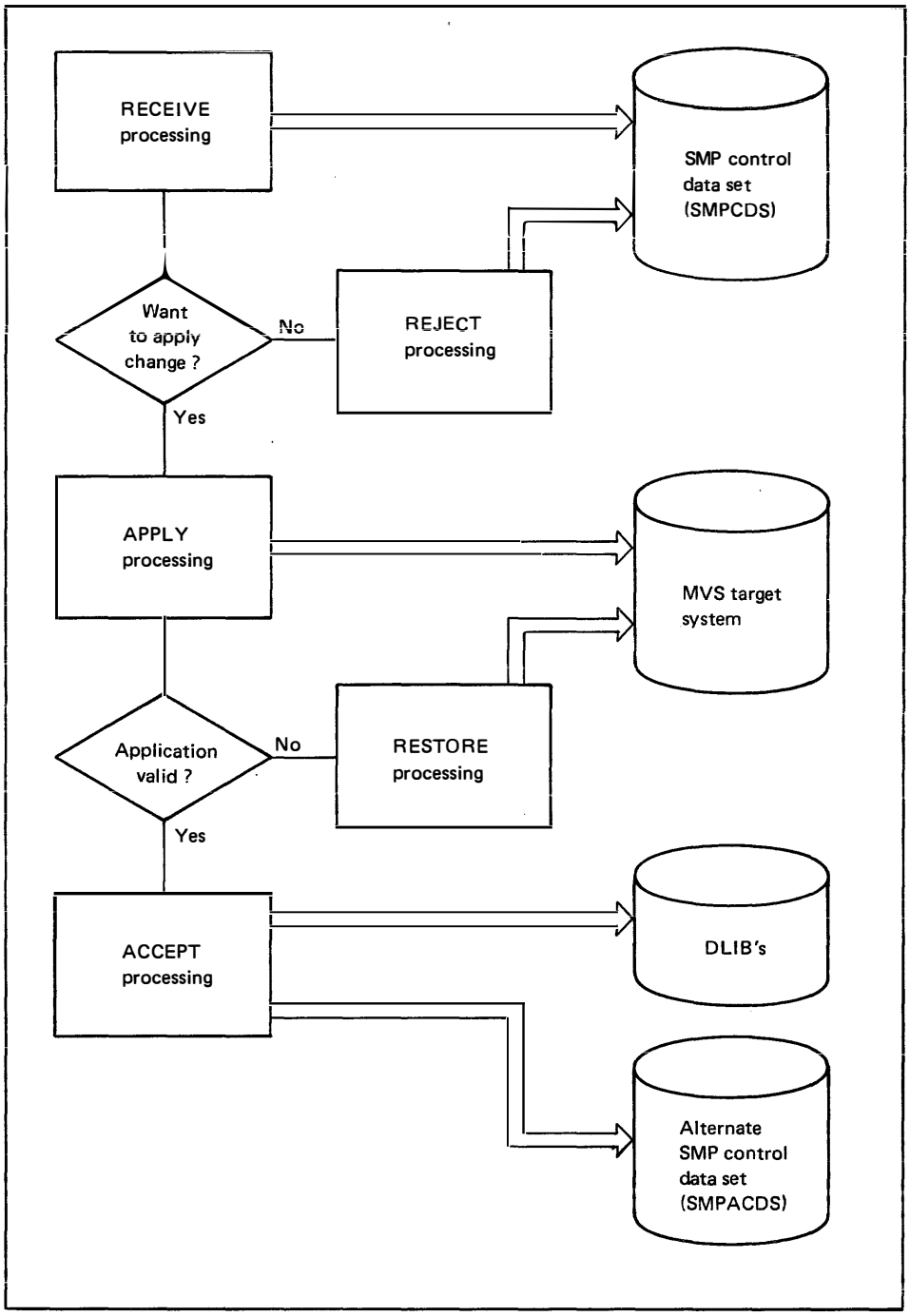



**Figure 3.9. SMP Functions**

**RECEIVE Function:** The RECEIVE function creates essential control information used to determine whether or not to add the current modification to the system. This information is placed in an SMP control data set called SMPCDS. The RECEIVE function also checks the syntax of control statements and verifies that the current modification applies to your particular system. Additionally, it prints a listing to help you determine which changes should be applied to the system or rejected.

**REJECT Function:** If you decide not to apply a particular change after RECEIVE processing, the REJECT function deletes the appropriate control information from the SMPCDS data set.

**APPLY Function:** The APPLY function first determines that all necessary changes are either on the system or being applied. It also identifies any previous changes that might precede this change. When you are satisfied that you can proceed with the change, the APPLY function makes the modification.

**RESTORE Function:** If you find during a testing period that a change does not work or that you must remove one or more changes for any reason, the RESTORE function will remove the changes from the system and update the SMPCDS data set.

**ACCEPT Function:** The ACCEPT function places into permanent libraries or into the distribution libraries any changes that the RECEIVE and APPLY functions have processed. An SMP alternate control data set (SMPACDS) is updated to reflect any changes to the distribution libraries.

# Chapter 4: Preparing the System For Work

Before productive work can be done, the MVS system must be initialized to specific starting values. These values, some of which were previously established during the system generation process and some of which may be provided by the system operator during the initialization process, provide installation tailoring to the MVS system.

## Overview of the Initialization Process

As shown in Figure 4.1, the initialization process consists largely of locating, loading, and initializing the nucleus, initializing system resources, initializing the master scheduler, and initiating the primary job entry subsystem (JES). In the course of the initialization process, an initial program loader (IPL), a nucleus initialization procedure (NIP), various resource initialization modules (RIMs), and a master scheduler initializer are loaded and activated to perform the appropriate initialization steps. To provide additional flexibility to the initialization process, the system operator can interact with the various initialization routines through a system console.

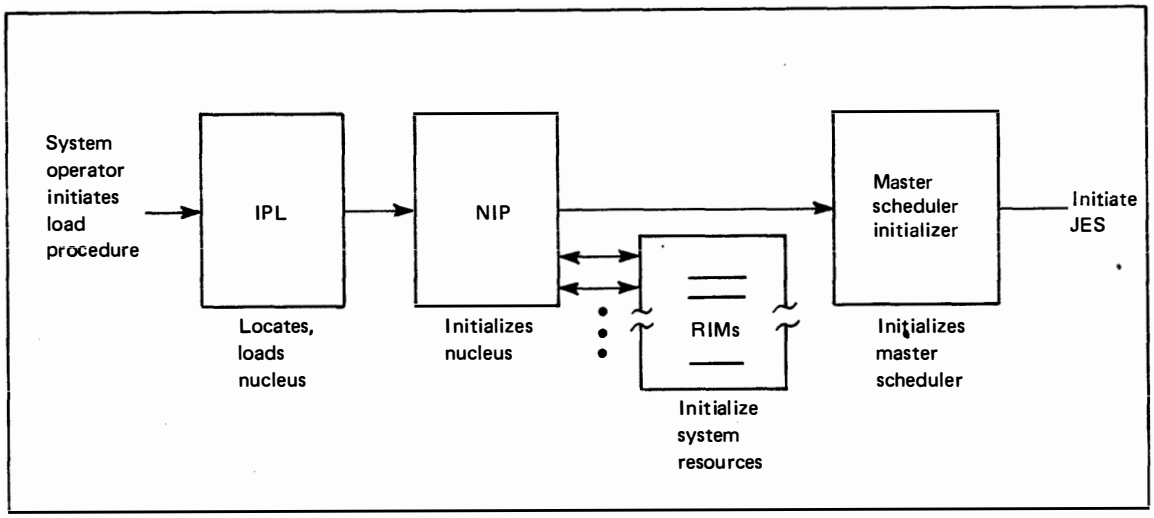



Figure 4.1. System Initialization Summary

## *Initiating the Load Procedure*

The load procedure is initiated by the system operator. He ensures that the system residence volume (SYSRES) is mounted and that the load device is readied. Then, using the system console, he selects the load device and initiates the load procedure.

### The System Residence Volume

The system residence volume (SYSRES) must be online and ready during system initialization because it contains the initial program loader and some of the system data sets necessary during the initialization process. For example, three such data sets that must be on the SYSRES volume are:

```
SYS1.NUCLEUS
SYS1.LOGREC
SYS1.SVCLIB
```

SYS1.NUCLEUS contains the resident nucleus to be loaded and initialized. It also contains the nucleus initialization procedure modules (NIP), the resource initialization modules (RIMs), and the modules used to initialize the master scheduler.

SYS1.LOGREC contains a record of hardware, software, and input/output errors that occur during system operation. The data set is opened during initialization so that error recording can take place.

SYS1.SVCLIB is an authorized program library that contains certain supervisor routines that are not part of the resident nucleus but that are invoked by NIP.

### The System Console

The operator uses the system console to operate and control the system. The system console consists of a control panel and a console device. On some System/370 models, the operator uses the control panel to select the load device and initiate the load procedure. On other models, he or she uses the console device, which includes a keyboard, a light pen, and a display screen. In the case in which the console device is used, the operator must first perform an initial micro program load (IMPL) after powering up the processor. The initial micro program controls the display screen, thereby permitting function selections to be made available as "menu" items. In any case, the operator's initial actions bring the initial program loader into storage.

## Initial Program Loading

When the operator initiates the load process, the stand-alone initial program loader (IPL) is loaded from SYSRES into real storage starting at location zero, as shown in Figure 4.2. Then IPL receives program control.

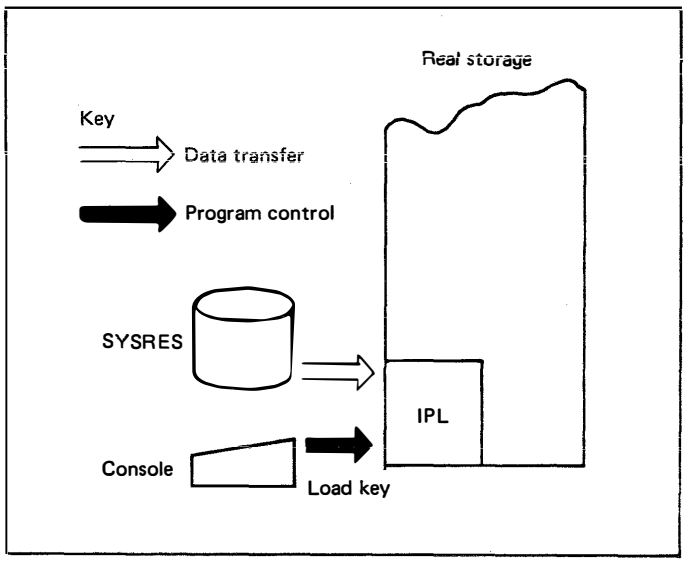



**Figure 4.2. Initial Program Loading**

The initial program loader has two major functions: clearing storage and loading the nucleus.

### *Clearing Storage*

IPL clears the general registers and floating point registers. Then it limits the effective size of real storage to a size specified by the system operator. (Or, if no size is specified, the system default size contained in the system parameter library is used.) Next, IPL clears effective real storage and resets the storage keys.

### *Loading the Nucleus*

After storage has been cleared, IPL searches the system residence volume for the nucleus, or, if applicable, for an operator-specified alternative nucleus. When it finds the nucleus, IPL relocates itself and then loads the nucleus load module (IEANUC0x) and the NIP module (IEAVNIP0) starting at location zero. IPL then passes control to NIP. This is illustrated in Figure 4.3.

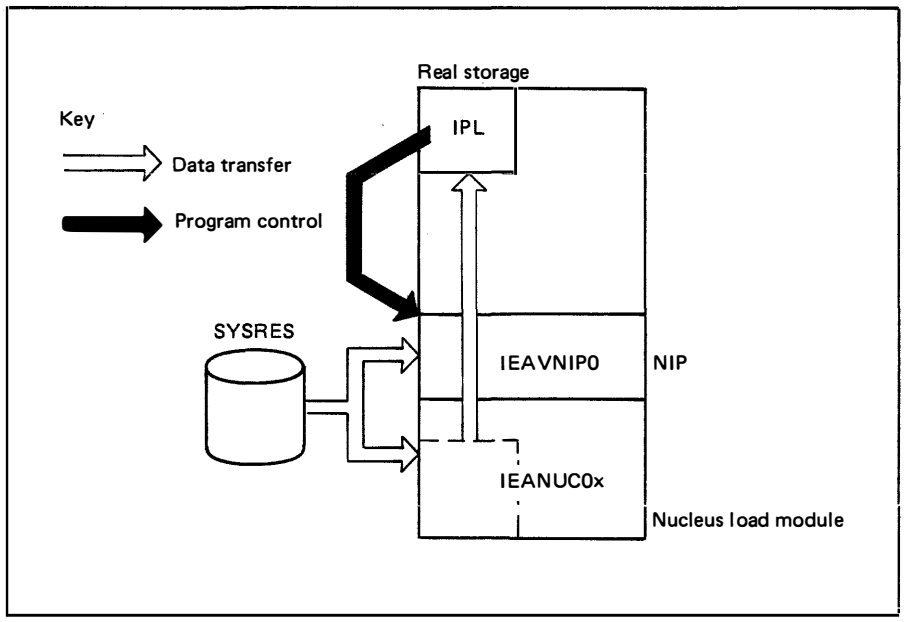



**Figure 4.3. Loading the Nucleus**

## Nucleus Initialization via NIP

After NIP receives control from IPL, it first performs a few preliminary initialization functions such as verifying that the nucleus has been properly loaded, initializing the SYSRES unit control block (UCB), and building a SYS1.NUCLEUS data extent block (DEB). Then NIP performs three major initialization functions. It:

- Initializes real storage
- Establishes an address space
- Processes SYS1.PARMLIB-specified and operator-specified initialization parameters

In addition, NIP controls initialization of system resources. (The appropriate resource initialization modules actually initialize the resources, however.)

## *Initializing Real Storage*

As previously described, IPL cleared effective real storage, as specified by the system operator or as an installation default limit. In a multiprocessing (MP) system, NIP overrides this limit, clearing **all** real storage and setting all storage keys to zero. Then NIP reserves space for permanent data areas and control blocks in real storage, after which it initializes these items.

As shown in Figure 4.4, space at the high end of real storage is reserved for the system queue area (SQA), and the control blocks necessary for the management of virtual storage and the processor are built and initialized.

Once SQA space is reserved and initialized, space for the master scheduler's local system queue area (LSQA) is obtained from the next available real storage frame below SQA. As with the SQA, appropriate control blocks are built in that area. Finally, NIP0 initializes the NIP transient area, which is used to execute the various load modules that constitute NIP.

The bottom of the NIP transient area is the top of the system area, as shown in Figure 4.4. If an installation attempts to extend the system area beyond this limit, MVS abnormally terminates and needs to be reinitialized.

NIP also initializes the page frame table entry (PFTE) for each real storage frame it allocates.

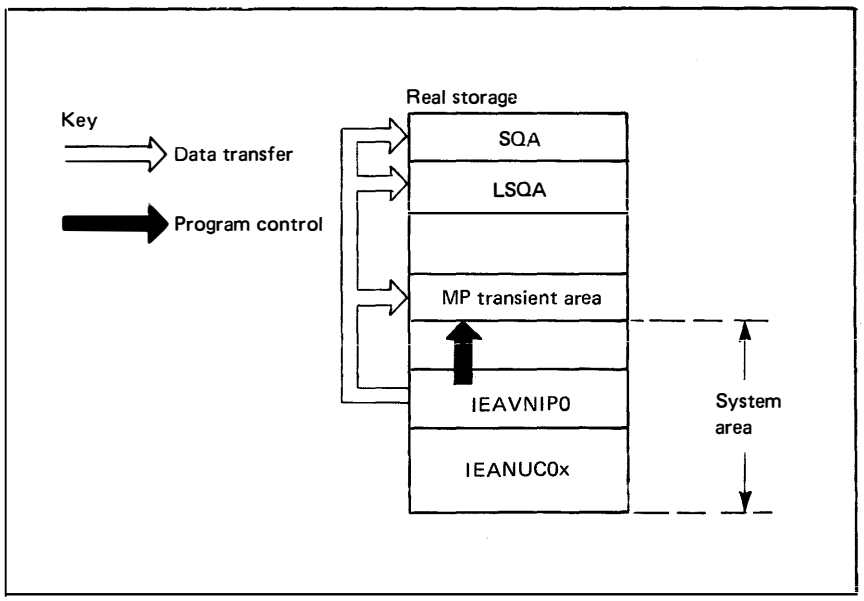



**Figure 4.4. Initializing Real Storage**

## *Initializing A Master Address Space*

NIP establishes a master address space in virtual storage. The master address space contains a **system** area, a **common** area, and a **private** area. (NIP and the master scheduler execute in the private area.) As shown in Figure 4.5, virtual space is allocated in the common area for SQA, PLPA, MLPA, and CSA. Space is allocated in the private area for the master scheduler LSQA and SWA, the master scheduler region, and the system region. Space is also allocated in the system area for the nucleus load module and, optionally, for fixed LPA and fixed BLDL.

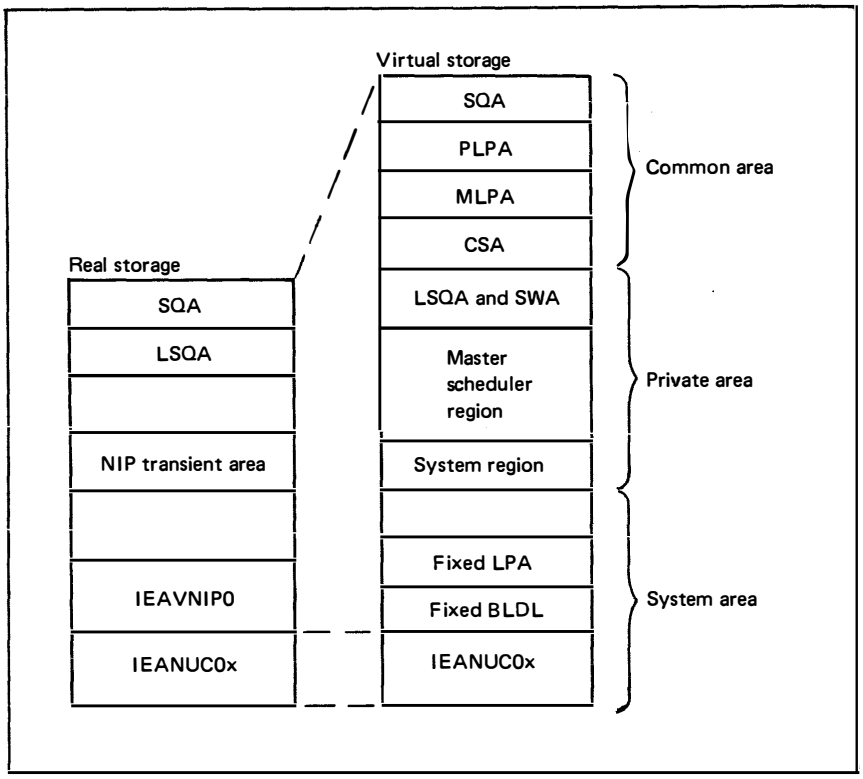



**Figure 4.5. Initializing the Master Address Space**

Next, NIP builds a segment table in the master scheduler's LSQA and initializes it with pointers to page tables for the nucleus and NIP. These page tables are built and initialized in SQA. At this point, NIP is ready to initialize system resources. However, before going into system resource initialization, a discussion on where NIP gets its initialization values is in order.

## *Obtaining System Parameters*

NIP depends on **system parameters** to tell it what initialization functions to perform, what values to use, and which SYS1.PARMLIB members to use to initialize the system. Figure 4.6 provides an overview of all system parameters. While these parameters are not discussed here at any length, some of them should be meaningful to the installation from previous discussions. Others will be discussed later. (Many of them, for example, directly affect the initialization of system resources, a topic that will be covered later in this chapter.)

| IEASYSxx Parameter | Function Performed/Value Specified/Data Set Named | SYS1.PARMLIB List Real |
|---|---|---|
| APF | Authorized library name | IEAAPFxx |
| APG | Automatic priority group for system resources manager | |
| BLDL | Pageable directory for SYS1.LINKLIB | IEABLDxx |
| BLDLF | Nonpageable directory for SYS1.LINKLIB | IEABLDxx |
| CLPA | New link pack area to be created | IEALOD00 |
| CMD | Command to be issued internally | COMMNDxx |
| CSA | Size of the common service area | |
| CVIO | Delete all VIO data sets from paging space | |
| DUMP | Data sets for SYS1.DUMP | |
| DUPLEX | Duplex data set name | |
| FIX | Reenterable routines for nonpageable LPA | IEAFIXxx |
| HARDCPY | Hard copy log | |
| IOS | specifies parmlib member containing options used by I/O Supervisor | IECIOSxx |
| IPS | Installation performance specification | IEAIPSxx |
| LNK | Names of data sets concatenated to SYS1.LINKLIB | LNKLSTxx |
| LOGCLS | Output class for log data set | |
| LOGLMT | WTL limit for log data set | |
| MAXUSER | Maximum number of virtual address spaces | |

**Figure 4.6. System Parameters (Part 1 of 2)**

| IEASYSxx Parameter | Function Performed/Value Specified/Data Set Named | SYS1.PARMLIB List Real |
|---|---|---|
| MLPA | Modifications to pageable LPA | IEALPAxx |
| OPI | SYS1.PARMLIB operator intervention restrictions | |
| OPT | System resources manager tuning parameters | IEAOPTxx |
| PAGE | Page data set names | |
| PAGNUM | Number of page and swap data sets that may be added | |
| PURGE | Demounts all mass storage system volumes | |
| REAL | V = R address area size | |
| RSU | Number of storage units available for storage reconfiguration in an MP system | |
| SMF | SMF parameters | SMFPRMxx |
| SQA | Size of the system queue area | |
| SWAP | Swap data set names | |
| SYSP | System parameter list to be merged with IEASYS00 | IEASYSxx |
| VAL | Volume characteristics | VATLSTxx |
| VRREGN | Default region size for a V = R request | |
| WTOBFRS | Number of buffers for WTO (write to operator) routine use | |
| WTORPLY | Number of operator reply elements for WTOR routine use | |

**Figure 4.6. System Parameters (Part 2 of 2)**

System parameters are provided to the initialization process from two sources: from **system parameter lists**, which are established on the system residence volume when the system is generated, and directly from the **system operator** during the initialization process.

## The System Parameter Lists

System parameter lists are contained in SYS1.PARMLIB. NIP always reads the **primary system parameter list** (IEASYS00). This list contains basic initialization instructions, installation-specified initialization defaults, and other initialization values that will not change from IPL to IPL.

SYS1.PARMLIB may also contain **secondary parameter lists** (IEASYSxx's other than IEASYS00) that can be merged with the primary parameter list at initialization time. The secondary lists, sometimes called alternate lists, contain values that override previous values in the primary list. They may also contain additional values not originally specified in the primary list. Secondary lists should contain parameters that are subject to change -- for example, they might contain the kinds of changes that are necessary between shifts. For more information on these parameters, refer to *OS/VS2 System Programming Library: Initialization and Tuning Guide.*

## System Operator Activity

The system operator is the key to a successful initialization. After console communication has been established and the system catalog opened, NIP asks the system operator to:

```
SPECIFY SYSTEM PARAMETERS.
```

If one or more secondary parameter lists are to be merged with the primary list, the system operator identifies them at this time. In addition, the system operator may directly specify certain system parameters at this time. Such a "direct specification" would include parameters that are unique for a specific IPL. If no secondary parameter lists or direct specifications are indicated by the system operator, the primary system parameter list is the sole source of initialization values.

Parameters specified in secondary parameter lists override previous parameters in the primary list. Likewise, directly supplied parameters override previous parameters in primary and secondary lists. For example, if IEASYS00 contains:

```
MLPA=00,BLDL=00
```

and IEASYS01 contains:

```
MLPA=(01,02),BLDL=01
```

and IEASYS02 contains:

```
MLPA=03,SQA=10
```

and the system operator specifies:

```
R   00,'SYSP=(01,02),SQA=2'
```

***Note:*** The SYSP parameter specifies which secondary lists are to be merged with the primary list.

then the system parameters used by NIP will be:

```
MLPA=03,BLDL=01,SQA=2.
```

While the use of secondary lists and operator-supplied parameters provides flexibility in tailoring MVS, it increases dependence on the system operator and tends to slow down the initialization process. By specifying OPI=NO in the primary system parameter list, the installation can forego operator intervention. And by specifying OPI=NO for secondary lists or for selected "critical" parameters in these lists, the installation can restrict operator intervention.

## Resource Initialization Via RIMs

NIP controls the initialization of each system resource. However, the actual initialization is done by a **resource initialization module** (RIM) that belongs to the function owning the resource. For example, because the input/output supervisor (IOS) uses and controls the unit control blocks (UCBs) that represent the I/O devices, the RIM that initializes these devices belongs to the input/output supervisor. Likewise, the RIM that initializes the system consoles belongs to the communications task because that task owns the consoles, and so on. Developing and distributing RIMs in this way tends to increase system reliability and simplify service.

This section deals with the initialization of the following system resources:

- I/O devices
- System consoles
- System catalog

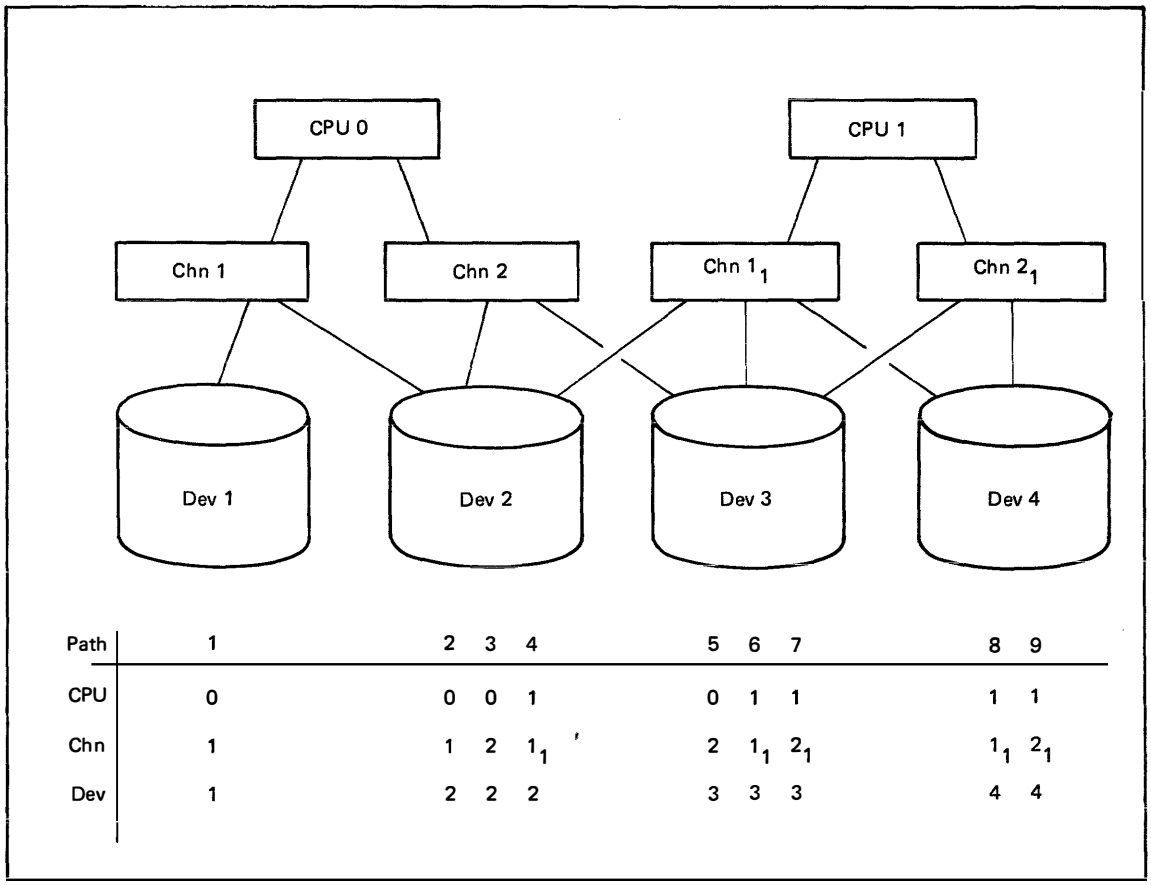
and the following resource managers:

- System resources manager
- Auxiliary storage manager
- Program manager

## *Initializing I/O Devices*

Each device is represented by a unit control block (UCB) that is used for subsequent device allocation and to control I/O operations. The I/O RIM initializes each device's UCB by setting status and condition flags in the UCB and, for DASD, by recording volume information in the UCB. However, before device UCBs can be initialized, the I/O RIM must ensure that the devices and paths to those devices are **available** and **accessible**.

