Contents

© Copyright IBM Corp. 2005, 2006. All rights reserved.

Preface

This course provides students of information systems technology with the background knowledge and skills necessary to begin using the basic facilities of a mainframe computer.

For optimal learning, students are assumed to have successfully completed an introductory course in computer system concepts, such as computer organization and architecture, operating systems, data management, or data communications. They should also have successfully completed courses in one or more programming languages, and be PC literate.

Note that this course can also be used as a prerequisite for courses in advanced topics such as compiler algorithms, or for internships and special studies.

Others who will benefit from this course include data processing professionals who have experience with non-mainframe platforms, or who are familiar with some aspects of the mainframe but want to become knowledgeable with other facilities and benefits of the mainframe environment.

When moving through this course, instructors are encouraged to alternate between course, lecture, discussions, and hands-on exercises. The instructor-led discussions and hands-on exercises are an integral part of the learning experience, and can include topics not covered in this course.

After reading this course, students will have received:

- A general introduction to mainframe concepts, usage, and architecture
- A comprehensive overview of z/OS, a widely used mainframe operating system
- An understanding of mainframe workloads and an overview of the major middleware applications in use on mainframes today
- The basis for subsequent course work in more advanced, specialized areas of z/OS, such as system administration or application programming

How this course is organized

This course is organized in four parts, as follows:

- **Part 1. "Introduction to z/OS and the mainframe environment"** provides an overview of the types of workloads commonly processed on the mainframe, such as batch jobs and online transactions. This part of the course helps students explore the

© Copyright IBM Corp. 2005, 2006. All rights reserved.

user interfaces of z/OS, a widely used mainframe operating system. Discussion topics include TSO/E and ISPF, UNIX interfaces, job control language, file structures, and job entry subsystems. Special attention is paid to the users of mainframes and to the evolving role of mainframes in today's business world.

- **Part 2. "Application programming on z/OS"** introduces the tools and utilities for developing a simple program to run on z/OS. This part of the course guides the student through the process of application design, choosing a programming language, and using a runtime environment.
- **Part 3. "Online workloads for z/OS"** examines the major categories of interactive workloads processed by z/OS, such as transaction processing, database management, and Web-serving. This part of the course includes discussions of several popular middleware products, including DB2®, CICS®, and WebSphere® Application Server.
- **Part 4. "System programming on z/OS"** provides topics to help the student become familiar with the role of the z/OS system programmer. This part of the course includes discussions of system libraries, starting and stopping the system, security, network communications and the clustering of multiple systems. Also provided is an overview of mainframe hardware systems, including processors and I/O devices.

In this course, we use simplified examples and focus mainly on basic system functions. Hands-on exercises are provided throughout the course to help students explore the mainframe style of computing. Exercises include entering work into the system, checking its status, and examining the output of submitted jobs.

How each topic is organized

Each chapter follows a common format:

- Objectives for the student
- Topics that teach a central theme related to mainframe computing
- Summary of the main ideas of the chapter
- A list of key terms introduced in the chapter
- Questions for review to help students verify their understanding of the material
- Topics for further discussion to encourage students to explore issues that extend beyond the chapter objectives
- Hands-on exercises intended to help students reinforce their understanding of the material

Part 1

Introduction to z/OS and the mainframe environment

Welcome to mainframe computing! We begin this course with an overview of the mainframe computer and its place in today's information technology (IT) organization. We explore the reasons why public and private enterprises throughout the world rely on the mainframe as the foundation of large-scale computing. We discuss the types of workloads that are commonly associated with the mainframe, such as batch jobs and online or interactive transactions, and the unique manner in which this work is processed by a widely used mainframe operating system—z/OS.

Throughout this course, we pay special attention to the people who use mainframes and to the role of ***The New Mainframe*** in today's business world.

© Copyright IBM Corp. 2005, 2006. All rights reserved.

1

Introduction to the new mainframe

Objective: As a technical professional in the world of mainframe computing, you will need to understand how mainframe computers support your company's IT infrastructure and business goals. You will also need to know the job titles of the various members of your company's mainframe support team.

After completing this chapter, you will be able to:

- List ways in which the mainframe of today challenges the traditional thinking about centralized computing versus distributed computing.
- Explain how businesses make use of mainframe processing power, the typical uses of mainframes, and how mainframe computing differs from other types of computing.
- Outline the major types of workloads for which mainframes are best suited.
- Name five jobs or responsibilities that are related to mainframe computing.
- Identify four mainframe operating systems.

© Copyright IBM Corp. 2005, 2006. All rights reserved.

1.1 The new mainframe

Today, mainframe computers play a central role in the daily operations of most of the world's largest corporations, including many Fortune 1000 companies. While other forms of computing are used extensively in various business capacities, the mainframe occupies a coveted place in today's e-business environment. In banking, finance, health care, insurance, public utilities, government, and a multitude of other public and private enterprises, the mainframe computer continues to form the foundation of modern business.

The long-term success of mainframe computers is without precedent in the information technology (IT) field. Periodic upheavals shake world economies and continuous—often wrenching—change in the Information Age has claimed many once-compelling innovations as victims in the relentless march of progress. As emerging technologies leap into the public eye, many are just as suddenly rendered obsolete by some even newer advancement. Yet today, as in every decade since the 1960s, mainframe computers and the mainframe *style* of computing dominate the landscape of large-scale business computing.

Why has this one form of computing taken hold so strongly among the world's largest corporations? In this chapter, we look at the reasons why mainframe computers continue to be the popular choice for large-scale business computing.

1.2 The S/360: A turning point in mainframe history

When did mainframe computers come into being? The origin of mainframe computers dates back to the 1950s, if not earlier. In those days, mainframe computers were not just the largest computers; they were the *only* computers and few businesses could afford them.

Mainframe development occurred in a series of *generations* starting in the 1950s. First generation systems, such as the IBM 705 in 1954 and the IBM 1401 in 1959, were a far cry from the enormously powerful machines that were to follow, but they clearly had characteristics of mainframe computers. These computers were sold as business machines and served then—as they do now—as the central data repository in a corporation's data processing center. [1]

In the 1960s, the course of computing history changed dramatically when mainframe manufacturers began to standardize the hardware and software they offered to customers. The introduction of the IBM System/360 ™ (or S/360™) in 1964 signaled the start of the third generation: the first general purpose computers. Earlier systems such as the 1401

[1] According to the IBM product brochure, typical customer uses for a 1401 were "payroll, railroad freight car accounting, public utility customer accounting, inventory control, and accounts receivable."

were dedicated as either commercial or scientific computers. The revolutionary S/360 could perform both types of computing, as long as the customer, a software company, or a consultant provided the programs to do so. In fact, the name S/360 refers to the architecture's wide scope: 360 degrees to cover the entire circle of possible uses.

With standardized mainframe computers to run their workloads, customers could, in turn, write business applications that didn't need specialized hardware or software. Moreover, customers were free to upgrade to newer and more powerful processors without concern for compatibility problems with their existing applications. The first wave of customer business applications were mostly written in Assembler, COBOL, FORTRAN, or PL/1, and a substantial number of these older programs are still in use today.

In the decades since the 1960s, mainframe computers have steadily grown to achieve enormous processing capabilities. ***The New Mainframe*** has an unrivaled ability to serve end users by the tens of thousands, manage petabytes of data, and reconfigure hardware and software resources to accommodate changes in workload—all from a single point of control.

1.3 An evolving architecture

An *architecture* is a set of defined terms and rules that are used as instructions to build products. In computer science, an architecture describes the organizational structure of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components, and subsystems.

Starting with the first large machines, which arrived on the scene in the 1960s and became known as "Big Iron" (in contrast to smaller departmental systems), each new generation of mainframe computers has included improvements in one or more of the following areas of the architecture:[2]

- More and faster processors
- More physical memory and greater memory addressing capability
- Dynamic capabilities for upgrading both hardware and software
- Increased automation of hardware error checking and recovery
- Enhanced devices for input/output (I/O) and more and faster paths (*channels*) between I/O devices and processors
- More sophisticated I/O attachments, such as LAN adapters with extensive inboard processing
- A greater ability to divide the resources of one machine into multiple, logically independent and isolated systems, each running its own operating system

[2] Since the introduction of the S/360 in 1964, IBM has significantly extended the platform roughly every ten years: System/370™ in 1970, System/370 Extended Architecture (370-XA) in 1983, Enterprise Systems Architecture/390® (ESA/390) in 1990, and z/Architecture™ in 2000.

- Advanced clustering technologies, such as *Parallel Sysplex,* and the ability to share data among multiple systems.

Despite the continual change, mainframe computers remain the most stable, secure, and compatible of all computing platforms. The latest models can handle the most advanced and demanding customer workloads, yet continue to run applications that were written in the 1970s or earlier.

How can a technology change so much, yet remain so stable? It can by evolving to meet new challenges. In the early 1990s, the client/server model of computing, with its distributed nodes of less powerful computers, emerged to challenge the dominance of mainframe computers. Industry pundits predicted a swift end for the mainframe computer and called it a "dinosaur." In response, mainframe designers did what they have always done when confronted with changing times and a growing list of user requirements: they designed new mainframe computers to meet the demand. With a tip of the hat to the dinosaur naysayers, IBM, as the leading manufacturer of mainframe computers, code-named its then-current machine *T-Rex*.

With the expanded functions and added tiers of data processing capabilities such as Web-serving, autonomics, disaster recovery, and grid computing, the mainframe computer is poised to ride the next wave of growth in the IT industry. Mainframe manufacturers such as IBM are once again reporting annual sales growth in the double digits.

And the evolution continues. While the mainframe computer has retained its traditional, central role in the IT organization, that role is now defined to include being the primary hub in the largest distributed networks. In fact, the Internet itself is based largely on numerous, interconnected mainframe computers serving as major hubs and routers.

As the image of the mainframe computer continues to evolve, you might ask: is the mainframe computer a self-contained computing environment, or is it one part of the puzzle in distributed computing? The answer is that ***The New Mainframe*** is both: a self-contained processing center, powerful enough to process the largest and most diverse workloads in one secure "footprint," and one that is just as effective when implemented as the primary server in a corporation's distributed server farm. In effect, the mainframe computer is the definitive server in the client/server model of computing.

1.4 Mainframes in our midst

Despite the predominance of mainframes in the business world, these machines are largely invisible to the general public, the academic community, and indeed many experienced IT professionals. Instead, other forms of computing attract more attention, at least in terms of visibility and public awareness. That this is so is perhaps not surprising. After all, who among us needs direct access to a mainframe? And, if we did, where

would we find one to access? The truth, however, is that we are *all* mainframe users, whether we realize it or not (more on this later).

Most of us with some personal computer (PC) literacy and sufficient funds can purchase a notebook computer and quickly put it to good use—running software, browsing Web sites, and perhaps even writing papers for college professors to grade. With somewhat greater effort and technical prowess, we can delve more deeply into the various facilities of a typical Intel®-based workstation and learn its capabilities through direct, hands-on experience—with or without help from any of a multitude of readily available information sources in print or on the Web.

Mainframes, however, tend to be hidden from the public eye. They do their jobs dependably—indeed, with almost total reliability—and are highly resistant to most forms of insidious abuse that afflict PCs, such as e-mail-borne viruses and Trojan Horses. By performing stably, quietly, and with negligible downtime, mainframes are the example by which all other computers are judged. But at the same time, this lack of attention tends to allow them to fade into the background.

Furthermore, in a typical customer installation, the mainframe shares space with many other hardware devices: external storage devices, hardware network routers, channel controllers, and automated tape library "robots," to name a few. ***The New Mainframe*** is physically no larger than many of these devices and generally does not stand out from the crowd of peripheral devices.

So, how can we explore the mainframe's capabilities in the real world? How can we learn to interact with the mainframe, learn its capabilities, and understand its importance to the business world? Major corporations are eager to hire new mainframe professionals, but there's a catch: Some previous experience would help.

Would we even know a mainframe if we saw one, given that these machines have evolved to flourish in the twenty-first century IT organization? What we need is an experienced guide to lead us on a *dinosaur safari*, which is where this course comes in!

1.5 What is a mainframe?

First, let's tackle the terminology. Today, computer manufacturers don't always use the term *mainframe* to refer to mainframe computers. Instead, most have taken to calling any commercial-use computer—large or small—a *server*, with the mainframe simply being the largest type of server in use today. IBM, for example, refers to its latest mainframe as the IBM System z9™ server. We use the term mainframe in this course to mean computers that can support thousands of applications and input/output devices to simultaneously serve thousands of users.

Servers are proliferating. A business might have a large server collection that includes transaction servers, database servers, e-mail servers and Web servers. Very large collections of servers are sometimes called *server farms* (in fact, some data centers cover areas measured in *acres*). The hardware required to perform a server function can range from little more than a cluster of rack-mounted personal computers to the most powerful mainframes manufactured today.

A mainframe is the central data repository, or *hub,* in a corporation's data processing center, linked to users through less powerful devices such as workstations or terminals. The presence of a mainframe often implies a centralized form of computing, as opposed to a distributed form of computing. Centralizing the data in a single mainframe repository saves customers from having to manage updates to more than one copy of their business data, which increases the likelihood that the data is current.

The distinction between centralized and distributed computing, however, is rapidly blurring as smaller machines continue to gain in processing power and mainframes become ever more flexible and multi-purpose. Market pressures require that today's businesses continually reevaluate their IT strategies to find better ways of supporting a changing marketplace. As a result, mainframes are now frequently used in combination with networks of smaller servers in a multitude of configurations. The ability to dynamically reconfigure a mainframe's hardware and software resources (such as processors, memory, and device connections), while applications continue running, further underscores the flexible, evolving nature of the modern mainframe.

While mainframe hardware has become harder to pigeon-hole, so, too, have the operating systems that run on mainframes. Years ago, in fact, the terms defined each other: a mainframe was any hardware system that ran a major IBM operating system.[3] This meaning has been blurred in recent years because these operating systems can be run on very small systems.

Computer manufacturers and IT professionals often use the term *platform* to refer to the hardware and software that are associated with a particular computer architecture. For example, a mainframe computer and its operating system (and their predecessors[4]) are considered a platform; UNIX on a Reduced Instruction Set Computer (RISC) system is considered a platform somewhat independently of exactly which RISC machine is involved; personal computers can be seen as several different platforms, depending on which operating system is being used.

So, let's return to our question now: "What is a mainframe?" Today, the term *mainframe* can best be used to describe a *style* of operation, applications, and operating system facilities. To start with a working definition, "a mainframe is what businesses use to host

[3] The name was also traditionally applied to large computer systems that were produced by other vendors.

[4] IBM System/390® (S/390®) refers to a specific series of machines, which have been superseded by the IBM zSeries machines. Nevertheless, many S/390 systems are still in use. Therefore, keep in mind that although we discuss the zSeries systems in this course, almost everything discussed also applies to S/390 machines. One major exception is 64-bit addressing, which is used only with zSeries.

the commercial databases, transaction servers, and applications that require a greater degree of security and availability than is commonly found on smaller-scale machines."

Early mainframe systems were housed in enormous, room-sized metal boxes or frames, which is probably how the term mainframe originated. The early mainframe required large amounts of electrical power and air-conditioning, and the room was filled mainly with I/O devices. Also, a typical customer site had several mainframes installed, with most of the I/O devices connected to all of the mainframes. During their largest period, in terms of physical size, a typical mainframe occupied 2,000 to 10,000 square feet (600 to 3000 square meters). Some installations were even larger than this.

Starting around 1990, mainframe processors and most of their I/O devices became physically smaller, while their functionality and capacity continued to grow. Mainframe systems today are much smaller than earlier systems—about the size of a large refrigerator.

In some cases, it is now possible to run a mainframe operating system on a PC that emulates a mainframe. Such emulators are useful for developing and testing business applications before moving them to a mainframe production system.

Clearly, the term "mainframe" has expanded beyond merely describing the physical characteristics of a system. Instead, the word typically applies to some combination of the following attributes:

- Compatibility with mainframe operating systems, applications, and data.
- Centralized control of resources.
- Hardware and operating systems that can share access to disk drives with other systems, with automatic locking and protection against destructive simultaneous use of disk data.
- A *style* of operation, often involving dedicated operations staff who use detailed *operations procedure books* and highly organized procedures for backups, recovery, training, and disaster recovery at an alternative location.
- Hardware and operating systems that routinely work with hundreds or thousands of simultaneous I/O operations.
- Clustering technologies that allow the customer to operate multiple copies of the operating system as a single system. This configuration, known as Parallel Sysplex, is

analogous in concept to a UNIX cluster, but allows systems to be added or removed as needed, while applications continue to run. This flexibility allows mainframe customers to introduce new applications, or discontinue the use of existing applications, in response to changes in business activity.

- Additional data and resource sharing capabilities. In a Parallel Sysplex, for example, it is possible for users across multiple systems to access the same databases concurrently, with database access controlled at the *record level*.

As the performance and cost of such hardware resources as central processing unit (CPU) power and external storage media improve, and the number and types of devices that can be attached to the CPU) increase, the operating system software can more fully take advantage of the improved hardware. Also, continuing improvements in software functionality help drive the development of each new generation of hardware systems.

1.6 Who uses mainframe computers?

So, who uses mainframes? Just about *everyone* has used a mainframe computer at one point or another. If you ever used an *automated teller machine* (ATM) to interact with your bank account, you used a mainframe.

Today, mainframe computers play a central role in the daily operations of most of the world's largest corporations. While other forms of computing are used extensively in business in various capacities, the mainframe occupies a coveted place in today's e-business environment. In banking, finance, health care, insurance, utilities, government, and a multitude of other public and private enterprises, the mainframe computer continues to be the foundation of modern business.

Until the mid-1990s, mainframes provided the *only* acceptable means of handling the data processing requirements of a large business. These requirements were then (and are often now) based on large and complex batch jobs, such as payroll and general ledger processing.

The mainframe owes much of its popularity and longevity to its inherent reliability and stability, a result of careful and steady technological advances that have been made since the introduction of the System/360 in 1964. No other computer architecture can claim as much continuous, evolutionary improvement, while maintaining compatibility with previous releases.

Because of these design strengths, the mainframe is often used by IT organizations to host the most important, *mission-critical* applications. These applications typically include customer order processing, financial transactions, production and inventory control, payroll, as well as many other types of work.

One common impression of a mainframe's user interface is the 80x24-character "green screen" terminal, named for the old cathode ray tube (CRT) monitors from years ago that glowed green. In reality, mainframe interfaces today look much the same as those for personal computers or UNIX systems. When a business application is accessed through a Web browser, there is often a mainframe computer performing crucial functions "behind the scenes."

Many of today's busiest Web sites store their production databases on a mainframe host. New mainframe hardware and software products are ideal for Web transactions because they are designed to allow huge numbers of users and applications to rapidly and simultaneously access the same data without interfering with each other. This security, scalability, and reliability is critical to the efficient and secure operation of contemporary information processing.

Corporations use mainframes for applications that depend on scalability and reliability. For example, a banking institution could use a mainframe to host the database of its customer accounts, for which transactions can be submitted from any of thousands of ATM locations worldwide.

Businesses today rely on the mainframe to:

- Perform large-scale transaction processing (thousands of transactions per second)[5]
- Support thousands of users and application programs concurrently accessing numerous resources
- Manage terabytes of information in databases
- Handle large-bandwidth communication

The roads of the information superhighway often lead to a mainframe.

1.7 Factors contributing to mainframe use

The reasons for mainframe use are many, but most generally fall into one or more of the following categories:

- Reliability, availability, and serviceability
- Security
- Scalabilty
- Continuing compatibility
- Evolving architecture

Let's look at each of these categories in more detail.

[5] IBM's latest series of mainframe computers, the IBM System z9™ 109 (also known as the z9-109) can process a staggering *one billion* transactions per day.

1.7.1 Reliability, availability, and serviceability

The *reliability, availability,* and *serviceability* (or "RAS") of a computer system have always been important factors in data processing. When we say that a particular computer system "exhibits RAS characteristics," we mean that its design places a high priority on the system remaining in service at all times. Ideally, RAS is a central design feature of all aspects of a computer system, including the applications.

RAS has become accepted as a collective term for many characteristics of hardware and software that are prized by mainframe users. The terms are defined as follows:

Reliability The system's hardware components have extensive self-checking and self-recovery capabilities. The system's software reliability is a result of extensive testing and the ability to make quick updates for detected problems.

Availability The system can recover from a failed component without impacting the rest of the running system. This applies to hardware recovery (the automatic replacing of failed elements with spares) and software recovery (the layers of error recovery that are provided by the operating system).

Serviceability The system can determine why a failure occured.Allows for the replacement of hardware and software elements while impacting as little of the operational system as possible. This term also implies well-defined units of replacement, either hardware or software.

A computer system is available when its applications are available. An available system is one that is reliable; that is, it rarely requires downtime for upgrades or repairs. And, if the system is brought down by an error condition, it must be serviceable; that is, easy to fix within a relatively short period of time.

Mean time between failure (MTBF) refers to the availability of a computer system. ***The New Mainframe*** and its associated software have evolved to the point that customers often experience months or even *years* of system availability between system downtimes. Moreover, when the system is unavailable because of an unplanned failure or a scheduled upgrade, this period is typically very short. The remarkable availability of the system in processing the organization's mission-critical applications is vital in today's 24-hour, global economy. Along with the hardware, mainframe operating systems exhibit RAS through such features as storage protection and a controlled maintenance process.

Beyond RAS, a state-of-the-art mainframe system might be said to provide *high availability* and *fault tolerance*. Redundant hardware components in critical paths, enhanced storage protection, a controlled maintenance process, and system software designed for unlimited availability all help to ensure a consistent, highly available environment for business applications in the event that a system component fails. Such

an approach allows the system designer to minimize the risk of having a *single point of failure* undermine the overall RAS of a computer system.

1.7.2 Security

One of a firm's most valuable resources is its data: customer lists, accounting data, employee information, and so on. This critical data needs to be securely managed and controlled, and, simultaneously, made available to those users authorized to see it. The mainframe computer has extensive capabilities to simultaneously share, but still protect, the firm's data among multiple users.

In an IT environment, data security is defined as protection against unauthorized access, transfer, modification, or destruction, whether accidental or intentional. To protect data and to maintain the resources necessary to meet the security objectives, customers typically add a sophisticated security manager product to their mainframe operating system. The customer's security administrator often bears the overall responsibility for using the available technology to transform the company's security policy into a usable plan.