An available path includes an online processor, a physical channel attached to an online processor, and at least one online device to complete the path. Figure 4.7 illustrates a configuration in which I/O device 1 has a single path, and devices 2, 3, and 4 have multiple paths. Note that for a device to be available, there must be at least one path to that device. Devices generated offline and devices generated online but with no available paths are unavailable.




| Path | 1 | 2 3 4 | 5 6 7 | 8 9 |
|---|---|---|---|---|
| CPU | 0 | 0 0 1 | 0 1 1 | 1 1 |
| Chn | 1 | 1 2 $1_1$ | 2 $1_1$ $2_1$ | $1_1$ $2_1$ |
| Dev | 1 | 2 2 2 | 3 3 3 | 4 4 |

**Figure 4.7. Paths to a Device**

The I/O RIM tests the accessibility of each available device on all available paths. To do this, the RIM requests an I/O operation on each available path. The results of these I/O operations will determine on which paths a device can be accessed. For DASD, the first of these I/O operations attempts to read the volume label to determine the volume serial number and the location of the volume table of contents (VTOC). For shared DASD, the RIM will issue an I/O operation to see if the device is actually sharable. Unavailable devices are not tested for accessibility.

After the applicable UCBs have been initialized, the RIM scans online DASD UCBs for duplicate volume serial numbers. If any duplicate volumes are found, the operator is requested to remove them.

### *Initializing Volume Attributes*

Volume attributes are actually initialized toward the end of NIP processing by a separate RIM called the volume attribute RIM. The installation can specify **mount** and **use** attributes for DASD volumes in a volume attribute list (VATLSTxx), a member of SYS1.PARMLIB. The list is selected at initialization time when the VAL system parameter is encountered in a system parameter list or is specified by the operator.

As shown in Figure 4.8, the volume attribute RIM processes the VATLSTxx and, accordingly, sets the mount and use attributes in the UCBs for all mounted volumes. If a volume is not mounted, the system operator is asked to mount it.

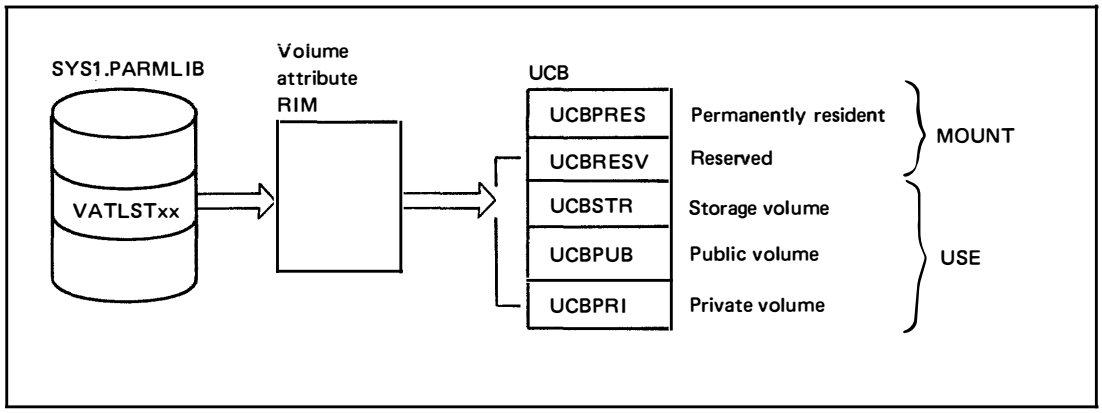



**Figure 4.8. Specifying Volume Attributes**

The MOUNT attribute indicates the conditions under which a volume can be subsequently demounted. You'll remember that a permanently resident volume (PRES) cannot be physically removed, or cannot be demounted until the device is varied offline. Such volumes, which include the system residence volume and volumes containing critical system data sets such as SYS1.LINKLIB or the paging data sets, are always marked PRES. Their MOUNT attributes should **not** be included in VATLSTxx.

Reserved volumes, on the other hand, are demountable. They remain mounted only until the operator issues a subsequent UNLOAD or a VARY OFFLINE command. A volume is marked RESV if so specified in a VATLSTxx, or if the operator issues a MOUNT command for the volume.

The use attributes indicate the types of requests for which a volume can be allocated. Volumes will be marked as storage volumes (STR), public volumes (PUB), or private volumes (PRV), as applicable.

## *Initializing System Consoles*

The system console is the I/O device the system operator uses to provide system parameters and otherwise control the initialization process. Because it is used for operator-to-system communication, it is actually one of the first devices to be initialized.

The RIM that initializes the system console must locate an available console, designate it as the **master console**, and initialize it. To do this, it looks first for the installation-specified master console. If the installation-specified master is not available, it will search for an available, installation-specified, **alternate** console to designate as master. If no alternate consoles are available, it will search for any other available console to designate as master.

Figure 4.9 shows how the RIM locates a master console. The RIM first locates the UCB for the installation-specified master console by searching the unit control module table (UCM), which contains an entry for each console in the system. The RIM checks the online flag in the appropriate UCB. If the console is online and available, it is selected as the master console.

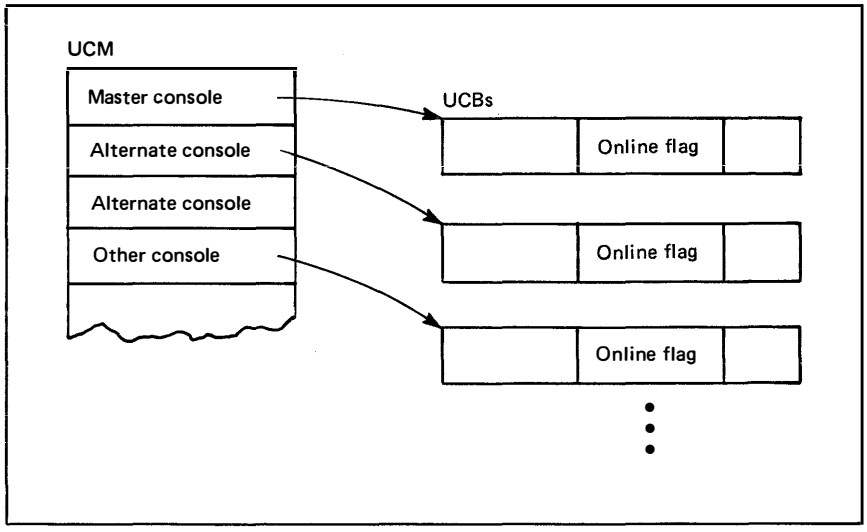



**Figure 4.9. Locating a Master System Console**

If the installation-specified master console is not available, the RIM searches the UCM for an online, available, alternate console. If it finds one, it selects it as the master console, it resets flags in the UCM entry for the installation-specified master console, and it sets like flags in the entry for the selected alternate console. If no suitable alternate console is located, the first other available console the RIM finds is designated as the master console, and the appropriate UCM entries are modified accordingly.

After a master console has been selected, the RIM passes the UCB address to NIP so that the console can be opened and used to communicate with the system operator. Finally, the RIM acquires buffer space in SQA for messages isssued by NIP. NIP uses this space to pass messages to the communication tasks so that the messages can be written as hardcopy during master scheduler initialization.

System parameters RIM uses to initialize the system consoles include: HARDCPY, LOGCLS, LOGLMT, WTOBFRS, and WTORPLY. You may want to review the explanation for these parameters given in Figure 4.6.

### *Initializing the System Catalog*

The system catalog is used to locate cataloged data sets and other catalogs. It contains the volume serial number and device type of each cataloged data set. Unlike MVT and SVS, the MVS system catalog is a **VSAM** (virtual sequential access method) data set serving as the VSAM master catalog. It can contain entries for VSAM data sets and VSAM user catalogs, as well as entries for OS data sets and OS user catalogs.

NIP can open data sets residing on the system residence volume whether or not the system catalog has been opened. However, system data sets residing on volumes other than the system residence volume are located through system catalog pointers and cannot be opened or accessed until the system catalog is initialized. For example, before NIP can complete the opening of SYS1.LINKLIB, and before any parameters can be read from SYS1.PARMLIB, the system catalog must be opened.

Various VSAM RIMs open, initialize, and close the system catalog at initialization time. As shown in Figure 4.10, one of the VSAM RIMs obtains the volume serial number and device type of the system catalog from SYS1.NUCLEUS. It then locates the UCB representing the device on which the volume is mounted. If the volume containing the system catalog is not mounted, the operator is requested to mount it. A VSAM RIM then searches the VTOC of the mounted volume to locate the system catalog. When it has been found, another VSAM RIM builds the control blocks necessary to access a VSAM data set. It then opens the data set and initializes it as the system catalog.

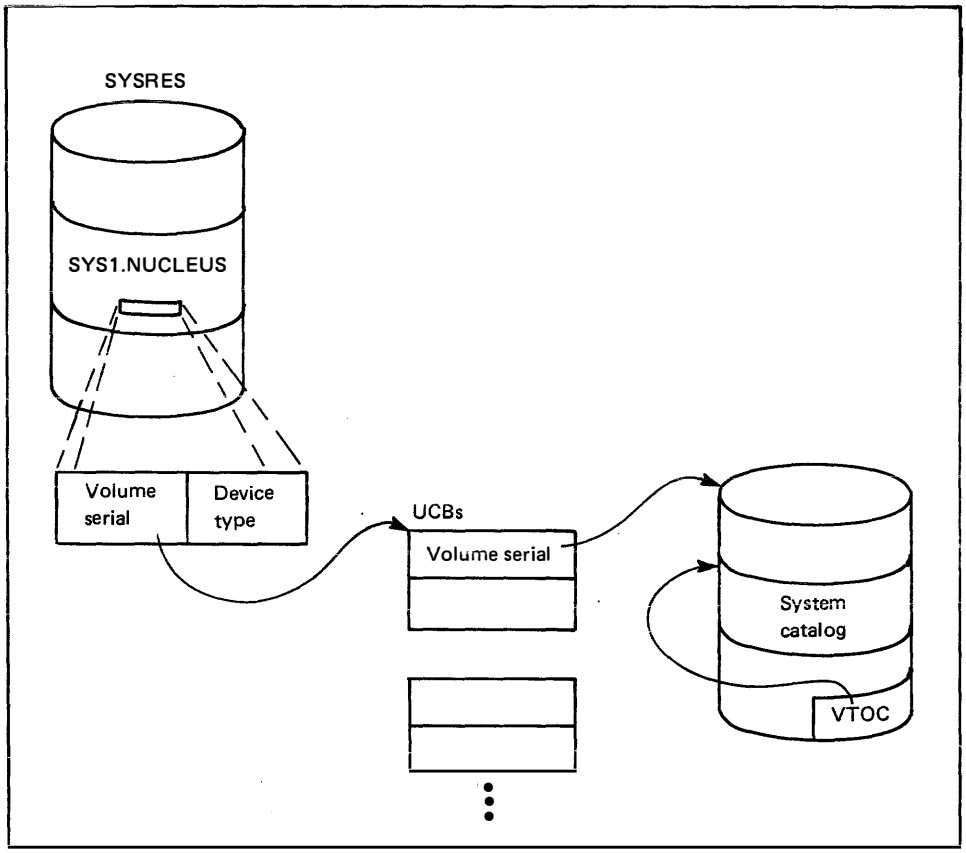



**Figure 4.10. Locating the System Catalog**

After NIP initialization has completed (before NIP terminates), a VSAM RIM is again invoked to close the system catalog. After system initialization is complete, the first reference to a cataloged system data set will cause the system catalog to be opened normally.

## *Initializing the System Resources Manager*

It is the job of the system resources manager (SRM) to provide an installation-specified level of acceptable user service while making the most efficient use of available system resources. SRM initialization consists of establishing system constants and processing certain SRM system parameters.

System constants are used to adjust processor, storage, and I/O loads, and are based on such variables as the processor model, the number of online processors, and the number of logical channels. (A logical channel is the set of all paths to a specific device or group of devices. Figure 4.7, for example, depicts four logical channels, one for each device.)

The installation establishes the level of user service in various system parameter lists and values selected at initialization time. The APG, IPS, and OPT system parameters specify or point to:

- The automatic priority group (APG)
- Installation performance specifications (IPS)
- Optional system tuning parameters (OPT)

### Automatic Priority Group (APG) Initialization

Through use of the APG system parameter, the installation establishes a range of dispatching priorities designated as an automatic priority group. During subsequent system operation, the APG value is one of the values used to determine the position of APG group address spaces on the dispatching queue. If the installation chooses not to set this value initially, a default value is established at initialization time. During a subsequent IPL, the system operator can override an existing APG value by specifying a system parameter directly.

### Installation Performance Specification Initialization (IPS)

The SRM manages the workload and apportions appropriate service to the current users of the system based on an installation-specified service rate provided as the installation performance specification. The installation performance specification is included in one of the IEAIPSxx lists, each of which is a member of SYS1.PARMLIB. The IPS system parameter tells the SRM RIM which list to use at initialization time.

### Optional System Tuning Parameter Initialization (OPT)

The SRM makes tuning decisions based on recommendations from the workload manager and the various resource managers. Optional system tuning parameters are used to weight the recommendations of the processor and I/O resource managers and to attempt to prevent the users from tying up serially reusable resources.

Optional system tuning parameters are supplied to the SRM in one of the IEAOPTxx system parameter lists, each of which is a member of SYS1.PARMLIB. The OPT system parameter tells the SRM RIM which list to use at initialization time.

### Additional SRM Initialization

After processing the APG, IPS, and OPT system parameters, the SRM RIM builds an SRM user control block (OUCB) and a user extension block (OUXB) for the master scheduler address space. Once the master scheduler is initialized, these blocks, used by SRM to control each user, is subsequently built for each address space as the address space is created.

After the SRM is initialized, NIP passes control to the RIM for the auxiliary storage manager.

## *Initializing the Auxiliary Storage Manager*

The auxiliary storage manager (ASM) controls the auxiliary storage used for paging and swapping, and the I/O operations associated with these activities. To page efficiently, the ASM divides paging requirements into pageable link pack area (PLPA), common, and local pages. When the system is generated, the installation allocates, catalogs, and formats page data sets to meet the requirements of the three types of page data sets mentioned above. The installation places the names of the data sets into the primary system parameter list. Additional page data sets can be specified in secondary system parameter lists or supplied directly by the system operator at initialization.

Optionally, the names of installation-defined swap data sets and/or duplex data sets can be specified in the same manner. Also, the installation can indicate how it wants VIO data sets to be reestablished when subsequent IPLs are performed.

After initialization, additional page and swap data sets can be dynamically added to the system. To do this, the system operator uses the PAGEADD command and names the page and/or swap data sets to be added. The total number of page and swap data sets is limited at initialization by the PAGNUM system parameter, which is obtained from a system parameter list or supplied directly by the operator.

### Page Data Set Initialization

Page data sets are opened and initialized by the ASM RIM according to the type of IPL start — cold or warm (quick starts are handled like warm starts). During a cold start (defined as the first IPL after the system is generated or any IPL in which the CLPA--create link pack area--system parameter is specified), the PAGE system parameter specifies applicable page data set names. The PAGE parameter is included in a primary system parameter list. Additional page data sets can be specified in secondary lists or directly by the system operator.

During a warm start (a start following a system failure), page data set names are "remembered" from the previous IPL. In these cases, the PAGE parameter can still be used to specify additional data sets. (Note that the PAGE parameter cannot be used to replace data sets. That is, secondary or directly-specified PAGE parameter values are concatenated to those specified in the primary list. They do not override existing values).

To successfully initialize the ASM, one PLPA, one common, and at least one local page data set must be specified and available. All page data sets (a maximum of 64) must be allocated, cataloged, and formatted as VSAM data sets prior to IPL. The sum of the local page data sets should be large enough to hold all of the private area pages and any VIO pages. The PLPA page data set should be large enough to hold all PLPA pages, and the common page data set large enough to hold all other pages in the common area (SQA, CSA, PSA).

### Swap Data Set Initialization

Swap data sets are optional, but their use can significantly improve performance. (If no swap data sets are specified, LSQA pages will be directed to a local page data set.) Swap data set names are specified by the SWAP system parameter contained in one of the system parameter lists or supplied directly by the operator. Unlike the PAGE parameter, the SWAP parameter permits both the addition and replacement of data set names specified in the system parameter lists.

Swap data sets must be allocated, cataloged, and formatted prior to the IPL. A maximum of 25 swap data set names can be specified. When SWAP is specified, at least one swap data set must be available at IPL time.

### Duplex Data Set Initialization

The installation can define a duplex data set to hold a duplicate copy of all pages written to the pageable link pack area (PLPA) and common page data sets. The DUPLEX system parameter, contained in a system parameter list or specified directly by the system operator, specifies the data set name.

Only one duplex data set can be specified, and then only on cold starts. For warm starts, the ASM RIM uses the duplex data set name specified on the most recent cold start.

If the duplex parameter is used, there must be a duplex data set available. It must be allocated, cataloged, and formatted as a VSAM data set prior to IPL.

### VIO Data Set Initialization

For warm starts, the ASM RIM will reestablish all VIO data sets if the volumes containing the previous local page data sets are available. However, for all cold starts, or if the clear VIO (CVIO) system parameter is specified for warm starts, the ASM RIM will delete all VIO data sets from local page space.

## *Initializing the Program Manager*

The program manager locates, loads, deletes, and transfers control between load modules residing in either the link pack area (LPA) or job pack area. This section discusses initialization of the LPA. (Modules in the job pack area are associated with job steps and are not discussed here.) During initialization, the program manager RIM loads LPA modules into the common area and builds and initializes related control blocks and queues. The following areas are initialized:

- The pageable link pack area (PLPA)
- The fixed link pack area (FLPA)
- The modified link pack area (MLPA)
- Various tables and lists

## Pageable Link Pack Area Initialization

The pageable link pack area is allocated in the common area of virtual storage directly below SQA. For cold starts, the program manager RIM loads the LPA modules from the link pack area library (SYSl.LPALIB) into the PLPA, as shown in Figure 4.11. Each module is represented by an entry that is built and initialized in the PLPA directory (PLPAD) as the module is loaded.

For warm starts, the PLPA is still in auxiliary storage from a previous IPL, and is not reloaded. Instead, the program manager RIM calls a real storage RIM to reconstruct PLPA page tables and segment table entries, and to place the auxiliary storage slot addresses in the appropriate external page table entries. This procedure speeds up the IPL process.

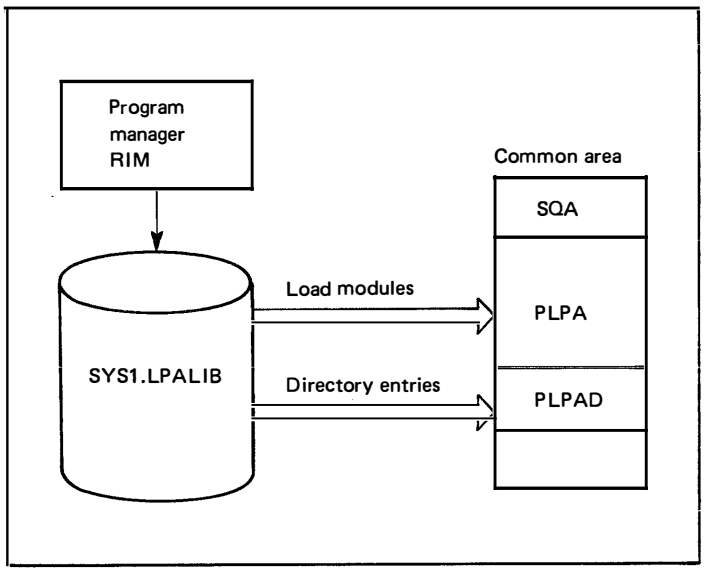



**Figure 4.11. Initializing the PLPA**

To reduce page faults and improve performance, it is sometimes appropriate to group PLPA modules that refer to each other or that execute in sequence. In this manner, the grouped modules will tend to occupy the same page, or at least be in real storage at the same time. The **system pack list** (IEAPAK00),,which is a member of SYS1.PARMLIB created when the system is generated, is used to provide such a grouping. It contains the names of the modules to be grouped.

As shown in Figure 4.12, the program manager RIM refers to the system pack list to determine the order in which SYS1.LPALIB modules are to be loaded into PLPA. If no pack list exists, modules are loaded as they are encountered, starting at the top of PLPA space. Note that there are no alternate pack lists; however, IEAPAK00 can be modified (or eliminated) by the installation prior to initialization.

If it is important to speed up the search procedure for certain link pack area modules, the load list (IEALOD00) can be used to do this. As shown in Figure 4.12, the program manager RIM creates and initializes an entry in the active link pack area queue (ALPAQ), within the SQA, for each module in the load list. During subsequent MVS system operation, the program manager searches the ALPAQ before searching the LPA directory.

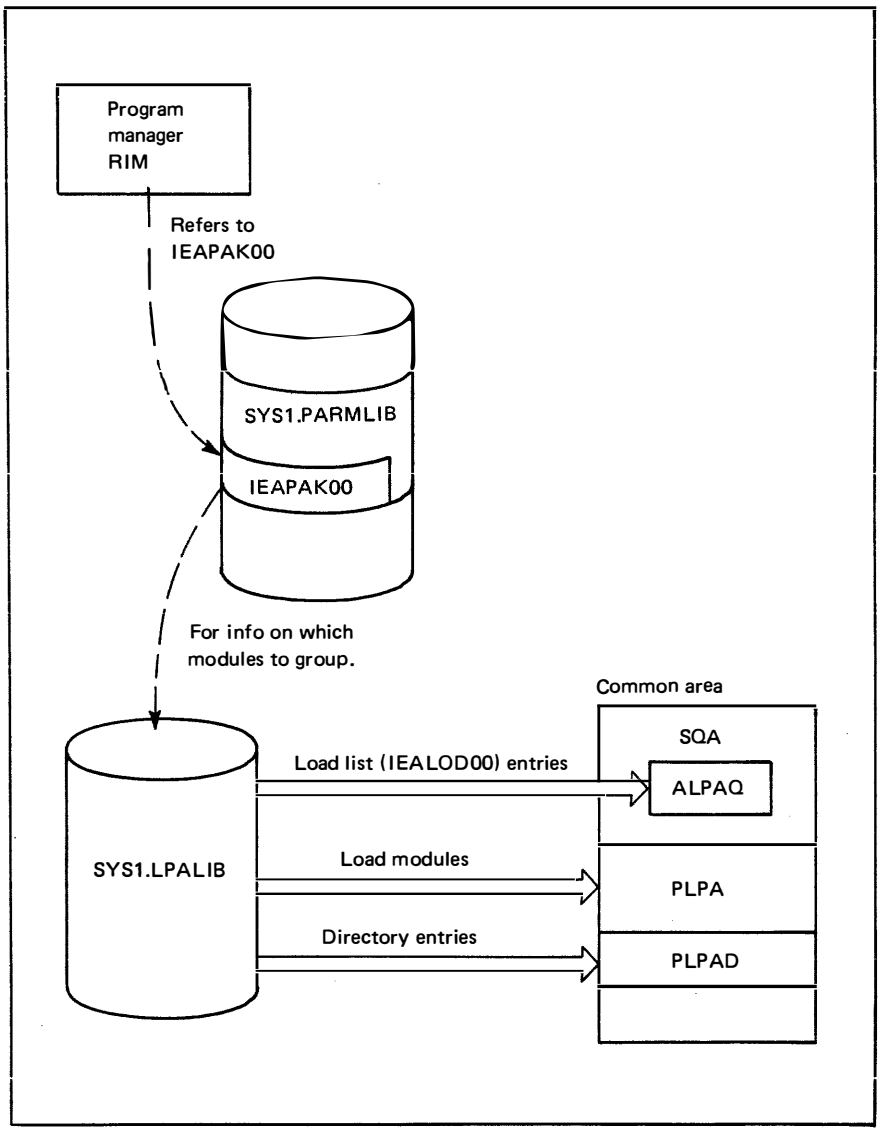



**Figure 4.12. System Pack List and ALPAQ Initialization**

## Fixed Link Pack Area Initialization

The fixed link pack area is an extension of the nucleus and is located directly above it in the system area of virtual storage. It contains reentrant modules in fixed V=R pages, which can be used by any task in the system.

As shown in Figure 4.13, FLPA modules are loaded by the program manager RIM as directed by a fix list (IEAFIXxx) in SYS1.PARMLIB. Since there can be multiple fix lists, the FIX system parameter is used to specify which list is to be used. If FIX is not specified, no FLPA modules will be loaded. The fix list can contain names from SYS1.LPALIB, SYS1.SVCLIB, and SYS1.LINKLIB.

In addition, up to 15 libraries, from which FLPA modules can be loaded, can be concatenated with SYS1.LINKLIB. To concatenate libraries, the installation creates and/or modifies one or more link lists (LNKLST00 or LNKLSTxx). The link lists contain the names of libraries to be concatenated. At initialization, the LNK system parameter is used to specify which link lists are to be used. If LNK is not specified, only the default LNKLST00 will be used. (This list, as created when the system is generated, contains only the name SYS1.LINKLIB.)

As the program manager RIM loads FLPA, it builds an SQA ALPAQ entry for each module. After FLPA is loaded, it is possible for modules from SYS1.LPALIB to now exist in both PLPA and FLPA. In these cases, the FLPA module represented in the ALPAQ is the one used. The PLPA module will be in the PLPA directory, but not on the active queue.