A secure computer system prevents users from accessing or changing any objects on the system, including user data, except through system-provided interfaces that enforce authority rules. ***The New Mainframe*** can provide a very secure system for processing large numbers of heterogeneous applications that access critical data.

1.7.3 Scalability

It has been said that the only constant is *change.* Nowhere is that statement truer than in the IT industry. In business, positive results can often trigger a growth in IT infrastructure to cope with increased demand. The degree to which the IT organization can add capacity without disruption to normal business processes or without incurring excessive overhead (nonproductive processing) is largely determined by the *scalability* of the particular computing platform.

By scalability, we mean the ability of the hardware, software, or a distributed system to continue to function well as it is changed in size or volume; for example, the ability to retain performance levels when adding processors, memory, and storage. A scalable system can efficiently adapt to work, with larger or smaller networks performing tasks of varying complexity.

As a company grows in employees, customers, and business partners, it usually needs to add computing resources to support business growth. One approach is to add more processors of the same size, with the resulting overhead in managing this more complex setup. Alternatively, a company can consolidate its many smaller processors into fewer, larger systems. Using a mainframe system, many companies have significantly lowered

their total cost of ownership (*TCO*), which includes not only the cost of the machine (its hardware and software), but the cost to run it.

Mainframes exhibit scalability characteristics in both hardware and software, with the ability to run multiple copies of the operating system software as a single entity called a system complex, or *sysplex*.

1.7.4 Continuing compatibility

Mainframe customers tend to have a very large financial investment in their applications and data. Some applications have been developed and refined over decades. Some applications were written many years ago, while others may have been written "yesterday." The ability of an application to work in the system or its ability to work with other devices or programs is called *compatibility.*

The need to support applications of varying ages imposes a strict compatibility demand on mainframe hardware and software, which have been upgraded many times since the first System/360 mainframe computer was shipped in 1964. Applications *must* continue to work properly. Thus, much of the design work for new hardware and system software revolves around this compatibility requirement.

The overriding need for compatibility is also the primary reason why many aspects of the system work as they do, for example, the syntax restrictions of the job control language (JCL) that is used to control batch jobs. Any new design enhancements made to JCL must preserve compatibility with older jobs so that they can continue to run without modification. The desire and need for continuing compatibility is one of the defining characteristics of mainframe computing.

Absolute compatibility across decades of changes and enhancements is not possible, of course, but the designers of mainframe hardware and software make it a top priority. When an incompatibility is unavoidable, the designers typically warn users *at least a year* in advance that software changes might be needed.

1.8 Typical mainframe workloads

Most mainframe workloads fall into one of two categories: batch processing or online transaction processing, which includes Web-based applications (Figure 1-1).

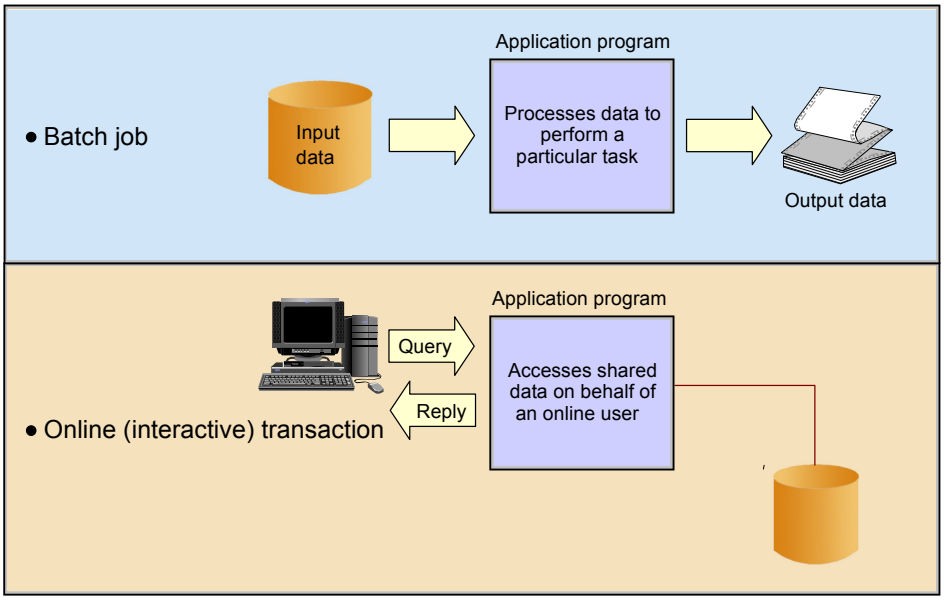


Figure 1-1 Typical mainframe workloads

These workloads are discussed in several chapters in this course; the following sections provide an overview.

1.8.1 Batch processing

One key advantage of mainframe systems is their ability to process terabytes of data from high-speed storage devices and produce valuable output. For example, mainframe systems make it possible for banks and other financial institutions to perform end-of-quarter processing and produce reports that are necessary to customers (for example, quarterly stock statements or pension statements) or to the government (for example, financial results). With mainframe systems, retail stores can generate and consolidate nightly sales reports for review by regional sales managers.

The applications that produce these statements are *batch* applications; that is, they are processed on the mainframe without user interaction. A *batch job* is submitted on the computer, reads and processes data in bulk —perhaps terabytes of data— and produces output, such as customer billing statements. An equivalent concept can be found in a UNIX script file or a Windows command file, but a z/OS batch job might process millions of records.

While batch processing is possible on distributed systems, it is not as commonplace as it is on mainframes because distributed systems often lack:

- Sufficient data storage
- Available processor capacity, or *cycles*
- Sysplex-wide management of system resources and job scheduling

Mainframe operating systems are typically equipped with sophisticated job scheduling software that allows data center staff to submit, manage, and track the execution and output of batch jobs[6].

Batch processes typically have the following characteristics:

- Large amounts of input data are processed and stored (perhaps terabytes or more), large numbers of records are accessed, and a large volume of output is produced.
- Immediate response time is usually not a requirement. However, batch jobs often must complete within a "batch window," a period of less-intensive online activity, as prescribed by a *service level agreement* (SLA).
- Information is generated about large numbers of users or data entities (for example, customer orders or a retailer's stock on hand).
- A scheduled batch process can consist of the execution of hundreds or thousands of jobs in a pre-established sequence.

During batch processing, multiple types of work can be generated. Consolidated information such as profitability of investment funds, scheduled database backups, processing of daily orders, and updating of inventories are common examples. Figure 1-2 shows a number of batch jobs running in a typical mainframe environment.

In Figure 1-2, consider the following elements at work in the scheduled batch process:

1. At night, numerous batch jobs running programs and utilities are processed. These jobs consolidate the results of the online transactions that take place during the day.
2. The batch jobs generate reports of business statistics.
3. Backups of critical files and databases are made before and after the batch window.
4. Reports with business statistics are sent to a specific area for analysis the next day.
5. Reports with exceptions are sent to the branch offices.
6. Monthly account balance reports are generated and sent to all bank customers.
7. Reports with processing summaries are sent to the partner credit card company.

[6] In the early days of the mainframe, punched cards were often used to enter jobs into the system for execution. "Keypunch operators" used card punches to enter data, and decks of cards (or batches) were produced. These were fed into card readers, which read the jobs and data into the system. As you can imagine, this process was cumbersome and error-prone. Nowadays, it is possible to transfer the equivalent of punched card data to the mainframe in a PC course file. We discuss various ways of introducing work into the mainframe in Chapter 5, "Batch processing and JES" on page 135.

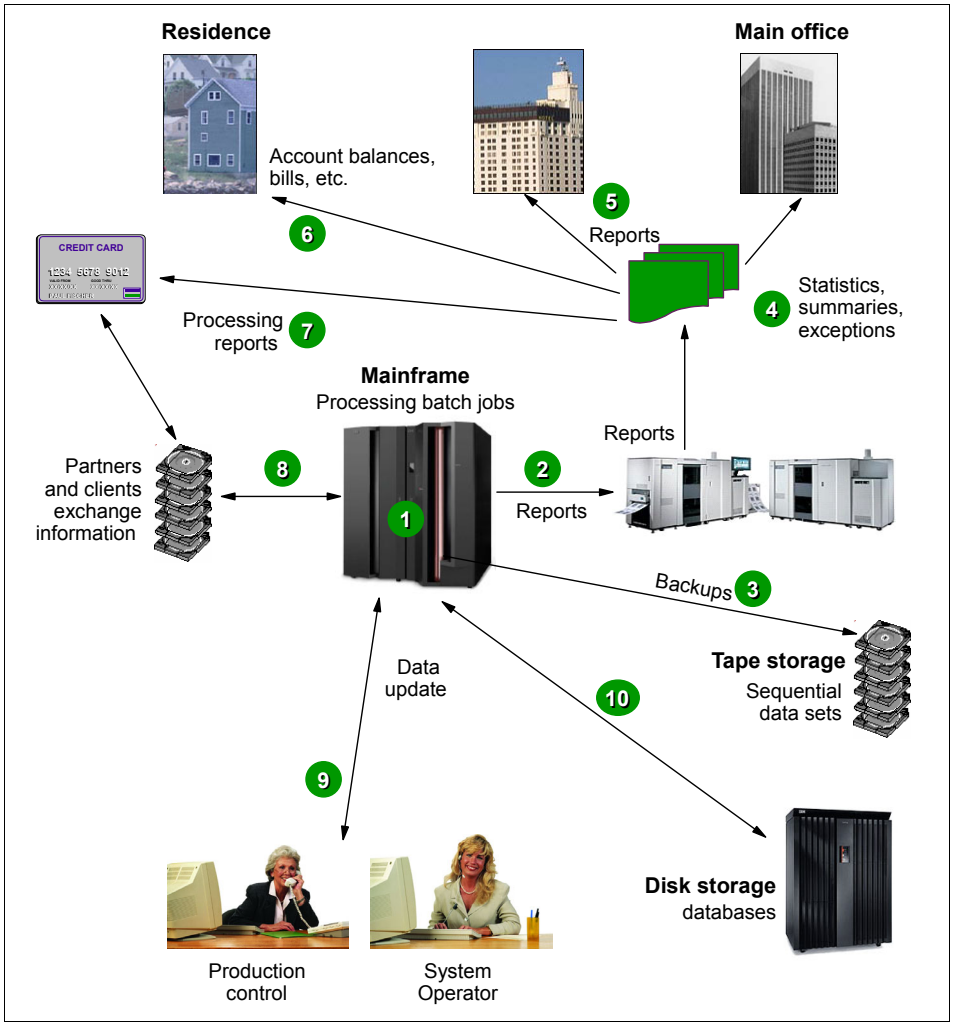


Figure 1-2 Typical batch use

8. A credit card transaction report is received from the partner company.
9. In the production control department, the operations area is monitoring the messages on the system console and the execution of the jobs.
10. Jobs and transactions are reading or updating the database (the same one that is used by online transactions) and many files are written to tape.

1.8.2 Online transaction processing

Transaction processing that occurs interactively with the end user is referred to as *Online transaction processing* or *OLTP*. Typically, mainframes serve a vast number of *transaction systems*. These systems are often mission-critical applications that businesses depend on for their core functions. Transaction systems must be able to support an unpredictable number of concurrent users and transaction types. Most transactions are executed in short time periods—fractions of a second in some cases.

One of the main characteristics of a transaction system is that the interactions between the user and the system are very short. The user will perform a complete business transaction through short interactions, with immediate response time required for each interaction. These systems are currently supporting mission-critical applications; therefore, continuous availability, high performance, and data protection and integrity are required.

Online transactions are familiar to most people. Examples include:

- ATM machine transactions such as deposits, withdrawals, inquiries, and transfers
- Supermarket payments with debit or credit cards
- Purchase of merchandise over the Internet

For example, inside a bank branch office or on the Internet, customers are using online services when checking an account balance or directing fund balances.

In fact, an online system performs many of the same functions as an operating system:

- Managing and dispatching tasks
- Controlling user access authority to system resources
- Managing the use of memory
- Managing and controlling simultaneous access to data files
- Providing device independence

Some industry uses of mainframe-based online systems include:

- Banks—ATMs, teller systems for customer service
- Insurance—agent systems for policy management and claims processing
- Travel and transport—airline reservation systems
- Manufacturing—inventory control, production scheduling
- Government—tax processing, license issuance and management

How might the end users in these industries interact with their mainframe systems? Multiple factors can influence the design of a company's transaction processing system, including:

- Number of users interacting with the system at any one time.
- Number of *transactions per second* (TPS).

- Availability requirements of the application. For example, must the application be available 24 hours a day, seven days a week, or can it be brought down briefly one night each week?

Before personal computers and intelligent workstations became popular, the most common way to communicate with online mainframe applications was with 3270 terminals. These devices were sometimes known as “dumb” terminals, but they had enough intelligence to collect and display a full screen of data rather than interacting with the computer for each keystroke, saving processor cycles. The characters were green on a black screen, so the mainframe applications were nicknamed “green screen” applications.

Based on these factors, user interactions vary from installation to installation. With applications now being designed, many installations are reworking their existing mainframe applications to include Web browser-based interfaces for users. This work sometimes requires new application development, but can often be done with vendor software purchased to “re-face” the application. Here, the end user often does not realize that there is a mainframe behind the scenes.

In this course, there is no need to describe the process of interacting with the mainframe through a Web browser, as it is exactly the same as any interaction a user would have through the Web. The only difference is the machine at the other end!

Online transactions usually have the following characteristics:

- A small amount of input data, a few stored records accessed and processed, and a small amount of data as output
- Immediate response time, usually less than one second
- Large numbers of users involved in large numbers of transactions
- Round-the-clock availability of the transactional interface to the user
- Assurance of security for transactions and user data

In a bank branch office, for example, customers use online services when checking an account balance or making an investment.

Figure 1-3 shows a series of common online transactions using a mainframe.

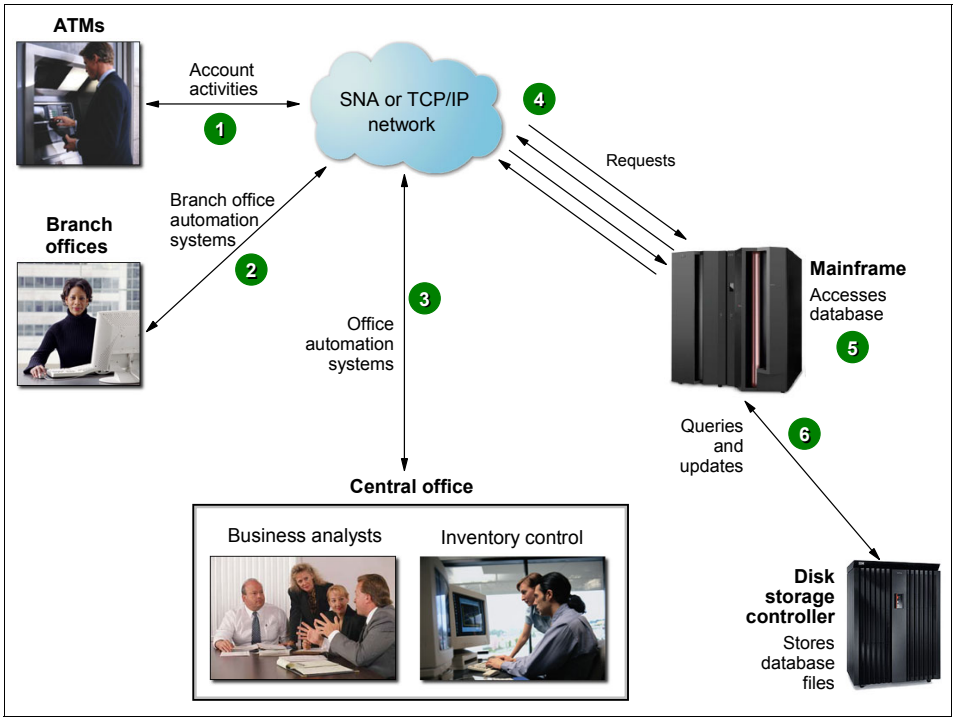


Figure 1-3 Typical online use

1. A customer uses an ATM, which presents a user-friendly interface for various functions: Withdrawal, query account balance, deposit, transfer, or cash advance from a credit card account.
2. Elsewhere in the same private network, a bank employee in a branch office performs operations such as consulting, fund applications, and money ordering.
3. At the bank's central office, business analysts tune transactions for improved performance. Other staff use specialized online systems for office automation to perform customer relationship management, budget planning, and stock control.
4. All requests directed to the mainframe computer for processing.
5. Programs running on the mainframe computer perform updates and inquires to the database management system (for example, DB2).
6. Specialized disk storage systems store the database files.

1.9 Roles in the mainframe world

Mainframe systems are designed to be used by large numbers of people. Most of those who interact with mainframes are end users— people who use the applications that are hosted on the system. However, because of the large number of end users, applications running on the system, and the sophistication and complexity of the system software that supports the users and applications, a variety of roles are needed to operate and support the system.

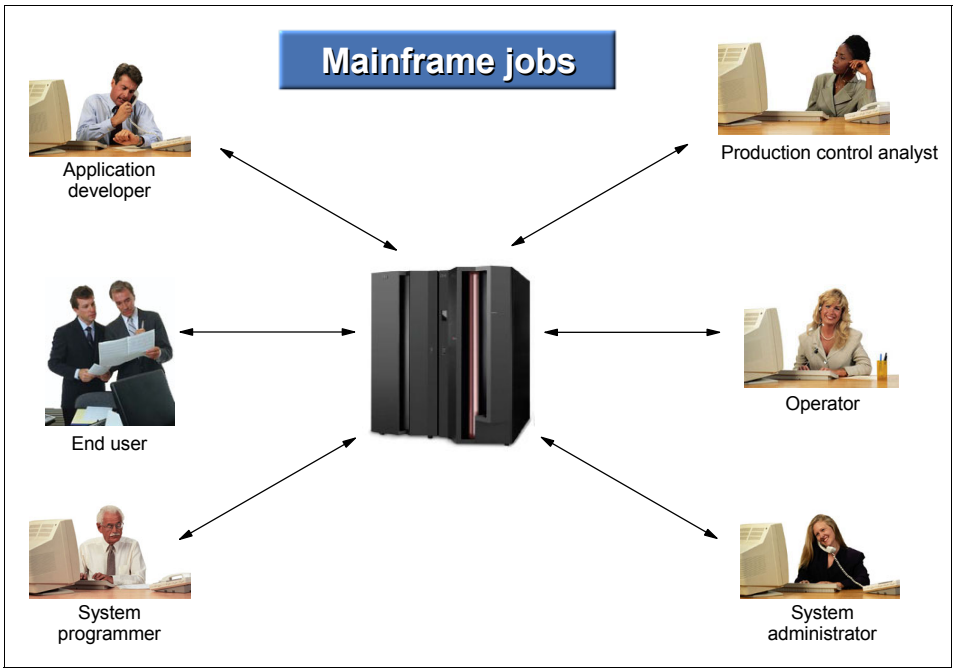


Figure 1-4 Who's who in the mainframe world?

In the IT field, these roles are referred to by a number of different titles; this course uses the following:

- System programmers
- System administrators
- Application designers and programmers
- System operators
- Production control analysts

In a distributed systems environment, many of the same roles are needed as in the mainframe environment. However, the job responsibilities are often not as well-defined. Since the 1960s, mainframe roles have evolved and expanded to provide an environment in which the system software and applications can function smoothly and effectively and

serve many thousands of users efficiently. While it may seem that the size of the mainframe support staff is large and unwieldy, the numbers become comparatively small when one considers the number of users supported, the number of transactions run, and the high business value of the work that is performed on the mainframe.

This course is concerned mainly with the system programmer and application programmer roles in the mainframe environment. There are, however, several other important jobs involved in the “care and feeding” of the mainframe, and we touch on some of these roles to give you a better idea of what’s going on behind the scenes.

Mainframe activities, such as the following, often require cooperation among the various roles:

- Installing and configuring system software
- Designing and coding new applications to run on the mainframe
- Introduction and management of new workloads on the system, such as batch jobs and online transaction processing
- Operation and maintenance of the mainframe software and hardware

In the following sections, we describe each role in more detail.

1.9.1 Who is the system programmer?

In a mainframe IT organization, the *system programmer* (or *systems programmer*) plays a central role. The system programmer installs, customizes, and maintains the operating system, and also installs or upgrades products that run on the system. The system programmer might be presented with the latest version of the operating system to upgrade the existing systems. Or, the installation might be as simple as upgrading a single program, such as a sort application.

The system programmer performs such tasks as the following:

- Planning hardware and software system upgrades and changes in configuration
- Training system operators and application programmers
- Automating operations
- Capacity planning
- Running installation jobs and scripts
- Performing installation-specific customization tasks
- Integration-testing the new products with existing applications and user procedures
- System-wide performance tuning to meet required levels of service

The system programmer must be skilled at debugging problems with system software. These problems are often captured in a copy of the computer's memory contents called a *dump*, which the system produces in response to a failing software product, user job, or transaction. Armed with a dump and specialized debugging tools, the system programmer can determine where the components have failed. When the error has occurred in a software product, the system programmer works directly with the software

vendor's support representatives to discover whether the problem's cause is known and whether a patch is available.

System programmers are needed to install and maintain the *middleware* on the mainframe, such as database management systems, online transaction processing systems and Web servers. Middleware is a software "layer" between the operating system and the end user or end user application. It supplies major functions that are not provided by the operating system. Major middleware products such as DB2, CICS, and IMS™ can be as complex as the operating system itself, if not more so.

1.9.2 Who is the system administrator?

The distinction between *system programmer* and *system administrator* varies widely among mainframe sites. In smaller IT organizations, where one person might be called upon to perform several roles, the terms may be used interchangeably.

In larger IT organizations with multiple departments, the job responsibilities tend to be more clearly separated. System administrators perform more of the day-to-day tasks related to maintaining the critical business data that resides on the mainframe, while the system programmer focuses on maintaining the system itself. One reason for the separation of duties is to comply with auditing procedures, which often require that no one person in the IT organization be allowed to have unlimited access to sensitive data or resources. Examples of system administrators include the database administrator (DBA) and the security administrator.

While system programmer expertise lies mainly in the mainframe hardware and software areas, system administrators are more likely to have experience with the applications. They often interface directly with the application programmers and end users to make sure that the administrative aspects of the applications are met. These roles are not necessarily unique to the mainframe environment, but they are key to its smooth operation nonetheless.

In larger IT organizations, the system administrator maintains the system software environment for business purposes, including the day-to-day maintenance of systems to keep them running smoothly. For example, the database administrator must ensure the integrity of, and efficient access to, the data that is stored in the database management systems.

Other examples of common system administrator tasks can include:

- Installing software
- Adding and deleting users and maintaining user profiles
- Maintaining security resource access lists
- Managing storage devices and printers
- Managing networks and connectivity
- Monitoring system performance

In matters of problem determination, the system administrator generally relies on the software vendor support center personnel to diagnose problems, read dumps, and identify corrections for cases in which these tasks aren't performed by the system programmer.

1.9.3 Who are the application designers and programmers?

The *application designer* and *application programmer* (or *application developer*) design, build, test, and deliver mainframe applications for the company's end users and customers. Based on requirements gathered from business analysts and end users, the designer creates a design specification from which the programmer constructs an application. The process includes several iterations of code changes and compilation, application builds, and unit testing.

During the application development process, the designer and programmer must interact with other roles in the enterprise. For example, the programmer often works on a team of other programmers who are building code for related application program modules. When completed, each module is passed through a testing process that can include function, integration, and system-wide tests. Following the tests, the application programs must be acceptance tested by the user community to determine whether the code actually satisfies the original user requirement.

In addition to creating new application code, the programmer is responsible for maintaining and enhancing the company's existing mainframe applications. In fact, this is often the primary job for many of today's mainframe application programmers. While mainframe installations still create new programs with Common Business Oriented Language (COBOL) or PL/I, languages such as Java™ have become popular for building new applications on the mainframe, just as they have on distributed platforms.

Widespread development of mainframe programs written in high-level languages such as COBOL and PL/I continues at a brisk pace, despite rumors to the contrary. Many thousands of programs are in production on mainframe systems around the world, and these programs are critical to the day-to-day business of the corporations that use them. COBOL and other high-level language programmers are needed to maintain existing code and make updates and modifications to existing programs. Also, many corporations continue to build new application logic in COBOL and other traditional languages, and IBM continues to enhance their high-level language compilers to include new functions and features that allow those languages to continue to take advantage of newer technologies and data formats.

We will look at the roles of application designer and application programmer in more detail in Part 2 of this course.

1.9.4 Who is the system operator?

The *system operator* monitors and controls the operation of the mainframe hardware and software. The operator starts and stops system tasks, monitors the system consoles for unusual conditions, and works with the system programming and production control staff to ensure the health and normal operation of the systems.

Console messages can be so voluminous that operators often have a difficult time determining whether a situation is really a problem. In recent years, tools to reduce the volume of messages and automate message responses to routine situations have made it easier for operators to concentrate on unusual events that might require human intervention.

The operator is also responsible for starting and stopping the major subsystems, such as transaction processing systems, database systems, and the operating system itself. These *restart operations* are not nearly as commonplace as they once were, as the availability of the mainframe has improved dramatically over the years. However, the operator must still perform an orderly shutdown and startup of the system and its workloads, when it is required.

In case of a failure or an unusual situation, the operator communicates with system programmers, who assist the operator in determining the proper course of action, and with the production control analyst, who works with the operator to make sure that production workloads are completing properly.

1.9.5 Who is the production control analyst?

The *production control analyst* is responsible for making sure that batch workloads run to completion—without error or delay. Some mainframe installations run interactive workloads for online users, followed by batch updates that run after the prime shift when the online systems are not running. While this execution model is still common, world-wide operations at many companies—with live, Internet-based access to production data—are finding the “daytime online/night time batch” model to be obsolete. Batch workloads continue to be a part of information processing, however, and skilled production control analysts play a key role.

A common complaint about mainframe systems is that they are inflexible and hard to work with, specifically in terms of implementing changes. The production control analyst often hears this type of complaint, but understands that the use of well-structured rules and procedures to control changes— a strength of the mainframe environment— helps to prevent outages. In fact, one reason that mainframes have attained a strong reputation for high levels of availability and performance is that there are controls on change and it is difficult to introduce change without proper procedures.

1.9.6 What role do vendors play?

A number of vendor roles are commonplace in the mainframe shop. Because most mainframe computers are sold by IBM, and the operating systems and primary online systems are also provided by IBM, most vendor contacts are IBM employees. However, *independent software vendor* (ISV) products are also used in the IBM mainframe environment, and customers use *original equipment manufacturer* (OEM) hardware, such as disk and tape storage devices, as well.

Typical vendor roles follow:

- *Hardware support or customer engineer*

  Hardware vendors usually provide on-site support for hardware devices. The IBM hardware maintenance person is often referred to as the *customer engineer* (CE). The CE provides installation and repair service for the mainframe hardware and peripherals. The CE usually works directly with the operations teams when hardware fails or new hardware is being installed.

- *Software support*

  A number of vendor roles exist to support software products on the mainframe[7]. IBM has a centralized *Support Center* that provides entitled and extra-charge support for software defects or usage assistance. There are also information technology specialists and architects who can be engaged to provide additional pre- and post-sales support for software products, depending upon the size of the enterprise and the particular customer situation.

- *Field technical sales support*, *systems engineer*, and *client representative*

  For larger mainframe accounts, IBM and other vendors provide face-to-face sales support. The vendor representatives specialize in various types of hardware or software product families and call on the part of the customer organization that influences the product purchases. At IBM, the technical sales specialist is referred to as the *field technical sales support* (FTSS) person, or by the older term, *systems engineer* (SE).

  For larger mainframe accounts, IBM frequently assigns a *client representative,* who is attuned to the business issues of a particular industry sector, to work exclusively with a small number of customers. The client representative acts as the general “single point of contact” between the customer and the different organizations within IBM.

[7] This course does not examine the marketing and pricing of mainframe software. However, the availability and pricing of middleware and other licensed programs is a critical factor affecting the growth and use of mainframes.

1.10 z/OS and other mainframe operating systems

Much of this course is concerned with teaching you the fundamentals of z/OS, which is IBM's foremost mainframe operating system. We begin discussing z/OS concepts in Chapter 2, "z/OS overview" on page 31. It is useful for mainframe students, however, to have a working knowledge of other mainframe operating systems. One reason is that a given mainframe computer might run multiple operating systems. For example, the use of z/OS, z/VM®, and Linux on the same mainframe is common.

Mainframe operating systems are sophisticated products with substantially different characteristics and purposes, and each could justify a separate book for a detailed introduction. Besides z/OS, four other operating systems dominate mainframe usage: z/VM, z/VSE™, Linux for zSeries, and z/TPF.

1.10.1 z/VM

z/Virtual Machine (z/VM) has two basic components: a c*ontrol program* (CP) and a single-user operating system, CMS. As a control program, z/VM is a *hypervisor* because it runs other operating systems in the virtual machines it creates. Any of the IBM mainframe operating systems such as z/OS, Linux for zSeries, z/VSE, and z/TPF can be run as *guest systems* in their own virtual machines, and z/VM can run any combination of guest systems.

The control program artificially creates multiple virtual machines from the real hardware resources. To end users, it appears as if they have dedicated use of the shared real resources. The shared real resources include printers, disk storage devices, and the CPU. The control program ensures data and application security among the guest systems. The real hardware can be shared among the guests, or dedicated to a single guest for performance reasons. The system programmer allocates the real devices among the guests. For most customers, the use of guest systems avoids the need for larger hardware configurations.

z/VM's other major component is the *Conversational Monitor System* or CMS. This component of z/VM runs in a virtual machine and provides both an interactive end user interface and the general z/VM application programming interface.

1.10.2 z/VSE

Virtual Storage Extended (z/VSE) is popular with users of smaller mainframe computers. Some of these customers eventually migrate to z/OS when they grow beyond the capabilities of z/VSE.

Compared to z/OS, the z/VSE operating system provides a smaller, less complex base for batch processing and transaction processing. The design and management structure of z/VSE is excellent for running routine production workloads consisting of multiple batch

jobs (running in parallel) and extensive, traditional transaction processing. In practice, most z/VSE users also have the z/VM operating system and use this as a general terminal interface for z/VSE application development and system management.

z/VSE was originally known as *Disk Operating System* (DOS), and was the first disk-based operating system introduced for the System/360 mainframe computers. DOS was seen as a temporary measure until OS/360 would be ready. However, some mainframe customers liked its simplicity (and small size) and decided to remain with it after OS/360 became available. DOS became known as DOS/VS (when it started using virtual storage), then VSE/SP and later VSE/ESA™, and most recently z/VSE. The name VSE is often used collectively to refer to any of the more recent versions.

1.10.3 Linux for zSeries

Several (non-IBM) Linux distributions can be used on a mainframe. There are two generic names for these distributions:

- Linux for S/390 (uses 31-bit addressing and 32-bit registers)
- Linux for zSeries (uses 64-bit addressing and registers)

The phrase *Linux on zSeries* is used to refer to Linux running on an S/390 or zSeries system, when there is no specific need to refer explicitly to either the 31-bit version or the 64-bit version. We assume students are generally familiar with Linux and therefore we mention only those characteristics that are relevant for mainframe usage. These include the following:

- Linux uses traditional count key data (CKD) disk devices and SAN-connected SCSI type devices. Other mainframe operating systems can recognize these drives as Linux drives, but cannot use the data formats on the drives. That is, there is no sharing of data between Linux and other mainframe operating systems.
- Linux does not use 3270 display terminals, while all other mainframe operating systems use 3270s as their basic terminal architecture.[8] Linux uses X Window System based terminals or X-Window System emulators on PCs; it also supports typical ASCII terminals, usually connected through the **telnet** protocol. The X-Window System is the standard for graphical interfaces in Linux. It is the middle layer between the hardware and the window manager.
- With the proper setup, a Linux system under z/VM can be quickly cloned to make another, separate Linux image. The z/VM emulated LAN can be used to connect multiple Linux images and to provide an external LAN route for them. Read-only file systems, such as a typical **/usr** file system, can be shared by Linux images.
- Linux on a mainframe operates with the ASCII character set, not the EBCDIC[9] form of stored data that is typically used on mainframes. Here, EBCDIC is used only when writing to such character-sensitive devices as displays and printers. The Linux drivers for these devices handle the character translation.

[8] There is a Linux driver for minimal 3270 operation, in very restrictive modes, but this is not commonly used.

1.10.4 z/TPF

The z/*Transaction Processing Facility* (z/TPF) operating system is a special-purpose system that is used by companies with very high transaction volume, such as credit card companies and airline reservation systems. z/TPF was once known as Airline Control Program (ACP). It is still used by airlines and has been extended for other very large systems with high-speed, high-volume transaction processing requirements.

z/TPF can use multiple mainframes in a loosely-coupled environment to routinely handle tens of thousands of transactions per second, while experiencing uninterrupted availability that is measured in years. Very large terminal networks, including special-protocol networks used by portions of the reservation industry, are common.

1.11 Summary

Today, mainframe computers play a central role in the daily operations of most of the world's largest corporations, including many Fortune 1000 companies. While other forms of computing are used extensively in business in various capacities, the mainframe occupies a coveted place in today's e-business environment. In banking, finance, health care, insurance, utilities, government, and a multitude of other public and private enterprises, the mainframe computer continues to form the foundation of modern business.

The New Mainframe owes much of its popularity and longevity to its inherent reliability and stability, a result of continuous technological advances since the introduction of the IBM System/360 in 1964. No other computer architecture in existence can claim as much continuous, evolutionary improvement, while maintaining compatibility with existing applications.

The term *mainframe* has gradually moved from a physical description of IBM's larger computers to the categorization of a style of computing. One defining characteristic of the mainframe has been a continuing compatibility that spans decades.

The roles and responsibilities in a mainframe IT organization are wide and varied. It takes skilled staff to keep a mainframe computer running smoothly and reliably. It might seem that there are far more resources needed in a mainframe environment than for small, distributed systems. But, if roles are fully identified on the distributed systems side, a number of the same roles exist there as well.

[9] EBCDIC, which stands for extended binary coded decimal interchange code, is a coded character set of 256 8-bit characters that was developed for the representation of courseual data. EBCDIC is not compatible with ASCII character coding.

Several operating systems are currently available for mainframes. This course concentrates on one of these, z/OS. However, mainframe students should be aware of the existence of the other operating systems and understand their positions relative to z/OS.

Key terms in this chapter				
architecture	availability	batch processing	compatibility	e-business
mainframe	online transaction processing (OLTP)	platform	production control analyst	scalabilty
server farm	system administrator	system operator	system programmer	System/360

2

z/OS overview

Objective: As the newest member of your company's mainframe IT group, you will need to know the basic functional characteristics of the mainframe operating system. The operating system taught in this course is z/OS, a widely used mainframe operating system. z/OS is known for its ability to serve thousands of users concurrently and for processing very large workloads in a secure, reliable, and expedient manner.

After completing this chapter, you will be able to:

- Give examples of how z/OS differs from a single-user operating system.
- List the major types of storage used by z/OS.
- Explain the concept of virtual storage and its use in z/OS.
- State the relationship between pages, frames, and slots.
- List several defining characteristics of the z/OS operating system.
- List several software products used with z/OS to provide a complete system.
- Describe several differences and similarities between the z/OS and UNIX operating systems.

© Copyright IBM Corp. 2005, 2006. All rights reserved.

2.1 What is an operating system?

In simplest terms, an *operating system* is a collection of programs that manage the internal workings of a computer system. Operating systems are designed to make the best use of the computer's various resources, and ensure that the maximum amount of work is processed as efficiently as possible. Although an operating system cannot increase the speed of a computer, it can maximize its use, thereby making the computer seem faster by allowing it to do more work in a given period of time.

A computer's *architecture* consists of the functions the computer system provides. The architecture is distinct from the physical design, and, in fact, different machine designs might conform to the same computer architecture. In a sense, the architecture is the computer as seen by the user, such as a system programmer. For example, part of the architecture is the set of machine instructions that the computer can recognize and execute.

In a mainframe environment, the system software and hardware comprise a highly advanced computer architecture, the result of decades of continuous technological innovation.

2.2 Overview of z/OS facilities

An extensive set of system facilities and unique attributes makes z/OS well suited for processing large, complex workloads, such as those that require many I/O operations, access to large amounts of data, or comprehensive security. Typical mainframe workloads include long-running applications that update millions of records in a database and online applications that can serve many thousands of users concurrently.

Figure 2-1 provides a "snapshot" view of the z/OS operating environment.

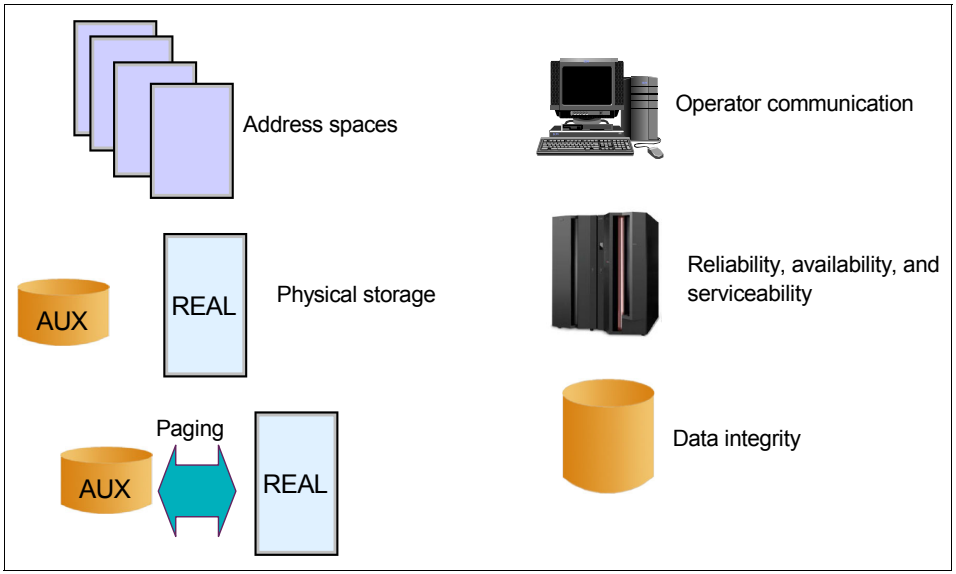


Figure 2-1 z/OS operating environment

These facilities are explored in greater depth in the remaining portions of this course, but are summarized here as follows:

- An address space describes the virtual storage addressing range available to an online user or a running program.
- Two types of physical storage are available: real storage and auxiliary storage (AUX). Real storage is also referred to as real memory or central storage.
- z/OS moves programs and data between real storage and auxiliary storage through processes called paging and swapping.
- z/OS dispatches work for execution (not shown in the figure). That is, it selects programs to be run based on priority and ability to execute and then loads the program and data into real storage. All program instructions and data must be in real storage when executing.
- An extensive set of facilities manages files stored on direct access storage devices (DASDs) or tape cartridges.
- Operators use consoles to start and stop z/OS, enter commands, and manage the operating system.

z/OS is further defined by many other operational characteristics, such as security, recovery, data integrity and workload management.

2.3 What is z/OS?

The operating system we discuss in this course is z/OS[1], a widely used mainframe operating system. z/OS is designed to offer a stable, secure, and continuously available environment for applications running on the mainframe.

z/OS today is the result of decades of technological advancement. z/OS evolved from an operating system that could process a single program at a time to an operating system that can handle many thousands of programs and interactive users concurrently. To understand how and why z/OS functions as it does, it is important to understand some basic concepts about z/OS and the environment in which it functions. This chapter introduces some of the concepts that you will need to understand the z/OS operating system.

In most early operating systems, requests for work entered the system one at a time. The operating system processed each request or *job* as a unit, and did not start the next job until the one ahead of it had completed. This arrangement worked well when a job could execute continuously from start to completion. But often a job had to wait for information to be read in from, or written out to a device such as a tape drive or a printer. Input and output (I/O) take a long time compared to the electronic speed of the processor. When a job waited for I/O, the processor was idle.

Finding a way to keep the processor working while a job waited would increase the total amount of work the processor could do without requiring additional hardware. z/OS gets work done by dividing it into pieces and giving portions of the job to various system components and subsystems that function interdependently. At any point in time, one component or another gets control of the processor, makes its contribution, and then passes control along to a user program or another component.

2.3.1 Hardware resources used by z/OS

The z/OS operating system executes in a processor and resides in processor storage during execution. z/OS is commonly referred to as the *system* software.

Mainframe hardware consists of processors and a multitude of peripheral devices such as disk drives (called direct access storage devices or *DASD*), magnetic tape drives, and various types of user consoles; see Figure 2-2. Tape and DASD are used for system functions and by user programs executed by z/OS.

[1] z/OS is designed to take advantage of the IBM zSeries architecture, or z/Architecture, which was introduced in 2000.

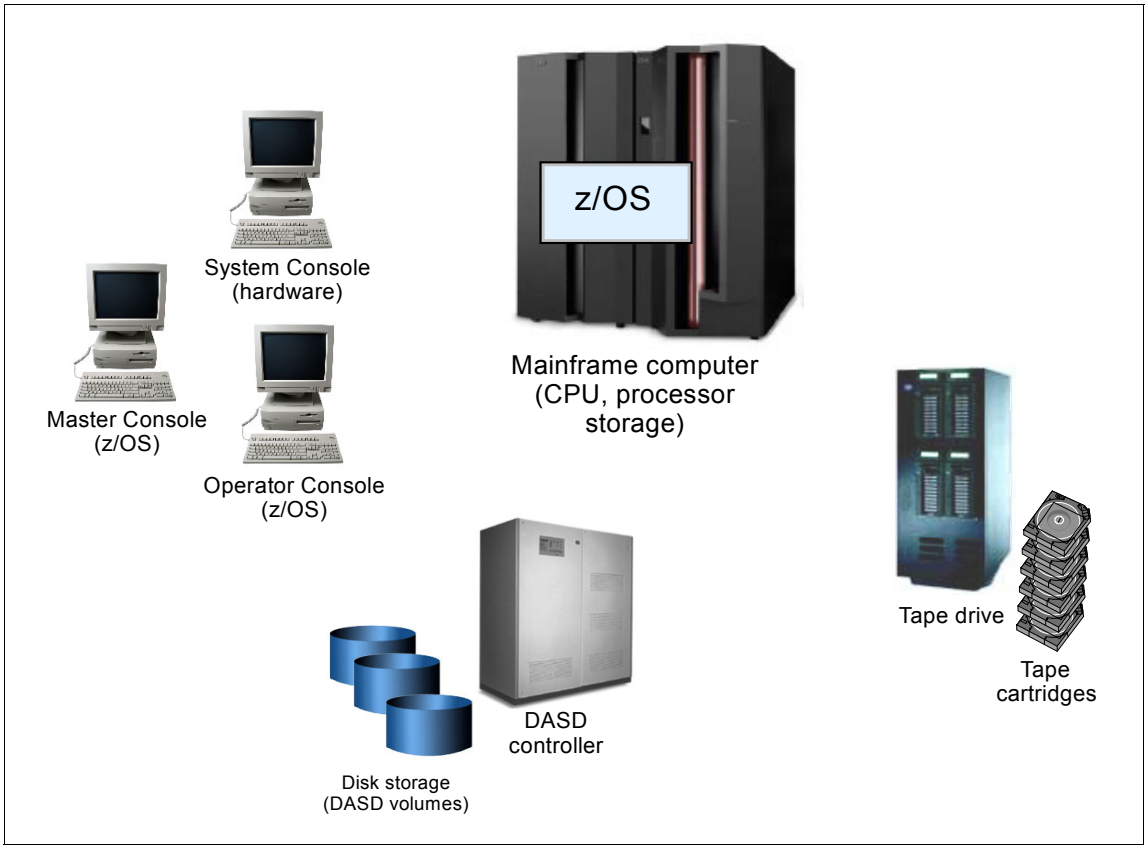


Figure 2-2 Hardware resources used by z/OS

To fulfill a new order for a z/OS system, IBM ships the system code to the customer through the Internet or (depending on customer preference) on physical tape cartridges. At the customer site, a person such as the z/OS system programmer receives the order and copies the new system to DASD volumes. After the system is customized and ready for operation, system consoles are required to start and operate the z/OS system.