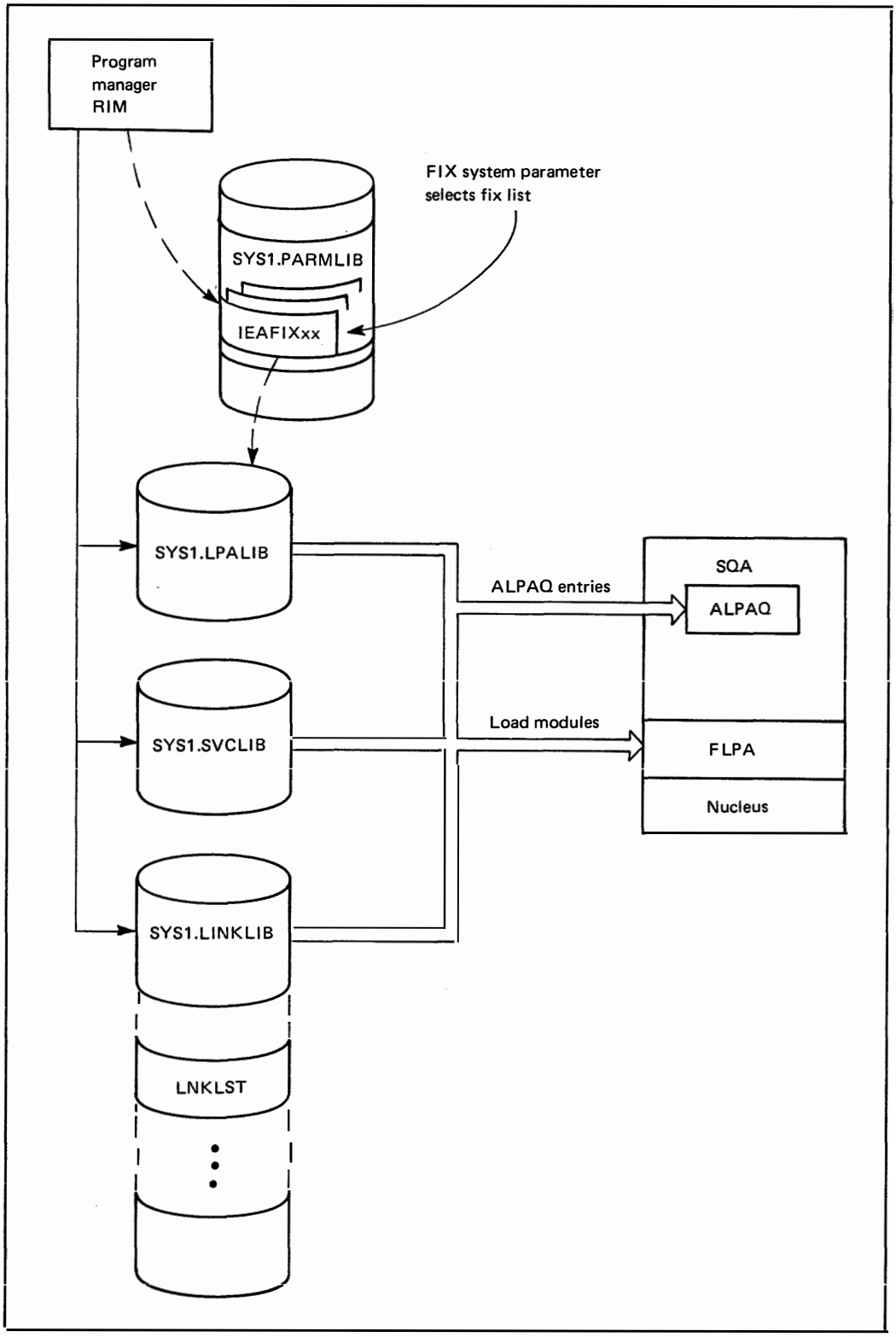



**Figure 4.13. Initializing FLPA**

## Modified Link Pack Area Initialization

The modified link pack area (MLPA) is an optional area located directly below the PLPA directory in virtual storage. It constitutes an extension to PLPA that remains on the paging data sets and on the ALPAQ only until the next IPL. With the next IPL, the area is cleared.

As a choice of modules to put in the MLPA, the installation might select those modules that have been tentatively modified and are being tested. The original module is not removed from PLPA, but the MLPA module is substituted for the original module during the current IPL.

Modules to be included in MLPA must be named in one of the modified LPA lists (IEALPAxx) specified by the MLPA system parameter. If MLPA is not specified, no MLPA modules are loaded. The program manager RIM loads each MLPA module and builds an entry for that module on the ALPAQ. MLPA modules, like FLPA modules, can be loaded from SYS1.LPALIB, SYS1.SVCLIB, and SYS1.LINKLIB. Additional concatenated libraries can be included.

## Table and List Initialization

In addition to initializing LPA, the program manager RIM initializes tables and lists used by the program manager. These include:

- BLDL list
- SVC table
- APF table

**BLDL List:** The BLDL list contains directory entries for frequently-used modules from SYS1.LINKLIB or any of the concatenated libraries. The program manager uses the BLDL list to eliminate the I/O required to bring the directory into storage when accessing a module that is not in virtual storage. (An in-storage copy of the directory is used.) A well thought out BLDL list can significantly improve performance. It can be in fixed storage directly above FLPA, or in pageable storage directly below MLPA. (A fixed BLDL list improves performance even more by eliminating the page faults that might otherwise be encountered in searching the list itself.)

The names of the modules to be included are contained in a IEABLDxx list. The BLDLF system parameter specifies the fixed BLDL list to be used. The BLDL system parameter specifies the pageable list. The program manager builds and initializes either a fixed list or a pageable list.

**SVC Table:** The SVC table contains an entry for each available SVC routine. The program manager RIM initializes entries for SVC routines that are not a part of the resident nucleus but have been placed in the LPA. It searches the ALPAQ and the PLPA directory for SVC load modules and places their addresses in the appropriate entries within the SVC table. If a load module cannot be found, the RIM places the address of the SVC error routine in the SVC table.

**APF Table:** The authorized program facility (APF) permits an installation to identify the system and user libraries that contain programs authorized to use restricted functions. The names of these authorized libraries are placed in an APF table that the program manager RIM builds in SQA. Entries in the table are established at initialization for SYS1.LINKLIB and

SYS1.SVCLIB. As a result, these libraries are always authorized. ***Note***: Because concatenated libraries are assumed to be part of SYS1.LINKLIB, they are authorized when accessed through SYS1.LINKLIB. If accessed any other way (e.g., through STEPLIB) they are not necessarily authorized. (That is, they are authorized in these cases only if included in the APF table.)

In addition, the installation can specify authorized libraries in any APF list (IEAAPFxx) contained in SYS1.PARMLIB. The list to be used is specified by the APF system parameter. The program manager RIM initializes an entry in the APF table for each library named in the applicable IEAAPFxx list.

## Master Scheduler Initialization

Master scheduler initialization can be broken into three steps, as shown in Figure 4.14. In the first step, the base initialization routine performs some basic initialization functions. In the second step, the initiator initiates the master scheduler by attaching the master scheduler region initialization routine as a job step task. To do this, it processes, through the subsystem interface, a set of master JCL (MSTRJCL) statements obtained from SYS1.LINKLIB. In the third step, additional tasks are attached by the region initialization routine. In addition, automatic commands contained in a command list (COMMNDxx) on SYS1.PARMLIB are executed or scheduled for execution, as the case may be. After region initialization is completed, control is transferred to the master scheduler wait routine.

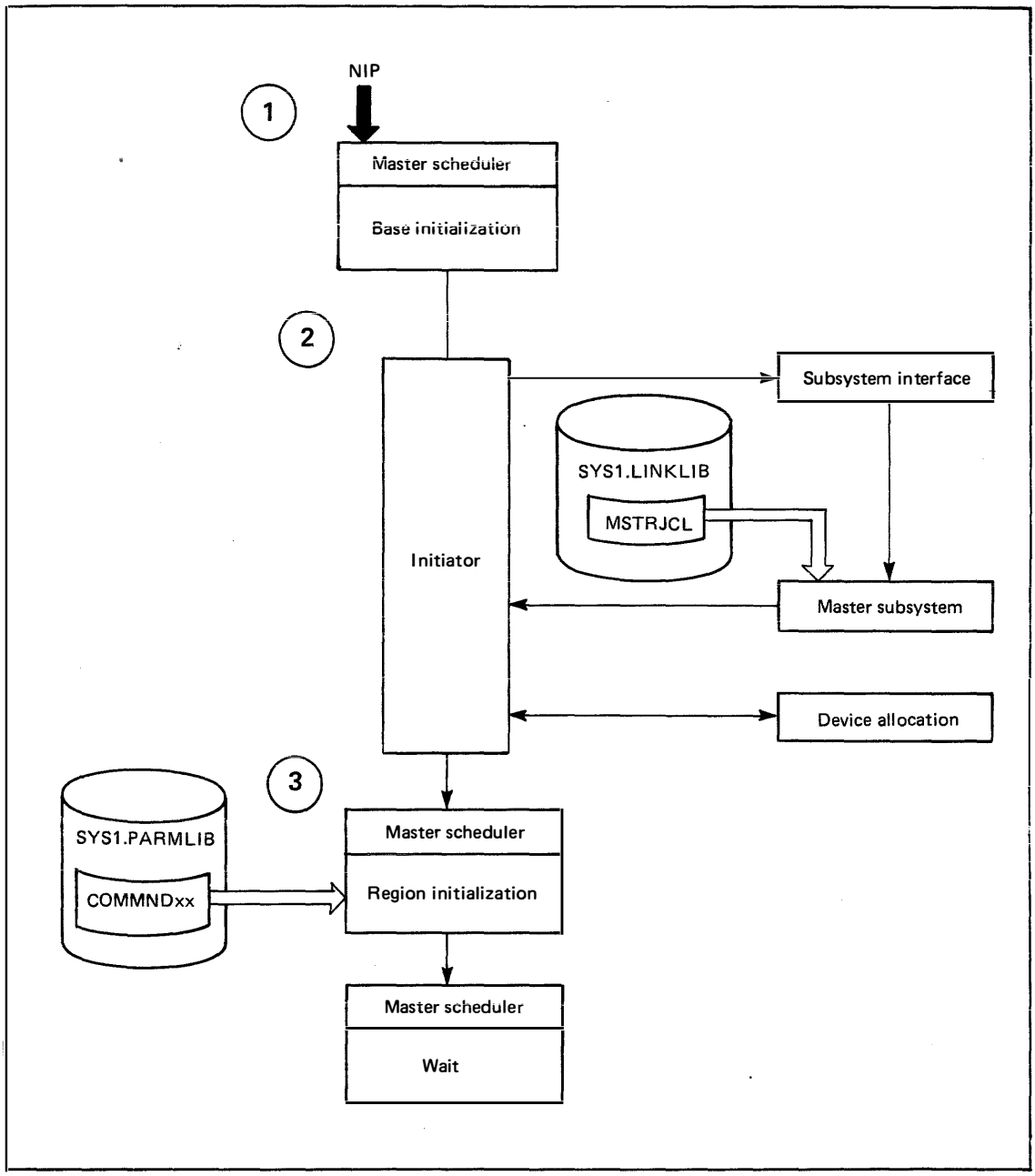



**Figure 4.14. Master Scheduler Initialization**

### *Initializing the Master Scheduler Base*

The master scheduler base initialization routine is entered from NIP. It creates and initializes the control blocks needed to invoke the initiator. Then it locates and stores entry points for certain job scheduler routines. It initializes the subsystem interface, the communications task, some TSO addresses and parameters, and the time-of-day clock. Finally, it attaches the initiator to initiate the master scheduler.

### *Initiating the Master Scheduler*

Before the initiator can attach the master scheduler region initialization routine, it must read the JCL to do so. (Applicable job step task control blocks must be created and data sets must be allocated.) As yet, however, no JES readers are active and no procedure libraries are open. So the initiator gets the necessary JCL from a load module (MSTRJCL) established on SYS1.LINKLIB at system generation time.

To read and process MSTRJCL, the initiator uses the subsystem interface to request job entry services, as shown in Figure 4.14. The request is passed to the master subsystem, which reads the MSTRJCL and invokes job scheduler routines to process the JCL and initialize necessary control blocks. The last statement in MSTRJCL is a command to START JES. This command is passed to the command processor portion of the master scheduler and scheduled for execution.

The initiator uses the device allocation routine to allocate the data sets indicated in MSTRJCL and required by the master scheduler (data sets such as SYS1.PROCLIB and SYS1.PARMLIB). These are required when JES is subsequently started. Two internal readers are also allocated. They are used later to pass JCL from system routines to JES. Lastly, the initiator attaches master scheduler region initialization as the job step task, and the master scheduler is active.

### *Initializing the Master Scheduler Region*

The region initialization routine attaches other tasks to be run in the master scheduler region and passes commands located in SYS1.PARMLIB to the command processor for execution or scheduling. These commands are contained in a command list (COMMNDxx) a member of SYS1.PARMLIB. Because there can be multiple command lists, the CMD system parameter is used to tell master scheduler initialization which list to use.

When initialization is complete, control is transferred to the master scheduler wait routine, which eventually encounters START JES in a command scheduler control block (CSCB).

## Job Entry Subsystem (JES) Start-Up

When the master scheduler wait routine scans the CSCB chain and finds the START JES CSCB, it attaches a new address space. (START JES is a task-creating command requiring a new address space.) Then a region control task (RCT) prepares the address space for execution. After these preliminaries have been taken care of, started task control (STC) builds the JCL necessary to invoke the JES procedure. Then the initiator starts JES.

### *Creating an Address Space*

The master scheduler attaches the address space create routine. This routine asks SRM if a new address space can be created. Because the only thing running at this point is the master scheduler, there should be no contention for system resources. After the address space create routine receives permission to proceed, it invokes the virtual storage manager to create a virtual address space for JES. Then it builds LSQA in the private area and initializes page table entries. Lastly, it builds task control blocks for a region control task (RCT) and places the address space control block (ASCB) on the dispatching queue.

### *Initializing the Region Control Task*

The region control task (RCT) is the highest priority task in the new address space. Therefore, when the JES address space becomes active, the RCT is the first task dispatched. RCT controls the address space and prepares it for execution (RESTORE) after a swap-in. It also prepares the address space (QUIESCE) for a swap-out, and frees the address space when the initiator terminates.

After RCT is initialized, it attaches the started task control routine to initiate JES.

### *Initiating JES*

The started task control (STC) routine uses START JES CSCB information to build the JCL necessary to invoke the JES procedure. Next, STC invokes the master subsystem to see if JES is already started (as in subsequent system operation). If JES was already started, STC uses it. But since JES is not yet started, STC links to the initiator.

The initiator invokes the master subsystem, which uses job scheduler routines much as it did when initiating the master scheduler. However, to start JES, it uses the internal JCL built by STC rather than MSTRJCL. After SYS1.PROCLIB has been allocated to the master scheduler, the JES procedure can be read.

After all JCL has been processed and after job scheduler control blocks have been built in the SWA, the initiator links to device allocation to allocate JES data sets specified in the JES procedure. Then, using the program name from the EXEC statement of the JES procedure, the initiator attaches the primary job entry subsystem. JES is started and MVS is ready for work.

# Chapter 5: Entering and Scheduling Work

When system initialization is complete and the job entry subsystem is active, MVS can accept jobs for processing. All jobs, started tasks (other than the job entry subsystem), and time-sharing LOGON requests must enter the system through the job entry subsystem. Also, the job entry subsystem processes all output data produced by the jobs.

MVS works with either of two job entry subsystems — JES2 or JES3. Only one of these subsystems can be specified during system generation to run as the primary job entry subsystem for MVS. Both JES2 and JES3 perform the following basic functions:

- Reading jobs into the system
- Scheduling jobs for execution
- Maintaining all data submitted with jobs
- Supporting the system management facilities
- Handling output from jobs and time-sharing users

This book uses the acronym JES when referring to the basic functions supported by both JES2 and JES3. JES2 and JES3 are described separately in more detail later in this chapter.

## Terminology and Concepts

The following information describes several terms and concepts that are essential to understanding how a job entry subsystem works.

### *Input Stream*

When you present a job to the system for processing, the job consists of JCL statements and input data. The JCL statements specify job information, data set characteristics, and device requirements for the job's execution. The input data is the data to be processed. The sequence of JCL statements and input data for one or more jobs being submitted is called an input stream. The job entry subsystem reads an input stream from card readers, magnetic tapes, direct access devices, remote and local terminals, and internal readers.

### *Internal Reader*

An internal reader is not an actual hardware device such as a card reader; it is a special output data set that other programs can use to submit jobs, control statements, and commands to the job entry subsystem. The job entry subsystem can receive multiple jobs simultaneously through the internal reader facility. MVS uses the following two internal readers, allocated during system initialization, to pass JCL for started tasks and TSO logon requests to the job entry subsystem:

- STCINRDR, which is used by the started task control (STC) routine when processing a START command. For example, if you issue a command to start VTAM, STC creates JCL to refer to the VTAM procedure and passes the JCL to the job entry subsystem through the STCINRDR internal reader.
- TSUINRDR, which is used by the TSO LOGON command to initiate a TSO terminal session. The LOGON command writes the user-specified data set(s), consisting of JCL and input data, into the TSUINRDR internal reader.

IBM supplies an external reader procedure named RDR that uses the internal reader facility to submit an input stream from tape or disk. Also, any job executing in MVS can use the internal reader facility to pass an input stream to the job entry subsystem.

### *Initiators and Job Classes*

An initiator is a job scheduler function that:

- Receives jobs and job steps to be executed
- Causes input/output devices to be allocated for them
- Places them under task control
- Supplies (at completion of the job) control information for writing job output on a system output device

Normally, the system operator or the job entry subsystem starts several initiators after system initialization is complete.

A job class is any one of many job categories that an installation can define using JES initialization parameters. By assigning jobs to job classes, the installation can attempt to:

- Avoid contention between jobs that require the same resources by preventing these jobs from running concurrently
- Provide a better mix of jobs for more efficient use of the system
- Process high-priority work quickly

To define a job class, first determine which characteristics are most important in achieving a good balance of jobs in your installation. Generally, jobs of similar characteristics and identical processing requirements should be assigned to the same class. For example, assume that several jobs are time-dependent and are executed in nonpageable dynamic storage. Running these jobs concurrently may not be desirable because it will tie up much nonpageable dynamic storage. The jobs can all be assigned to class B (or C or D — class names have no inherent meaning); then, if only one initiator is started that can accept class B jobs, more than one of these jobs will never be in execution at once.

Suppose you make the following assignments:

```
Class B = jobs that are time-dependent
Class C = jobs with high instruction-processing
          requirements
Class D = jobs with high I/O-request requirements
```

And you specify initiator parameters such as:

```
I1 (initiator 1) can process classes B, C, and D
I2 (initiator 2) can process classes C, D, and B
I3 (initiator 3) can process classes D, C, and B
```

Initiator 1 can accept jobs in classes B, C, and D, but the lowest-priority job in class B will be executed ahead of the highest-priority job in class C, and so on. That is, initiator 1 will only process class C jobs when class B is empty, and class D jobs when classes B and C are empty. If the three initiators are processing jobs with the same priority and all necessary resources (for example, I/O devices and data sets) are available, then three jobs (one from each of the three different classes) run concurrently. If a job within one of the classes has higher priority than the others in that class, it will be initiated first.

To specify a job's class, you code the CLASS parameter on the JOB statement. Classes are automatically associated with each initiator during JES initialization or dynamically by the operator. During execution, the initiator receives jobs from JES in priority order within their class. That is, the lowest priority job in the first non-empty class is selected ahead of the highest priority job of the next class.

When an initiator becomes active, it asks the job entry subsystem for a job that is ready for execution. The job entry subsystem selects the highest priority job in a class associated with the initiator, prepares the job for execution, and returns the job to the initiator. The initiator attaches the first job step within the job and waits for it to complete before attaching the next job step, and so on. When all job steps in the job have completed, the initiator cleans up the address space and asks for another job. This continues until the operator stops the initiator or the job entry subsystem, or until the job class associated with the initiator is exhausted.

For more information on establishing job priorities, see *OS/VS2 MVS JCL.*

## *Address Space Creation*

During system initialization, the control program relates an address space for the master scheduler, and one for the job entry subsystem. After initialization is complete, the control program creates additional address spaces in response to START, MOUNT, and LOGON commands, which represent requests to use system resources. The control program creates one address space for each program started by a START command (such as TCAM, IMS, or an initiator), each MOUNT command, and each logged-on time-sharing user.

When you want to start a job from a console device, reserve a volume on a device for all jobs that need that volume, or start a TSO terminal session, you enter a START, MOUNT, or LOGON command, respectively. The master scheduler, in conjunction with other system components, creates a task that performs the requested function (initiating a job, reserving a volume, or initiating a TSO session) in the task's own address space. Figure 5.1 summarizes the process of creating an address space.

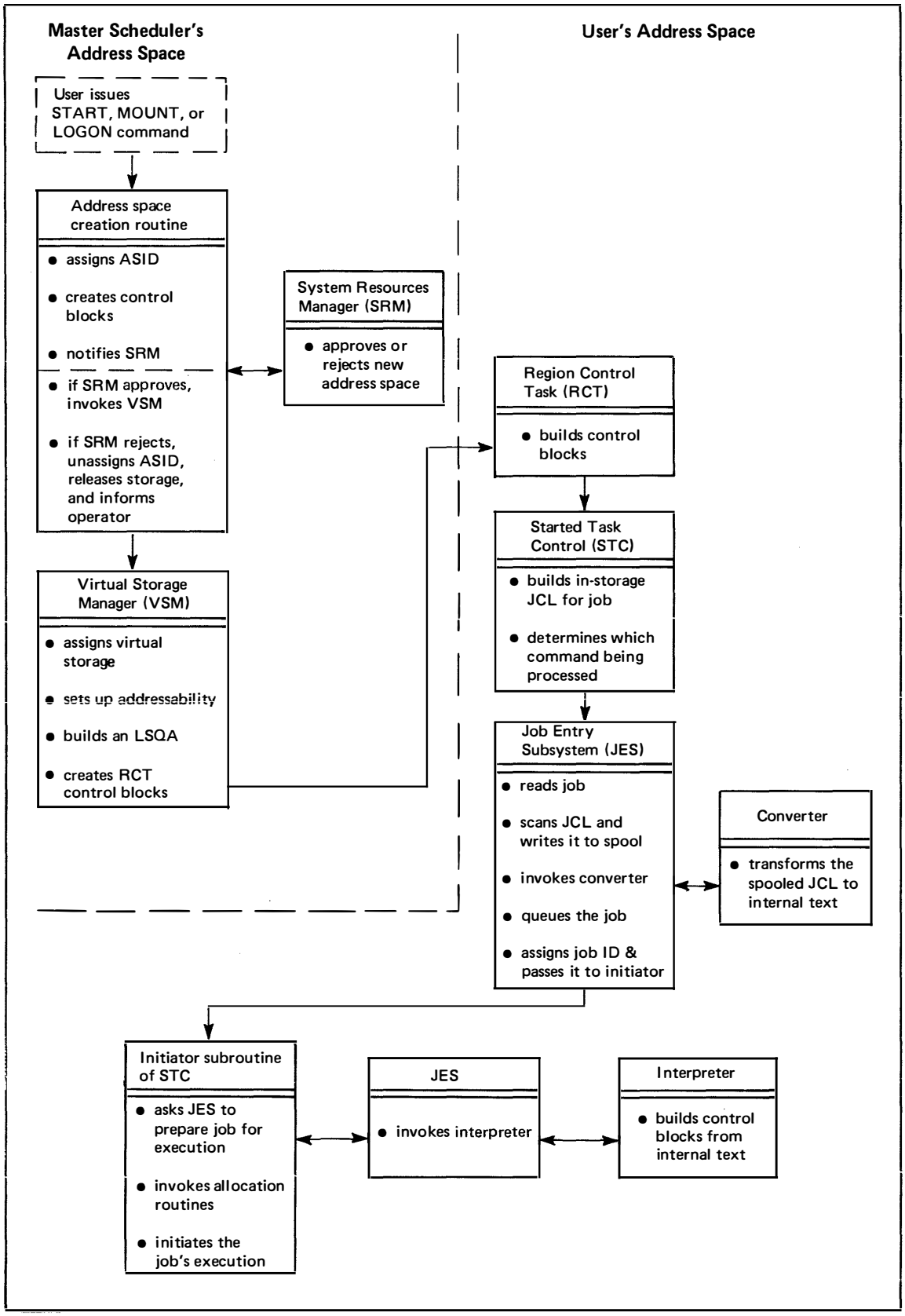



**Figure 5.1. Creating an Address Space**

The **address space creation** routine, operating in the master scheduler's address space, assigns the new address space an ASID (address space identifier) and creates control blocks for it. Then the routine notifies the **system resources manager** (SRM) that a new address space is to be created. SRM decides (based on the availability of system resources) whether the creation of an address space is advantageous. If system conditions are unfavorable for creating a new address space, SRM does not allow the address space to be created. The address space creation routine unassigns the ASID and frees the storage used by the control blocks. The operator receives a message indicating that the address space could not be created. If current system conditions are favorable to creating the new address space, the address space creation routine invokes **virtual storage management** (VSM) to assign virtual storage and set up addressability for the address space. VSM builds an LSQA (local system queue area) and sets up a segment table, page table, and external page tables in it. VSM also creates control blocks to operate the **region control task** (RCT) for the address space.

***Note:*** The MAXUSER parameter specified during system initialization limits the number of address spaces that can exist at any one time; within the MAXUSER limit, SRM controls the number of address spaces that actually exist at any one time.

Next the RCT receives control in the new address space. One RCT exists for each address space. When the address space is created, the RCT is the only task associated with it. The RCT builds control blocks that further define the address space, then attaches the **started task control** (STC) routine.

STC determines which command is being processed (START, MOUNT, or LOGON), builds in-storage JCL for the task associated with the command, the passes the JCL to the **job entry subsystem**. The job entry subsystem reads the job, scans the JCL and writes it on a spool data set, invokes the **converter** to transform the spooled JCL into internal text, queues the job on an internal queue, and assigns a job ID which it returns to STC.

Next, STC uses its **initiator subroutine** to pass this job ID back to the job entry subsystem with a request to prepare the job for execution. The job entry subsystem invokes the **interpreter** to build and initialize the scheduler control blocks for the address space from the internal text created by the converter. Upon return from the job entry subsystem, the initiator subroutine invokes the allocation routines and issues an ATTACH macro instruction for the task related to the address space: any started program (START), the MOUNT command processor (MOUNT), or the terminal monitor program (LOGON).

## Job Entry Subsystem Processing

Job entry subsystem (JES) processing consists of five stages:

- Input
- Conversion
- Execution
- Output
- Purge

The system operator can communicate with the job entry subsystem in all stages by using JES commands that control and monitor the devices, jobs, and functions. For descriptions of the JES commands, see either *Operator's Library: OS/VS2 MVS JES2 Commands* or *Operator's Library: OS/VS2 MVS JES3 Commands.*

The following descriptions of the five stages apply to JES2 and JES3. Functions unique to JES2 or JES3 are described briefly in "JES2 Features" and "JES3 Features" later in this chapter.

### *Input*

The job entry subsystem reads an input stream from card readers, magnetic tapes, direct access devices, remote and local terminals, and internal readers. Before passing control to the converter, JES stores the JCL and input data on a direct access device called a **spool data set**.

### *Conversion*

The converter takes the JCL from the spool data set, merges it with JCL from a procedure library (for any job that requests inclusion of a procedure), and converts the JCL to internal text (a form of data that is recognizable by the control program). The internal text is also stored on the spool data set. If the converter detects any JCL syntax errors, it issues diagnostic messages and places the job on the output queue. If the job has no syntax errors, the job entry subsystem assigns it a job ID and puts it on a priority queue to await processing.

### *Execution*

Jobs are selected in priority sequence within each job class. JES selects a job for execution when an initiator eligible to process the job is available. The use of the word "priority" for JES refers to input queues and output queues and the order in which jobs will be selected for processing.

JES invokes the interpreter to build and initialize SWA control blocks from the internal text created by the converter. The initiator calls the allocation routines to analyze the I/O device requirements of the job and to allocate the required devices and data sets. Then the initiator activates the job. JES provides an access method for reading and writing data to and from the spool data sets in response to requests from executing jobs. When the job completes its processing, JES places it on a queue to await output processing.

### *Output*

During its execution, a job creates system messages that must be printed, and data sets that must be printed or punched. Upon termination of a job, JES analyzes the characteristics of the job's output in terms of output class, setup requirements (such as mounting a carriage control tape or pre-printed forms on a printer, or inserting identification cards between sets of punched card output on a card punch), and output priority. JES queues the output data according to these characteristics; the output data on each output queue has identical characteristics and is eligible for processing only on an output device that matches these setup characteristics. Thus JES minimizes operator interaction with the output devices by grouping together similar output data.

The JES print/punch routines or the external writer process job output. The external writer facility allows the user to write to devices other than printers and punches (such as disks or magnetic tapes) and allows the user to control all output written by installation-supplied writers.

### *Purge*

When all processing for a job is completed, JES releases the spool space assigned to the job, making it available for allocation to subsequent jobs. JES also issues a message to the operator to indicate that the job has been purged from the system.

## JES2 Features

JES2 provides four features that extend the basic job scheduling functions of a job entry subsystem. JES2 allows an installation to:

- Dynamically control a job's input priority (priority aging)
- Reduce job-scheduling overhead for certain types of jobs (execution batch scheduling)
- Automatically schedule a given set of commands at specified times (automatic commands)
- Share a common workload across several processors (multi-access spool)

### *Priority Aging*

JES2 can increase the priority of a job based on the length of time the job has been in the system. JES2 initialization parameters specify an upper and a lower limit for priority aging, and an integer that represents the number of times the priority can be increased in a 24-hour period. By using the priority aging facility, an installation can ensure the eventual processing of low priority jobs. The longer a low-priority job remains on an input queue, the greater its chance for execution.

### *Execution Batch Scheduling*

Execution batch scheduling is an extension of normal job-scheduling that can increase throughput by reducing job-scheduling overhead for certain types of jobs. The jobs eligible for execution batch scheduling are jobs of

relatively short duration, especially multistep jobs with common setup requirements that are run frequently. Examples of such jobs are: compile-and-go debugging jobs and order-entry and file-inquiry jobs.

To use the execution batch scheduling facility, an installation must write an execution batch (XBATCH) processing program and a procedure to initiate it, and assign the jobs a unique job class associated with the execution batch procedure. Also the installation must include execution batch scheduling parameters when initializing JES2. When JES2 recognizes a job with the execution-batch-scheduling job class, JES2 builds and submits JCL through an internal reader to invoke the XBATCH procedure. Once the XBATCH procedure initiates the XBATCH program, the program remains active as long as it has jobs to process. Thus execution batch scheduling involves gathering related jobs into a single input stream and passing them as an input data set to the user-written XBATCH program. This process reduces the overhead associated with setting up for and processing numerous individual jobs or job steps.

For more information on the XBATCH program, see *OS/VS2 MVS System Programming Library: JES2*.

### *Automatic Commands*

You can specify from the console or through a local reader that certain commands or strings of commands take effect automatically at specific times or at regular intervals. By performing common preset routines through automatic command processing, you can eliminate the operator's involvement in such tasks as:

- Providing periodic status displays
- Starting, stopping, and modifying initiators
- Starting or stopping remote lines

### *Multi-Access Spool*

Previous topics have described JES2 functions on a single system (a uniprocessor or a multiprocessor) operating under a single copy of the MVS control program. JES2 can also operate two to seven such systems (each a uniprocessor or multiprocessor) as members of a multi-access spool configuration, as shown in Figure 5.2.

Each system in the configuration operates independently and includes all functions previously described for single JES2 systems. That is, each JES2 system can read jobs from local and remote card readers, schedule jobs for conversion and execution under MVS initiators, print and punch results from local and remote output devices, and communicate with operators and time-sharing users. However, all systems share a common JES2 workload queue, which resides on spool volumes.

By sharing a common queue, the systems can balance the workload by allowing jobs to execute on whatever system has an idle initiator with the correct class, and to print or punch on whatever system has an idle device with the correct class, routing, setup, and other requirements.

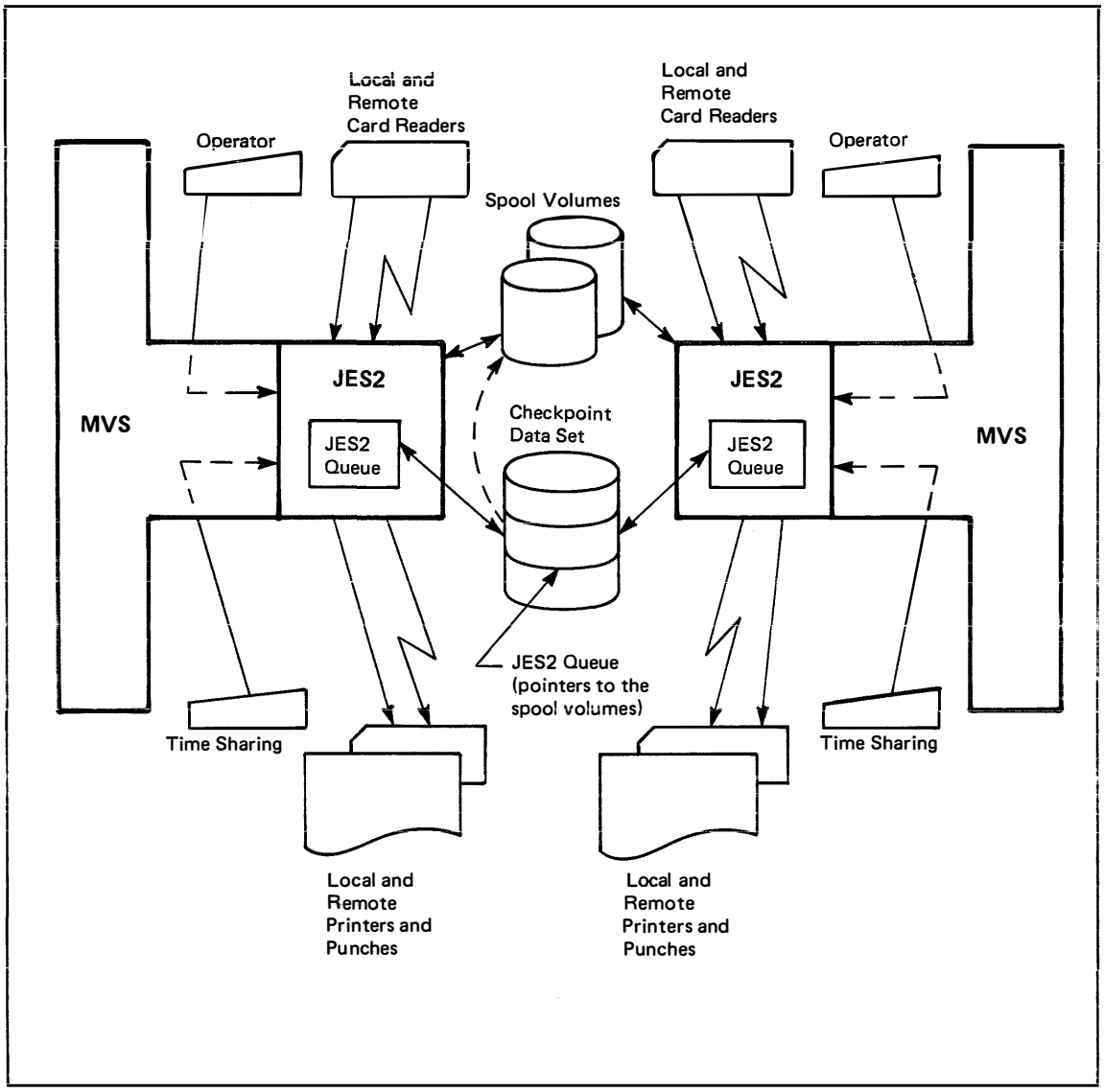



Figure 5.2. A JES2 Multi-Access Spool Configuration

Because all systems are functionally the same, if one system in the configuration fails, the others may continue processing from the common queue. Only work in process on the failed system is interrupted; this work may be recovered by a warm start of the failed system while the other systems continue processing.

## JES3 Features

Under MVS, JES3 supports configurations of one to eight physically-connected uniprocessors or multiprocessors. In addition, the configuration can include a number of ASP main processors, up to a combined maximum of 32 processors.

The **global** JES3 function, which can reside in any one of the MVS systems that make up the configuration, actively controls the other processors in the complex. The other MVS processors in which JES3 resides are called **local processors**. All processors on which jobs can execute are called **main processors;** therefore, there may be a global main processor, one or more local main processors, and one or more ASP main processors. Figure 5.3 shows a typical JES3 complex.

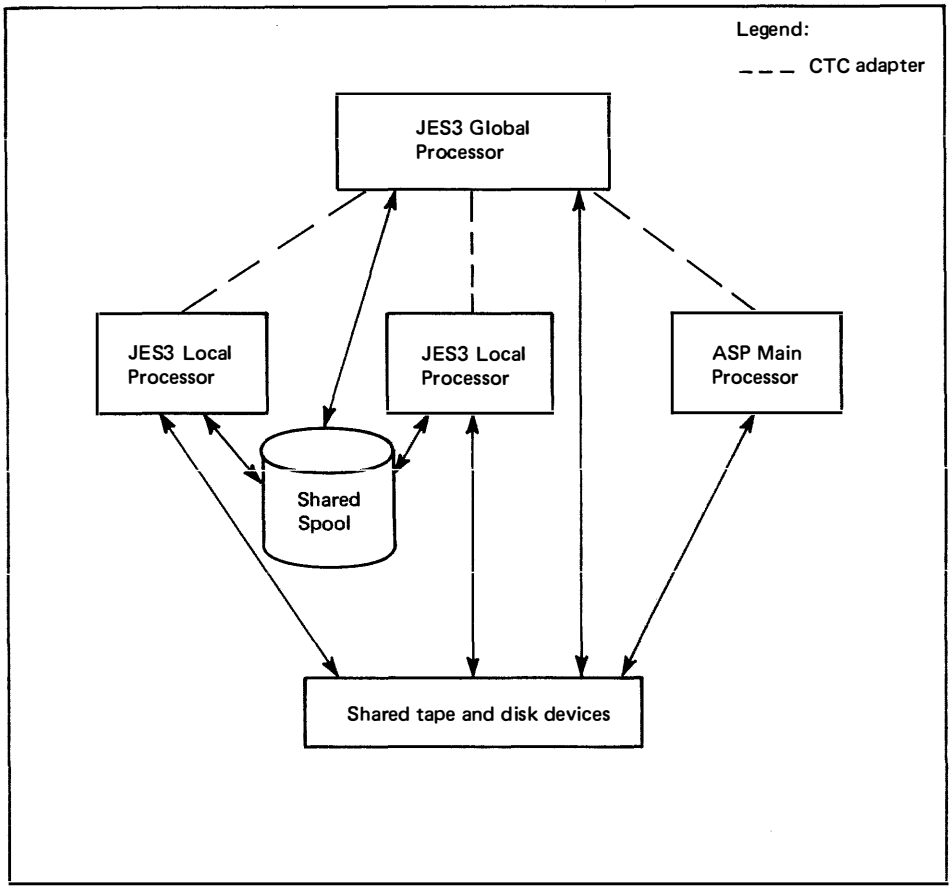



**Figure 5.3. A JES3 Complex**

The spool volumes, containing SYSIN and SYSOUT data, JCL, internal text, and the job queues for the entire complex, are shared by the MVS processors. Each local processor is also connected to the global processor by a channel-to-channel adapter for the interchange of control information.

JES3 must run under MVS in all systems sharing the spool volumes. As a compatibility aid, JES3 also supports ASP Version 3.1 main processors on a System/360 or System/370 connected through a channel-to-channel (CTC) adapter to the global JES3 processor. These ASP main processors do not share the spool volumes. Their access to the centralized job queue is through the CTC adapter to the global system.

JES3 performs the same basic job entry, scheduling, and output services that JES2 performs. JES3 also offers the following features, some of which JES2 also provides:

- Dependent job control
- Device fencing
- Priority aging (also provided by JES2)
- Deadline scheduling
- Network job processing (also provided by JES2)
- Remote job processing (also provided by JES2)
- Dynamic system interchange

### *Dependent Job Control*

Dependent job control (DJC) is a JES3 function that causes JES3 to control job selection based on dependencies among jobs. With JES3 control statements, the user can specify that one set of jobs (predecessor jobs) is to be completed before other jobs (successor jobs). The success or failure of a predecessor job can cause execution, holding, or cancellation of its successor jobs.

### *Device Fencing*

Device fencing involves reserving devices for use only by jobs within a specified job group, or jobs within a specific job network. By reserving devices for certain jobs, the user can improve overall job turnaround although device utilization will be less efficient. Jobs in a DJC network or job class may optionally use devices other than the pool of reserved devices.

### *Priority Aging*

JES3 can increase the priority of a job depending on the number of times the job has been passed over for selection. However, at an installation-specified priority barrier, JES3 attempts to prevent lower priority jobs from using idle resources if the resources are known to be needed by a higher priority job.

### *Deadline Scheduling*

Deadline scheduling allows the user to specify a time of day by which a job should be scheduled. If the job is not scheduled by this time, JES3 increases the job's priority at user-defined intervals until it is scheduled.

### *Network Job Processing*

Network job processing (NJP) permits two or more global processors to schedule and route JES3 jobs from one global processor to another using telecommunication lines. The system programmer must determine which jobs can be sent where, based on data dependencies. By specifically defining the types of jobs (by job class) that can execute at various locations, the system programmer can improve the workload balancing among JES3 global processors.

### *Remote Job Processing*

Remote job processing (RJP) permits a user located many miles from a particular JES3 installation to submit jobs to that installation. The unit record devices at the remote site are logically operated by JES3 as if they were local readers, printers, punches, and consoles. Thus, while operating all local unit record devices, JES3 can simultaneously read jobs from several remote readers into the queue of jobs awaiting processing. It can also send the results of previously entered jobs that have completed execution to several remote printers and punches.

### *Dynamic System Interchange*

Dynamic system interchange (DSI) allows the operator to assign the JES3 global function to a capable, active local processor. The purpose of DSI is to sustain the operation of the JES3 complex when a long-term global system failure occurs. Such a failure might be the result of a hardware failure of the processor, a channel, or a control unit on the global processor.

The local processor assuming the global function must have channel-to-channel paths to all other local and ASP processors in the complex. Any processors for which no path exists cannot be supported by the new global processor

DSI is invoked by the *CALL,DSI operator command on the processor that is to assume the global function.

## Allocation of Devices

Device allocation is the assignment of a resource (I/O device, volume, data set) for use by a specific job step. When a user submits a job for processing, the job consists of statements and any related input data. The JCL statements identify the job (JOB statement), each job step within the job (EXEC statement), and the data sets to be used by the job (DD statements).

When JES selects a job and passes it to an eligible initiator for execution, the job's JCL has already been converted into internal text, which the interpreter uses to build and initialize the SWA control blocks. The parameters on the DD statement provide such control block information as:

- The name of the data set
- The name of the volume on which it resides
- The type of I/O device that holds the data set
- The format of the records in the data set
- Whether the data set is old or new
- The size of newly-created data sets
- The access method that will be used to create or refer to the data

The initiator passes control to the allocation routines which use the control block information to analyze the job's device, volume, and data set requirements.

During step initiation, the allocation routines assign the requested devices, volumes, and data sets to the job step. The initiator does not start the job step's execution until the allocation process is complete. That is, the job step does not receive control until it has all the resources it needs to execute successfully. A similar process occurs when a time-sharing user issues a LOGON command to start a TSO terminal session.

These are the major functions that device allocation performs:

- Locating a requested data set's volume and unit information
- Resolving relationships between two or more requests
- Creating, via data management, new data sets
- Assigning I/O devices to the requests
- Instructing the operator to mount necessary volumes
- Allowing dynamic concatenation and deconcatenation of data sets

These are the major functions that device unallocation performs:

- Directing the processing of a data set's disposition
- Releasing data sets, reserved by an initiator, for use by other job steps
- Releasing I/O devices for use by other job steps

## *Dynamic Allocation*

The allocation performed during job step initiation can be altered prior to job step unallocation (or LOGOFF command to end a TSO terminal session) by invoking dynamic allocation. Because resource requirements may not be fully known prior to execution, dynamic allocation routines enable jobs and time-sharing users to acquire resources as the need develops. Dynamic allocation also allows resources to be used more efficiently because the resources can be acquired just before use and released immediately after use.

A typical use for dynamic allocation is in a program that needs temporary use of a device, volume, or data set for which there is heavy contention. In such a case, dynamic allocation provides the means for a job to tie up the resource for only as long as necessary rather than for the life of the job.

Another common use for dynamic allocation is in a job whose need for allocation resources may vary according to input. Dynamic allocation permits such jobs to dynamically allocate and free only the data sets necessary to process the input, so the specific resources supporting the required data sets can be in use for the minimum time.

For more information on dynamic allocation, see *OS/VS2 System Programming Library: Job Management.*

# Chapter 6: Supervising the Execution of Work

As described in the preceding chapters, work enters the system, is assigned a private address space, and is scheduled for execution. Once the work is brought into real storage (where it has access to the processor), it becomes the responsibility of the **supervisor**.

The supervisor provides the controls needed for multiprogramming. This chapter describes the following functions of the supervisor:

- **Interruption processing**. In order to achieve multiprogramming, some technique must exist to switch control from one routine to another — so that, for example, when routine A must wait for an I/O request to be satisfied, routine B can be executing. In MVS, as in MVT and SVS, this is achieved by **interruptions**, which are events that alter the sequence in which the processor executes instructions. When an interruption occurs, the supervisor receives control, saves the execution status of the interrupted routine, analyzes the interruption, and passes control to the appropriate routine to process the interruption.
- **Creating dispatchable units of work**. The supervisor requires some way of identifying and keeping track of all the work in the system. It does this by representing each unit of work with a control block. Two types of control blocks represent dispatchable units of work in MVS systems: **task control blocks (TCBs)**, which also exist in MVT and SVS systems and which represent tasks executing within an address space; and **service request blocks (SRBs)**, which were introduced in MVS as an efficient way to provide high priority for system services.
- **Dispatching work**. After supervisor routines process interruptions, they either return control to the routine that was interrupted or pass control to a routine called the **dispatcher**. (Which action occurs is described in detail in the topic "The Interruption Handler (IH) Routines.") The dispatcher determines which unit of ready work, of all the ready units of work in the system, has the highest priority and passes control to that unit of work.
- **Serializing the use of resources**. In a multiprogramming system, almost any sequence of instructions can be interrupted, to be resumed later. If that set of instructions manipulates or modifies a resource (for example, a control block or a record in a data set), the supervisor must prevent other programs from using the resource until the interrupted program has completed its processing of the resource.

In MVS, the supervisor provides two techniques for serializing the use of resources: **enqueuing** (via the ENQ or, for shared DASD, RESERVE macro instruction), which is also available in MVT and SVS systems; and **locking** using multiple locks, which was introduced in MVS as an efficient way to serialize the use of resources by supervisor routines and, in a tightly-coupled multiprocessing environment, by processors.

For detailed information on supervisor functions see *System Programming Library: Supervisor* and *Supervisor Services and Macro Instructions*.

## Interruption Processing

An interruption is an event that alters the sequence in which the processor executes instructions. An interruption may be planned (specifically requested by the task the processor is currently executing) or unplanned (caused by an event that may be either related or unrelated to the task currently executing). There are six types of interruptions:

- **SVC (supervisor call) interruptions**, which occur when the program issues an SVC instruction. An SVC is a request for a particular system service — for example, to open a data set (SVC 19 — OPEN), to obtain storage (SVC 4 — GETMAIN), to write a message to the operator (SVC 35 — WTO/WTOR).
- **I/O interruptions**, which occur when a channel or device signals a change of status. For example, an I/O operation completes, an error occurs, or a device becomes ready.
- **External interruptions**, which indicate any of several events **for example**, a time interval expires, the operator presses the interrupt key on the console, or a signal is received from another processor.
- **Restart interruptions**, which occur when the operator presses the restart button on the console or when a restart SIGP (signal processor) instruction is received from another processor.
- **Program interruptions**, which are caused by program errors (for example, the program attempts an invalid operation), page faults program references a page that is not in real storage), or requests to monitor an event.
- **Machine check interruptions**, which are caused by machine malfunctions.

The supervisor includes six routines called **interruption handlers (IHs)** to process the six types of interruptions: an SVC IH, I/O IH, external IH, restart IH, program IH, and machine check IH. When an interruption occurs, the system must save the status of the program that was interrupted and route control to the appropriate interruption handler routine. This is accomplished by means of a hardware feature called program status words (PSWs).

### *The Role of Program Status Words*

**Program status words (PSWs)** are used to control the order in which instructions are executed and to hold and indicate the status of the system in relation to the program currently being executed. There are three types of PSWs: current PSW, new PSWs, and old PSWs.

The **current PSW** indicates the next instruction to be executed. It also indicates whether the processor is **enabled** or **disabled** for I/O interruptions, external interruptions, machine check interruptions, and certain program interruptions. When the processor is enabled, these interruptions can occur. When the processor is disabled, these interruptions are ignored or remain pending, to be processed when the unit of work that is executing disabled completes the processing that requires disablement. (The processor is never disabled for SVC, restart, or certain program interruptions.)

A **new PSW** and an **old PSW** are associated with each of the six types of interruptions. The new PSW contains the address of the interruption handler routine that can process its associated interruption. If the processor is not disabled when an interruption occurs, the System/370 hardware switches PSWs by:

- Storing the current PSW in the old PSW associated with the type of interruption that occurred
- Moving the contents of the new PSW for the type of interruption that occurred into the current PSW

The current PSW, which indicates the next instruction to be executed, now contains the address of the appropriate IH routine to handle the interruption (see figure 6.1); this has the effect of transferring control to the appropriate interruption-handling routine.

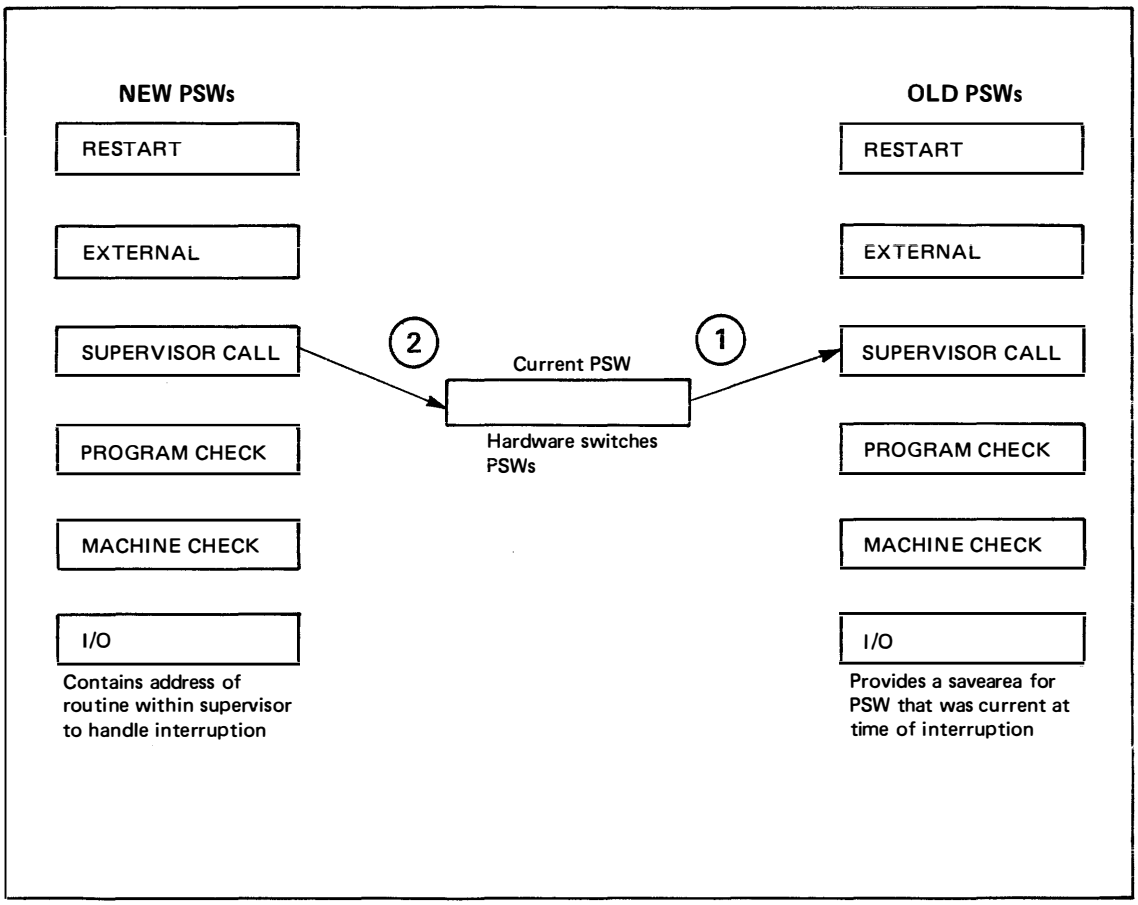



**Figure 6.1. The Use of Program Status Words (PSWs) in Interruption Processing**

### *The Interruption Handler (IH) Routines*

The interruption handler (IH) that receives control saves the status (general registers and the old PSW) of the unit of work that was interrupted, analyzes the interruption, and determines the control program action required. Specifically:

- The **SVC interruption handler** determines the type and location of the requested SVC routine and, if the requested SVC requires that the caller be authorized, checks that the caller has the appropriate authorization. (The request is denied if the caller lacks necessary authorization.) There are several types of SVCs, each type having different execution characteristics. For example, some types of SVCs reside in the nucleus, others in the link pack area; some types can issue other SVCs, other types cannot. If the requested SVC is a type that can issue other SVCs, the SVC IH builds a control block called an SVC request block (SVRB) for the requested routine. The SVRB is needed to save status information about the routine so that it can be resumed after an SVC interruption has been processed. After checking for proper authorization and, if necessary, building an SVRB, the SVC IH passes control to the requested SVC routine.
- The **I/O interruption handler** passes control to the input/output supervisor (IOS). IOS performs all processing for I/O requests and controls all I/O error processing. For more information on IOS, see chapter 8.
- The **external interruption handler** determines the cause of the external interruption and passes control to the appropriate external service routine.
- The **restart interruption handler** routes control to the recovery termination manager (RTM). For more information on RTM, see chapter 9.
- The **machine check interruption handler** records all machine checks and, if the machine check cannot be corrected by hardware, calls the recovery termination manager (RTM) —see chapter 9.
- The **program interruption handler** determines the cause of the program interruption and, depending on the cause, passes control to one of the following:
  - Real storage management (RSM), if the program interruption was caused by a page fault. RSM determines if the page fault is valid and, if it is, starts the processing necessary to bring the referenced page into real storage.
  - Generalized trace facility (GTF), if the interruption occurred as the result of a request to monitor an event. GTF (if it is active) records the event.
  - A user-provided program-interruption exit routine, if the program interruption was caused by an error in user code (for example, using an incorrect address or attempting to execute privileged instructions) and the user provided an error-handling routine (by means of the SPIE —set-program-interruption-element— macro instruction).
  - The recovery termination manager (RTM), if the program interruption was caused by an error in system code or, if the user does not provide his own error-handling routine, in user code.

The routine that receives control after the interruption is processed depends on whether the interrupted unit of work was non-preemptive. A non-preemptive unit of work can be interrupted but must receive control after the interruption is processed. All SRBs are non-preemptive; a TCB is non-preemptive if it is executing a non-preemptive SVC (the installation identifies which SVCs will be non-preemptive during system generation). If the interrupted unit of work was preemptive, the dispatcher receives control and determines which unit of work should be performed next.

Figure 6.2 summarizes the processing of interruptions; for more information on the dispatcher, see the topic "Dispatching Work."