The z/OS operating system is designed to make full use of the latest IBM mainframe hardware and its many sophisticated peripheral devices. Figure 2-2 on page 35 presents a simplified view of mainframe concepts that students will build upon throughout this course:

- Software - The z/OS operating system consists of load modules or *executable code.* During the install process, the system programmer copies these load modules to *load libraries* residing on DASD volumes.
- Hardware - The system hardware consists of all the devices, controllers, and processors that constitute a mainframe environment.

- Peripheral devices - These include tape drives, DASD, and consoles. There are many other types of devices, some of which are discussed later in the course.
- Processor storage - Often called real or central storage (or memory), this is where the z/OS operating system executes. Also, all user programs share the use of processor storage with the operating system.

As a "Big Picture" of a typical mainframe hardware configuration, Figure 2-2 is far from complete. Not shown, for example, are the hardware control units that connect the mainframe to the other tape drives, DASD, and consoles.

While this course is primarily concerned with teaching the z/OS system software, it is important for students to understand that many z/OS design characteristics exist to take advantage of ongoing mainframe hardware innovations.

Related Reading: The standard reference for descriptions of the major facilities of z/Architecture is the IBM publication *z/Architecture Principles of Operation*. You can find this and other related publications at the z/OS Internet Library Web site:

http://www.ibm.com/servers/eserver/zseries/zos/bkserv/

2.3.2 Multiprogramming and multiprocessing

The earliest operating systems were used to control single user computer systems. In those days, the operating system would read in one job, find the data and devices the job needed, let the job run to completion, and then read in another job. In contrast, the computer systems that z/OS manages are capable of *multiprogramming,* or executing many programs concurrently. With multiprogramming, when a job cannot use the processor, the system can suspend, or *interrupt*, the job, freeing the processor to work on another job.

z/OS makes multiprogramming possible by capturing and saving all the relevant information about the interrupted program before allowing another program to execute. When the interrupted program is ready to begin executing again, it can resume execution just where it left off. Multiprogramming allows z/OS to run thousands of programs simultaneously for users who might be working on different projects at different physical locations around the world.

z/OS can also perform *multiprocessing,* which is the simultaneous operation of two or more processors that share the various hardware resources, such as memory and external disk storage devices.

The techniques of multiprogramming and multiprocessing make z/OS ideally suited for processing workloads that require many I/O operations. Typical mainframe workloads include long-running applications that write updates to millions of records in a database, and online applications for thousands of interactive users at any given time.

By way of contrast, consider the operating system that might be used for a single-user computer system. Such an operating system would need to execute programs on behalf of one user only. In the case of a personal computer (PC), for example, the entire resources of the machine are often at the disposal of one user.

Many users running many separate programs means that, along with large amounts of complex hardware, z/OS users need large amounts of memory to ensure suitable system performance. Large companies run sophisticated business applications that access large databases and industry-strength middleware products. Such applications require the operating system to protect privacy among users, as well as enable the sharing of databases and software services.

Thus, multiprogramming, multiprocessing, and the need for a large amount of memory means that z/OS must provide function beyond simple, single-user applications. The sections that follow explain, in a general way, the attributes that enable z/OS to manage complex computer configurations. Subsequent portions of this course explore these features in more detail.

2.3.3 Modules and macros

z/OS is made up of programming instructions that control the operation of the computer system. These instructions ensure that the computer hardware is being used efficiently and is allowing application programs to run. z/OS includes sets of instructions that, for example, accept work, convert work to a form that the computer can recognize, keep track of work, allocate resources for work, execute work, monitor work, and handle output.

A group of related instructions is called a *routine* or *module*. A set of related modules that make a particular system function possible is called a *system component*. The workload management (WLM) component of z/OS, for instance, controls system resources, while the recovery termination manager (RTM) handles recovery for the system.

Sequences of instructions that perform frequently used system functions can be invoked with executable macro instructions, or *macros*. z/OS macros exist for functions such as opening and closing data files, loading and deleting programs, and sending messages to the computer operator.

2.3.4 Control blocks

As instructions execute the work of a computer system, they keep track of this work in storage areas known as *control blocks*. Generally speaking, there are four types of z/OS control blocks:

- System-related control blocks
- Resource-related control blocks
- Job-related control blocks

- Task-related control blocks

Each system-related control block represents one z/OS system and contains system-wide information, such as how many processors are in use. Each resource-related control block represents one resource, such as a processor or storage device. Each job-related control block represents one job executing on the system. Each task-related control block represents one unit of work.

Control blocks serve as vehicles for communication throughout z/OS. Such communication is possible because the structure of a control block is known to the programs that use it, and thus these programs can find needed information about the unit of work or resource.

Control blocks representing many units of the same type may be chained together on queues, with each control block pointing to the next one in the chain. The operating system can search the queue to find information about a particular unit of work or resource, which might be:

- An address of a control block or a required routine
- Actual data, such as a value, a quantity, a parameter, or a name
- Status flags (usually single bits in a byte, where each bit has a specific meaning)

2.3.5 Physical storage used by z/OS

Conceptually, mainframes and all other computers have two types of physical storage[2].

- The physical storage located on the mainframe processor itself. This is called processor storage or *real storage;* think of it as *memory* for the mainframe.
- The physical storage external to the mainframe, including storage on direct access devices, such as disk drives and tape drives. This storage is called *auxiliary storage.*

The primary difference between the two kinds of storage relates to the way in which it is accessed, as follows:

- Real storage is accessed synchronously with the processor. That is, the processor must wait while data is retrieved from real storage[3].
- Auxiliary storage is accessed asynchronously. The processor accesses auxiliary storage through an input/output (I/O) request, which is scheduled to run amid other

[2] Many computers also have a fast memory, local to the processor, called the processor cache. The cache is not visible to the programmer or application programs or even the operating system directly. It is beyond the scope of this book.

[3] Some processor implementations use techniques such as instruction or data pre-fetching or “pipelining” to enhance performance. These techniques are not visible to the application program or even the operating system, but a sophisticated compiler can organize the code it produces to allow the best use of these techniques.

work requests in the system. During an I/O request, the processor is free to execute other, unrelated work.

As with memory for a personal computer, mainframe real storage is tightly coupled with the processor itself, whereas mainframe auxiliary storage is located on (comparatively) slower, external disk and tape drives. Because real storage is more closely integrated with the processor, it takes the processor much less time to access data from real storage than from auxiliary storage. However, the processor is free to do other work while waiting for an I/O request to be satisfied. Auxiliary storage is less expensive than real storage, so it provides the capability for many jobs to be running while keeping real storage costs down.

2.4 Virtual storage and other mainframe concepts

z/OS uses both types of physical storage (real and auxiliary) to enable another kind of storage called *virtual storage*. In z/OS, each user has access to virtual storage, rather than physical storage.

This use of virtual storage is central to the unique ability of z/OS to interact with large numbers of users concurrently, while processing the largest workloads. To make virtual storage possible, z/OS requires sufficient amounts of real storage and auxiliary storage. z/OS uses a system of tables and special settings (bit settings) to relate the location of data on disk storage and real storage, and keep track of the identity and authority of each user or program. z/OS uses a variety of storage manager components to manage virtual storage. This chapter briefly covers the key points in the process.

2.4.1 What is virtual storage?

Virtual storage means that each running program can assume it has access to all of the storage defined by the architecture's addressing scheme. The only limit is the number of bits in a storage address. This ability to use a large number of storage locations is important because a program may be long and complex, and both the program's code and the data it requires must be in real storage for the processor to access them.

z/OS supports 64-bit long addresses, which allows a program to address up to 18,446,744,073,709,600,000 bytes (16 exabytes) of storage locations. In reality, the mainframe might have *much less* real storage installed. How much less depends on the model of computer and the system configuration.

To allow each user to act as though this much storage really exists in the computer system, z/OS keeps only the active portions of each program in real storage. z/OS keeps the rest of the code and data in files called *page data sets* on auxiliary storage, which usually consists of a number of high-speed direct access storage devices (DASDs).

Virtual storage, then, is this combination of real and auxiliary storage. z/OS uses a system of tables and indexes to relate the auxiliary storage locations to real storage locations and keep track of the identity and authority of each program. This process is shown in more detail in 2.4.4, "Virtual storage overview" on page 41.

Terms: The terms *real storage*, *real memory*, *central storage*, and *main storage* are used interchangeably. Likewise, *virtual memory* and *virtual storage* are used synonymously.

2.4.2 What is an address space?

The range of virtual addresses that the operating system assigns to a user or separately running program is called an *address space.* This is the area of contiguous virtual addresses available for executing instructions and storing data. The range of virtual addresses in an address space starts at zero and can extend to the highest address permitted by the operating system architecture.

z/OS provides each user with a unique address space and maintains the distinction between the programs and data belonging to each address space. Within each address space the user can start multiple tasks, using *task control blocks* or *TCBs* that allow user multiprogramming.

In some ways an address space in z/OS is analogous to a UNIX process and the address space identifier (ASID) is like a process ID (PID). Further, TCBs are like threads, in that the UNIX kernel supports multiple threads at once.

The use of address spaces in z/OS, however, holds some special advantages. Both z/OS and UNIX, for example, provide APIs to allow in-memory data to be shared between processes. In z/OS, a user can access another user's address spaces directly through *cross-memory services.* Similarly, UNIX has the concept of Shared Memory functions, and these can be used on UNIX without special authority.

In contrast to UNIX, z/OS cross-memory services require the issuing program to have special authority, controlled by the authorized program facility (APF). This method allows efficient and secure access to data owned by others, data owned by the user but stored in another address space for convenience, and for rapid and secure communication with services like transaction managers and database managers.

The private areas within one user's address space are isolated from the private areas within other address spaces, and this provides much of the operating system's security. Yet, each address space also contains a common area that is accessible to every other address space.

Because it maps all of the available addresses, an address space includes system code and data as well as user code and data. Thus, not all of the mapped addresses are available for user code and data.

The ability of many users to share the same resources implies the need to protect users from one another and to protect the operating system itself. Along with such methods as "keys" for protecting real storage and code words for protecting data files and programs, separate address spaces ensure that users' programs and data do not overlap.

An active z/OS system uses many address spaces. There is at least one address space for each job in progress and one address space for each user logged on through TSO, telnet, rlogin or FTP[4]. There are many address spaces for operating system functions, such as operator communication, automation, networking, security, and so on.

Each student in this course causes at least one address space to be created whenever they log on to z/OS.

2.4.3 What is dynamic address translation?

Dynamic address translation, or *DAT*, is the process of translating a virtual address during a storage reference into the corresponding real address. If the virtual address is already in real storage, the DAT process may be accelerated through the use of a translation lookaside buffer; if the virtual address is not in real storage, a page fault interrupt occurs, z/OS is notified, and z/OS brings the page in from auxiliary storage.

Looking at this process more closely reveals that the machine can present any one of a number of different types of faults. A type, region, segment, or page fault will be presented depending on at which point in the DAT structure invalid entries are found. The faults will repeat, down the DAT structure until ultimately a page fault is presented and the virtual page is brought into real storage either for the first time (there is no copy on auxiliary storage) or by bringing the page in from auxiliary storage.

DAT is implemented by both hardware and software through the use of page tables, segment tables, region tables and translation lookaside buffers. DAT allows different address spaces to share the same program or other data that is for read only. This is because virtual addresses in different address spaces can be made to translate to the same frame of real storage. Otherwise, there would have to be many copies of the program or data, one for each address space.

2.4.4 Virtual storage overview

Recall that for the processor to execute a program instruction, both the instruction and the data it references must be in real storage. The convention of early operating systems was

[4] Users logged on z/OS through a major subsystem, such as CICS or IMS, are using an address space belonging to the subsystem, not their own address spaces.

to have the entire program reside in real storage when its instructions were executing. However, the entire program does not really need to be in real storage when an instruction executes. Instead, by bringing pieces of the program into real storage only when the processor is ready to execute them—moving them out to auxiliary storage when it doesn't need them, an operating system can execute more and larger programs concurrently.

How does the operating system keep track of each program piece? How does it know whether it is in real storage or auxiliary storage, and where? It is important for z/OS professionals to understand how the operating system makes this happen.

Physical storage is divided into areas, each the same size and accessible by a unique address. In real storage, these areas are called *frames*; in auxiliary storage, they are called *slots*.

Similarly, the operating system can divide a program into pieces the size of frames or slots and assign each piece a unique address. This arrangement allows the operating system to keep track of these pieces. In z/OS, the program pieces are called *pages*. These areas are discussed further in 2.4.8, "Frames, pages, and slots" on page 46.

Pages are referenced by their virtual addresses and not by their real addresses. From the time a program enters the system until it completes, the virtual address of the page remains the same, regardless of whether the page is in real storage or auxiliary storage. Each page consists of individual locations called bytes, each of which has a unique virtual address.

2.4.5 What is paging?

z/OS maintains tables to determine whether a page is in real or auxiliary storage, and where. To find a page of a program, z/OS checks the table for the virtual address of the page, rather than searching through all of physical storage for it. z/OS then transfers the page into real storage or out to auxiliary storage as needed. This movement of pages between auxiliary storage slots and real storage frames is called *paging*. Paging is key to understanding the use of virtual storage in z/OS.

z/OS paging is transparent to the user. During job execution, only those pieces of the application that are required are brought in, or *paged in*, to real storage. The pages remain in real storage until no longer needed, or until another page is required by the same application or a higher priority application and no empty real storage is available.

To select pages for paging out to auxiliary storage, z/OS follows a "Least Used" algorithm. That is, z/OS assumes that a page that has not been used for some time will probably not be used in the near future.

2.4.6 Format of a virtual address

As mentioned, virtual storage is an illusion created by the architecture, in that the system seems to have more memory than it really has. Each user or program gets an address space, and each address space contains the same range of storage addresses. Only those portions of the address space that are needed at any one point in time are actually loaded into real storage. z/OS keeps the inactive pieces of address spaces in auxiliary storage.

z/OS manages address spaces in units of various sizes, as follows:

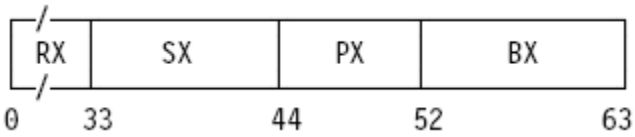
Page*:* Address spaces are divided into 4-kilobyte units of virtual storage called pages.

Segment*:* Address spaces are divided into 1-megabyte units called segments. A segment is a block of sequential virtual addresses spanning megabytes, beginning at a 1-megabyte boundary. A 2-gigabyte address space, for example, consists of 2048 segments.

Region*:* Address spaces are divided into 2-8 gigabyte units called regions. A region is a block of sequential virtual addresses spanning 2-8 gigabytes, beginning at a 2-gigabytes boundary. A 2-terabyte address space, for example, consists of 2048 regions.

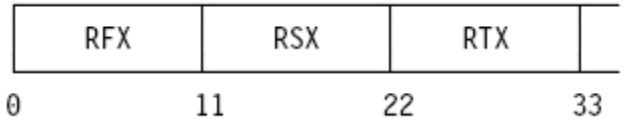
A virtual address, accordingly, is divided into four principal fields. Bits 0-32 are called the region index (RX), bits 33-43 are called the segment index (SX), bits 44-51 are called the page index (PX), and bits 52-63 are called the byte index (BX).

A virtual address has the following format:



As determined by its address-space-control element, a virtual address space can be a 2G-byte space consisting of one region, or it may be up to a 16-exabyte space consisting of up to 8G regions. The RX part of a virtual address applying to a 2G-byte address space must be all zeros; otherwise, an exception is recognized.

The RX part of a virtual address is itself divided into three fields. Bits 0-10 are called the region first index (RFX), bits 11-21 are called the region second index (RSX), and bits 22-32 are called the region third index (RTX). Bits 0-32 of the virtual address have the following format:



A virtual address in which the RTX is the left most significant part (a 42-bit address) is capable of addressing 4T bytes (2K regions), one in which the RSX is the left most significant part (a 53-bit address) is capable of addressing 8P bytes (4M regions), and one in which the RFX is the left most significant part (a 64-bit address) is capable of addressing 16 exabytes (8G regions).

2.4.7 How is an address used in virtual storage?

The use of virtual storage in z/OS means that only the pieces of a program that are currently active need to be in real storage at processing time. The inactive pieces are held in auxiliary storage.

Figure 2-3 shows the virtual storage concept at work in z/OS.

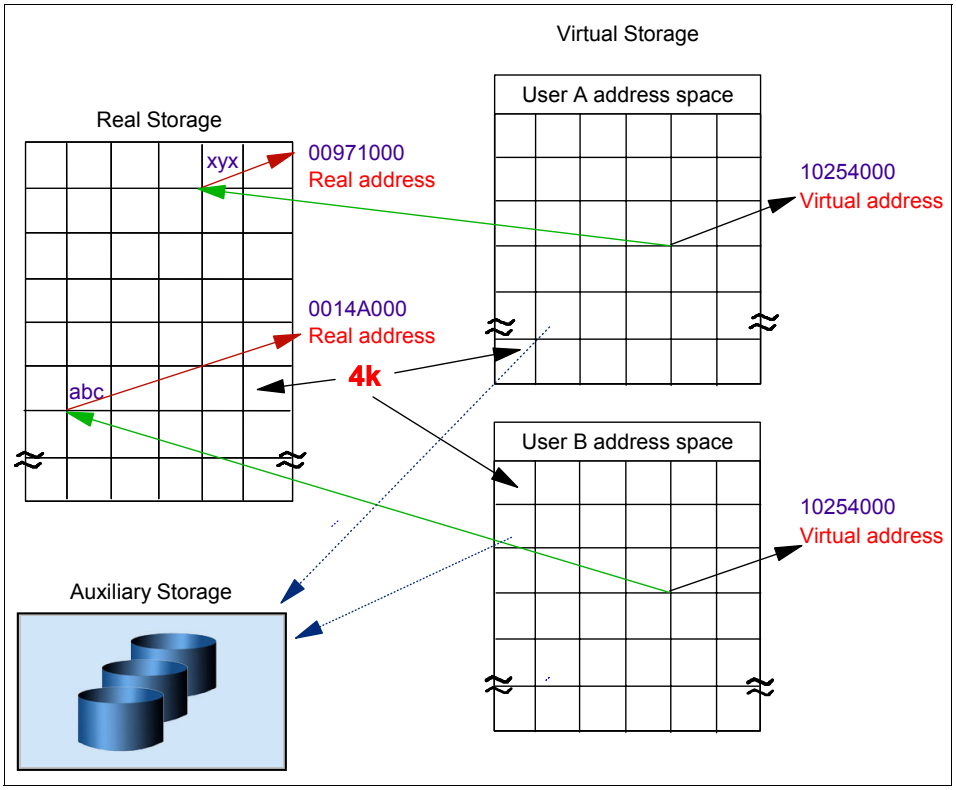


Figure 2-3 Real and auxiliary storage combine to create the illusion of virtual storage

In Figure 2-3, observe the following:

- An address is an identifier of a required piece of information, but not a description of where in real storage that piece of information is. This allows the size of an address space (that is, all addresses available to a program) to exceed the amount of real storage available.
- For most user programs, all real storage references are made in terms of virtual storage addresses. [5]
- Dynamic address translation (DAT) is used to translate a virtual address during a storage reference into a physical location in real storage. As shown in Figure 2-3, the virtual address 10254000 can exist more than once, because each virtual address maps to a different address in real storage.
- When a requested address is not in real storage, a hardware interruption is signaled to z/OS and the operating system pages in the required instructions and data to real storage.

[5] Some instructions, primarily those used by operating system programs, require real addresses.

2.4.8 Frames, pages, and slots

When a program is selected for execution, the system brings it into virtual storage, divides it into pages of 4 kilobytes (4K), transfers the pages into real storage for execution. To the programmer, the entire program appears to occupy contiguous space in storage at all times. Actually, not all pages of a program are necessarily in real storage, and the pages that *are* in real storage do not necessarily occupy contiguous space.

The pieces of a program executing in virtual storage must be moved between real and auxiliary storage. To allow this, z/OS manages storage in units, or *blocks*, of 4 kilobytes. The following blocks are defined:

- 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: 4096 bytes (4 kilobytes). An active virtual storage page resides in a real storage frame. A virtual storage page that becomes inactive resides in an auxiliary storage slot (in a paging data set). Figure 2-4 shows the relationship of pages, frames, and slots in the system.

In Figure 2-4, z/OS is performing paging for a program running in virtual storage. The lettered boxes represent parts of the program. In this simplified view, program parts A, E, F, and H are active and running in real storage frames, while parts B, C, D, and G are inactive and have been moved to auxiliary storage slots. All of the program parts, however, reside in virtual storage and have virtual storage addresses.

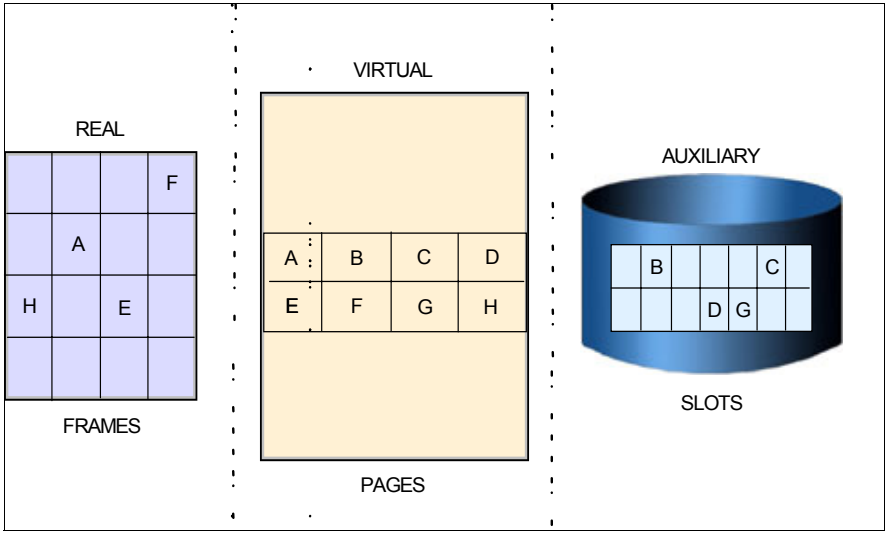


Figure 2-4 Frames, pages, and slots

2.4.9 The paging process

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 handle the movement of pages and several additional tables to keep track of the most current version of each page.