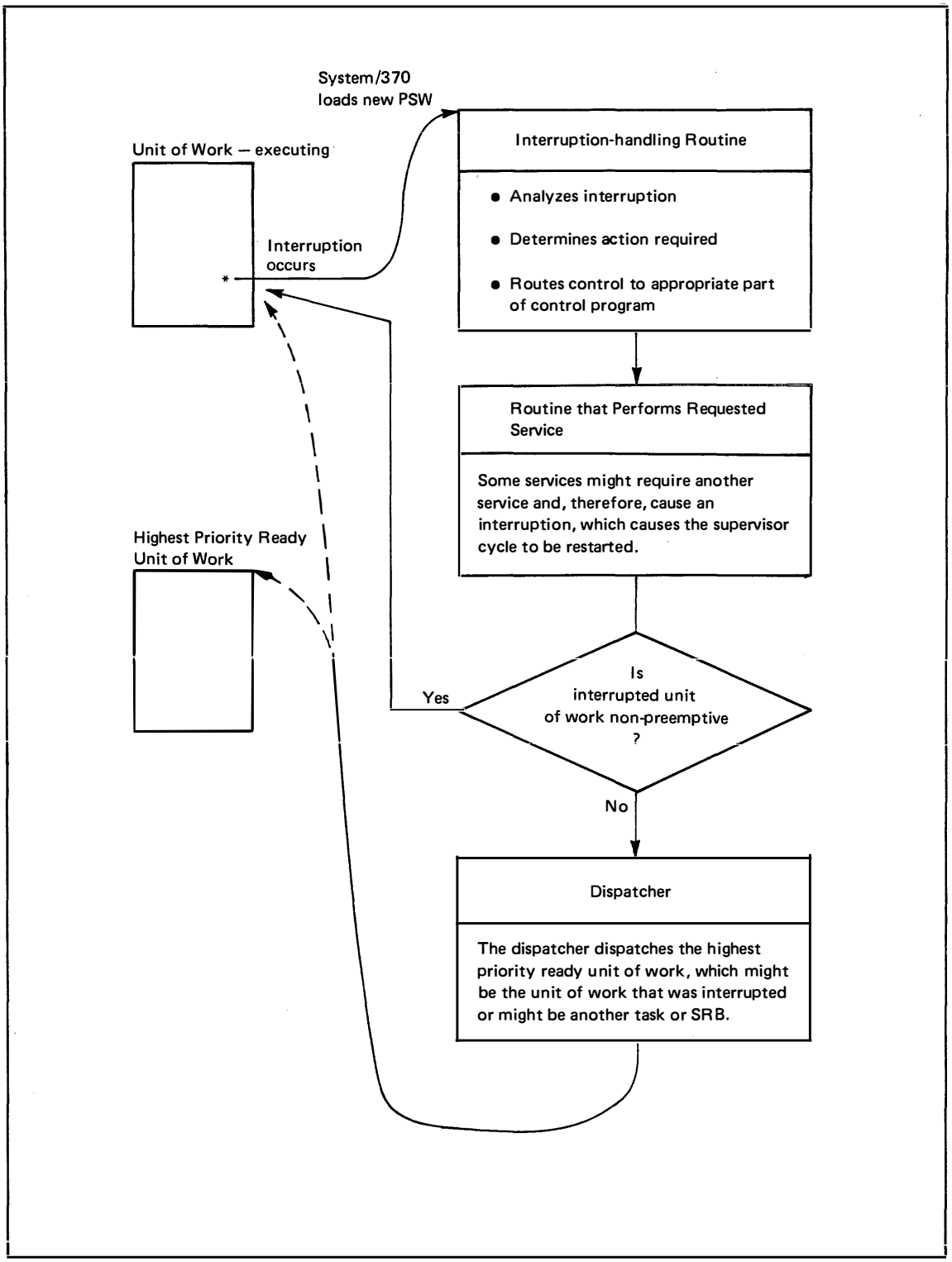



Figure 6.2. Summary of Interruption Processing

## Creating Dispatchable Units of Work

In MVS, dispatchable units of work are represented by two different control blocks:

- Task control blocks (TCBs), which represent tasks executing within an address space —user programs and system programs executed to support the user programs.
- Service request blocks (SRBs), which represent requests to execute a service routine. SRBs are typically created when one address space is executing and an event occurs that affects a different address space; they provide the mechanism for almost all communication between address spaces.

### *Task Control Blocks (TCBs)*

Task control blocks (TCBs) are created in response to an ATTACH macro instruction. By issuing ATTACH, a user or system routine causes the supervisor to begin the execution of the program specified on the ATTACH macro as a subtask of the caller's task. As a subtask, the specified program can compete for processor time and may use certain resources already allocated to the caller's task.

The ATTACH macro instruction causes an SVC interruption. The SVC interruption handler branches to the ATTACH SVC routine to perform the requested service. The ATTACH routine does the following:

- Obtains storage for a new TCB
- Places in the new TCB information needed to control the subtask
- Places the new TCB on the chain of TCBs for that address space
- Branches to program management routines to locate the first program to be executed for the new subtask and, if necessary, fetch the program from a program library.

The region control task (RCT), which is responsible for preparing an address space for swap-in and swap-out, is the highest priority task in an address space. All tasks within an address space are subtasks of the RCT. The RCT's TCB is pointed to from the address space control block extension (ASXB) and points to the next TCB in the address space. Figure 6.3 illustrates the basic TCB structure for batch jobs, operator-started jobs, and TSO users.

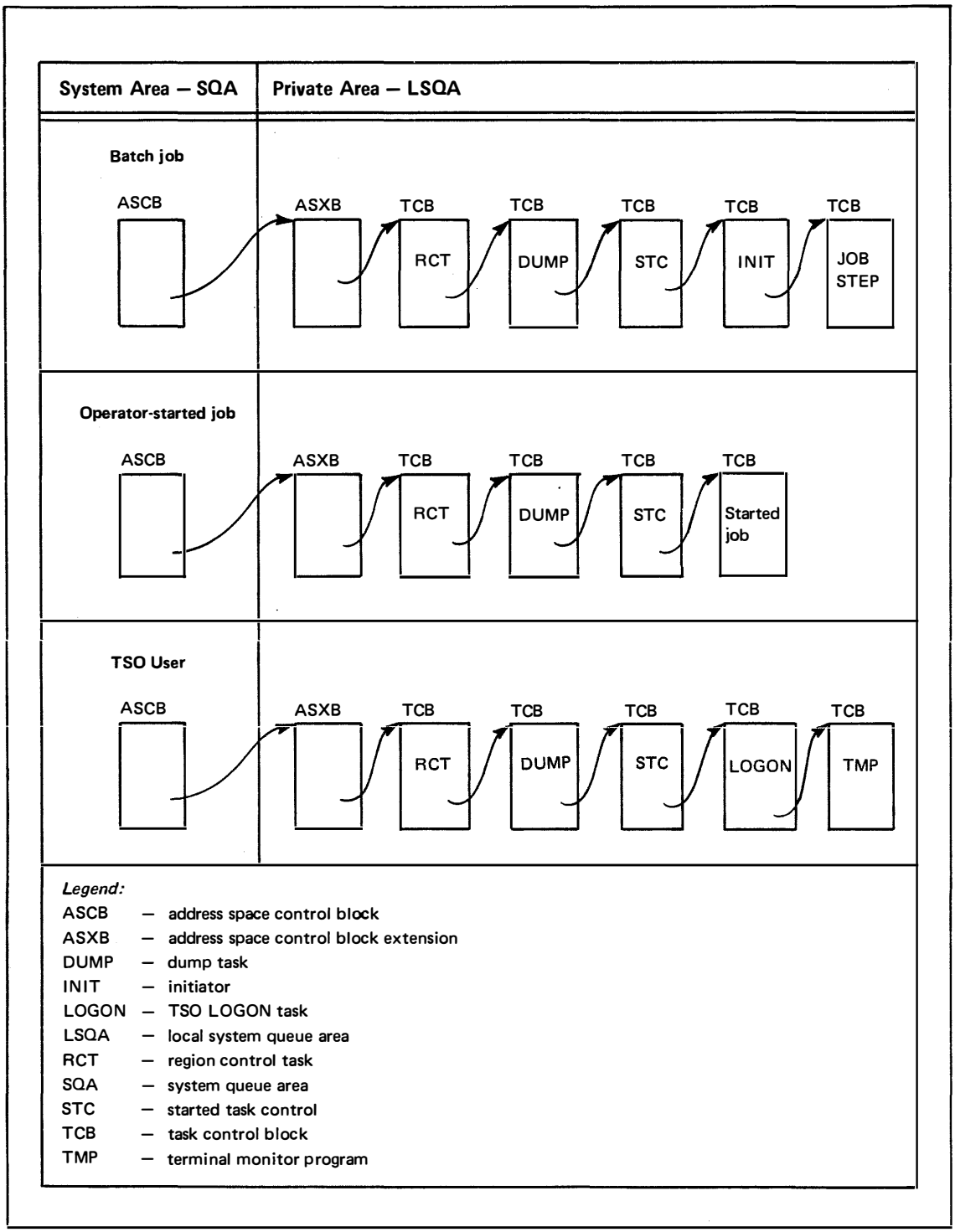



Figure 6.3. Task Control Block (TCB) Structure

### *Service Request Blocks (SRBs)*

Service request blocks (SRBs) are typically created when one address space is executing and an event occurs that affects a different address space. For example, address space A is executing and an I/O interruption occurs because an I/O operation requested by address space B has completed. The I/O interruption handler collects the necessary information about the interruption and builds and schedules a service request block (SRB). The I/O interruption handler can then start I/O requests that were waiting for the I/O path used by the request that just completed and can accept any additional pending interruptions. Delaying complete processing of the interruption by building the SRB allows faster re-use of the I/O path and less disabled interruption time.

The SRB identifies the routine to be executed and the address space in which the routine should be executed. In the preceding example, the SRB would be executed in address space B, because that address space had requested the I/O operation. To schedule an SRB, the routine that builds the SRB issues the SCHEDULE macro instruction. On the SCHEDULE macro instruction, the routine indicates the priority of the request relative to other requests in the system by specifying either GLOBAL or LOCAL. SRBs with a global priority are given a priority higher than that of any address space, regardless of the actual address space in which they will be executed. SRBs with a local priority receive a priority equal to that of the address space in which they will be executed, but higher than that of any TCB within that address space. The assignment of global or local priority depends on the "importance" of the request; for example, SRBs for I/O interruptions are scheduled at a global priority, so that I/O delays are minimized.

## Dispatching Work

Dispatching work consists of routing control to the highest priority unit of work that is ready to execute. The dispatcher, a supervisor routine, dispatches work in the following order:

1. Special exits. These are exits to routines that have a high priority because of specific conditions in the system. For example, if one processor of a tightly-coupled multiprocessing system fails, alternate CPU recovery (ACR) will be invoked by means of a special exit to recover work that was being executed on the failing processor.
2. SRBs that have global priority. If a global SRB cannot be dispatched (for example, the address space in which it will execute is swapped out), the dispatcher reschedules it at a local priority.

3. Ready address spaces in order of priority. An address space is ready to execute if it is swapped in and not waiting for some event to complete; an address space's priority is determined by the dispatching priority specified by the user or the installation. The address space control block (ASCB) contains the address space's dispatching priority; ASCBs that represent ready address spaces are queued in storage according to their dispatching priority. To select an address space, the dispatcher selects the first ready ASCB on the chain of ASCBs.

After selecting the highest-priority ASCB, the dispatcher first dispatches SRBs with a local priority that are scheduled for that address space and then TCBs in that address space.

If there is no ready work in the system, the dispatcher loads an enabled wait PSW.

The dispatcher receives control after a task is interrupted or becomes non-dispatchable, after an SRB completes or is suspended, (that is, an SRB is delayed because a required resource is not available), and from other supervisor routines that want higher priority work dispatched without waiting for an interruption to occur. The dispatcher saves the status of the unit of work relinquishing control, selects a unit of work, builds a program status word (PSW) for the selected unit of work, and issues a load PSW (LPSW) instruction, which results in the selected routine receiving control. That routine executes until an interruption occurs or until the routine voluntarily gives up control (for example, by issuing a WAIT SVC).

## Serializing the Use of Resources

The supervisor provides two techniques for serializing the use of resources: enqueuing, which was available in MVT and SVS systems; and locking using multiple locks, which is a new technique for MVS.

### *Enqueuing*

Enqueuing is accomplished by means of the ENQ (enqueue) and DEQ (dequeue) macro instructions, which can be used by both user and system programs; or, for devices shared between systems, by means of the RESERVE and DEQ macro instructions. On ENQ or RESERVE, a user specifies the name(s) of one or more resources and requests shared or exclusive control of those resources. If the resources are to be modified, the user must request exclusive control; if the resources are not to be modified, the user should request shared control, which allows the resource to be shared by other users that do not require exclusive control. The DEQ macro instruction is used to release control of a resource.

### *Locking*

Locking using multiple locks is a new technique in MVS that serializes the use of system resources by supervisor routines and, in a tightly-coupled multiprocessing system, by processors. A lock is simply a field in storage that indicates if a resource is being used and who is using it. In MVS, there are two kinds of locks: **global locks**, for resources related to more than one address space, and **local locks**, for resources assigned to a particular address space. Global locks are provided for non-reentrant routines and the following control blocks:

- Control blocks the dispatcher uses.
- Control blocks the auxiliary storage manager (ASM) uses.
- Routines of real storage management (RSM) and virtual storage management (VSM) that allocate storage.
- Control blocks and functions of the input/output supervisor (IOS). These include locks for the following: global IOS functions; the channel availability table (used by IOS to allocate a channel to an I/O request); each unit control block (updated by IOS when units are assigned to or released by I/O requests); each logical channel queue (maintained by IOS for requests waiting for a logical channel).
- Control blocks used by VTAM. There is one lock for each of the following types of control blocks: VTAM node control blocks; VTAM destination node controls blocks; VTAM data extent blocks.
- The control algorithms and control blocks the system resources manager (SRM) uses.
- Control blocks that provide cross-memory services that are not protected by any of the preceding locks.

A local lock is provided for each address space to serialize the allocation of storage and the use of control blocks within the address space.

To use a resource protected by a lock, a routine must first request the lock for that resource. A part of the supervisor called the lock manager acquires and maintains all locks. If the lock is unavailable (that is, already held by a different program or processor), the action taken by the program or processor that requested the lock depends on the type of lock; there are two types of locks —spin locks and suspend locks:

- If a **spin lock** is unavailable, the requesting processor continues testing the lock until the other processor releases it. As soon as the lock is released, the requesting processor can obtain the lock and, therefore, control of the protected resource. All of the global locks except the cross-memory-services lock are spin locks.
- If a **suspend lock** is unavailable, the unit of work requesting the lock is delayed until the lock is available; the requesting processor is dispatched to do other work. The cross-memory-services global lock and all local locks are suspend locks.

To prevent deadlocks, MVS locks are arranged in a hierarchy and a processor or routine may unconditionally request only locks higher in the hierarchy than locks it currently holds. For example, a deadlock could occur if processor 1 held lock A and required lock B; and processor 2 held lock B and required lock A. In MVS, this situation cannot occur because locks have to be acquired in hierarchical sequence. Assume, in the preceding example, lock A precedes lock B in the hierarchy. Processor 2, then, cannot hold lock B without already holding lock A; the deadlock cannot occur. Figure 6.4 summarizes the locks provided in MVS and lists them in hierarchical order.

| Class of lock | Name of lock* | Resource protected | Type of lock |
|---|---|---|---|
| Global | DISP | Dispatcher control blocks | |
| | ASM | ASM control blocks | |
| | SALLOC | RSM and VSM routines | |
| | IOSYNCH | Global IOS functions | Spin |
| | IOSCAT | Channel availability table | |
| | IOSUCB | Unit control blocks | |
| | IOSLCH | Logical channel queues | |
| | TPNCB | VTAM node control blocks | |
| | TPDNCB | VTAM destination node control blocks | |
| | TPACBDEB | VTAM data extent blocks | |
| | SRM | SRM algorithms and control blocks | |
| | CMS | Cross memory services | Suspend |
| Local | LOCAL | Address space storage and control blocks | |

*Locks are listed in hierarchical order, from highst. to lowest..

**Figure 6.4. Summary of MVS Locks**

The design of locking in MVS allows supervisor routines to execute and allows one processor in a tightly-coupled multiprocessing system to use one resource while the other processor uses a different resource — two benefits that were not provided by earlier techniques to serialize the use of resources.

# Chapter 7: Managing System Resources

Managing system resources in MVS is the responsibility of a component called the **system resources manager** (SRM). SRM has two objectives:

- To distribute the system's resources (processor time, I/O resources, and real storage) among individual address spaces as specified in the installation performance specification (IPS)
- To achieve the optimal use of processor time, real storage, and I/O resources by active address spaces, as seen from the viewpoint of system throughput

This chapter describes how SRM attempts to meet these objectives: the decisions it makes and the factors it considers in making those decisions. The system programmer can influence almost all of the decisions made by SRM routines by means of the installation performance specification (IPS) an the IEAOPTxx member of the SYS1.PARMLIB data set. The *Initialization and Tuning Guide* contains detailed information on SRM's processing and how the installation can influence it.

***Note:*** Except where noted, this chapter describes SRM as it exists when SU7 (Supervisor Performance #2) has been installed.

## How SRM Meets Its Objectives

SRM's two objectives are contradictory in terms of the availability of resources. Optimizing throughput implies keeping resources busy; meeting the installation's objectives for response and turnaround time (as reflected in the IPS) implies the availability of any resource when it's required. SRM makes decisions that represent trade-offs between its two conflicting objectives.

The decisions SRM makes include the following:

- Which address spaces should be permitted access to the system's resources (that is, swapped in)
- When to steal pages and which pages to steal
- When to change the dispatching priority of address spaces (called "chapping")
- Which device should be allocated, when allocation routines have a choice of devices
- When to inhibit the creation of new address spaces

These decisions are the controls SRM uses to meet its objectives.

## *Major Functional Areas of SRM*

To reach its decisions, SRM is divided into three major functional areas:

- **SRM control**, which determines the processing required by SRM and routes control to the appropriate SRM routines. SRM control decides when and which address spaces will be swapped in or out. To make this decision, it obtains recommendations from the other functional areas of SRM: the workload manager and the resource manager.
- **Workload manager**, which monitors the use of resources by the various address spaces. It gives the SRM control function swapping recommendations that attempt to maintain each address space's use of system resources as specified in the IPS.
- **Resource manager**, which monitors system-wide use of resources to determine if they are over- or under-utilized. It makes swapping recommendations to the SRM control function that are intended to optimize throughput — to optimize use of the processor(s), I/O resources, and storage. In addition, the resource manager is responsible for implementing other SRM controls related to the use of resources: inhibiting the creation of new address spaces or stealing pages when certain shortages of storage exist; changing the dispatching priority of address spaces, which controls the rate at which the address spaces are allowed to consume resources; choosing the device to be allocated if a choice of devices exists, in order to balance the use of I/O resources.

## *Communicating with SRM*

Other system components communicate with SRM by means of the **SYSEVENT macro instruction**. All SYSEVENTs have a code, which indicates the processing SRM is to do. Essentially, all codes fall into one of two categories:

- SYSEVENTs that notify SRM of a change in status for a particular address space or for the system as a whole. For example: real storage has been configured into or out of the system; an address space has been deleted; an initiator selects or terminates a job; a swap-in is started or a swap-out completes. In response to these SYSEVENTs, SRM updates, builds, or releases control blocks that contain information on system and address space activity.
- SYSEVENTs that invoke SRM's decision-making functions. For example: an address space enters a long wait (SRM will swap the address space out of real storage); an address space is to be created (if a shortage of SQA or pageable storage exists, SRM will prohibit creation of the address space); allocation routines have a choice of devices to be allocated to a request (SRM will recommend one of the devices); a time interval expires. The timer-interval SYSEVENT is the exclusive means to invoke most of SRM's algorithms, which provide data on which SRM bases its decisions.

Most SYSEVENTs cause the SRM control function to be called, which in turn can call the resource or workload manager for the processing of various algorithms. The remainder of this chapter describes in greater detail SRM control, the workload manager, and the resource manager.

## SRM Control

SRM control is the dispatcher of SRM. It schedules actions and algorithms to be performed by other SRM routines and is responsible for the swapping of address spaces.

The installation provides guidelines for SRM's swap decisions by defining a **domain** for each distinct type of work (for example, batch work). For each domain, the installation defines a **minimum** and **maximum MPL** (multiprogramming level) and the domain's importance relative to other domains. The definition of each domain's importance is used by resource manager routines, as described in the topic "Resource Monitoring." The MPLs state the minimum and maximum number of address spaces in each domain that should be in real storage (that is, swapped in) at the same time. Within the boundaries of the minimum and maximum MPL and based on factors such as the total utilization of system resources, SRM periodically computes an optimal MPL for each domain, called the **target MPL.** The objective of the swap analysis performed by SRM control is to maintain the MPL of each domain at its target value.

### *Swap Analysis*

Swap analysis is triggered by several events —for example, a user becomes ready to execute or a time interval expires. The swap analysis must answer two questions: **whether** a swap is necessary; and, if so, **which** address space(s) to swap.

To determine **whether** a swap is necessary, SRM control goes through the following steps:

1. SRM control examines each domain, to locate any domain(s) whose current MPL exceeds its target MPL. SRM control swaps out the required number of address spaces to lower the domain's MPL to its target value.
2. If a user is swapped out and enqueued on a resource requested by another user, SRM control swaps in the enqueued user.
3. SRM control examines each domain, to locate any domain(s) whose current MPL is less than its target MPL. SRM control swaps in the required number of users to raise the current MPL to its target value.
4. If a domain's MPL equals its target value, SRM control analyzes swapped-in users and swapped-out users to determine if an exchange swap should occur (that is, a swapped-in user and swapped-out user change places).

Each time swap analysis is called, SRM control proceeds with the preceding steps until it reaches the end of a step that has resulted in at least one swap or it determines no swap is required.

To determine **which** address space(s) within a domain to swap in or out, SRM control asks the workload manager and resource manager for swap recommendations, which take the form of **swap recommendation values (RVs).** The workload manager's RVs aim to maintain an address space's

use of resources as specified in the IPS. The resource manager's RVs aim to correct imbalances in I/O or processor utilization. By combining the RVs of the workload manager and resource manager, SRM control makes trade-offs between its two objectives: distributing resources as specified in the IPS and optimizing throughput.

## The Workload Manager

The workload manager has three basic functions:

- To monitor service rates —the rates at which system resources are being provided to individual address spaces
- To provide swapping recommendations requested by SRM control
- To collect data for certain measurement tools —for example, the system activity measurement facility (MF/1) or the Resource Measurement Facility (RMF), Program Product #5740-XXH

The workload manager measures the rate at which resources are used in terms of **service units** per second. Service units are computed as a combination of three basic system resources: processor time used, I/O activity (EXCP counts for data sets associated with the address space), and real storage frames occupied. Service rate, then, is the result of dividing the number of service units by a time interval, which includes both the time an address space is swapped into real storage and the time it is swapped out but otherwise ready to execute.

To arrive at a swapping recommendation, the workload manager measures the service rates of different address spaces and compares them in light of factors defined by the installation in the IPS (installation performance specification). By means of these factors, the installation can instruct SRM to give certain users better service at the expense of other users. For example, assume two address spaces exist in real storage and one must be swapped out; the installation-defined IPS factors will dictate how the workload manager views measured service rates:

- Address space A has a higher service rate than address space B. Based on IPS factors associated with these two address spaces, the workload manager determines that address space B should be swapped out. (A different IPS could result in the opposite decision — that address space A should be swapped out.)
- Address space A has a lower service rate than address space B. The IPS indicates that address space A is more important and, based on the IPS, the workload manager determines that address space B should be swapped out.
- Address space A and address space B have identical service rates. Again, IPS factors indicate which address space is more important and which, therefore, should remain in storage.

The IPS factors that dictate the workload manager's swap recommendations are described in detail in the *Initialization and Tuning Guide*. The workload manager passes its swap recommendations to SRM control, which combines them with recommendations from the resource manager.

## The Resource Manager

The resource manager includes algorithms that are concerned with improving the system-wide use of resources (as contrasted to an individual address space's use of resources, which is the concern of the workload manager). The resource manager's routines can be divided into four functional areas:

- Storage management, which is concerned with SRM's decisions to steal pages and to prevent the creation of new address spaces
- I/O management, which is concerned with SRM's swap decisions and device allocation decisions
- Processor management, which is concerned with SRM's swap decisions and decisions to change an address space's dispatching priority
- Resource monitoring, which is concerned with adjusting the target MPLs of individual domains based on the need to raise or lower the system-wide multiprogramming level

### *Storage Management*

Storage management routines of SRM take action when shortages of the following are detected: available frames in real storage; space in the system queue area (SQA); slots on auxiliary storage; and pageable frames in real storage.

The system maintains an **available frame queue**, which indicates the number of available frames in real storage. When the number of available frames falls below a "LOW" threshold, SRM storage management routines begin to steal the least-recently used pages from the working sets of address spaces in real storage. The storage management routines continue stealing pages until the count of available frames plus the number of pages stolen exceeds an "OK" threshold for the available frame queue.

SQA shortages are detected by the virtual storage manager (VSM), which calls SRM's storage management routines when a shortage is detected. The storage management routines prevent the creation of new address spaces until the shortage is relieved. The routines also write messages to the operator when the shortage is detected and when the shortage is relieved.

SRM's storage management routines periodically check that the number of available auxiliary storage slots has not fallen below a certain limit. Shortages of pageable real storage are detected by real storage management (RSM) when the percentage of fixed frames to total frames exceeds a certain limit; RSM then notifies SRM's storage management routines. The action taken by SRM for shortages of auxiliary storage slots or pageable real storage is the same; SRM:

- Prevents the creation of new address spaces
- Delays newly-initiated jobs
- Sets the multiprogramming level in each domain to its minimum MPL
- Swaps out the user who is acquiring slots at the greatest rate (for shortages of auxiliary storage) or the user who has the most fixed frames (for shortages of real storage)
- Notifies the operator of the shortage and the identity of the swapped-out user

When the shortage is relieved, creation of address spaces is again allowed, the operator is notified, and address spaces that were swapped out are again made eligible for swap-in.

### *I/O Management*

SRM's I/O management routines are called to:

- Choose a device when allocation routines have a choice of devices to allocate
- Give swap recommendations to SRM control

In both cases, the objective of I/O management is to balance I/O activity across logical channels. When choosing a device for allocation, I/O management seeks candidates on the logical channel that has the lowest utilization; for direct access devices, it then chooses the device with the lowest number of allocated data sets. When giving swap recommendations to SRM control, I/O management bases its recommendations on the extent to which the swap-in or swap-out of a user would correct a detected I/O imbalance: it recommends, via swap recommendation values, that a significant user of an over-utilized logical channel be swapped out; or that a significant user of an under-utilized logical channel be swapped in.

### *Processor Management*

Processor management routines have three responsibilities:

- Controlling the APG (automatic priority group) subset of dispatching priorities
- Preventing the swap-out of users who are enqueued on resources required by other users
- Making swap recommendations to correct under- or over-utilization of the processor

The **APG** is a range of dispatching priorities under the control of SRM. Dispatching priority controls the rate at which address spaces are allowed to consume resources after they have been given access to those resources. By placing jobs in the APG range, the installation, via the IPS and SRM, can alter the dispatching priorities of address spaces as their execution characteristics change.

The APG is divided into three groups: the mean-time-to-wait (MTTW) group, rotate priority, and fixed priorities. (If MVS System Extensions, Program Product #5740-XE1, is installed, the installation can define more than one MTTW group and more than one rotate priority.)

- The **MTTW group** can be used to increase system throughput by increasing processor and I/O overlap (that is, the processor is not waiting while I/O requests are satisfied). Users in the MTTW group have a dispatching priority based on the user's mean execution time before entering a wait state; users who quickly release the processor receive a high priority within the MTTW group.
- The **rotate priority** can be used to ensure that one address space does not dominate the processor in relation to other address spaces also assigned the rotate priority. Processor management routines periodically reposition the address space that is highest in the rotate priority group to the bottom of the group.
- SRM does not change **fixed priorities**; they are available so that the installation can associate, via the IPS, a different fixed priority with different periods in the life of a job or transaction.

By means of the APG, the installation can give SRM control even over nonswappable address spaces.

For users enqueued on resources in demand by other users, processor management routines prevent their swap-out until they have released the resource or executed for a fixed period of time (whichever occurs first). The installation can specify the execution time interval via an SRM tuning parameter.

If processor management routines determine that the processor is over- or under-utilized, they will search for heavy processor users and calculate swap recommendation values for swap-out (to correct over-utilization) or swap-in (to correct under-utilization). A heavy processor user is one that meets or exceeds a certain mean execution time before entering the wait state. The processor is considered over-utilized if, during the period under consideration, it did not enter the wait state and any ready address space on the dispatching queue was not dispatched. The processor is considered under-utilized when its utilization is less than a certain percentage. Processor management routines take into account the extent to which the processor is over- or under-utilized when computing swap recommendation values for SRM control.

## *Resource Monitoring*

The resource monitoring function of the resource manager periodically checks several system resource usage indicators, such as length of the ASM queue, which indicates paging and swapping requests not yet satisfied, and processor utilization. If measured resource usage (averaged over a number of sample intervals) is greater than a "high" threshold or less than a "low" threshold for that indicator, the resource monitoring function recommends that the system-wide multiprogramming level (MPL) be lowered or raised. (The system-wide MPL is the total number of address spaces in the system that are swapped in.)

If the system-wide MPL is to be raised or lowered, resource monitoring routines then identify the individual domain whose MPL will be raised or lowered to achieve the recommended system-wide MPL. The domain selected for MPL adjustment depends on the relative importance of the domains, as defined by the installation in the IPS.

# Chapter 8: Satisfying I/O Requests and Data Management

An input/output operation — I/O — involves the movement of data between main storage and a data set on an I/O device, such as a tape, disk, card reader, or printer. In comparison to the time the processor requires to execute a series of instructions, an I/O operation is very lengthy.

Under MVS, the processor initiates the I/O operation by signalling a channel. The channel, a link between the processor and the device, then executes independently of the processor, thus allowing an overlap of the I/O operation with processor activity. Overlap is one of the key techniques for achieving efficiency in handling I/O operations.

Data moves between main storage and a device along a path; the logical path for data consists of main storage, a channel, and a device. MVS allows the definition of multiple logical paths to a single device, thus giving more flexibility in scheduling I/O requests to balance the load over physical channels and devices.

Under MVS, where many jobs execute concurrently and efficient system operation requires overlap between I/O operations and processor activity, both the information the system must have to perform an I/O operation and the decisions it must make to balance its resources are complex. However, MVS provides a number of services and facilities that make the complexity of an I/O operation largely transparent to the user. One of these services is the access method.

## Access Method

An access method is a data management routine that a user program selects based on the organization of the data set and the access technique used to process the records in the data set. The access method moves data between main storage and an I/O device in response to macro instructions issued by the user program.

### *Data Set Organization*

A data set is a collection of related records that are associated with a particular device. If the device is a tape or a disk, the data set occupies a specific area on a volume mounted on the device. A data set can be organized in four ways:

- **Sequential.** Records are stored and retrieved according to their physical position in the data set.
- **Indexed sequential.** Records are arranged in sequence according to a key. An index or set of indexes maintained by the access method gives access to the records.

- **Direct.** The records in the data set, which must be on a direct access device, can be organized in any way that meets the user's needs. Records are stored and retrieved according to the address of each record within the data set.
- **Partitioned.** The data set, which must be on a direct access volume, consists of members. A member is an independent group of sequentially-organized records that is accessed through its name in the directory of the data set. Partitioned data sets are generally used to store programs and are often referred to as libraries.

### *Access Techniques*

The records in a data set can be accessed by two techniques: the queued access technique and the basic access technique. Some data sets can be accessed by either technique.

With queued access, GET and PUT macro instructions are used to transfer data. The queued technique assumes that the records are to be accessed sequentially. The access method automatically blocks and deblocks the records and, on input, anticipates I/O requests so that the record is generally available before the request is actually made. After a request, control (that is, the ability to execute, to use the resources of the system) does not return to the user program until the requested operation has completed.

With basic access, READ and WRITE macro instructions are used to transfer data. The basic technique is used for direct access of any of the records in the data set. Therefore, the access method does not block or deblock records and does not perform I/O operations in advance of the request. The user program must test for the completion of the I/O operation.

### *Access Method Types*

MVS provides an access method, the virtual sequential access method (VSAM), that is specifically designed to run in virtual storage; it is described under "Virtual Sequential Access Method (VSAM)" later in this chapter. MVS also supports the following access methods:

- **Basic sequential access method (BSAM).** Records in a data set processed by BSAM are sequentially organized and are stored and retrieved in physical blocks. The READ and WRITE macro instructions are used to initiate I/O operations. The user program tests for completion of the operation and performs any required blocking or deblocking.
- **Queued sequential access method (QSAM).** Records in a data set processed by QSAM are stored and retrieved as logical records; QSAM handles any physical blocking or deblocking required. On input, QSAM anticipates the need for a record based on its physical order; normally, the desired record is in storage, ready for use, before the request for it is made. On output, QSAM holds the logical records in a buffer and performs physical output only when the buffer is filled.

- **Basic direct access method (BDAM).** Records in a data set processed by BDAM can be organized in any manner chosen by the programmer. The data set must reside on a direct access volume. Records are stored and retrieved by actual or relative addresses within the data set.
- **Indexed sequential access method (ISAM).** Records in a data set processed by ISAM are arranged in sequential order according to the contents of a key. ISAM maintains an index structure that is used to locate a particular record. Access to the records can be either sequential (QISAM) or direct (BISAM).
- **Basic partitioned access method (BPAM).** A data set processed by BPAM consists of a number of members and a directory that holds the name and location of each member. A member contains a group of records that are organized sequentially. BPAM maintains and accesses the directory; once BPAM locates the desired member, the records within the member are processed by BSAM.

A user program can also request I/O operations without using a specific access method by issuing the execute channel program (EXCP or EXCPVR) macro instruction.

To request an I/O operation, either the access method or the user program presents information about the operation to the components of the MVS system control program that manage the actual physical I/O operation. These components are the EXCP driver and the I/O supervisor (IOS). How the EXCP driver and IOS handle the I/O operation and how their functions and responsibilities fit together with those of the user program and the access method are described under "Scheduling I/O" later in this chapter.

As a means of improving system performance by eliminating much of the overhead and time required to allocate a device and move data physically between main storage and an I/O device, MVS provides virtual input/output (VIO). VIO can be used only for temporary data sets; it uses the system paging routines to transfer data into and out of a page data set and attempts to keep as much data as possible in real storage. "Virtual Input/Output (VIO)" later in this chapter describes how the system intercepts a VIO request and branches to VIO.

## Scheduling I/O

To satisfy an I/O request, the user program, with or without an access method, describes the operation required, and the system components perform the operation, handle the interruption that signals the completion of the operation, and post its status.

Figure 8.1 shows the major steps required to perform an I/O operation. The figure summarizes the responsibilities and functions of the user program, the access method, and the system components; the circled numbers show the chronological sequence of events. The figure assumes the use of an access method and that the user is executing in a virtual region. When a program does not use an access method, or when it executes in a real region, the process differs slightly from the one shown in the figure. However, the I/O services provided by MVS can handle these special cases.

The following text explains the standard operation in more detail and describes the actions taken to handle special cases, such as the user who must get control during the execution of an I/O operation.

### *User Program Functions*

The user program that issues the I/O request must describe the data set to be used and the specific operation to be performed on the data set. To describe the data set to the system, the user program creates a data control block (DCB) and issues an OPEN macro instruction.

### OPEN Processing

When the user program issues an OPEN macro instruction, it invokes the system OPEN routines. These routines merge information from various sources to build a complete description of the data set. The information used comes from:

- The job file control block (JFCB) and a task I/O table (TIOT) entry built from information in the DD statement included in the JCL for the user program. After the device for the data set has been allocated, the TIOT entry points to the unit control block (UCB) for the required device.
- The data set control block (DSCB) that describes the data set. For data sets on a direct access device, for example, the DSCB comes from the volume table of contents (VTOC) for the volume containing the data set.
- The data control block (DCB) the user program builds. The DCB includes a great deal of information, one piece of which is the access method that the user program needs to perform I/O operations on the data set. Other information might include how the data set is organized and how its records are to be accessed.

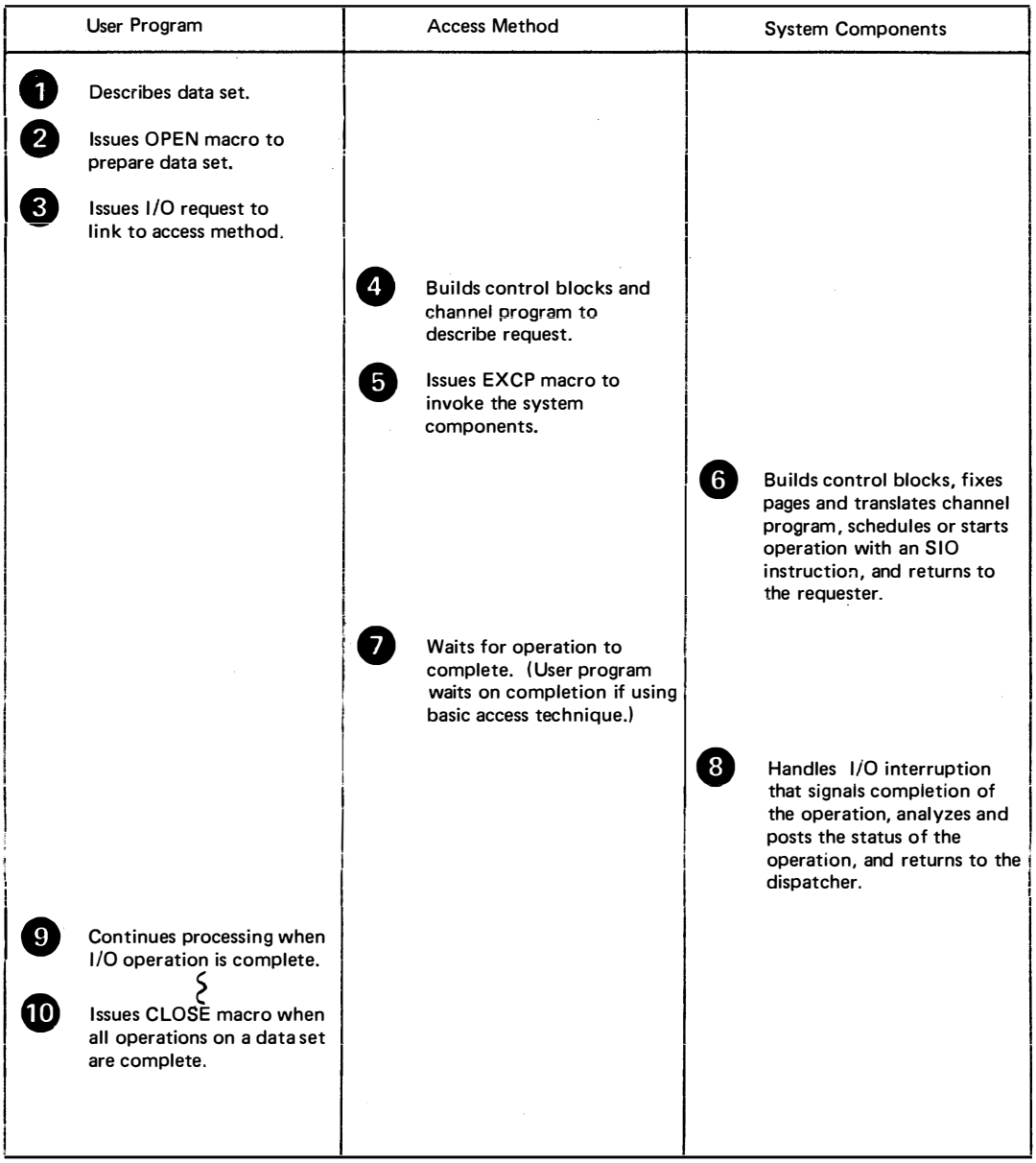



**Figure 8.1. Major Steps in a Standard I/O Operation**

The OPEN routines can acquire the information they need from any of these sources, giving the user a great deal of flexibility in specifying I/O operations. To achieve device independence, for example, a user can specify a minimal amount of DCB information in the program and supply the rest of the information on the JCL for a particular execution of his program.

The OPEN routines build a data extent block (DEB), which specifies the device on which the volume is mounted and the physical extent of the data set on that volume. OPEN processing also places addresses in the DCB that provide linkage between the user program and the access method. If the user program needs access method appendages or user exits to perform such functions as analyzing data errors or processing end-of-data conditions, linkage between the user program and the required routines is also built into the DCB. Figure 8.2 summarizes the relationships the OPEN routines establish between the control blocks and between the user program and the access method.

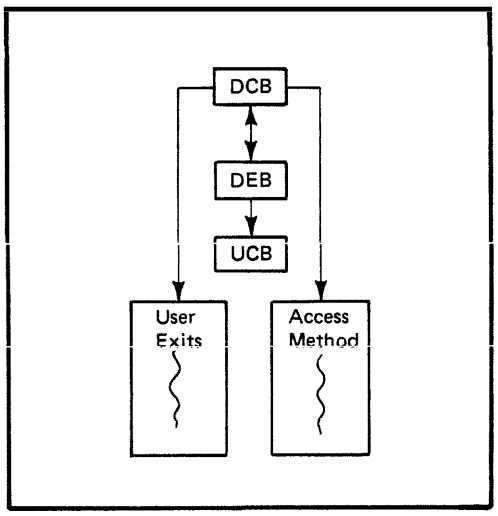



**Figure 8.2. Relationships Established by OPEN**

Once the data set to be used for the operation is successfully opened, it is ready to be used. The user program can then issue an I/O request.

## I/O Request

To transfer data between a data area in storage and an I/O device using an access method, the user program issues a macro instruction. GET and PUT are used for queued input and output requests; the access method does not return control to the user program until the I/O operation is complete. READ and WRITE are used for basic input and output requests; control returns to the user program once the I/O operation is initiated, and the user program must test for the completion of the operation.

Either type of request causes a branch to the access method. The access method routines reside in PLPA, but, as shown in Figure 8.3, both the user program and the access method run in the user's address space.

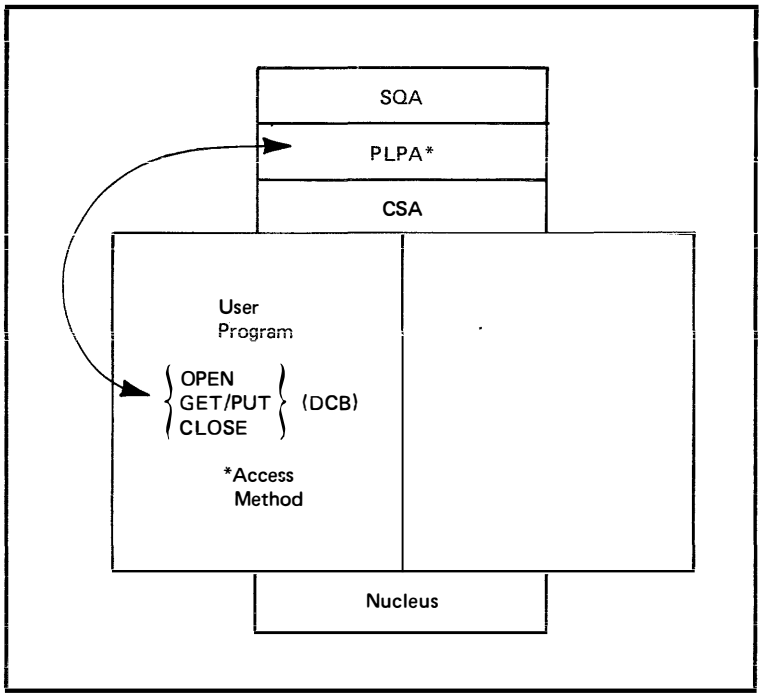



**Figure 8.3. Access Method and User Program in an Address Space**

If the access method cannot satisfy the request because of a specification error in the request, the access method immediately returns control to the user with indicators set to describe the nature of the error. If the request was made correctly, processing of the I/O operation continues as described later in this chapter under "Access Method Functions."

A user program can also issue an I/O request with an EXCP or EXCPVR macro instruction to invoke the EXCP driver directly. See "EXCP Driver Front End" later in this chapter for more information.

When the user program has made all its requests for work to be done on a data set, it must free the data set by issuing a CLOSE macro instruction.

## CLOSE Processing

Issuing a CLOSE macro instruction causes the user program to invoke the system CLOSE routines. The CLOSE routines modify the DCB to break the logical connections between control blocks and between the user program and the access method; these connections were established when the data set was opened. The CLOSE routines free any storage acquired by the OPEN routines.

These routines also rewrite the DSCB for the data set to the volume. Because the DSCB can be modified during OPEN processing, a user program can change the specifications for the data set by opening and closing it.

Figure 8.4 summarizes the control blocks used as input to the CLOSE routines, the functions the CLOSE routines perform, and the modified control blocks that are created during CLOSE processing.

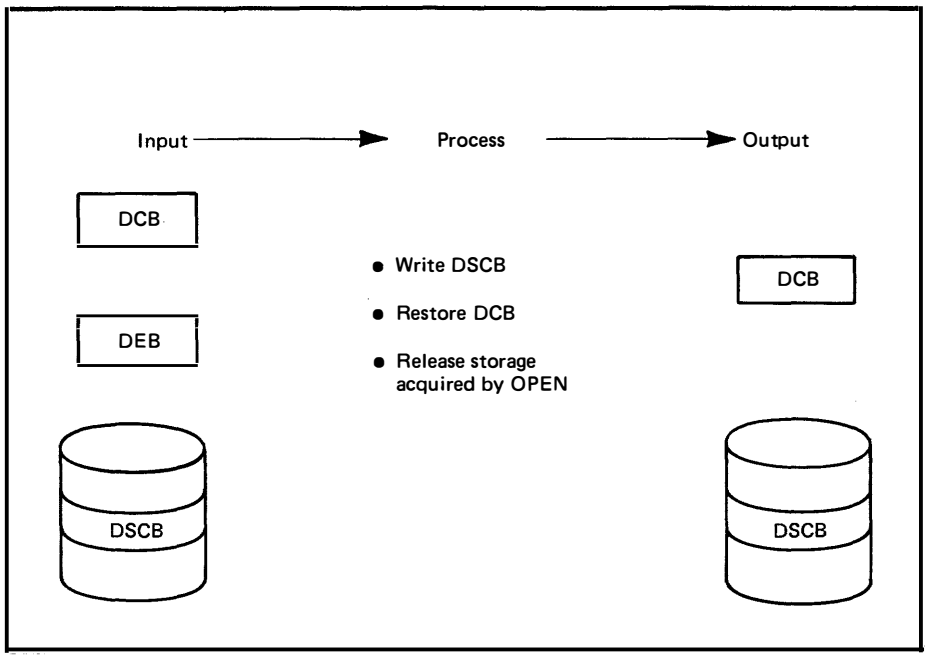



**Figure 8.4. CLOSE Processing Summary**

## *Access Method Functions*

Because the OPEN routines place the address of the required access method in the DCB for the data set, the access method gets control when the user program issues an I/O macro instruction. The access method uses the control block structure built by the OPEN routines to build control blocks for the EXCP driver and a channel program for the I/O request. The access method then issues an EXCP macro instruction to pass control to the EXCP driver.

### Control Blocks

The access method builds two control blocks: the input/output block (IOB) and the event control block (ECB). The IOB points to the DCB; through the DCB, the EXCP driver can access the contents of the DEB and the UCB. The IOB also points to the ECB and to the channel program. The IOB thus contains pointers to all of the information IOS needs about the I/O request.

The ECB is logically empty when it is built; it is used when the operation is complete to post the status of the operation. The access method or the user program can thus test the contents of the ECB to find out when the I/O operation is finished.

## Channel Program

The access method builds a channel program for the I/O operation. A channel program consists of a string of channel command words (CCWs) that describe the operation to the channel. Channel command words provide the channel with all of the information that it needs to perform the operation, such as the address of the data area and the number of bytes of data to be transferred.

## EXCP Macro Instruction

When the IOB and ECB have been built and initialized and the channel program has been created, the access method issues an EXCP macro instruction. The EXCP macro instruction causes an SVC interruption to occur. As a result of this interruption, the SVC interruption handler causes control to be passed to the EXCP driver and then to IOS to schedule and execute the physical I/O operation.

Figure 8.5 summarizes the control block structure and the channel program built by the access method and the pointers it sets before causing control to pass to the EXCP driver.

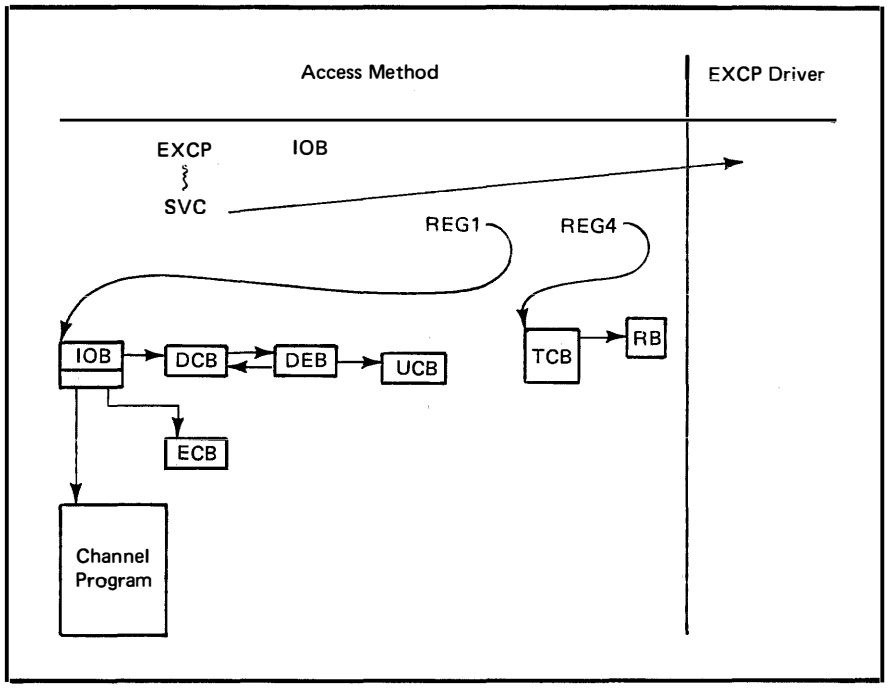



**Figure 8.5. Control Block Structure for the EXCP Driver**

When the EXCP driver and IOS have completed or scheduled the operation, control returns to the access method. If the request used a GET or PUT macro instruction (queued access technique), the access method issues a WAIT against the ECB for the operation. In this case, the access method waits until the ECB is posted complete, and then it returns control to the user program. If the request used a READ or WRITE macro instruction (basic access technique), the access method returns control to the user program, which issues the WAIT macro instruction against the ECB and waits until the request is completed.

### Appendages

Appendages are routines that enable a user to get control at various points during the execution of an I/O operation. Some are entered before execution of the I/O operation, others after execution, and one, the PCI appendage, enables a user to get control during execution to modify the channel program while it is executing.

To establish these exits, authorized routines from authorized libraries identified during system generation can be loaded during OPEN processing for authorized users. The DEB contains the pointers to the appendage routines.

## *Input/Output Supervisor (IOS) Functions*

The MVS input/output supervisor (IOS) has been rewritten and restructured to:

- Support multiprocessing
- Increase system responsiveness
- Make effective use of virtual storage
- Use the MVS recovery capabilities

To maintain compatibility and achieve the improved function described in the preceding list, new interfaces to IOS were created. These interfaces are the IOS drivers. Because the standard access methods use the EXCP driver as an interface to IOS, the balance of this description is concerned only with the relationship between IOS and the EXCP driver. As this relationship is explained, you will see that the EXCP driver is tailored to meet the needs of its intended users.

Figure 8.6 shows some of the drivers that were developed to meet the needs of various IOS users.

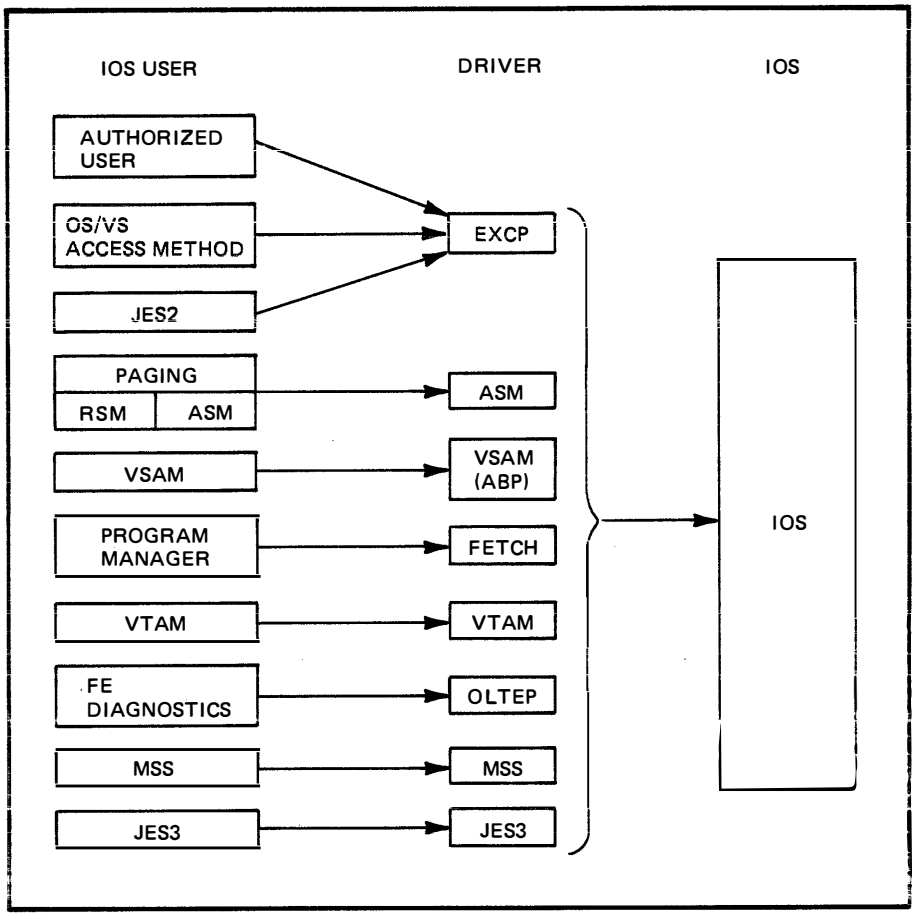



**Figure 8.6. IOS Drivers**

The EXCP driver has three major parts: the front end, the disabled interruption exit (DIE), and the back end. These parts function in response to the needs of the I/O request to interact with the three major parts of IOS: the channel scheduler, the I/O interruption handler, and the post status routines. The driver is separate from IOS, acting primarily as an interface between the I/O requestor and IOS. However, the following description of the functions of the driver and IOS is presented in chronological order to show the steps involved in satisfying a single I/O request.

## EXCP Driver Front End

The front end of the EXCP driver gets control from the SVC interruption handler when an I/O requestor issues an EXCP or EXCPVR macro instruction. The EXCP macro instruction is used by the standard access methods and most user programs. The EXCPVR macro instruction is used by programs that have special I/O needs, such as a program that must dynamically modify a channel program.

Most user programs and the standard access methods run with virtual addresses. Thus, user data areas, control blocks, and the channel programs built by the standard access methods are in virtual storage, use virtual addresses, and are pageable. However, the System/370 channels transfer data into and out of real storage locations. Therefore, the data areas, the control blocks, and the channel program for the I/O operation must be fixed and use real addresses.

The front end of the EXCP driver performs the address translation and page fixing required by the user running in a virtual (V=V) region. Such users invoke the driver with an EXCP macro instruction.

However, users that run in a real (V=R) region do not require address translation or page fixing. The EXCP driver recognizes a V=R user and bypasses the address translation and page fixing functions.

Users who invoke the driver with an EXCPVR macro instruction must construct their own channel programs and build a list of pages to be fixed by the EXCP driver.

Thus, a user who needs to dynamically modify his channel program must either run V=R or use the EXCPVR macro instruction to invoke the driver. Note that the disabled interruption exit (DIE) of the EXCP driver can be invoked only by a user who runs in a V=R region or issues the EXCPVR macro instruction.

Whether or not address translation and page fixing are performed, the EXCP driver front end processing constructs the control blocks IOS requires and branches to the IOS channel scheduler.