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 bring the page in from auxiliary storage. First, however, it must locate an available real storage frame. If none are available, the request must be saved and an assigned frame freed. To free a frame, the system copies its contents to auxiliary storage and marks its corresponding page table entry as invalid. This operation is called a page-out.

After a frame is located for the required page, the contents of the page are copied from auxiliary storage to real storage and the page table invalid bit is set off. This operation is called a page-in.

Paging can also take place when z/OS loads an entire program into virtual storage. z/OS obtains virtual storage for the user program, and allocates a real storage frame to each page. 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 might be paged out to auxiliary storage.

Page stealing

z/OS tries to keep an adequate supply of available real storage frames on hand. When a program refers to a page that is not in real storage, z/OS uses a real storage page frame from a supply of available frames.

When this supply becomes low, z/OS uses *page stealing* to replenish it, that is, it 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 the activity history of each page currently residing in a real storage frame. Pages that have not been active for a relatively long time are good candidates for page stealing.

Unreferenced interval count

z/OS uses a sophisticated paging algorithm to efficiently manage virtual storage based on which pages were most recently used. An unreferenced interval count indicates how long it has been since a program referenced the page. At regular intervals, the system checks the reference bit for each page frame. If the reference bit is off -- that is, the frame has not been referenced -- the system adds to the frame's unreferenced interval count. It adds the number of seconds since this address space last had the reference count checked. If the reference bit is on, the frame has been referenced and the system turns it off and sets the

unreferenced interval count for the frame to zero. Those frames with the highest unreferenced interval counts are those most likely to be stolen.

z/OS also uses various storage managers to keep track of all pages, frames, and slots in the system. These are described in 2.4.12, “Role of storage managers” on page 49.

2.4.10 Swapping and the working set

Swapping is the process of transferring all of the pages of an address space between real storage and auxiliary storage. A swapped-in address space is active, having pages in real storage frames and pages in auxiliary storage slots. A swapped-out address space is inactive; the address space resides on auxiliary storage and cannot execute until it is swapped in.

While only a subset of the address space’s pages (known as its **working set**) would likely be in real storage at any time, swapping effectively moves the entire address space. It is one of several methods that z/OS uses to balance the system workload and ensure that an adequate supply of available real storage frames is maintained.

Swapping is performed by the System Resource Manager (SRM) component, in response to recommendations from the Workload Manager (WLM) component. WLM is described in 2.5, “What is workload management?” on page 57.

2.4.11 What is storage protection?

Up to now, we’ve discussed virtual storage in the concourse of a single user or program. In reality, of course, many programs and users would be competing for the use of the system. z/OS uses the following techniques to preserve the integrity of each user’s work:

- A private address space for each user
- Page protection
- Low-address protection
- Multiple storage protect keys, as described in this section.

How storage protect keys are used

Under z/OS, 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 4K frame of real storage.

When a request is made to modify the contents of a real storage location, the key associated with the request is compared to the storage protect key. If the keys match or the program is executing in key 0, the request is satisfied. If the key associated with the request does not match the storage key, the system rejects the request and issues a program exception interruption.

When a request is made to read (or fetch) the contents of a real storage location, the request is automatically satisfied unless the fetch protect bit is on, indicating that the frame 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, or the requestor is in key 0, the request is satisfied. If the keys do not match, and the requestor is not in key 0, the system rejects the request and issues a program exception interruption.

How storage protect keys are assigned

z/OS uses sixteen different storage protect keys. A specific key is assigned according to the type of work being performed.

Storage protect keys 0 through 7 are used by the z/OS base control program (BCP) and various subsystems and middleware products. Storage protect key 0 is the master key. Its use is restricted to those parts of the BCP that require almost unlimited store and fetch capabilities. In almost any situation, a storage protect key of 0 associated with a request to access or modify the contents of a real storage location means that the request is satisfied.

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 called V=V (virtual = virtual) users and are assigned a key of 8. Some users, however, must run in a real storage region. These users are known as V=R (virtual = real) users and require individual storage protect keys because their addresses are not protected by the DAT process that keeps each address space distinct. Without separate keys, V=R users might reference each other's code and data. These keys are in the range of 9 through 15.

2.4.12 Role of storage managers

Real storage frames and auxiliary storage slots, and the virtual storage pages that they support, are managed by separate components of z/OS. You might not be surprised to learn that these components are known as the real storage manager, the auxiliary storage manager, and the virtual storage manager. In this section, we describe the role of each briefly.

Real storage manager

The **real storage manager** or **RSM** keeps track of the contents of real storage. It manages the paging activities described earlier, such as page-in, page-out, and page stealing, and helps with swapping an address space in or out. RSM also performs **page fixing** (marking pages as unavailable for stealing).

Auxiliary storage manager

The **auxiliary storage manager** or **ASM** keeps track of the contents of special use files called **page data sets**. Page data sets contain slots representing virtual storage pages that are not currently occupying a real storage frame. Page data sets also contain slots representing pages that do not occupy a frame, but, because the frame's contents have not been changed, the slots are still valid.

When a page-in or page-out is required, ASM works with RSM to locate the proper central storage frames and auxiliary storage slots.

Virtual storage manager

The **virtual storage manager** or **VSM** responds to requests to obtain and free virtual storage. VSM also manages storage allocation for any program that must run in real, rather than virtual storage. Storage is allocated to code and data when they are loaded in virtual storage. As they run, programs can request more storage by means of a system service, such as the **GETMAIN macro**. Programs can release storage with the **FREEMAIN macro**.

VSM keeps track of the map of virtual storage for each address space. In so doing, it sees an address space as a collection of 256 **subpools**, which are logically related areas of virtual storage identified by the numbers 0 to 255. Being logically related means the storage areas within a subpool share characteristics such as:

- Storage protect key
- Whether they are fetch protected, pageable, or swappable
- Where they must reside in virtual storage (above or below 16 megabytes)
- Whether they can be shared by more than one task

Some subpools (numbers 128 to 255) are predefined by use by system programs. Subpool 252, for example, is for authorized programs from authorized program libraries. Others (numbered 0 to127) are defined by user programs.

2.4.13 A brief history of virtual storage and 64-bit addressability

In 1970, IBM introduced System/370, the first of its architectures to use virtual storage and address spaces. Since that time, the operating system has changed in many ways. One key area of growth and change is addressability.

A program running in an address space can reference all of the storage associated with that address space. In this course, a program's ability to reference all of the storage associated with an address space is called *addressability*.

System/370 defined storage addresses as 24 bits in length, which meant that the highest accessible address was 16,777,215 bytes (or 2^{24}-1 bytes)[6]. The use of 24-bit

[6] Addressing starts with 0, so the last address is always one less than the total number of addressable bytes.

addressability allowed MVS/370, the operating system at that time, to allot to each user an address space of 16 megabytes. Over the years, as MVS/370 gained more functions and was asked to handle more complex applications, even access to 16 megabytes of virtual storage fell short of user needs.

With the release of the System/370-XA architecture in 1983, IBM extended the addressability of the architecture to 31 bits. With 31-bit addressing, the operating system (now called MVS Extended Architecture or MVS/XA™) increased the addressability of virtual storage from 16 MB to 2 gigabytes (2 GB). In other words, MVS/XA provided an address space for users that was 128 times larger than the address space provided by MVS/370. The 16 MB address became the dividing point between the two architectures and is commonly called the *line* (see Figure 2-5).

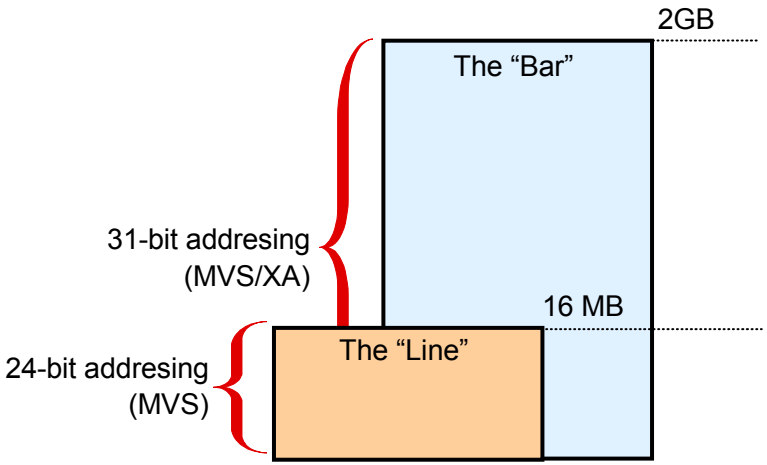


Figure 2-5 31-bit addressability allows for 2 gigabyte address spaces in MVS/XA

The new architecture did not require customers to change existing application programs. To maintain compatibility for existing programs, MVS/XA remained compatible for programs originally designed to run with 24-bit addressing on MVS/370, while allowing application developers to write new programs to exploit the 31-bit technology.

To preserve compatibility between the different addressing schemes, MVS/XA did not use the *high-order bit* of the address (Bit 0) for addressing. Instead, MVS/XA reserved this bit to indicate how many bits would be used to resolve an address: 31-bit addressing (Bit 0 on) or 24-bit addressing (Bit 0 off).

With the release of zSeries mainframes in 2000, IBM further extended the addressability of the architecture to 64 bits. With 64-bit addressing, the potential size of a z/OS address space expands to a size so vast we need new terms to describe it. Each address space, called a 64-bit address space, is 16 exabytes (EB) in size; an exabyte is slightly more than

one billion gigabytes. The new address space has logically 2^{64} addresses. It is 8 billion times the size of the former 2-gigabyte address space, or 18,446,744,073,709,600,000 bytes (Figure 2-6).

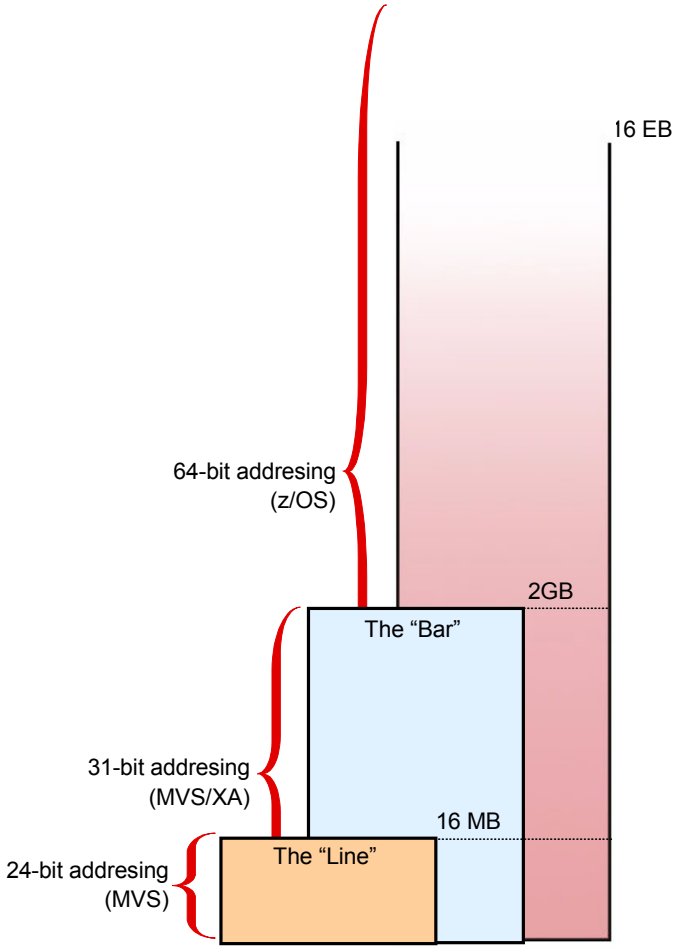


Figure 2-6 64-bit addressability allows for 16 exabytes of addressable storage

We say that the potential size is 16 exabytes because z/OS, by default, continues to create address spaces with a size of 2 gigabytes. The address space exceeds this limit only if a program running in it allocates virtual storage above the 2-gigabyte address. If so, the z/OS operating system increases the storage available to the user from 2 gigabytes to 16 exabytes.

A program running on z/OS and the zSeries mainframe can run with 24-, 31-, or 64-bit addressing (and can switch among these if needed). To address the high virtual storage

available with the 64-bit architecture, the program uses 64-bit specific instructions. Although the architecture introduces unique 64-bit exploitation instructions, the program can use both 31-bit and 64-bit instructions, as needed.

For compatibility, the layout of the storage areas for an address space is the same below 2 gigabytes, providing an environment that can support both 24-bit and 31-bit addressing. The area that separates the virtual storage area below the 2-gigabyte address from the user private area is called *the bar*, as shown in Figure 2-7. The user private area is allocated for application code rather than operating system code.

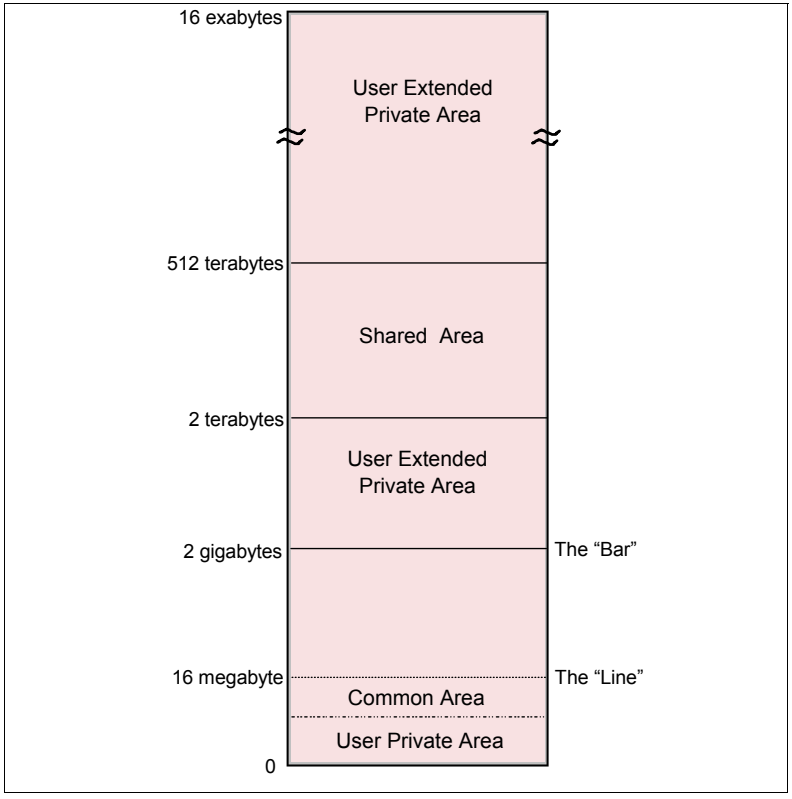


Figure 2-7 Storage map for a 64-bit address space

0 to 2^{31} The layout is the same; see Figure 2-7.

2^{31} to 2^{32} From 2 GB to 4 GB is considered the *bar*. Below the bar can be addressed with a 31-bit address. Above the bar requires a 64-bit address.

2^{32} - 2^{41} The low non-shared area (user private area) starts at 4 GB and extends to 2^{41}.

2^{41} - 2^{50}	Shared area (for storage sharing) starts at 2^{41} and extends to 2^{50} or higher, if requested.
2^{50} - 2^{64}	High non-shared area (user private area) starts at 2^{50} or wherever the shared area ends and goes to 2^{64}.

In a 16-exabyte address space with 64-bit virtual storage addressing, there are three additional levels of translation tables, called region tables. They are called region third table (R3T), region second table (R2T), and region first table (R1T). The region tables are 16 KB in length, and there are 2048 entries per table. Each region has 2 GB.

Segment tables and page table formats remain the same as for virtual addresses below the bar. When translating a 64-bit virtual address, once the system has identified the corresponding 2 GB region entry that points to the Segment table, the process is the same as that described previously.

2.4.14 What's *in* an address space?

Another way of thinking of an address space is as a programmer's map of the virtual storage available for code and data. An address space provides each programmer with access to all of the addresses available through the computer architecture (earlier, we defined this as *addressability*).

z/OS provides each user with a unique address space and maintains the distinction between the programs and data belonging to each address space. Because it maps all of the available addresses, however, an address space includes system code and data as well as user code and data. Thus, not all of the mapped addresses are available for user code and data.

Understanding the division of storage areas in an address space is made easier with a diagram. The diagram shown in Figure 2-8 on page 55 is more detailed than needed for this part of the course, but is included here to show that an address space maintains the distinction between programs and data belonging to the user, and those belonging to the operating system.

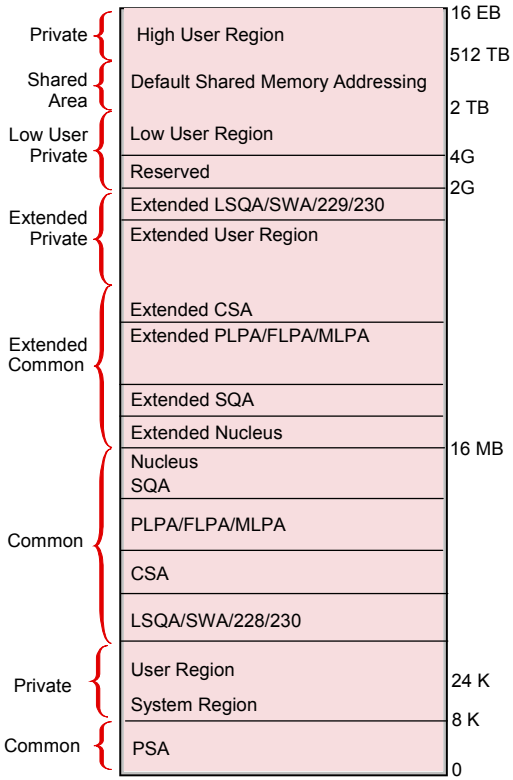


Figure 2-8 Storage areas in an address space

Figure 2-8 shows the major storage areas in each address space. These are described briefly as follows:

- **All storage above 2 gigabytes** is termed high virtual storage and is only addressable by programs running in 64-bit mode. The high virtual region is divided by the high virtual shared area, which is an area of installation-defined size that can be used to establish cross address space viewable connections to obtained areas within this area.
- **Extended areas above 16 megabytes:** This range of areas, which lies above The Line but below The Bar (two gigabytes) is a kind of "mirror image" of the common area below The Line. These areas have the same attributes as their equivalent areas below The Line, but because of the additional storage above The Line, their sizes are much larger.
- **Nucleus:** The nucleus is a key 0, read-only area of common storage that contains operating system control programs.
- **SQA:** This area of common storage contains system level data (key 0) that is accessed by multiple address spaces. The SQA area is not pageable (fixed), which means that it resides in real storage until it is freed by the requesting program. The

size of the SQA area is predefined by the installation and cannot change while the operating system is active. Yet it has the unique ability to "overflow" into the CSA area as long as there is unused CSA storage that can be converted to SQA.

- **PLPA/FLPA/MLPA:** This area of storage contains the link pack areas (the pageable link pack area, fixed link pack area, and modified link pack area). The link pack areas contain system level programs that are often run by multiple address spaces. For this reason, the link pack areas reside in the common area which is addressable by every address space, therefore eliminating the need for each address space to have its own copy of the program. This storage area is below The Line and is therefore addressable by programs running in 24-bit mode.
- **CSA:** The section of common area storage (addressable by all address spaces) that is available to all applications. The CSA is often used to contain data frequently accessed by multiple address spaces. The size of the CSA area is established at system initialization time (IPL), and cannot change while the operating system is active.
- **LSQA/SWA/subpool 228/subpool 230:** An assortment of subpools, each with specific attributes used primarily by system functions when the functions require address space level storage isolation. This storage area is below The Line and is therefore addressable by programs running in 24-bit mode.
- **User Region:** The storage area obtainable by any executing code in the address space, including user key programs. This area resides below The Line and is therefore addressable by programs running in 24-bit mode.
- **System Region:** A small area of storage (usually 4 pages) reserved for use by the region control task of each address space.
- **PSA:** Prefixed Save Area, which is often referred to as "Low Core." The PSA is a common area of virtual storage from address zero through 8191 in every address space. There is one unique PSA for every processor installed in a system. The PSA maps architecturally fixed hardware and software storage locations for the processor. Because there is a unique PSA for each processor, from the view of a program running on z/OS, the contents of the PSA can change any time the program is dispatched on a different processor. This feature is unique to the PSA area and is accomplished through a unique DAT manipulation technique called *prefixing*.

Given the vast range of addressable storage in an address space, the drawing in Figure 2-8 on page 55 is not to scale.

2.4.15 System address spaces and the master scheduler

Many z/OS system functions run in their own address spaces. The *master scheduler subsystem*, for example, runs in the address space called *MASTER* and is used to establish communication between z/OS and its own address spaces.

When you start z/OS, master initialization routines initialize system services, such as the system log and communication task, and start the master scheduler address space. Then, the master scheduler may start the job entry subsystem (JES2 or JES3). JES is the

primary job entry subsystem. On many production systems JES is not started immediately; instead, the automation package starts all tasks in a controlled sequence. Then other defined subsystems are started. All subsystems are defined in a special file of system settings called a parameter library or *PARMLIB*. These subsystems are *secondary subsystems*.

Each address space created has a number associated with it, called the address space ID (or ASID). Because the master scheduler is the first address space created in the system, it becomes address space number one (ASID=1). Other system address spaces are then started during the initialization process of z/OS.

At this point, you need only understand that z/OS and its related subsystems require address spaces of their own to provide a functioning operating system. A short description of each type of address space follows:

- System

  z/OS system address spaces are started after initialization of the master scheduler. These address spaces perform functions for all the other types of address spaces that start in z/OS.

- Subsystem

  z/OS requires the use of various subsystems, such as a primary job entry subsystem or *JES (*described in Chapter 5, “Batch processing and JES” on page 135). Also, there are address spaces for middleware products such as DB2, CICS, and IMS.

Besides system address spaces, there are, of course, typically many address spaces for users and separately running programs, for example:

- TSO/E address spaces are created for every user who logs on to z/OS (described in Chapter 3, “TSO/E, ISPF, and UNIX: Interactive facilities of z/OS” on page 75).
- Address space is created for every batch job that runs on z/OS. Batch job address spaces are started by JES.

2.5 What is workload management?

A computer system has three broad categories of hardware resources: processors, processor storage, and I/O devices. For z/OS, the management of system resources is the responsibility of the **workload management** (**WLM**) component. WLM manages the processing of workload in the system according to the company’s business goals, such as response time. WLM also manages the use of system resources, such as processors and storage, to accomplish these goals.

2.5.1 What does WLM do?

In simple terms, WLM has three objectives:

- To achieve the business goals that are defined by the installation, by automatically assigning sysplex resources to workloads based on their importance and goals. This objective is known as *goal achievement*.
- To achieve optimal use of the system resources from the system point of view. This objective is known as *throughput*.
- To achieve optimal use of system resources from the point of view of the individual address space. This objective is known as *response and turnaround time*.

Goal achievement is the first and most important task of WLM. Optimizing throughput and minimizing turnaround times of address spaces come after that. Often, these later two objectives are contradictory. Optimizing throughput means keeping resources busy. Optimizing response and turnaround time, however, requires resources to be available when they are needed. Achieving the goal of an important address space might result in worsening the turnaround time of a less important address space. Thus, WLM must make decisions that represent trade-offs between conflicting objectives.

To balance throughput with response and turnaround time, WLM does the following:

- Monitors the use of resources by the various address spaces
- Monitors the system-wide use of resources to determine whether they are fully utilized
- Determines which address spaces to swap out (and when)
- Inhibits the creation of new address spaces or steals pages when certain shortages of real storage exist
- Changes the dispatching priority of address spaces, which controls the rate at which the address spaces are allowed to consume system resources
- Selects the devices to be allocated, if a choice of devices exists, in order to balance the use of I/O devices.

Other z/OS components, transaction managers and database managers can communicate to WLM a change in status for a particular address space (or for the system as a whole), or to invoke WLM's decision-making power.

For example, WLM is notified when:

- Real storage is configured into or out of the system
- An address space is to be created
- An address space is deleted
- A swap-out starts or completes
- Allocation routines can choose the devices to be allocated to a request.

Up to this point, we have discussed WLM only in concourse of a single z/OS system. In real life use, customer installations often use clusters of multiple z/OS systems in concert

to process complex workloads. Later in the course, we discuss the concept of clustered z/OS systems (a sysplex), but for now, it is important to observe that WLM is particularly well-suited to a sysplex environment. WLM keeps track of system utilization and workload goal achievement across all the systems in parallel sysplex processing and data sharing environments. For example, WLM can decide which z/OS system a batch job can run in based on the availability of resources to process the job quickly.

2.5.2 How is WLM used?

A mainframe installation can influence almost all decisions made by WLM by establishing a set of *policies*, which allow the installation to closely link system performance to its business needs. Workloads are assigned goals (for example, a target average response time) and an importance. Importance represents how important it is to the business that a workload meet its goals.

Before the introduction of WLM, the only way to inform z/OS about the company's business goals was for the system programmer to translate from high-level objectives about what work needs to be done into the extremely technical terms that the system can understand. This translation required highly skilled staff, and could be protracted, error-prone, and eventually in conflict with the original business goals.

Further, it was often difficult to predict the effects of changing a system setting, which might be required, for example, following a system capacity increase. This could result in unbalanced resource allocation, that is, feeding work one resource while starving it of another. This way of operation, known as compatibility mode, was becoming unmanageable as new workloads were introduced, and as multiple systems were being managed together.

When in goal mode system operation, WLM provides fewer, simpler, and more consistent system externals that reflect goals for work expressed in terms commonly used in business objectives, and WLM and System Resource Manager (SRM) match resources to meet those goals by constantly monitoring and adapting the system. Workload Manager provides a solution for managing workload distribution, workload balancing, and distributing resources to competing workloads.

WLM polices are often based on a service level agreement (SLA), which is a written agreement of the information systems (I/S) service to be provided to the users of a computing installation. WLM tries to achieve the needs of workloads (response time) as described in an SLA by attempting the appropriate distribution of resources without over-committing them. Equally important, WLM maximizes system use (throughput) to deliver maximum benefit from the installed hardware and software platform.

2.6 I/O and data management

Nearly all work in the system involves data input or data output. In a mainframe, the channel subsystem manages the use of I/O devices, such as disks, tapes, and printers. The operating system must associate the data for a given task with a device, and manage file allocation, placement, monitoring, migration, backup, recall, recovery, and deletion.

These data management activities can be done either manually or through the use of automated processes. When data management is automated, the system determines object placement, and automatically manages object backup, movement, space, and security. A typical z/OS production system includes both manual and automated processes for managing data.

Depending on how a z/OS system and its storage devices are configured, a user or program can directly control many aspects of data management, and in the early days of the operating system, users were required to do so. Increasingly, however, z/OS installations rely on installation-specific settings for data and resource management, and add-on storage management products to automate the use of storage. The primary means of managing storage in z/OS is through the DFSMS component, which is discussed in 4.3, “Role of DFSMS in managing space” on page 109.

2.7 Supervising the execution of work in the system

To enable multiprogramming, z/OS requires the use of a number of supervisor controls, as follows:

- **Interrupt processing.** Multiprogramming requires that there be some technique for switching 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 execute. In z/OS, this switch is achieved by **interrupts**, which are events that alter the sequence in which the processor executes instructions. When an interrupt occurs, the system saves the execution status of the interrupted routine and analyzes and processes the interrupt.
- **Creating dispatchable units of works.** To identify and keep track of its work, the z/OS operating system represents each unit of work with a control block. Two types of control blocks represent dispatchable units of work: **task control blocks** (**TCBs**) and **service request blocks** (**SRBs**). TCBs represent tasks executing within an address space; SRBs represent higher priority system services.
- **Dispatching work.** After interrupts are processed, the operating system determines which unit of work that is ready to run (**ready work**), of all the units of ready 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

data file), the operating system must prevent other programs from using the resource until the interrupted program has completed its processing of the resource.

Several techniques exist for serializing the use of resources; **enqueuing** and **locking** are the most common (a third technique is called *latching*). All users can use enqueuing, but only authorized routines can use locking to serialize the use of resources.

2.7.1 What is interrupt processing?

An interrupt is an event that alters the sequence in which the processor executes instructions. An interrupt might be planned (specifically requested by the currently running program) or unplanned (caused by an event that might or might not be related to the currently running program). z/OS uses six types of interrupts, as follows:

- **Supervisor call** or **SVC** interrupts, which occur when the program issues an SVC instruction. An SVC is a request for a particular system service. These services are requested through macros such as OPEN (open a file), GETMAIN (obtain storage), or WTO (write a message to the system operator).
- **I/O interrupts**, which occur when the channel subsystem signals a change of status. For example, an I/O operation completes, an error occurs, or an I/O device such as a printer becomes ready.
- **External interrupts**, which indicate any of several events: a time interval expires, the operator presses the interrupt key on the console, or the processor receives a signal from another processor.
- **Restart interrupts**, which occur when the operator selects the restart function at the console or when a restart SIGP (signal processor) instruction is received from another processor.
- **Program interrupts**, which are caused by program errors (for example, the program attempts to perform an invalid operation), **page faults** (the program references a page that is not in real storage), or requests to monitor an event.
- **Machine check interrupts**, which are caused by machine malfunctions.

When an interrupt occurs, the hardware saves pertinent information about the program that was interrupted and, if possible, disables the processor for further interrupts of the same type. The hardware then routes control to the appropriate interrupt handler routine. The program status word or PSW is a key resource in this process.

How is the program status word used?

The program status word (PSW) is a 128-bit data area in the processor that, along with a variety of other types of registers (control registers, timing registers, and prefix registers) provides details crucial to both the hardware and the software. The current PSW includes the address of the next program instruction and control information about the program that is running (every application running on z/OS has its own control information). Each

processor has only one current PSW. Thus, only one task can execute on a processor at a time.

The PSW controls the order in which instructions are fed to the processor, and indicates the status of the system in relation to the currently running program. Although each processor has only one PSW, it is useful to think of three types of PSWs in order to understand interrupt processing. The three PSWs are: the current PSW, the new PSW, and the old PSW.

The current PSW indicates the next instruction to be executed. It also indicates whether the processor is enabled or disabled for I/O interrupts, external interrupts, machine check interrupts, and certain program interrupts. When the processor is enabled, these interrupts can occur. When the processor is disabled, these interrupts are ignored or remain pending.

There is a new PSW and an old PSW associated with each of the six types of interrupts. The new PSW contains the address of the routine that can process its associated interrupt. If the processor is enabled for interrupts when an interrupt occurs, PSWs are switched through the following technique:

1. Storing the current PSW in the old PSW associated with the type of interrupt that occurred
2. Loading the contents of the new PSW for the type of interrupt that occurred into the current PSW

The current PSW, which indicates the next instruction to be executed, now contains the address of the appropriate routine to handle the interrupt . This switch has the effect of transferring control to the appropriate interrupt handling routine.

The routine that receives control after an interrupt is processed depends on whether the interrupted unit of work was non-preemptable. A non-preemptable unit of work can be interrupted but must receive control after the interrupt is processed. All SRBs are non-preemptable; a TCB is non-preemptable if it is executing an SVC. If the interrupted unit of work was preemptable, the system determines which unit of work should be performed next.

Registers and the PSW

Mainframe architecture provides registers to keep track of things. The PSW, for example, is a register used to contain information that is required for the execution of the currently active program. Mainframes provide other registers, as follows:

- *Access registers* are used to specify the address space data in which data is found.
- *General registers* are used to address data in storage, and also for holding user data.
- *Floating point registers* are used to hold numeric data in floating point form.

- *Control registers* are used by the operating system itself, for example, as references to translation tables.

Related Reading: The IBM publication *z/Architecture Principles of Operation* describes the hardware facilities for the switching of system status, including CPU states, control modes, the PSW, and control registers. You can find this and other related publications at the z/OS Internet Library Web site:

http://www.ibm.com/servers/eserver/zseries/zos/bkserv/

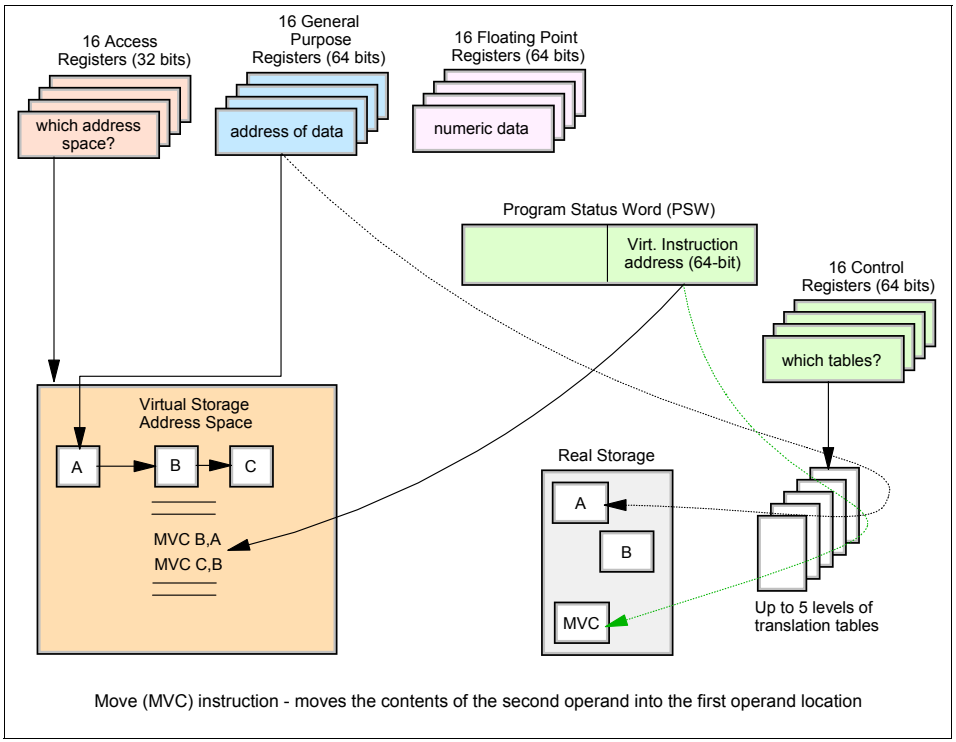


Figure 2-9 Registers and the PSW

2.7.2 Creating dispatchable units of work

In z/OS, dispatchable units of work are represented by two different control blocks, as follows:

- Task control blocks (TCBs), which represent tasks executing within an address space, such as user programs and system programs that support the user programs.
- Service request blocks (SRBs), which represent requests to execute a system service routine. SRBs are typically created when one address space detects an event that

affects a different address space; they provide one mechanism for communication between address spaces.

Task control blocks (TCBs)

TCBs are created in response to an ATTACH macro. By issuing the ATTACH macro, a user or system routine begins the execution of the program specified on the ATTACH macro, as a subtask of the attacher's task. As a subtask, the specified program can compete for processor time and can use certain resources already allocated to the attacher's task.

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.

Service request blocks (SRBs)

An SRB represents a request to perform a service in a specified address space. Typically, an SRB is created when one address space is executing and an event occurs that affects a different address space.

Only supervisor state, key 0 functions create an SRB. They obtain storage and initialize the control block with such things as the identity of the target address space and pointers to the code that will process the request. The component creating the SRB then issues the SCHEDULE macro and indicates whether the SRB has global (system-wide) or local (address space wide) priority. SCHEDULE places the SRB on the appropriate dispatching queue where it will remain until it becomes the highest priority work on the queue.

SRBs with a global priority have a higher priority than that of any address space, regardless of the actual address space in which they will be executed. SRBs with a local priority have a priority equal to that of the address space in which they will be executed, but higher than 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 interrupts are scheduled at a global priority, to minimize I/O delays.

As mentioned previously, SRBs are non-preemptable. Thus, if a routine represented by an SRB is interrupted, it will receive control after the interrupt has been processed. In contrast, a routine represented by a TCB is preemptable. If it is interrupted, control returns to the operating system when the interrupt handling completes. z/OS then determines which tasks, of all the ready tasks, executes next.

An SRB can execute concurrently and in a different address space from the task that created it. This means, among other things, that an SRB provides the means for asynchronous inter-address space communication. Such communication improves the availability of resources in a multiprocessing environment.

2.7.3 What does the dispatcher do?

New work is selected, for example, when a task is interrupted or becomes non-dispatchable, or after an SRB completes or is suspended (that is, an SRB is delayed because a required resource is not available).

In z/OS, the dispatcher component is responsible for routing control to the highest priority unit of work that is ready to execute. The dispatcher processes 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 in a multi-processing system fails, alternate CPU recovery is invoked by means of a special exit to recover work that was being executed on the failing processor.
2. SRBs that have a global 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 spaces's priority is determined by the dispatching priority specified by the user or the installation.

   After selecting the highest priority address space, z/OS (through 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, z/OS assumes a state called an **enabled wait** until fresh work enters the system.

Different models of the z/Series hardware can have from one to 54 central processors (CPs)[7]. Each and every CP can be executing instructions at the same time. Dispatching priorities determine when ready-to-execute address spaces get dispatched.

[7] The IBM z9-109 Model S54 can be ordered with up to 54 CPs (the model numbers correspond to the maximum number of processors that can be ordered in the server).

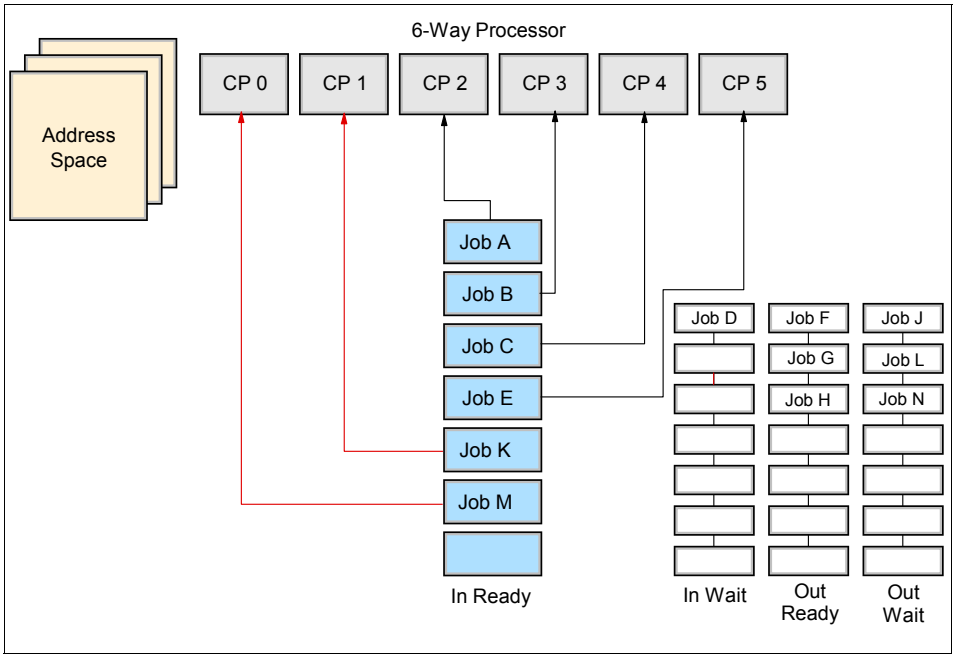


Figure 2-10 Dispatching work

- An address space can be in any one of four queues:
  - IN-READY - In real storage and waiting to be dispatched
  - IN-WAIT - In real storage but waiting for some event to complete
  - OUT-READY - Ready to execute but swapped out
  - OUT-WAIT - Swapped out and waiting for some event to complete
- Only IN-READY work will be selected for dispatching.

2.7.4 Serializing the use of resources

In a multitasking, multiprocessing environment, resource serialization is the technique used to coordinate access to resources that are used by more than one application. Programs that change data need exclusive access to the data. Otherwise, if several programs were to update the same data at the same time, the data could be corrupted (also referred to as a loss of data integrity). On the other hand, programs that need only to read data can safely share access to the same data at the same time.

The most common techniques for serializing the use of resources are enqueuing and locking. These techniques allow for orderly access to system resources needed by more than one user in a multiprogramming or multiprocessing environment. In z/OS, enqueuing is managed by the global resource serialization (or GRS) component and locking is managed by various lock manager programs in the supervisor component.

What is global resource serialization?

The global resource serialization (GRS) component processes requests for resources from programs running on z/OS. Global resource serialization serializes access to resources to protect their integrity. An installation can connect two or more z/OS systems with channel-to-channel (CTC) adapters to form a GRS complex to serialize access to resources shared among the systems.

When a program requests access to a reusable resource, the access can be requested as exclusive or shared. When global resource serialization grants shared access to a resource, exclusive users cannot obtain access to the resource. Likewise, when global resource serialization grants exclusive access to a resource, all other requestors for the resource wait until the exclusive requestor frees the resource.

Enqueuing

Enqueuing is accomplished by means of the ENQ (enqueue) and DEQ (dequeue) macros, which are available to all users and programs on the system. For devices shared between z/OS systems, enqueuing is accomplished through the RESERVE and DEQ macros.

On ENQ and RESERVE, a user specifies the names 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 others who do not require exclusive control. If the resource is not available, the requestor is suspended until it becomes available. The DEQ macro is used to release control of a resource.

Locking

Locking serializes the use of system resources by authorized routines and, in a parallel sysplex, by processors. A lock is simply a named field in storage that indicates whether a resource is being used and who is using it. In z/OS, 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-reusable or non-sharable routines and various resources.

To use a resource protected by a lock, a routine must first request the lock for that resource. If the lock is unavailable (that is, it is already held by another program or processor), the action taken by the program or processor that requested the lock depends on whether the lock is a spin lock or a suspend lock:

- 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, thus, control of the protected resource. Most global locks are spin locks. The holder of a spin lock should be disabled for most interrupts (if the holder were to be interrupted, it might never be able to gain control to give up the lock).

- If a suspend lock is unavailable, the unit of work requesting the lock is delayed until the lock is available. Other work is dispatched on the requesting processor. All local locks are suspend locks.

You might wonder what would happen in the case in which two users both request locks that are held by the other? Would they both wait forever for the other to release the lock first, in a kind of stalemate? In z/OS, such an occurrence would be known as a deadlock. Fortunately, the z/OS locking methodology prevents deadlocks.

To avoid deadlocks, locks are arranged in a hierarchy, and a processor or routine can 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. This situation cannot occur because locks must be acquired in hierarchical sequence. Assume, in this example, that lock A precedes lock B is the hierarchy. Processor 2, then, cannot unconditionally request lock A while holding lock B. It must, instead, release lock B, request lock A, and then request lock B. Because of this hierarchy, a deadlock cannot occur.

Related Reading: The IBM publication *z/OS Diagnosis Reference* includes a table that lists the hierarchy of z/OS locks, along with their descriptions and characteristics.

2.8 Defining characteristics of z/OS

The defining characteristics of z/OS are summarized as follows:

- The use of address spaces in z/OS holds many advantages: Isolation of private areas in different address spaces provides for system security, yet each address space also provides a common area that is accessible to every address space.
- The system is designed to preserve *data integrity*, regardless of how large the user population might be. z/OS prevents users from accessing or changing any objects on the system, including user data, except by the system-provided interfaces that enforce authority rules.
- The system is designed to manage a large number of concurrent batch jobs, with no need for the customer to externally manage workload balancing or integrity problems that might otherwise occur due to simultaneous and conflicting use of a given set of data.
- The security design extends to system functions as well as simple files. Security can be incorporated into applications, resources, and user profiles.
- The system allows multiple communications subsystems at the same time, permitting unusual flexibility in running disparate communications-oriented applications (with mixtures of test, production, and fall-back versions of each) at the same time. For example, multiple TCP/IP stacks can be operational at the same time, each with different IP addresses and serving different applications.

- The system provides extensive software recovery levels, making unplanned system restarts very rare in a production environment. System interfaces allow application programs to provide their own layers of recovery. These interfaces are seldom used by simple applications—they are normally used by more sophisticated applications.
- The system is designed to routinely manage very disparate workloads, with automatic balancing of resources to meet production requirements established by the system administrator.
- The system is designed to routinely manage large I/O configurations that might extend to thousands of disk drives, multiple automated tape libraries, many large printers, large networks of terminals, and so forth.
- The system is controlled from one or more operator terminals, or from application programming interfaces (APIs) that allow automation of routine operator functions.
- The operator interface is a critical function of z/OS. It provides status information, messages for exception situations, control of job flow, hardware device control, and allows the operator to manage unusual recovery situations.