The EXCP driver front end gets control again when the channel scheduler has initiated or scheduled the requested I/O operation. At that point, the front end returns control to the access method or user program that issued the EXCP or EXCPVR macro instruction.

### Channel Scheduler

The IOS channel scheduler gets control from the EXCP driver. The channel scheduler initiates the physical I/O operation by attempting to establish a path from the processor through a channel to a device.

If no path is available because the device, the control unit, or the physical channel is busy, the channel scheduler queues the request. To queue a request, the channel scheduler places it on a logical channel queue where it waits until the required path becomes available. (MVS allows the definition of multiple logical paths to a single device, thus giving more flexibility in scheduling I/O requests to balance the load over physical channels and devices.)

If a path is available, the channel scheduler initiates the I/O operation by issuing a start I/O (SIO) instruction to the channel. Before issuing the SIO instruction, the channel scheduler places the address of the channel program in the channel address word (CAW) in a fixed real storage location. When the SIO instruction is issued, the channel fetches and loads the CAW and uses its contents to locate the channel program, which it then proceeds to execute without requiring further intervention from the processor.

After queuing or initiating the I/O operation, the channel scheduler returns control to the front end of the EXCP driver.

During the course of system execution, the channel scheduler is also invoked by the I/O interruption handler each time an I/O interruption occurs, which usually signals the completion of an I/O operation. When the channel scheduler is invoked by the I/O interruption handler, it searches the logical channel queues for an operation that was queued but not initiated because a path was not available. If an operation is waiting for a path that is now available, the channel scheduler issues an SIO instruction to initiate the operation before returning to the I/O interruption handler. Control then passes to either the interrupted program or the dispatcher.

## I/O Interruption Handler

When the physical I/O operation completes, the channel sends an I/O interruption to the processor. The status of the operation is stored in a fixed real storage location called the channel status word (CSW). The hardware then passes control to the I/O interruption handler in the supervisor, called the first-level interruption handler. This routine passes control to the interruption handler in IOS, the second-level interruption handler.

If the I/O request was initiated from a V=R region or by means of an EXCPVR macro instruction **and** if the interruption was a program controlled interruption (PCI), control also passes to the disabled interruption exit (DIE) of the EXCP driver.

After analyzing the status information about the operation and, if required, taking the disabled interruption exit, the second-level I/O interruption handler schedules execution of the IOS post status routines and passes control to the channel scheduler so that any scheduled I/O operations can be initiated.

## EXCP Driver Disabled Interruption Exit (DIE)

The disabled interruption exit (DIE) of the EXCP driver is entered only when the I/O interruption that occurred was a program controlled interruption (PCI) and the user is either running in a V=R region or has issued the EXCPVR macro instruction.

In each CCW in a channel program, there is a PCI bit. When the PCI bit is on, an I/O interruption occurs when the CCW is loaded into the channel. Setting the PCI bit on, which indicates that the user might want to modify his channel program while it is executing, causes control to pass to the DIE.

When the DIE gets control, the processor is in supervisor state and disabled for I/O interruptions. For the DIE to function, the address of a valid PCI appendage must have been placed in the DEB during OPEN processing. The PCI appendage and the DIE make it possible for an authorized user to get control during the execution of the I/O request.

After the user program has processed the PCI, it returns control to the DIE. The DIE then returns control to the second-level I/O interruption handler.

### Post Status

The I/O interruption handler schedules an SRB to invoke IOS post status. When post status is dispatched, it passes control to the EXCP driver back end, which handles any appendages requested by the user, and returns control to the post status routine.

Post status then analyzes the status indicators from the completed operation and returns to the back end of the EXCP driver. If an error has occurred, post status passes control to an error recovery procedure (ERP) before returning to the back end of the EXCP driver. After the back end of the EXCP driver completes its processing and returns control, post status returns to the dispatcher.

### EXCP Driver Back End

The back end of the EXCP driver receives control after IOS has analyzed the status of the event. The back end exits to any access method appendages that are to receive control after the execution of an I/O request. Upon return from any appendages, the EXCP driver back end issues a POST macro instruction to post the status of the completed operation in the ECB and returns control to the post status routine.

The access method or user program that is waiting for the ECB to be posted then becomes ready for execution and is eventually dispatched. Control returns to the user program or access method at the instruction immediately following the WAIT for the completion of the I/O request.

## *Summary*

The preceding explanation described the part each component of the EXCP driver and IOS performs in satisfying an I/O request made by a user program directly or by an access method on behalf of a user program. Figure 8.7 presents an overview of the interaction between the user program, the access method, the EXCP driver, and IOS, showing the flow of a single operation and the means of passing control from step to step.

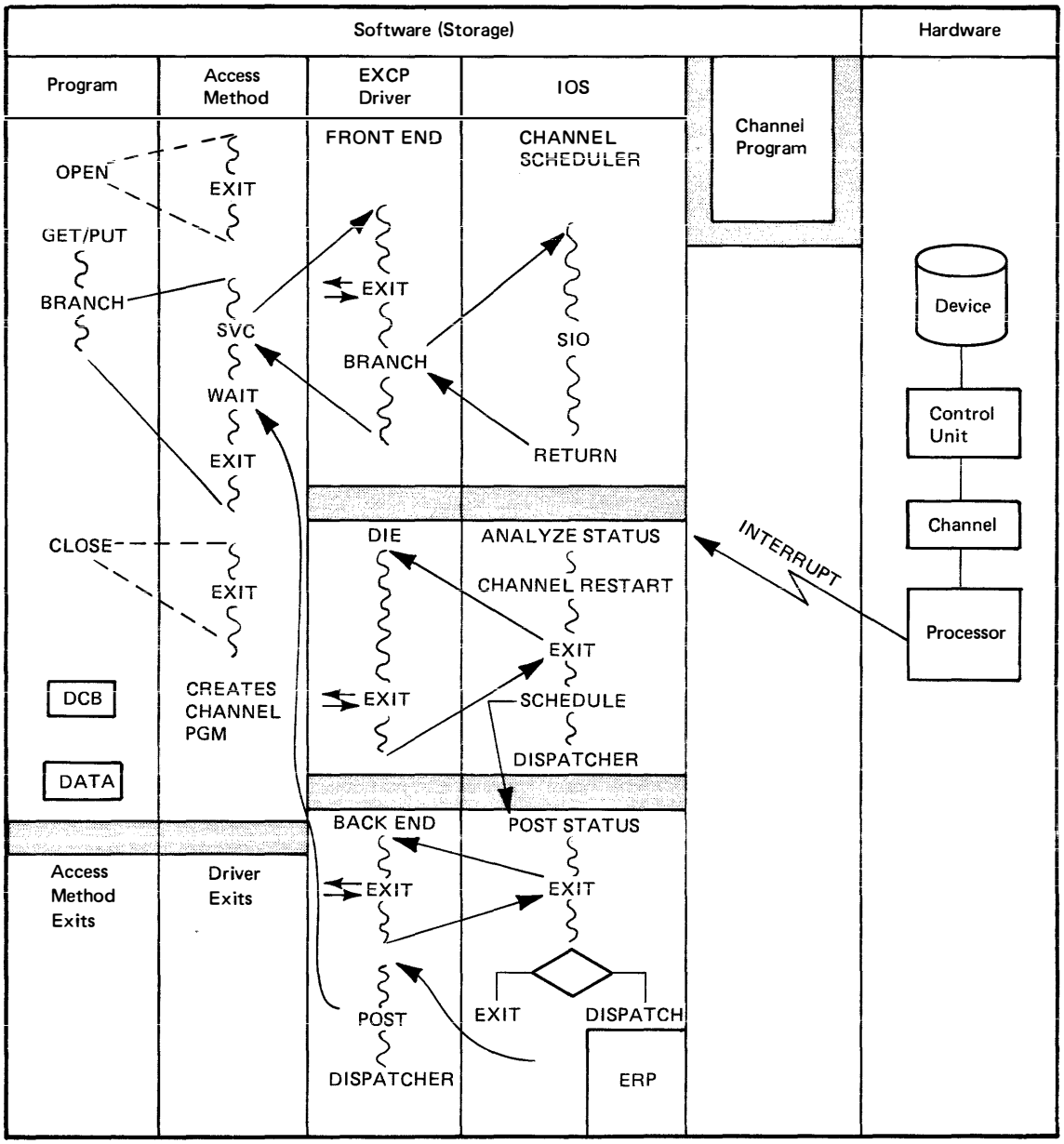



**Figure 8.7. Flow of an I/O Request**

## Virtual Input/Output (VIO)

A physical input/output operation reads data from or writes data to a data set on an I/O device. A virtual input/output (VIO) operation uses the system paging routines to transfer data.

To use VIO, an installation specifies one or more I/O unit names for VIO at system generation time. Then, a user program or access method can build a channel program to send input data to a system-named temporary data set on a unit that was specified for VIO. The EXCP driver intercepts such a channel program and branches to VIO instead of invoking IOS to transfer the data over a channel to a device. VIO uses the move character (MVC) instruction to move that data from the channel program buffers to a special buffer in the user's address space. This special buffer is called a **window**.

The window contains enough contiguous virtual storage pages to hold all of the data that could be placed on a track for a real device. For example, a 2314 track requires a two-page window, and a 3330 or 2305 track requires a four-page window. Figure 8.8 shows the movement of data between the channel program buffer and the VIO window.

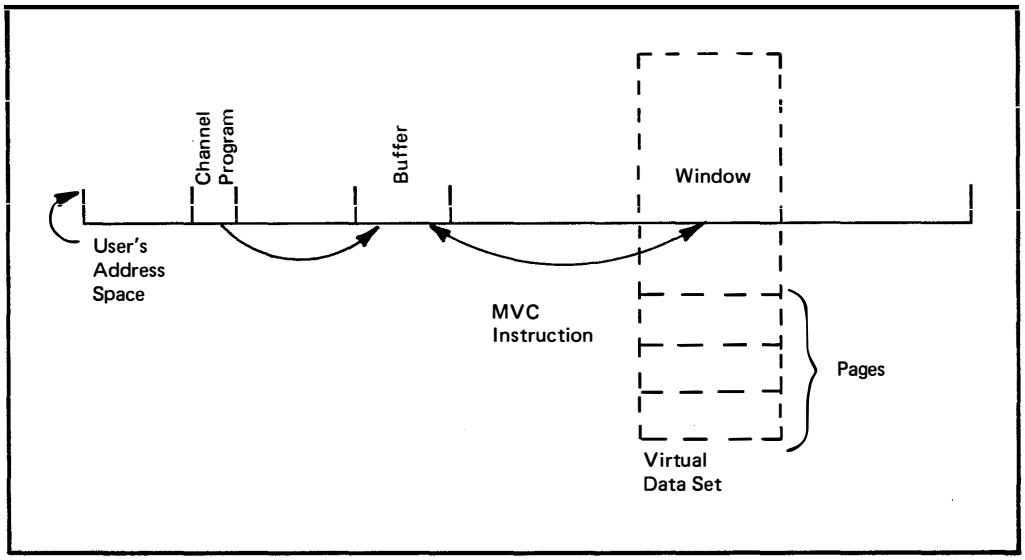



**Figure 8.8. VIO Window**

When VIO intercepts a channel program and issues the first MVC instruction, a page fault causes frames to be assigned to the window. One or more channel programs are then executed to fill the window. When the user program or access method determines that the track is full, it builds another channel program to place data on a second track. When VIO detects this track switch, it writes the contents of the window to a page

data set, using the system paging operations. The system provides special support to keep VIO data set pages in real storage after this page-out, whenever possible. VIO then disconnects the window from the frames that contain the VIO data set pages. When VIO moves new data (the second track) to the window, another page fault occurs, causing fresh frames to be assigned to the window.

As the data set is created and external page storage assigned, the system keeps track of the locations of each page of the VIO data set. The paging data set slots, like the real storage frames, are not necessarily contiguous; they are allocated dynamically throughout external page storage as the data set is created.

When data is to be retrieved from the VIO data set, VIO locates the pages that contain the required data. If the data is not currently in the window, VIO changes the appropriate page table entries to point to the required pages in external page storage. Then VIO uses the MVC instruction to move data from the window to the channel program buffers. This instruction causes a page fault, and the proper page is either reclaimed or brought into real storage and made addressable through the window.

Thus, VIO uses paging rather than explicit I/O to transfer data. VIO eliminates the channel program translation and page fixing done by the EXCP driver as well as some device allocation and data management overhead. It also provides dynamic allocation of DASD space as it is needed. Another advantage of VIO is that the data set can remain in real storage after it is created because VIO attempts to keep the pages in real storage as long as possible. In this case, no actual I/O operations are required to create or retrieve data from the VIO data set.

## Virtual Storage Access Method (VSAM)

The virtual storage access method (VSAM) is a high performance access method for direct access storage that runs in virtual storage and uses virtual storage to buffer input and output operations. VSAM supports batch users, online transactions, and data base applications.

Through a master catalog, VSAM controls allocation of data space on VSAM volumes and the location and use of VSAM data sets. An MVS system requires at least one VSAM master catalog; this required catalog is also the system catalog. It is maintained by VSAM, but, because it is required for system operation, it is discussed separately later in this chapter under "System Catalog."

VSAM can process three types of data sets: key-sequenced, entry-sequenced, and relative record. The order in which the data set is initially loaded and updated is different for each type.

For a **key-sequenced data set**, records are loaded, as the name implies, in key sequence. Each record must have a key, and the ordering of the records is determined by the collating sequence of the keys. Any new records subsequently added to the data set are added in key sequence.

For an **entry-sequenced data set**, records are loaded in sequential order as they are entered. New records are added at the end of the data set.

For a **relative record data set**, records are loaded according to a relative record number that can be assigned either by VSAM or by the user program. When VSAM assigns the relative record number, new records are added at the end of the data set. When the user program assigns the relative record number, new records can be added in relative record number sequence.

When a VSAM data set of any type is created, it is defined to VSAM as a **cluster**. A cluster for a key-sequenced data set consists of an index component and a data component. A cluster for an entry-sequenced or relative record data set consists of only a data component.

A VSAM data set of any type is allocated in a data space. A VSAM **data space** is an area of direct access storage defined in a volume table of contents (VTOC) for exclusive VSAM use. A data space can consist of a single extent (area) on a single volume, multiple extents on multiple volumes, or multiple data spaces on multiple volumes. A single volume can contain both VSAM data spaces and non-VSAM areas.

Within a VSAM data set, VSAM stores the records for each type of data set in the same way — in a fixed-length area of direct access storage called a control interval.

### *Control Interval*

A control interval is a continuous area of direct access storage that VSAM uses for storing data records and the control information that describes them. It is the area that VSAM transfers between virtual and direct access storage during an input or output operation. A control interval can contain stored records, free space, or both stored records and free space.

The size of the control interval for a data set can be chosen by either the user or VSAM. Once chosen, the size is fixed, and all control intervals within the data set are the same length. When VSAM chooses the size of the control interval, it considers the following factors:

- The type of direct access device used for the data set
- The size of the data records
- The smallest amount of virtual storage the user program can provide for I/O buffers

When the user chooses the size of the control interval, the size chosen must fall within limits that VSAM finds acceptable, based on the factors listed above.

The size of the control interval need not correspond to the size of a track on the device. Figure 8.9 shows the independence of control intervals from physical records, which are limited by the capacity of a track on a particular device.

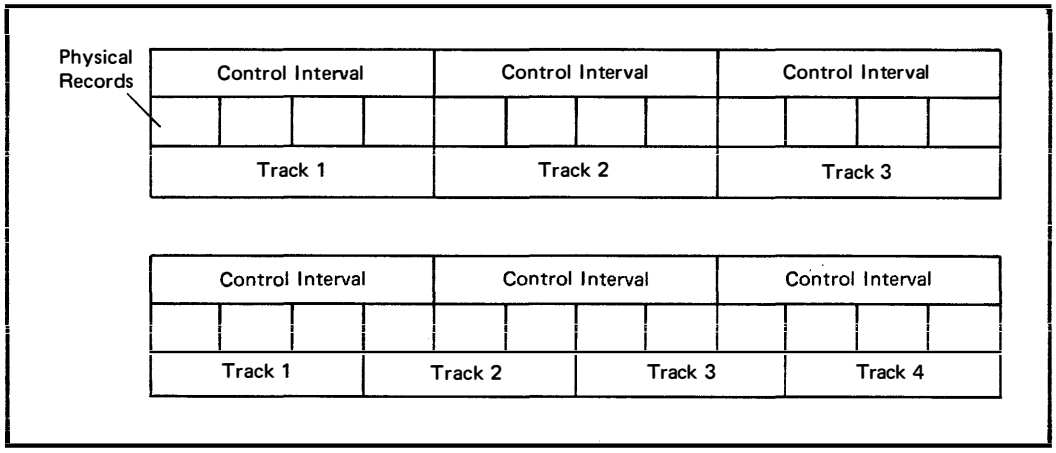



**Figure 8.9. Control Intervals and Physical Records**

Control intervals are grouped together in a control area. A **control area** is the unit of a data set that VSAM preformats for data integrity as records are added to the data set. The number of control intervals in a control area is fixed by VSAM; the minimum is two. In a key-sequenced data set, control areas are also used for placing portions of the index next to the data set and for distributing free space throughout the data set. Free space is distributed as a percentage of control intervals in each control area.

The records in a VSAM data set can be either fixed or variable; VSAM treats both types in the same way. It puts control information at the end of a control interval to describe the data records stored in that control interval. The combination of a data record and its control information, even though they are not physically adjacent, is called a **stored record**. When adjacent records are the same length, they share control information. Figure 8.10 shows how data records and control information are stored in a control interval.

Although the records for each type of VSAM data set are similar in that they are all stored in control intervals, there are significant differences in the way VSAM processes each data set type. These differences are explained in the following text.

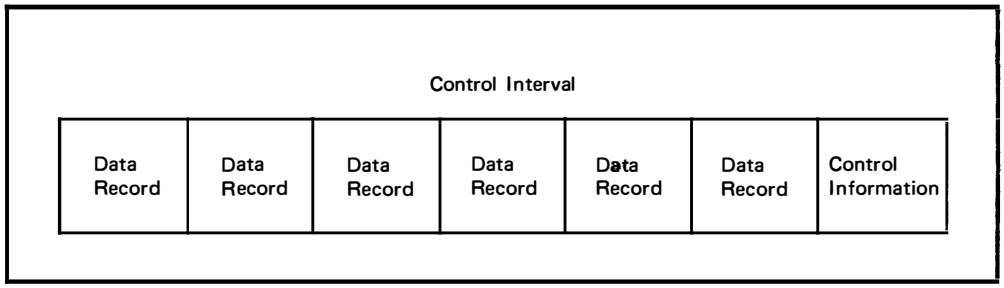



**Figure 8.10. Data Records and Control Information Placement**

### *Key-Sequenced Data Set*

A key-sequenced data set is always defined with an index and distributed free space. The index relates key values to the location of the associated record in the data set. The index created with the data set is the **prime index**; other indexes, called **alternate indexes**, can also be created for the data set, as described later in this chapter under "Alternate Indexes." **Distributed free space** is the number of control intervals within a control area that are initially left blank; VSAM uses the distributed free space to add records to the data set in key sequence. VSAM also reclaims space freed by the deletion or shortening of records; that is, such space is also available to hold additional records.

The index for a key-sequenced data set has one or more levels, each of which is a set of records that contains entries giving the location of the records in the next lower level. The index records at the lowest level are called the **sequence set**; they give the location of control intervals containing data records. The records in all higher levels are called the **index set**; they point to lower-level index records. The highest level always consists of only one record. The index of a small data set thus might consist of one record.

Figure 8.11 shows the levels of a prime index and the relationship between a sequence-set index record and a control area. Note that the highest-level index record (A) controls the entire next level (B through Z) and that each sequence-set index record points to a control area as well as to control intervals within the control area.

Figure 8.11 also shows both vertical and horizontal pointers. Vertical pointers are followed to access records directly by key. Horizontal pointers are followed between the sequence-set index records to access records sequentially by key. To reduce the size of the index, keys can be compressed; that is, VSAM retains only those characters required to distinguish one key from another.

Because VSAM transmits control intervals between direct access storage and virtual storage, index keys are compared and stored and records are accessed while they are in virtual storage.

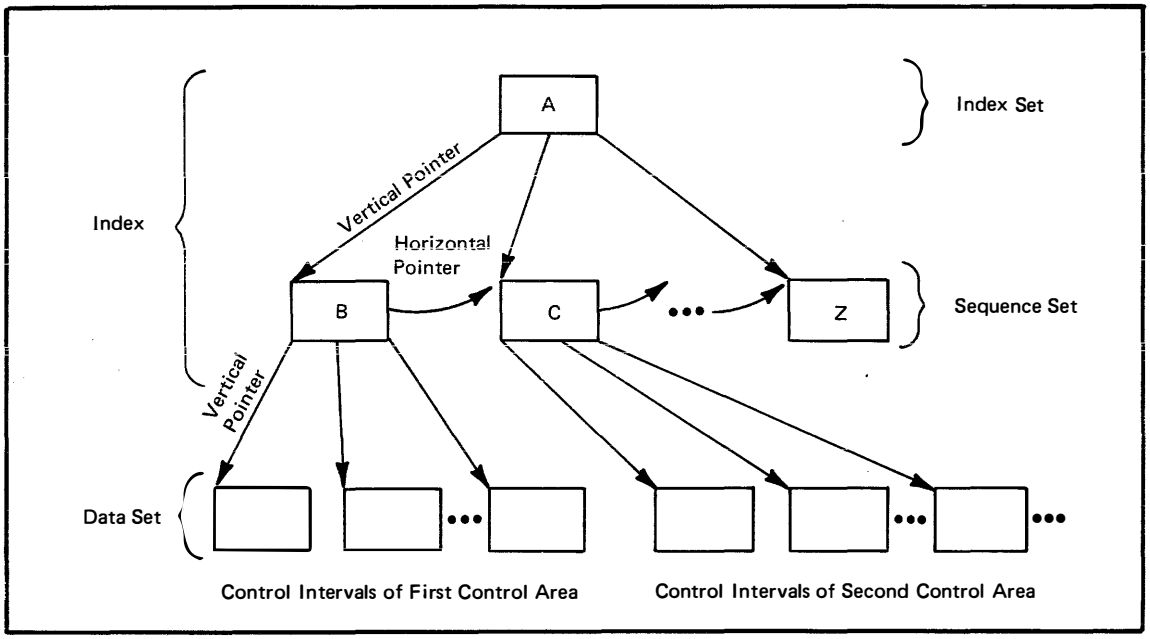



**Figure 8.11. Relationships Between Levels of a Prime Index**

## *Entry-Sequenced Data Set*

Records in an entry-sequenced data set are loaded in the order in which they are received. When VSAM places a record in the data set, it returns the relative byte address (RBA) of the record to the user program. Thus, the records could be accessed directly because the user program can create an index based on the RBAs returned by VSAM.

When the records are accessed sequentially, VSAM retrieves them in the order in which they were stored. Thus, an entry-sequenced data set is very useful for such applications as a journal or a log.

No prime index is associated with an entry-sequenced data set; however, it can have an alternate index. See "Alternate Indexes" later in this chapter.

## *Relative Record Data Set*

In a relative record data set, each record occupies a fixed-length slot, each of which has a relative record number ranging from one up to the total number of records in the data set. A record is stored and retrieved according to the number of the slot that it occupies.

Because a slot can contain data or be empty, a data record can be inserted, moved, or deleted without affecting the position of other data records. Records can be accessed either sequentially or directly but only by relative record number; a record cannot be accessed by its relative byte address (RBA).

A relative record data set is appropriate for many applications that use fixed-length records. A user program could, for example, process a field in each record to yield a unique relative record number for each record. Then, a record could be located directly through the contents of the field. In this way, a relative record data set could be accessed as if it were a key-sequenced data set but without the overhead required to search through index records to locate a particular record.

Like a key-sequenced or entry-sequenced data set, records in a relative record data set are grouped together in control intervals. Each control interval contains the same number of slots, the size of which is the record length specified when the data set is defined. The number of slots in a control interval is determined by the control interval size and the record length.

### *Alternate Indexes*

An alternate index provides another way to gain access to a single data set, thus eliminating the need to keep multiple copies of the same information organized in different ways for different applications. For example, a payroll data set indexed by employee number can also be indexed by other fields, such as employee name or department number. Thus, multiple alternate indexes can be associated with the same base data set, allowing multiple logical paths to the same data.

VSAM can build an alternate index for either a key-sequenced or an entry-sequenced data set. Each entry in an alternate index for a key-sequenced data set contains an alternate key and one or more prime key pointers. Each entry in an alternate index for an entry-sequenced data set contains a key and an RBA pointer. Alternate indexes can be used to access a data set either sequentially or directly.

Alternate indexes must, of course, be updated to reflect changes to the base data set. Either VSAM or the user program can maintain the alternate indexes.

### *System Catalog*

Under MVS, the VSAM master catalog, which acts as a central information point for volumes, data spaces, and data sets controlled by VSAM, is also the system catalog.

The system catalog contains pointers to VSAM data sets, to all system data sets that must be cataloged, to VSAM user catalogs, and to non-VSAM data sets and user catalogs. Non-VSAM data sets are called OS data sets, and non-VSAM user catalogs are called CVOLs. Figure 8.12 shows the structure of the system catalog.

There can be only one system catalog. It is established at system generation time and must be available to the system during system initialization and operation to locate user catalogs, data spaces, and data sets. The volume on which the system catalog is defined must be permanently mounted.

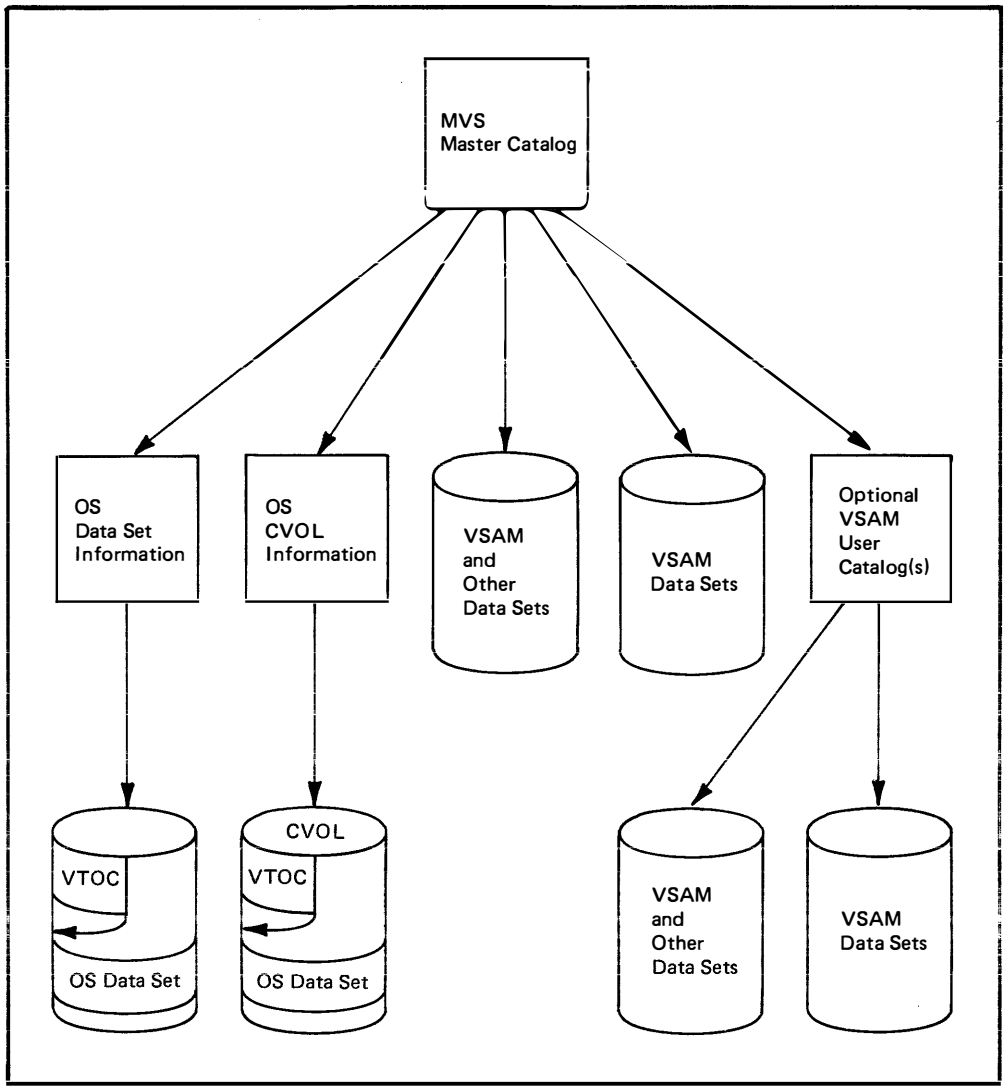



**Figure 8.12. Structure of the System Catalog**

# Chapter 9: Recovering From Errors

A system is available when both its hardware and software are capable of processing jobs. Error recovery in MVS is designed to increase the availability of the system and reduce the impact on users when errors occur in critical software and hardware components. If recovery is not possible, MVS attempts to continue without the damaged facility. In general, recovery is attempted in such a manner that the recovery processes are transparent to the user.