2.9 Licensed programs for z/OS

A z/OS system usually contains additional priced products that are needed to create a practical working system. These additional products are called *licensed programs*. For example, a production z/OS system typically includes a security manager product and a database manager product. When talking about z/OS, people often assume the inclusion of licensed programs. This is normally apparent from the concourse of a discussion, but it might sometimes be necessary to ask whether a particular function is part of "the base z/OS" or is an add-on product.

With a multitude of independent software vendors (ISVs*)* offering a large number of products with varying but similar functionality, such as security managers, the ability to choose from a variety of licensed programs to accomplish a task considerably increases the flexibility of the z/OS operating system and allows the mainframe IT group to tailor the products it runs to meet their company's specific needs.

We won't attempt to list all of the z/OS licensed programs in this course (hundreds exist); some common choices include:

- A security system. z/OS provides a framework for customers to add security through the addition of a security management product (IBM's licensed program is *Resource Access Control Facility or RACF*®). Non-IBM security system licensed programs are also available.
- Compilers. z/OS includes an assembler and a C compiler. Other compilers, such as the COBOL compiler, and the PL/1 compiler are offered as separate products.

- A relational database, such as DB2. Other types of database products, such as hierarchical databases, are also available.
- Transaction processing facilities. IBM offers several, including:
  - Customer Information Control System (CICS)
  - Information Management System (IMS)
  - WebSphere Application Server for z/OS
- A sort program. Fast, efficient sorting of large amounts of data is highly desirable in batch processing. IBM and other vendors offer sophisticated sorting products.
- A large variety of utility programs. For example, the System Display and Search Facility (SDSF) program that we use extensively in this course to view output from batch jobs is a licensed program. Not every installation purchases SDSF; alternative products are available.
- A large number of other products are available from various *independent software vendors* or ISVs as they are commonly called in the industry.

2.10 Middleware for z/OS

Middleware is typically something between the operating system and an end user or end-user applications. It supplies major functions not provided by the operating system. As commonly used, the term usually applies to major software products such as database managers, transaction monitors, Web servers, and so forth. *Subsystem* is another term often used for this type of software. These are usually licensed programs, although there are notable exceptions, such as the HTTP Server.

z/OS is a base for using many middleware products and functions. It is commonplace to run a variety of diverse middleware functions, with multiple instances of some. The routine use of wide-ranging workloads (mixtures of batch, transactions, Web serving, database queries and updates, and so on) is characteristic of z/OS.

Typical z/OS middleware includes:

- Database systems
- Web servers
- Message queueing and routing functions
- Transaction managers
- Java virtual machines
- XML processing functions

A middleware product often includes an *application programming interface* (API). In some cases, applications are written to run completely under the control of this middleware API, while in other cases it is used only for unique purposes. Some examples of mainframe middleware APIs include:

- The WebSphere suite of products, which provides a complete API that is portable across multiple operating systems. Among these, WebSphere MQ provides cross-platform APIs and inter-platform messaging.
- The DB2 database management product, which provides an API (expressed in the SQL language) that is used with many different languages and applications.

A Web server is considered to be middleware and Web programming (Web pages, CGIs, and so forth) is largely coded to the interfaces and standards presented by the Web server instead of the interfaces presented by the operating system. Java is another example in which applications are written to run under a *Java Virtual Machine* (JVM™)[8] and are largely independent of the operating system being used.

2.11 A brief comparison of z/OS and UNIX

What would we find if we compared z/OS and UNIX? In many cases, we'd find that quite a few concepts would be mutually understandable to users of either operating system, despite the differences in terminology.

For experienced UNIX users, Table 2-1 provides a small sampling of familiar computing terms and concepts. As a new user of z/OS, many of the z/OS terms will sound unfamiliar to you. As you work through this course, however, the z/OS meanings will be explained and you will find that many elements of UNIX have analogs in z/OS.

A major difference for UNIX users moving to z/OS is the idea that the user is just one of *many* other users. In moving from a UNIX system to the z/OS environment, users typically ask questions such as *"Can I have the root password because I need to do....."* or *"Would you change this or that and restart the system?"* It is important for new z/OS users to understand that potentially thousands of other users are active on the same system, and so the scope of user actions and system restarts in z/OS and z/OS UNIX are carefully controlled to avoid negatively effecting other users and applications.

Under z/OS, there does not exist a single root password or root user. User IDs are external to z/OS UNIX System Services. User IDs are maintained in a security database that is shared with both UNIX and non-UNIX functions in the z/OS system, and possibly even shared with other z/OS systems. Typically, some user IDs have root authority, but these remain individual user IDs with individual passwords. Also, some user IDs do not normally have root authority, but can switch to "root" when circumstances require it.

[8] A JVM is not related to the virtual machines created by z/VM.

Table 2-1 Mapping UNIX to z/OS terms and concepts

Term or concept	UNIX	z/OS
Start the operating system	Boot the system.	IPL (initial program load) the system.
Virtual storage given to each user of the system	Users get whatever virtual storage they need to reference, within the limits of the hardware and operating system.	Users each get an address space, a range of addresses extending to 2 GB (or even 16 EB) of virtual storage, though some of this storage contains system code that is common for all users.
Data storage	Files	Data sets (sometimes called files)
Data format	Byte orientation; organization of the data is provided by the application.	Record orientation; often an 80-byte record, reflecting the traditional punched card image.
System configuration data	The /etc file system controls characteristics.	Parameters in PARMLIB control how the system IPLs and how address spaces behave.
Scripting languages	Shell scripts, Perl, awk, and other languages	CLISTS (command lists) and REXX execs
Smallest element that performs work	A thread. The kernel supports multiple threads.	A task or a service request block (SRB). The z/OS base control program (BCP) supports multiple tasks and SRBs.
A long-running unit of work	A daemon	A started task or a long-running job; often this is a subsystem of z/OS.
Order in which the system searches for programs to run	Programs are loaded from the file system according to the user's PATH environmental variable (a list of directories to be searched).	The system searches the following libraries for the program to be loaded: TASKLIB, STEPLIB, JOBLIB, LPALST, and the linklist.

Term or concept	UNIX	z/OS
Using the system interactively	Users *log in* to systems and execute shell sessions in the shell environment. They can issue the **rlogin** or **telnet** commands to connect to the system. Each user can have many login sessions open at once.	Users *log on* to the system through TSO/E and its panel-driven interface, ISPF. A user ID is limited to having only one TSO/E logon session active at a time. Users can also login to a z/OS UNIX shell environment using telnet, rlogin, or ssh.
Editing data or code	Many editors exist, such as vi, ed, sed, and emacs.	ISPF editor[a]
Source and destination for input and output data	stdin and stdout	SYSIN and SYSOUT ► SYSUT1 and SYSUT2 are used for utilities. ► SYSTSIN and SYSTSPRT are used for TSO/E users.
Managing programs	The **ps** shell command allows users to view processes and threads, and kill jobs with the **kill** command.	SDSF allows users to view and terminate their jobs.

a. There is also a TSO editor, though it is rarely used. For example, when sending e-mail via TSO, the SENDNOTE exec opens a TSO EDIT session to allow the user to compose the e-mail.

2.12 Summary

An operating system is a collection of programs that manage the internal workings of a computer system. The operating system taught in this course is z/OS, a widely used mainframe operating system. The z/OS operating system's use of multiprogramming and multiprocessing, and its ability to access and manage enormous amounts of storage and I/O operations, makes it ideally suited for running mainframe workloads.

The concept of virtual storage is central to z/OS. Virtual storage is an illusion created by the architecture, in that the system seems to have more storage than it really has. Virtual storage is created through the use of tables to map virtual storage pages to frames in real storage or slots in auxiliary storage. Only those portions of a program that are needed are actually loaded into real storage. z/OS keeps the inactive pieces of address spaces in auxiliary storage.

z/OS is structured around address spaces, which are ranges of addresses in virtual storage. Each user of z/OS gets an address space containing the same range of storage addresses. The use of address spaces in z/OS allows for isolation of private areas in different address spaces for system security, yet also allows for inter-address space sharing of programs and data through a common area accessible to every address space.

In common usage, the terms real storage, real memory, central storage, and main storage are used as synonyms and are used interchangeably. Likewise, virtual memory and virtual storage are used interchangeably.

The amount of real storage needed to support the virtual storage in an address space depends on the working set of the application being used, and this varies over time. A user does not automatically have access to all the virtual storage in the address space. Requests to use a range of virtual storage are checked for size limitations and then necessary paging table entries are constructed to create the requested virtual storage.

Programs running on z/OS and zSeries mainframes can run with 24-, 31-, or 64-bit addressing (and can switch between these modes if needed). Programs can use a mixture of instructions with 16-bit, 32-bit, or 64-bit operands, and can switch between these if needed.

Mainframe operating systems seldom provide complete operational environments. They depend on additional software products for middleware and other functions. Many vendors, including IBM, provide middleware and various utility products.

Middleware is a relatively recent term that can embody several concepts at the same time. A common characteristic of middleware is that it provides a programming interface, and applications are written (or partially written) to this interface.

Key terms in this chapter			
address space	addressability	auxiliary storage	control block
dynamic address translation (DAT)	frame	input/output (I/O)	middleware
multiprogramming	multiprocessing	page / paging	page stealing
licensed programs	real storage	service level agreement (SLA)	slot
swapping	virtual storage	workload management (WLM)	z/OS

.

3

TSO/E, ISPF, and UNIX: Interactive facilities of z/OS

Objective: In working with the z/OS operating system, you will need to know its end-user interfaces. Chief among these is TSO and its menu-driven interface, ISPF. These programs allow you to log on to the system, run programs, and manipulate data files. Also, you will need to know the interactive facilities of the z/OS implementation of UNIX interfaces, known collectively as z/OS UNIX System Services, or z/OS UNIX for short.

After completing this chapter, you will be able to:

- Log on to z/OS
- Run programs from the TSO READY prompt
- Navigate through the menu options of ISPF
- Use the ISPF editor to make changes to a data set
- Use the UNIX interfaces on z/OS, including the z/OS UNIX command shell.

© Copyright IBM Corp. 2005, 2006. All rights reserved.

3.1 How do we interact with z/OS?

We've mentioned that z/OS is ideal for processing batch jobs – workloads that run in the background with little or no human interaction. However, z/OS is just as much an interactive operating system as it a batch processing system. By *interactive* we mean that end-users (sometimes tens of thousands of them concurrently in the case of z/OS) using the system through direct interaction, such as commands and menu style user interfaces.

z/OS provides a number of facilities to allow users to interact directly with the operating system. This chapter provides an overview of each facility, as follows:

- "TSO overview" on page 76 shows how to log on to z/OS and describes the use of a limited set of basic TSO commands available as part of the core operating system. Interacting with z/OS in this way is called using TSO in its *native mode.*
- "ISPF overview" on page 80 introduces the ISPF menu system, which is what many people use exclusively to perform work on z/OS. ISPF menus list the functions that are most frequently needed by online users.
- "z/OS UNIX interactive interfaces" on page 99 explores the z/OS UNIX shell and utilities. This facility allows users to write and invoke shell scripts and utilities, and use the shell programming language.

Hands-on exercises are provided at the end of the chapter to help students develop their understanding of these important facilities.

3.2 TSO overview

Time Sharing Option/Extensions (TSO/E) allows users to create an interactive session with the z/OS system. TSO[1] provides a single-user logon capability and a basic command prompt interface to z/OS.

Most users work with TSO through its menu-driven interface, *Interactive System Productivity Facility* (ISPF). This collection of menus and panels offers a wide range of functions to assist users in working with data files on the system. ISPF users include system programmers, application programmers, administrators, and others who access z/OS. In general, TSO and ISPF make it easier for people with varying levels of experience to interact with the z/OS system.

In a z/OS system, each user is granted a user ID and a password authorized for TSO logon. Logging on to TSO requires a 3270 display device or, more commonly, a TN3270 emulator running on a PC.

[1] Most z/OS users refer to TSO/E as simply "TSO," and that is how it is called in this course. Also, the word "user" is synonymous with "end user."

During TSO logon, the system displays the TSO logon screen on the user's 3270 display device or TN3270 emulator. The logon screen serves the same purpose as a Windows logon panel.

z/OS system programmers often modify the particular course layout and information of the TSO logon panel to better suit the needs of the system's users. Therefore, the screen captures shown in this book will likely differ from what you might see on an actual production system.

Figure 3-1 shows a typical example of a TSO logon screen.

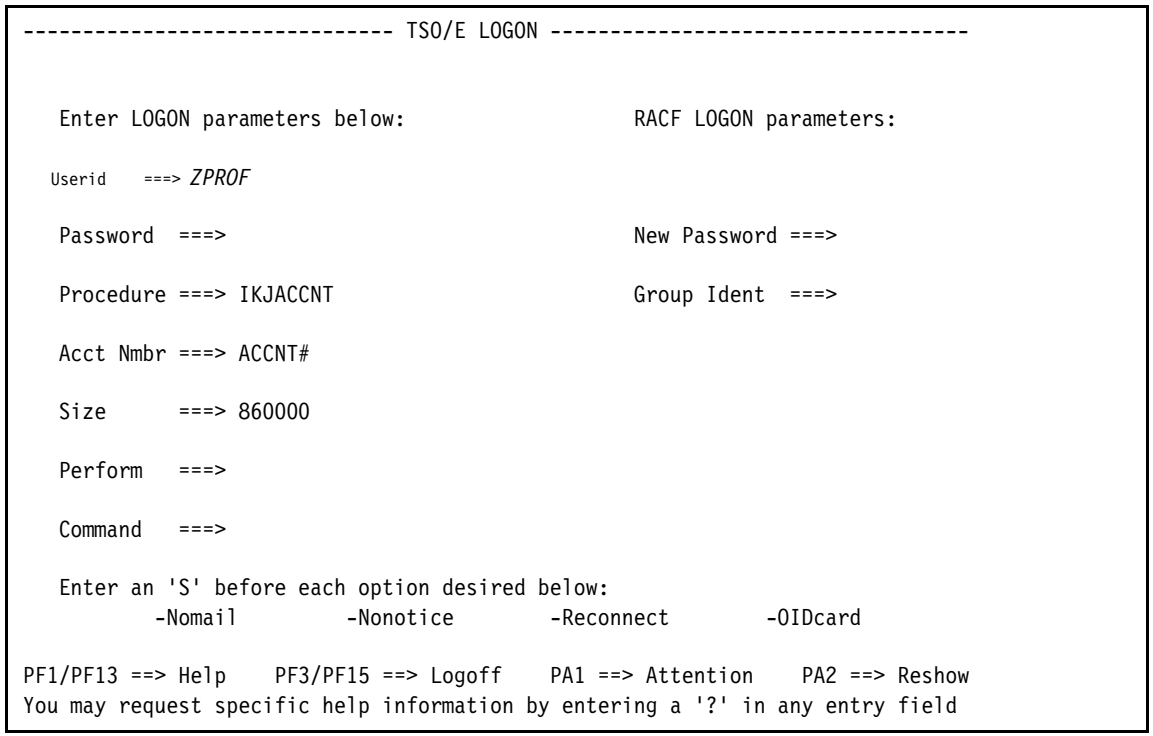


Figure 3-1 Typical TSO/E logon screen

Many of the screen capture examples used in this course show program function (PF) key settings. Because it is common practice for z/OS sites to customize the PF key assignments to suit their needs, the key assignments shown in this course might not match the PF key settings in use at your site.

A list of the PF key assignments used in this course is provided in 3.3.1, "Keyboard mapping used in this course" on page 86.

3.2.1 Using TSO commands in native mode

Most z/OS sites prefer to have the TSO user session automatically switch to the ISPF interface after TSO logon. This section, however, briefly discusses the limited set of basic TSO commands available independent of other complementary programs, such as ISPF. Using TSO in this way is called using TSO in its *native mode.*

When a user logs on to TSO, the z/OS system responds by displaying the READY prompt, and waits for input, such as in Figure 3-2.

```
ICH70001I ZPR0F  LAST ACCESS AT 17:12:12 ON THURSDAY, OCTOBER 7, 2004
ZPR0F LOGON IN PROGRESS AT 17:12:45 ON OCTOBER 7, 2004
You have no messages or data sets to receive.
READY
```

Figure 3-2 TSO logon READY prompt

The READY prompt accepts simple line commands such as HELP, RENAME, ALLOCATE, and CALL. Figure 3-3 shows an example of an ALLOCATE command that creates a *data set* (a file) on disk.

```
READY
 alloc dataset(zschol.test.cntl) volume(test01) unit(3390) tracks space(2,1)
recfm(f) lrecl(80) dsorg(ps)
READY
listds
 ENTER DATA SET NAME -
zschol.test.cntl
 ZSCHOL.TEST.CNTL
  --RECFM-LRECL-BLKSIZE-DSORG
    F     80    80      PS
 --VOLUMES--
   TEST01
 READY
```

Figure 3-3 Allocating a data set from the TSO command line

Native TSO is similar to the interface offered by the native DOS prompt. TSO also includes a very basic line mode editor, in contrast to the full screen editor offered by ISPF.

Figure 3-4 is another example of the line commands a user might enter at the READY prompt. Here, the user is entering commands to sort data.

```
 READY
ALLOCATE DATASET(AREA.CODES) FILE(SORTIN)    SHR
 READY
ALLOCATE DATASET(*)          FILE(SORTOUT)   SHR
 READY
ALLOCATE DATASET(*)          FILE(SYSOUT)    SHR
 READY
ALLOCATE DATASET(*)          FILE(SYSPRINT)  SHR
 READY
ALLOCATE DATASET(SORT.CNTL)  FILE(SYSIN)     SHR
 READY
CALL 'SYS1.SICELINK(SORT)'

 ICE143I 0 BLOCKSET     SORT  TECHNIQUE SELECTED
 ICE000I 1 - CONTROL STATEMENTS FOR Z/OS DFSORT V1R5
             SORT FIELDS=(1,3,CH,A)
 201  NJ
 202  DC
 203  CT
 204  Manitoba
 205  AL
 206  WA
 207  ME
 208  ID
 ***
```

Figure 3-4 Using native TSO commands to sort data

In this example, the user entered several TSO ALLOCATE commands to assign inputs and outputs to the workstation for the sort program. The user then entered a single CALL command to run the sort program, DFSORT™, an optional software product from IBM.

Each ALLOCATE command requires content (specified with the DATASET operand) associated with the following:

- SORTIN - in this case AREA.CODES
- SORTOUT - in this case *, which means the terminal screen
- SYSOUT
- SYSPRINT
- SYSIN

Following the input and output allocations and the user-entered CALL command, the sort program displays the results on the user's screen. As shown in Figure 3-4, the sort fields control statement causes the results to be sorted by area code. For example, NJ (New Jersey) has the lowest number telephone area code, 201.

Native TSO screen control is very basic. For example, when a screen fills up with data, three asterisks (***) are displayed to indicate a full screen. Here, you must press the Enter key to clear the screen of data and allow the screen to display the remainder of the data.

3.2.2 Using CLISTs and REXX under TSO

With native TSO, it is possible to place a list of commands, called *a command list* or *CLIST* (pronounced “see list”) in a file, and execute the list as if it were one command. When you invoke a CLIST, it issues the TSO/E commands in sequence. CLISTs are used for performing routine tasks; they enable users to work more efficiently with TSO.

For example, suppose that the commands shown in Example 3-4 on page 79 were grouped in a file called AREA.COMMND. The user could then achieve the same results by using just a single command to execute the CLIST, as follows:

```
EXEC 'CLIST AREA.COMMND'
```

TSO users create CLISTs with the CLIST command language. Another command language used with TSO is called *Restructured Extended Executor* or *REXX*. Both CLIST and REXX offer shell script-type processing. These are *interpretive* languages, as opposed to *compiled* languages (although REXX can be compiled as well).

Some TSO users write functions directly as CLISTs or REXX programs, but these are more commonly implemented as ISPF functions, or by various software products. CLIST programming is unique to z/OS, while the REXX language is used on many platforms.

3.3 ISPF overview

After logging on to TSO, users typically access the ISPF menu. In fact, many use ISPF exclusively for performing work on z/OS. ISPF is a full panel application navigated by keyboard. ISPF includes a course editor and browser, and functions for locating and listing files and performing other utility functions. ISPF menus list the functions that are most frequently needed by online users.

Figure 3-5 shows the allocate procedure to create a data set using ISPF.

```
 Menu  RefList  Utilities  Help
 ------------------------------------------------------------------------------
Allocate New Data Set
 Command ===>
Data Set Name  . . . : ZCHOL.TEST.CNTL
Management class . . .                (Blank for default management class)
 Storage class  . . . .                (Blank for default storage class)
  Volume serial . . . . TEST01         (Blank for system default volume) **
  Device type . . . . . .              (Generic unit or device address) **
 Data class . . . . . .                (Blank for default data class)
  Space units . . . . . TRACK          (BLKS, TRKS, CYLS, KB, MB, BYTES
                                        or RECORDS)
  Average record unit                  (M, K, or U)
  Primary quantity  . . 2              (In above units)
  Secondary quantity    1              (In above units)
  Directory blocks  . . 0              (Zero for sequential data set) *
  Record format . . . . F
  Record length . . . . 80
  Block size  . . . . .
  Data set name type  :                (LIBRARY, HFS, PDS, or blank)  *
                                       (YY/MM/DD, YYYY/MM/DD
  Expiration date . . .                 YY.DDD, YYYY.DDD in Julian form
 Enter "/" to select option             DDDD for retention period in days
    Allocate Multiple Volumes           or blank)

 ( * Specifying LIBRARY may override zero directory block)

 ( ** Only one of these fields may be specified)
  F1=Help F2=Split F3=Exit F7=Backward F8=Forward F9=Swap F10=Actions  F12=Cancel
```

Figure 3-5 Allocating a data set using ISPF panels

Figure 3-6 shows the results of allocating a data set using ISPF panels.

```
Data Set Information
Command ===>

Data Set Name  . . . : ZCHOL.TEST.CNTL

General Data                          Current Allocation
 Volume serial . . . : TEST01          Allocated tracks  . : 2
 Device type . . . . : 3390            Allocated extents . : 1
 Organization  . . . : PS
 Record format . . . : F
 Record length . . . : 80
 Block size  . . . . : 80             Current Utilization
 1st extent tracks . : 2               Used tracks . . . . : 0
 Secondary tracks  . : 1               Used extents  . . . : 0

 Creation date . . . : 2005/01/31
 Referenced date . . : 2005/01/31
 Expiration date . . : ***None***

 F1=Help F2=Split F3=Exit F7=Backward  F8=Forward   F9=Swap F12=Cancel
```

Figure 3-6 Result of data set allocation using ISPF

Figure 3-7 shows the ISPF menu structure.