Recovery routines have four objectives:

- To isolate the error
- To assess the damage, and attempt to confine it to one user or task
- To indicate the actions, such as dumping, that should be taken
- To repair the damage and perform clean-up processing so that the function is reinvokable

In MVS, error processing of software failures is handled by recovery termination, and error processing of hardware failures is handled by recovery management support (RMS). As a result of these facilities, MVS processing continues with minimal downtime.

## Recovery Termination

The recovery termination manager (RTM) monitors the flow of software recovery processing by handling all abnormal termination of tasks and address spaces, and passing control to recovery routines associated with the terminating functions. The RTM enables user programs to establish their own recovery protection and system programs to enhance system serviceability and reliability.

The RTM is invoked for the following conditions:

- I/O error during a page-in operation
- Program error not handled by a program interruption routine
- Machine error not handled by hardware recovery
- Supervisor call that is invalid
- Restart operation initiated by the console operator
- CALLRTM macro instruction directed towards another task (ABTERM)
- CALLRTM macro instruction directed towards an address space (MEMTERM)
- ABEND macro instruction
- Dynamic address translation (DAT) error
- Branch entries for abnormal termination requests
- Reentry for abnormal termination requests
- Reentry for machine checks

Two types of recovery routines are identified by the RTM: task recovery routines and functional recovery routines. These routines are described in the following sections. (For more information on the recovery routines and the RTM, see *OS/VS2 System Programming Library: Supervisor*, GC28-0628.)

### *Task Recovery Routines*

Task recovery routines (STAE/STAI, ESTAE/ESTAI) provide recovery for those programs that run enabled, unlocked, and in task mode. They are established by using the STAE or ESTAE macro instruction or the STAI or ESTAI parameter of the ATTACH macro instruction.

Issuance of the STAE or ESTAE macro instruction or ATTACH with the STAI or ESTAI option allows the user to intercept an anticipated abend. Control is given to a user-specified routine in which the user may perform pretermination processing, diagnose the cause of the abend, and specify a retry address if he wishes to avoid the termination. The routines operate in the mode (problem program or supervisor) that existed at the time the STAE/ESTAE request was made.

***Note***: The STAE macro instruction is available with OS/VS2 Release 1 (SVS) and with OS/MVT and OS/MFT. Although STAE is also available in MVS, it is recommended that ESTAE be used in MVS. ESTAE provides increased capabilities over STAE: it can schedule clean-up processing under certain instances for which STAE routines do not get control, and it can provide defaults for the most commonly used options.

If a task is scheduled for abnormal termination, the recovery routine specified by the most recently issued ESTAE (or STAE) macro instruction is given control. If the ESTAE routine cannot provide recovery for the error, the next higher-level ESTAE routine (if any) associated with the task is given control. (This process of passing control from a recovery routine to a higher-level recovery routine along a preestablished path is called percolation, and does not apply to STAE routine.) Each ESTAE routine for the task is then given control, one at a time in LIFO (last-in first-out) order, until retry is requested or all routines for the task are exhausted. When ESTAE processing is exhausted, abnormal termination occurs.

### *Functional Recovery Routines*

Functional recovery routines (FRRs) provide recovery for those system programs that run disabled, locked, or in SRB (service request block) mode. The system programs establish the FRRS by using the SETFRR macro instruction.

The SETFRR macro instruction provides each system program with the ability to define its own unique recovery environment. Each FRR established by a system program is placed in an FRR LIFO (last-in first-out) stack that is used during processing of the RTM. The SETFRR macro instruction can be used to add, delete, or replace FRRs in the stack, or to purge all FRRs in the stack.

Each FRR stack used by RTM contains the addresses of the FRRs established to protect a single path through the system control program. When an error occurs in a path, the RTM passes control to the last FRR in the associated stack. If the FRR cannot provide recovery for the error, the previously-established FRR in the stack is given control (percolation.) Each FRR in the stack is eventually given control, one at a time in LIFO order, until retry is requested or the stack is exhausted. When FRR processing is exhausted, appropriate task recovery routines (if any exist) are given control; otherwise, abnormal termination occurs.

Any user-written routines outside the control program that are qualified to issue the SETFRR macro instruction may add one, and only one, FRR to a stack. If more than one FRR is added to a stack, abnormal termination may occur when SETFRR is issued.

## Recovery Management Support

Recovery management support (RMS) includes those standard MVS facilities that gather information about hardware reliability and allow retry of operations that fail because of processor, I/O device, or channel errors. The facilities are designed to keep the system operational in the event of hardware failures.

The RMS facilities are:

- Machine check handler
  - Alternate CPU recovery
  - Channel reconfiguration hardware
- Channel check handler
- Dynamic device reconfiguration
- Missing interruption handler

For information on the RMS facilities in an MP environment, see *OS/VS2 MVS Multiprocessing: An Introduction and Guide to Writing Operating and Recovery Procedures*, GC28-0952.

### *Machine Check Handler*

The machine check handler (MCH) minimizes the impact of machine malfunctions on System/370 models supported by MVS. It alerts the control program of any hardware failures that could affect the successful execution of the control program.

Recovery from machine malfunctions is initially attempted by the hardware instruction retry (HIR) and error checking and correction (ECC) facilities of the hardware. If the hardware recovery attempts are unsuccessful, MCH is invoked to analyze the data and isolate the source of error. MCH then provides the recovery termination manager (RTM) with an analysis of the error.

When the RTM receives control, it records the error analysis on the SYS1.LOGREC data set and invokes the appropriate functional recovery routines to attempt recovery from the machine check. If recovery is possible, RTM resumes the interrupted program at the point of interruption; if recovery is not possible, RTM terminates the interrupted program.

In a uniprocessing environment, if MCH determines that processing cannot continue on the processor, it will terminate execution on that processor and place the processor in a disabled wait state. In a multiprocessing environment, however, MCH will invoke the alternate CPU recovery routine.

Figure 9.1 demonstrates the flow of control through the machine check handler and, also, through alternate CPU recovery and channel reconfiguration hardware.

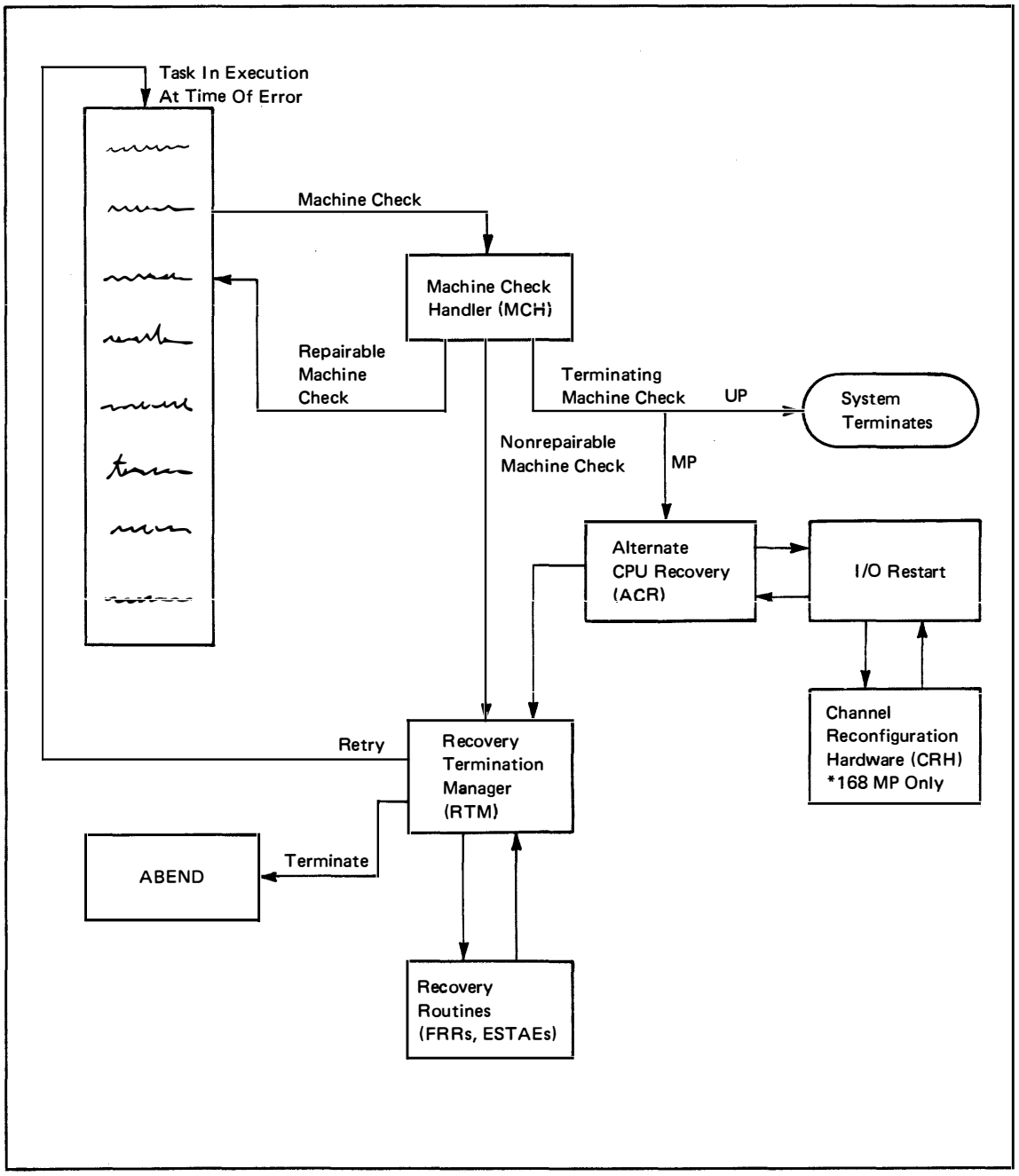



**Figure 9.1. MCH Control Flow**

### Alternate CPU Recovery

The alternate CPU recovery (ACR) routine provides a multiprocessing system with the ability to recover system operations on the operational processor after one processor fails. Where possible, it will take responsibility for all work in progress on the failing processor, including I/O.

In a multiprocessing environment, if MCH is unsuccessful because of a recursive error or a damaged processor, MCH invokes ACR on the operative processor to terminate execution on the failing processor. When ACR receives control, it attempts to transfer work that was in progress on the failing processor to the operative processor. The recovery termination manager then initiates recovery by invoking the appropriate functional recovery routines to free resources associated with the failing processor.

ACR then cleans up resources associated with the failing processor and frees them, where possible, for use by the operative processor. The failing processor is logically disconnected along with all devices uniquely affiliated with that processor. Since the remaining processor cannot continue to handle the load of two processors, it is important for the installation to take appropriate actions to reduce workload and reconfigure I/O.

In a system without channel reconfiguration hardware (CRH), a processor failure in a multiprocessing environment means the loss of all I/O paths through channels attached to the inoperative processor. However, if CRH was included during system generation for a Model 168 MP, then ACR passes control to the CRH routine.

### Channel Reconfiguration Hardware

Channel reconfiguration hardware (CRH) enables either processor in a Model 168 MP to control the operation of the channels normally dedicated to the other processor. The facility is intended as a short-term recovery aid, and can degrade system performance if kept active indefinitely.

CRH receives control when a hardware failure in one processor causes ACR to take that processor offline, or when the operator varies online a channel that is attached only to an offline processor. It is available only on a 168 MP and is included with the 168 hardware; however, it is activated only if included during system generation.

With CRH, since the operative processor can access the channels on the inoperative processor, all devices in the configuration remain accessible to the system. In addition, CRH allows access to symmetric devices when the paths through the operative processor are busy or offline, or when the device is reserved through a path on the inoperative processor.

Since the operation of CRH can result in significant system overhead, the installation should deactivate CRH as soon as possible.

## *Channel Check Handler*

The channel check handler (CCH) reduces the impact of channel malfunctions on System/370 models supported by MVS. It aids the I/O supervisor (IOS) in recovering from channel errors and informs the operator or system maintenance personnel when errors occur.

CCH receives control from the IOS after a channel malfunction is detected. It analyzes the type and extent of the error using the information stored by the channel. If the error condition affects the entire channel, CCH invokes the I/O restart function of IOS to recover the active I/O on the failing channel. If any other error condition occurs, CCH allows the device-dependent error recovery procedures to retry the failing I/O, forcing the retry on an alternate channel path (if one is available). Records describing the error are written to the SYS1.LOGREC data set.

CCH performs no error recovery itself: it does not retry any operation or make any changes to the system. Recovery from channel errors is performed only by the device-dependent routines.

### *Dynamic Device Reconfiguration*

Dynamic device reconfiguration (DDR) allows the system and user to circumvent an I/O failure, if possible, by moving a demountable volume (tape or disk) from one device to another or by substituting one unit record device (reader, punch, or printer) for another. DDR requests are processed without shutting down the system and may eliminate the need for terminating a job.

A DDR swap can be initiated by either the system or an operator. When a permanent I/O error occurs, the system initiates a swap along with a proposed alternate device to take over the processing of the device on which the error occurred. The operator can accept the swap and proposed device, accept the swap but select another device, or refuse the swap. The operator himself may initiate a swap (via the SWAP command) if a device cannot be made ready, if one unit record device is to be substituted for another, or if, for example, cleaning procedures are to be carried out on a device.

For additional information on DDR, see *Operator's Library: OS/VS2 MVS System Commands*, GC38-0229.

### *Missing Interruption Handler*

The missing interruption handler (MIH) checks whether expected I/O interruptions occur within a specified period of time. If an interruption does not occur, the operator is notified so that corrective steps can be taken before system status is harmed. MIH does not support teleprocessing devices or devices that are marked offline.

MIH is invoked as part of the master scheduler. It checks for missing interruptions caused by pending device and channel ends, DDR swaps, and MOUNT commands. The absence of such interruptions may indicate, for example, that a device is not ready, a MOUNT message has not been satisfied, or a device has malfunctioned. Channel and device end interruptions are recorded on the SYS1.LOGREC data set.

If a pending condition is found and remains pending after a useror system-specified time interval has elapsed, a missing interruption condition is determined to exist and the operator is notified. The specific pending condition determines what operator action is needed to correct the situation.

# Chapter 10: Multiprocessing

With the growth of multiple applications and the proliferation of online users, an installation may find that a single processor cannot service its needs. More capacity and higher speed are often required. A viable solution to the need for more computing power is a configuration of several processors sharing one or more critical resources. In such a configuration the processors share the workload and synchronize their activities.

Sharing, synchronizing, and controlling the work on several processors is generally called **multiprocessing**. The two basic types of multiprocessing are:

- Loosely-coupled multiprocessing, which allows processors to operate independently, yet share a common workload queue. The processors are connected by channel-to-channel adapters or by shared DASD.
- Tightly-coupled multiprocessing, which allows two processors to operate under the control of a single operating system. The processors are connected by a multisystem unit.

## Loosely-Coupled Multiprocessing

Loosely-coupled multiprocessing affords an easy growth path. The installation can connect many combinations of System/360 and System/370 processors into a single configuration with the following traits:

- JES2 or JES3 supports the processors' access to a common workload queue.
- Each processor has its own control program.
- The I/O device configurations on the various processors need not be identical. However, availability can be improved by including redundant components and by making the configuration symmetrical.
- Jobs can be routed to a particular processor, if necessary.

For further discussion of JES2 and JES3 multiprocessing support this book, see "Multi-Access Spool" under "JES2 Features" in chapter 5, and "JES3 Features" also in chapter 5.

For more detailed information about JES2 and JES3 multiprocessing support, refer to *OS/VS2 MVS System Programming Library: JES2* and *Introduction to JES3*, respectively.

## Tightly-Coupled Multiprocessing

In a tightly-coupled multiprocessor (MP), the two processors share all processor storage, communicate directly with each other, and operate under the control of a single system control program (OS/VS2 MVS). MVS supports tightly-coupled MPs and APs on the IBM System/370 Model 158 and Model 168.

A Model 158 or 168 tightly-coupled MP configuration in some respects has less complex operational requirements than two uncoupled 158 or 168 processors. The MP presents a single system image to the operator even though there are two processors available for work. The operator has one operational interface to the entire system, one job scheduling interface, and one point of control for all the resources available. In addition, the operator must communicate with and control only one operating system instead of two.

Three other important characteristics of a tightly-coupled MP are:

- The ability to dynamically change the hardware configuration to meet various needs
- The ability to communicate between the processors to coordinate their activity
- The ability to control the operation of the two processors and yet keep their individual control and status information separate

## *Configuration*

A tightly-coupled MP configuration consists of many hardware components, which MVS regards as resources. "Reconfiguration" refers to the process of changing the configuration of these hardware components. It involves varying system resources online or offline as well as changing some control switches on the processors' configuration control panel to establish the corresponding physical configuration.

Change to the configuration can occur for several reasons, such as:

- A segment of storage that experiences failures must be disabled from both processors. By removing the failing storage from the system while the system is still processing, the system operator can isolate the failure from the MVS system and allow the repairs to take place.
- A scheduled change from MP mode to UP mode can allow MVS to continue uninterrupted on one processor while the other processor runs a secondary operating system or undergoes repairs.

### Logical Reconfiguration

The process of varying system resources online and offline with the VARY command is called logical reconfiguration. The system operator uses the VARY command to make system resources (processor, storage, I/O device) either available or unavailable for system use, for example, changing from MP mode to UP mode by varying a processor offline. This command, along with other system commands and operator actions, can separate a system resource from an active MP system without necessarily interrupting the work being processed.

### Physical Reconfiguration

When the system operator changes the logical configuration, he must make corresponding changes to the physical configuration. This process, called physical reconfiguration, involves the configuration control panel which is housed in the multisystem unit that connects the two processors. The configuration control panel contains rotary switches, toggle switches, pushbuttons, and display lights that allow the operator to establish:

- System mode — MP mode in which the processors share real storage and communicate with each other, or UP mode in which the processors operate independently, do not share real storage, and do not communicate with each other.
- Storage configuration — Each storage switch assigns a real storage address range to its associated segment of storage (a storage element). Furthermore, each storage element can be enabled for access by one or both processors or disabled for access by both processors.
- I/O device configuration — A pair of I/O allocation switches (one for each processor) is assigned to each control unit connected to the configuration control panel. Each switch establishes the associated processor's access to a particular control unit. As with segments of storage, each control unit can be enabled for access by one or both processors or disabled for access by both processors.
- Validity of a desired configuration — The configuration-validity indicators show whether the desired configuration control panel settings are acceptable (valid). If the specified configuration is valid, pressing the ENTER CONFIG pushbutton causes the control panel settings to take effect.

## *Communication*

To control the system resources, the two processors must communicate with each other. Communication between the processors is referred to as interprocessor communication (IPC). The MVS software and the System/370 hardware both provide support for IPC.

### MVS-Initiated Communication

MVS establishes interprocessor communication for several purposes:

- To perform system initialization
- To dispatch work or start an I/O operation
- To stop or restart a processor during reconfiguration
- To attempt alternate CPU recovery

To accomplish this communication, MVS uses the signal processor (SIGP) instruction. A SIGP instruction indicates the address of the processor being signaled and transmits a request to that processor. The request indicates the function to be performed. When the addressed processor receives the signal, an external interruption occurs. As a result of the interruption, the addressed processor decodes the request, performs the requested function (if possible), and transmits a response to the calling processor. The response contains a condition code and status information.

The following topics describe some of the SIGP requests used by the system.

**Initialization:** During the initialization of a tightly-coupled MP system, MVS can determine whether the other processor is online by issuing a SIGP sense instruction. The addressed processor responds with an indication of its status. If the response indicates the processor is online, MVS can issue a SIGP start instruction. The addressed processor performs the start function just as though an operator had pressed the START key on the processor's console. When initialization is complete, multiprocessing operation can proceed on both processors.

**Operation:** Normal operation proceeds with each processor receiving work from the MVS dispatcher routine. The dispatcher is normally entered after a system event occurs or when a unit of work is complete. However, if one processor has entered the wait state because it had no work to perform, the other processor may wish to tell the idle processor that new work has arrived. This kind of communication is called "shoulder-tapping."

Other situations may arise that make shoulder-tapping necessary. For example, a program running on processor A may need to issue an I/O request to a device that is attached only to processor B. Using the SIGP external-call instruction, processor A can ask processor B to perform the operation.

**Reconfiguration:** When the operator varies a processor offline or online, MVS-initiated communication may be necessary. For example, if the master scheduler is running in processor A when a VARY command is received to vary processor B offline, processor A must tell processor B to stop. To do this, processor A issues a SIGP stop instruction. Processor B enters the stopped state just as it would if the STOP key on the processor's system console had been pressed. To vary processor B back online, processor A can issue a SIGP restart instruction. Processor B performs a restart function just as though the RESTART key had been pressed.

**Recovery:** When one processor wants the other to perform an action immediately, it executes a SIGP emergency-signal (EMS) instruction, which also results in an external interruption on the other processor. A SIGP emergency-signal is used to initiate actions such as a request from a failing processor for alternate CPU recovery activity on the operative processor. The operative processor can transmit a SIGP program-reset instruction to reset any pending I/O operations that were in progress on the failing processor. The operative processor may also issue a SIGP stop-and-store-status instruction to determine the status of the failing processor. If the status can be obtained, the MVS recovery routines have a better chance of succeeding.

## Hardware-Initiated Communication

In addition to the signals exchanged between processors through use of the SIGP instruction, the System/370 hardware supports direct communication between the processors. This communication is necessary to ensure:

- Clock synchronization
- Storage control
- Recovery

**Clock Synchronization:** In a tightly-coupled MP configuration, each processor has a time-of-day (TOD) clock. When the two processors operate in MP mode, the TOD clock in one processor transmits synchronizing pulses to the other processor to keep the TOD clocks synchronized. When the operator initializes (IPLs) a tightly-coupled MP system or varies a processor online, he must ensure that the TOD clocks are synchronized. If MVS detects that the clocks have become unsynchronized, an external interruption occurs and the processor that accepts the interruption first can reset the clocks and initiate operator intervention, if necessary.

**Storage Control:** Because storage is shared between the processors, the processors must communicate with each other to ensure that all references to shared storage refer to the most current data. Therefore, each processor (for example, processor A) notifies the other processor (for example, processor B) when it modifies the contents of a real storage location. Processor B then determines whether its high-speed buffer currently contains the contents of that same real storage location. If processor B's buffer contains this same storage, this copy of the storage is no longer current; processor B invalidates its copy in the buffer.

**Recovery:** When a processor experiences a failure that causes it to enter the check-stop state, the failing processor generates a malfunction-alert interruption on the other processor, which then attempts recovery. Alternate CPU recovery routines receive control and attempt to keep MVS running on the operative processor.

## *Control*

Although tightly-coupled MPs share all real storage and run under the control of a single MVS operating system, each processor must have a unique physical address for identification purposes. Likewise, each processor must have its own status and control information.

### Physical Addresses

In a tightly-coupled MP, one processor is called processor A and the other is called processor B, as indicated on the configuration control panel. Internally, the processors have addresses of 0 and 1, respectively, which the processors must use when signaling each other and when recording the processor identifier in operator messages, SMF records, and so on. The operator must use 0 and 1 when issuing the configuration commands (for example, VARY PATH, VARY CPU). These addresses are permanent and apply in both MP and UP modes.

### Status and Control Information

The System/370 hardware and MVS software maintain status and control information in specifically-assigned real storage locations. This information consists of data such as PSWs. A 4096-byte block of fixed storage is reserved for the information in the low-address range (storage locations 0-4095) of real storage. However, the two processors can execute two jobs concurrently, one in each processor. In order to keep the jobs separate, each processor must have its own storage area. The technique used to achieve this is called **prefixing,** whereby the two processors do not use absolute locations 0-4095 (0-4K) for status and control information. Each processor has its own separate 4K-byte prefixed storage area (PSA) of real storage. MVS can locate each PSA by referring to the address contained in the prefix register for each processor.

## Attached Processor System

An attached processor (AP) system consists of a System/370 Model 158 or Model 168 processing unit (called the host processor) and an attached processing unit. The host processor provides instruction processing, I/O control, and storage control. The attached processor has a similar instruction processing ability, but has no I/O or storage control of its own. The host processor shares its I/O and storage control with the attached processor.

Most communication and control facilities of a tightly-coupled MP also apply to an AP system. However, an AP system's availability is not significantly increased over a UP system because an AP system's ability to reconfigure is limited. An attached processor does not have the same configuration control panel that an MP has. If an attached processor fails, it can be varied offline and MVS can continue on the host processor in UP mode. But if the host processor fails, it cannot be varied offline and MVS cannot continue on the attached processor. [**Exception**: The Model 168 does allow the operator to reinitialize (re-IPL) an attached processor as a stand-alone host processor with access to channels and storage.]

The advantage of an attached processor system is increased performance. Just as in a tightly-coupled MP system, an AP system can execute two tasks concurrently, one in each processor. Part of the performance improvement results from less interprocessor communication (no need to communicate for I/O-device and storage control).

# Index

OS/VS2 MVS Overview
GC28-0984-0

**READER'S COMMENT FORM**

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate.

IBM shall have the nonexclusive right, in its discretion, to use and distribute all submitted information, in any form, for any and all purposes, without obligation of any kind to the submitter. Your interest is appreciated.

Note: ***Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.***

Possible topics for comments are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If comments apply to a Selectable Unit, please provide the name of the Selectable Unit _____.

If you wish a reply, give your name and mailing address:

_____

_____

_____

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

**Cut or Fold Along Line**

Please circle the description that most closely describes your occupation.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Customer** | (Q) Install Mgr. | (U) System Consult. | (X) System Analyst | (Y) System Prog. | (Z) Applica. Prog. | (F) System Oper. | (I) I/O Oper. | (L) Term. Oper. | | | | (O) Other |
| **IBM** | (S) System Eng. | (P) Prog. Sys. Rep. | (A) System Analyst | (B) System Prog. | (C) Applica. Prog. | (D) Dev. Prog. | (R) Comp. Prog. | (G) System Oper. | (J) I/O Oper. | (E) Ed. Dev. Rep. | (N) Cust. Eng. | (T) Tech. Staff Rep. |

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

## Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

First Class
Permit 40
Armonk
New York

**Business Reply Mail**
No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department D58, Building 706-2
PO Box 390
Poughkeepsie, New York 12602

Fold and tape | Please Do Not Staple | Fold and tape

OS/VS2 MVS Overview (SC28-0984) Printed in U.S.A. GC28-0984-0

**International Business Machines Corporation**
**Data Processing Division**
**1133 Westchester Avenue, White Plains, N.Y. 10604**

**IBM World Trade Americas/Far East Corporation**
**Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591**

**IBM World Trade Europe/Middle East/Africa Corporation**
**360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601**

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601