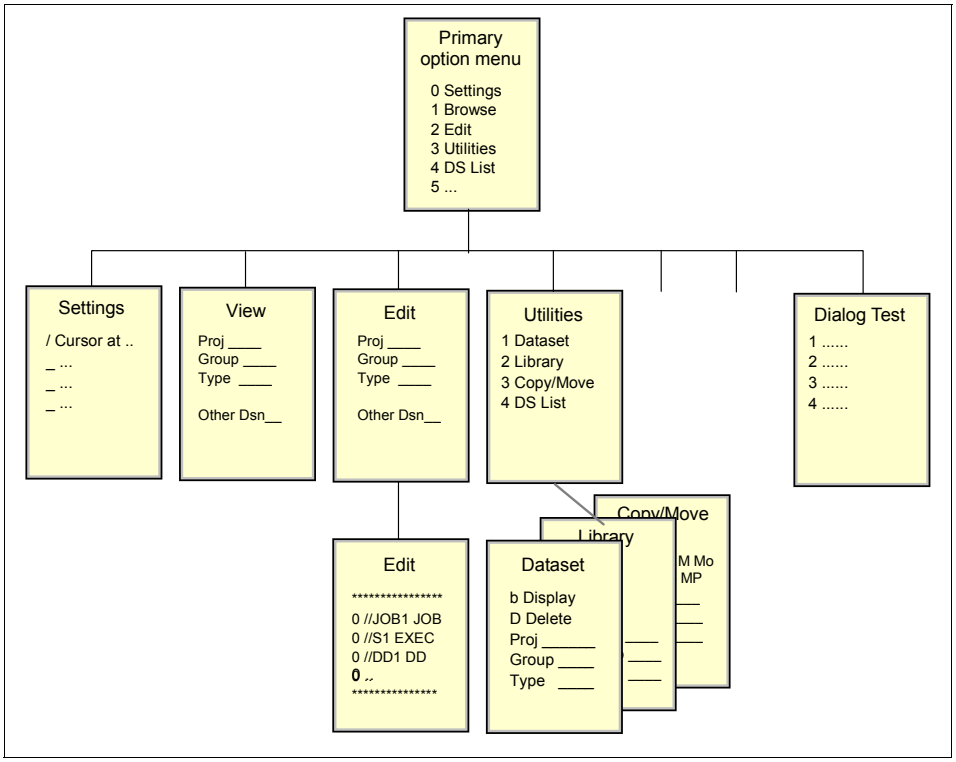


Figure 3-7 ISPF menu structure

To access ISPF under TSO, the user enters a command such as **`ISPPDF`** from the READY prompt to display the ISPF Primary Option Menu.

Figure 3-8 shows an example of the ISPF Primary Menu.

```
Menu  Utilities  Compilers  Options  Status  Help
------------------------------------------------------------------------------
                           ISPF Primary Option Menu
Option ===>

0  Settings      Terminal and user parameters           User ID . : ZPROF
1  View          Display source data or listings        Time. . . : 17:29
2  Edit          Create or change source data           Terminal. : 3278
3  Utilities     Perform utility functions              Screen. . : 1
4  Foreground    Interactive language processing        Language. : ENGLISH
5  Batch         Submit job for language processing     Appl ID . : PDF
6  Command       Enter TSO or Workstation commands      TSO logon : IKJACCT
7  Dialog Test   Perform dialog testing                 TSO prefix: ZPROF
8  LM Facility   Library administrator functions        System ID : SC04
9  IBM Products  IBM program development products       MVS acct. : ACCNT#
10 SCLM          SW Configuration Library Manager       Release . : ISPF 5.2
11 Workplace     ISPF Object/Action Workplace
M  More          Additional IBM Products

Enter X to Terminate using log/list defaults

 F1=Help F2=Split F3=Exit F7=Backward F8=Forward F9=Swap F10=Actions  F12=Cancel
```

Figure 3-8 ISPF Primary Option Menu

The ISPF panel can be customized with additional options by the local system programmer. Therefore, it can vary in features and content from site to site.

To reach the ISPF menu selections shown in Figure 3-9, you enter `M` on the option line.

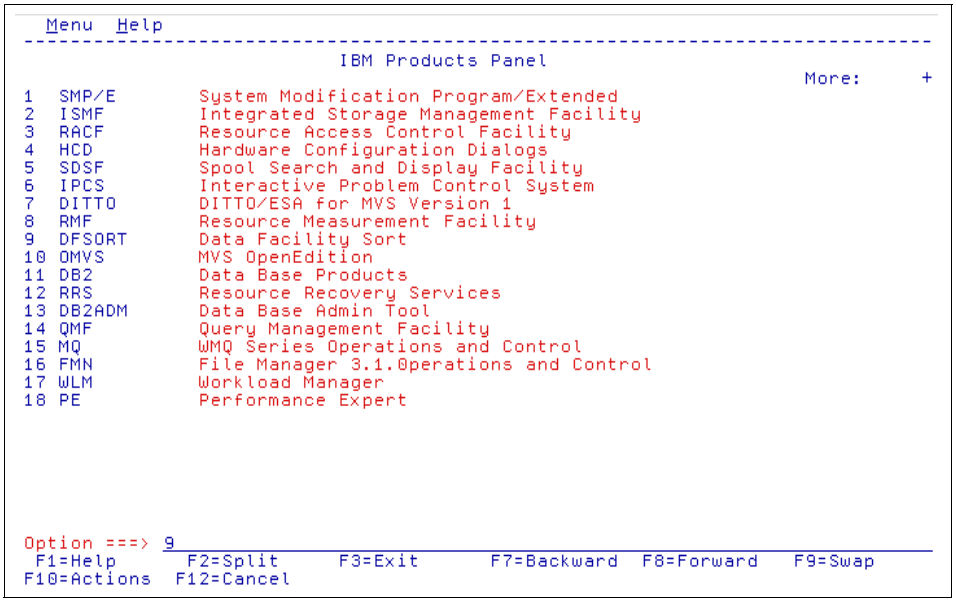


Figure 3-9 More ISPF options displayed

In Figure 3-9, SORT is offered as option 9 on this panel. We will select it now as a useful example of the ISPF panel-driven applications.

Figure 3-10 shows the panel that would be displayed for option 9 of ISPF.

```
DFSORT PRIMARY OPTION MENU
ENTER SELECTION OR COMMAND ===>

SELECT ONE OF THE FOLLOWING:

  0  DFSORT PROFILE              - Change DFSORT user profile
  1  SORT                        - Perform Sort Application
  2  COPY                        - Perform Copy Application
  3  MERGE                       - Perform Merge Application
  X  EXIT                        - Terminate DFSORT

           \-------------------------------------------------/
           | \-----------------------------------------------/ |
           | |      Licensed Materials - Property of IBM      | |
           | |                                                | |
           | |  5740-SM1 (C) Copyright IBM Corp. 1988, 1992.  | |
           | |   All rights reserved.  US Government Users    | |
           | |    Restricted Rights - Use, duplication or     | |
           | |   disclosure restricted by GSA ADP Schedule    | |
           | |           Contract with IBM Corp.              | |
           | /-----------------------------------------------\ |
           /-------------------------------------------------\

USE HELP COMMAND FOR HELP; USE END COMMAND TO EXIT.

 F1=HELP     F2=SPLIT    F3=END      F4=RETURN   F5=RFIND    F6=RCHANGE
 F7=UP       F8=DOWN     F9=SWAP     F10=LEFT    F11=RIGHT   F12=CURSOR
```

Figure 3-10 SORT panel

Recall that 3.2.1, “Using TSO commands in native mode” on page 78 showed an example of how a TSO user might perform a simple sort operation by entering TSO commands in TSO native mode. Here, the same sort function is available through ISPF as a menu-selectable option. Through the SORT option, the user can allow ISPF to handle the TSO allocations, create the SORT control statement, and call the SORT program to produce the results of the sort.

Notice the keyboard program function key (PF key) selections at the bottom of each panel; using PF3 (END) returns the user to the previous panel.

3.3.1 Keyboard mapping used in this course

Many of the screen capture examples used in this course show ISPF program function (PF) key settings at the bottom of the panel. As previously mentioned, because it is

common for z/OS users to customize the PF key assignments to suit their needs, the key assignments shown in this course might not match the PF key settings in use on your system. Actual function key settings vary from customer to customer.

Table 3-1 lists some of the most frequently used PF keys and other keyboard functions and their corresponding keys.

Table 3-1 Keyboard mapping

Function	Key
Enter	Ctrl (right side)
Exit, end, or return	PF3
Help	PF1
PA1 or Attention	Alt-Ins or Esc
PA2	Alt-Home
Cursor movement	Tab or Enter
Clear	Pause
Page up	PF7
Page down	PF8
Scroll left	PF10
Scroll right	PF11
Reset locked keyboard	Ctrl (left side)

The examples in this course use these keyboard settings. For example, directions to press Enter mean that you should press the keyboard's control key (Ctrl) at the lower right. If the keyboard locks up, press the control key at the lower left.

3.3.2 Using PF1-HELP and the ISPF tutorial

From the ISPF Primary Menu, press the PF1 HELP key to display the ISPF tutorial. New users of ISPF should acquaint themselves with the tutorial (Figure 3-11) and with the extensive online help facilities of ISPF.

```
Tutorial -------------------- Table of Contents -------------------- Tutorial

                  ISPF Program Development Facility Tutorial

The following topics are presented in sequence, or may be selected by entering
a selection code in the option field:
     G  General      - General information about ISPF
     0  Settings     - Specify terminal and user parameters
     1  View         - Display source data or output listings
     2  Edit         - Create or change source data
     3  Utilities    - Perform utility functions
     4  Foreground   - Invoke language processors in foreground
     5  Batch        - Submit job for language processing
     6  Command      - Enter TSO command, CLIST, or REXX exec
     7  Dialog Test  - Perform dialog testing
     9  IBM Products - Use additional IBM program development products
     10 SCLM         - Software Configuration and Library Manager
     11 Workplace    - ISPF Object/Action Workplace
     X  Exit         - Terminate ISPF using log and list defaults
     The following topics will be presented only if selected by number:
     A  Appendices   - Dynamic allocation errors and ISPF listing formats
     I  Index        - Alphabetical index of tutorial topics

F1=Help      F2=Split     F3=Exit      F4=Resize    F5=Exhelp    F6=Keyshelp
  F7=PrvTopic  F8=NxtTopic  F9=Swap     F10=PrvPage  F11=NxtPage  F12=Cancel
```

Figure 3-11 ISPF Tutorial Main Menu

You will most likely only use a fraction of content found in the entire ISPF tutorial.

Besides the tutorial, you can access online help from any of the ISPF panels. When you invoke help, you can scroll through information. Press the PF1-HELP key for explanations of common ISPF entry mistakes, and examples of valid entries. ISPF Help also contains help for the various functions found in the primary option menu.

3.3.3 Using the PA1 key

We interrupt your reading enjoyment with a brief commercial for the PA1 key. This is a very important key for TSO users and every user should know how to find it on the keyboard.

Back in the early days, the “real” 3270 terminals had keys labeled PA1, PA2, and PA3. These were called Program Action keys or *PA* keys. In practice, only PA1 is still widely used and it functions as a break key for TSO. In TSO terminology, this is an attention interrupt. That is, pressing the PA1 key will end the current task.

Finding the PA1 key on the keyboard of a 3270 terminal emulator such as TN3270 emulator can be a challenge. A 3270 emulator can be customized to many different key combinations. On an unmodified x3270 session, the PA1 key is Left Alt-1.

Let's give PA1 a try (you'll find it useful in the future). If you've got a TSO session open now, try this:

1. Go to ISPF option 6. This panel accepts TSO commands.
2. Enter LISTC LEVEL(SYS1) ALL on the command line and press Enter. This should produce a screen of output with three asterisks (*******) in the last line on the screen. In TSO, the *** indicates that there is more output waiting and you must press Enter to see it (this meaning is consistent in almost all TSO usage).
3. Press Enter for the next screen, and press Enter for the next screen, and so forth. This command produces lots of output, although it is not an endless loop.
4. Press the PA1 key, using whatever key combination is appropriate for your TN3270 emulator. This should terminate the output.

3.3.4 Navigating through ISPF menus

ISPF includes a course editor and browser, and functions for locating and listing data sets and performing other utility functions. This course has not yet discussed *data sets*, but you will need at least a working understanding of data sets to begin the lab exercises in this chapter.

For now, think of a data set as a file used on z/OS to store data or executable code. A data set can have a name up to 44 characters in length, such as `ZSCHOLAR.TEST.DATA`. Data sets are described in more detail in Chapter 4, "Working with data sets" on page 107.

A data set name is usually segmented, with one or more periods used to create the separate data set *qualifiers* of 1 to 8 characters. The first data set qualifier is the high level qualifier or HLQ. In this example, the HLQ is the `ZSCHOLAR` portion of the data set name.

z/OS users typically use the ISPF data set list utility to work with data sets. To access this utility from the ISPF Primary Option Menu, select **Utilities**, then select **Dslist** to display the Utility Selection Panel, which is shown in Figure 3-12.

```
Menu  RefList  RefMode  Utilities  Help
--------------------------------------------------------------------------------
                           Data Set List Utility
Option ===> ____________________________________________________________

   blank Display data set list               P Print data set list
       V Display VTOC information           PV Print VTOC information

Enter one or both of the parameters below:
  Dsname Level . . . ZPROF_____________________
   Volume serial  . .  ______
Data set list options
   Initial View . . . 1  1. Volume       Enter "/" to select option
                         2. Space        /  Confirm Data Set Delete
                         3. Attrib       /  Confirm Member Delete
                         4. Total        /  Include Additional Qualifiers

When the data set list is displayed, enter either:
  "/" on the data set list command field for the command prompt pop-up,
  an ISPF line command, the name of a TSO command, CLIST, or REXX exec, or
  "=" to execute the previous command.

 F1=Help F2=Split F3=Exit F7=Backward  F8=Forward   F9=Swap F10=Actions  F12=Cancel
```

Figure 3-12 Using the Data Set List utility

In the panel, you can use the Dsname Level data entry field to locate and list data sets. To search for one data set in particular, enter the complete (or *fully qualified*) data set name. To search for a range of data sets, such as all data sets sharing a common HLQ, enter only the HLQ in the Dsname Level field.

Qualifiers can be specified fully, partially, or defaulted. At least one qualifier must be partially specified. To search for a portion of a name, specify an asterisk (*) before or after part of a data set name. Doing so will cause the utility to return all data sets that match the search criteria. Avoid searching on * alone, because TSO has many places to search in z/OS so this could take quite awhile.

In the majority of ISPF panels, a fully qualified data set name needs to be enclosed in single quotes. Data set names not enclosed in single quotes will, by default, be prefixed with a high level qualifier specified in the TSO PROFILE. This default can be changed using the Profile Prefix command. In addition, an exception is ISPF option 3.4 DSLIST; do not enclose Dsname Level in quotes on the ISPF option 3.4 DSLIST panel.

For example, if you enter *ZPROF* in the Dsname field, the utility lists all data sets with *ZPROF* as a high-level qualifier. The resulting list of data set names (see Figure 3-13) allows the user to edit or browse the contents of any data set in the list.

```
 Menu  Options  View  Utilities  Compilers  Help
 ------------------------------------------------------------------------------
DSLIST - Data Sets Matching ZPROF                              Row 1 of 4
Command ===>                                                 Scroll ===> PAGE

Command - Enter "/" to select action                Message          Volume
------------------------------------------------------------------------------
         ZPROF                                                       *ALIAS
         ZPROF.JCL.CNTL                                              EBBER1
         ZPROF.LIB.SOURCE                                            EBBER1
         ZPROF.PROGRAM.CNTL                                          EBBER1
         ZPROF.PROGRAM.LOAD                                          EBBER1
         ZPROF.PROGRAM.SRC                                           EBBER1
****************************** End of Data Set list ****************************


F1=Help F2=Split F3=Exit F5=Rfind F7=Up F8=Down F9=Swap F10=Left F11=Right F12=Cancel
```

Figure 3-13 Data set list results for Dsname ZPROF

To see all of the possible actions you might take for a given data set, specify a forward slash (/) in the command column to the left of the data set name. ISPF will display a list of possible actions, as shown in Figure 3-14.

```
 Menu  Options  View  Utilities  Compilers  Help
 - +-------------------------------------------------------------+ ----------
 D !                    Data Set List Actions                      ! Row 1 of 4
 C !                                                               !  ===> PAGE
   ! Data Set: ZPROF.PROGRAM.CNTL                                !
 C !                                                               !     Volume
 - ! DSLIST Action                                                 ! -----------
   ! __  1.  Edit                    12. Compress                 !     *ALIAS
 / !     2.  View                    13. Free                     !     EBBER1
   !     3.  Browse                  14. Print Index              !     EBBER1
   !     4.  Member List             15. Reset                    !     EBBER1
 * !     5.  Delete                  16. Move                     ! ***********
   !     6.  Rename                  17. Copy                     !
   !     7.  Info                    18. Refadd                   !
   !     8.  Short Info              19. Exclude                  !
   !     9.  Print                   20. Unexclude 'NX'           !
   !     10. Catalog                 21. Unexclude first 'NXF'    !
   !     11. Uncatalog               22. Unexclude last 'NXL'     !
   !                                                              !
   ! Select a choice and press ENTER to process data set action.  !
   !  F1=Help        F2=Split       F3=Exit        F7=Backward    !
   !  F8=Forward     F9=Swap       F12=Cancel                     !
   +-------------------------------------------------------------+

F1=Help F2=Split F3=Exit F5=Rfind F7=Up F8=Down F9=Swap F10=Left F11=Right F12=Cancel
```

Figure 3-14 Displaying the Data Set List actions

3.3.5 Using the ISPF editor

To edit a data set's contents, enter an `e` (edit) to the left of the data set name. In a data set, each line of course is known as a *record*.

You can perform the following tasks:

- To view a data set's contents, enter a `v` (view) as a line command in the column.
- To edit a data set's contents, enter an `e` (edit) as a line command in the column.
- To edit the contents of a data set, move the cursor to the area of the record to be changed and type over the existing course.
- To find and change course, you can enter commands on the editor command line.
- To insert, copy, delete, or move course, place these commands directly on the line numbers where the action should occur.

To commit your changes, use PF3 or save. To exit the data set without saving your changes, enter `Cancel` on the edit command line.

Figure 3-15 shows the contents of data set *ZPROF*.PROGRAM.CNTL(SORTCNTL) opened in edit mode.

```
 File      Edit      Edit_Settings      Menu      Utilities      Compilers      Test      Help
-------------------------------------------------------------------------------

EDIT      ZPROF.PROGRAM.CNTL(SORTCNTL) - 01.00      Columns 00001 00072

Command ===>                                        Scroll ===> CSR

****** ***************************** Top of Data ******************************

000010 SORT FIELDS=(1,3,CH,A)

****** **************************** Bottom of Data ****************************
```

Figure 3-15 Edit a data set

Take a look at the line numbers, the course area, and the editor command line. Primary command line, line commands placed on the line numbers, and course overtype are three different ways in which you can modify the contents of the data set. Line numbers increment by 10 with the TSO editor so that the programmer can insert nine additional lines between each current line without having to renumber the program.

3.3.6 Using the online help

Remember your private tutor, F1=Help, when editing data sets. PF1 in edit mode displays the entire editor tutorial (Figure 3-16).

```
TUTORIAL ------------------------- EDIT ---------------------------- TUTORIAL
OPTION  ===>

                    ----------------------------------
                    |             EDIT               |
                    ----------------------------------

  Edit allows you to create or change source data.

The following topics are presented in sequence, or may be selected by number:
  0 - General introduction              8 - Display modes (CAPS/HEX/NULLS)
  1 - Types of data sets                9 - Tabbing (hardware/software/logical)
  2 - Edit entry panel                 10 - Automatic recovery
  3 - SCLM edit entry panel            11 - Edit profiles
  4 - Member selection list            12 - Edit line commands
  5 - Display screen format            13 - Edit primary commands
  6 - Scrolling data                   14 - Labels and line ranges
  7 - Sequence numbering               15 - Ending an edit session

The following topics will be presented only if selected by number:
 16 - Edit models
 17 - Miscellaneous notes about edit


 F1=Help      F2=Split     F3=Exit      F4=Resize    F5=Exhelp    F6=Keyshelp
 F7=PrvTopic  F8=NxtTopic  F9=Swap     F10=PrvPage  F11=NxtPage  F12=Cancel
```

Figure 3-16 Edit Help Panel and Tutorial

During the lab, you will edit a data set and use F1=Help to explore the Edit Line Commands and Edit Primary Commands functions. Within the help function, select and review the FIND, CHANGE, and EXCLUDE commands. This lab is important for developing further skills in this course.

A subset of the line commands includes:

i	insert a line
Enter key	Press Enter without entering anything to escape insert mode
i5	obtain 5 input lines
d	delete a line
d5	delete 5 lines
dd/dd	delete a block of lines
r	repeat a line
rr/rr	repeat a block of lines
c, then **a** or **b**	copy a line after or before
c5, then **a** or **b**	copy 5 lines after or before
cc/cc, then **a** or **b**	copy a block of lines after or before

m, **m5**, **mm/mm**	to move lines
x	exclude a line

3.3.7 Customizing your ISPF settings

The command line for your ISPF session might appear at the bottom of the display, while your instructor's ISPF command line might appear at the top. This is a personal preference, but traditional usage places it at the top of the panel.

If you want your command line to appear at the top of the panel, do the following:

1. Go to the ISPF primary option menu.
2. Select option **0** to display the Settings menu, as shown in Figure 3-17 on page 96.
3. In the list of Options, remove the "/" on the line that says "Command line at bottom." Use the Tab or New line key to move the cursor.

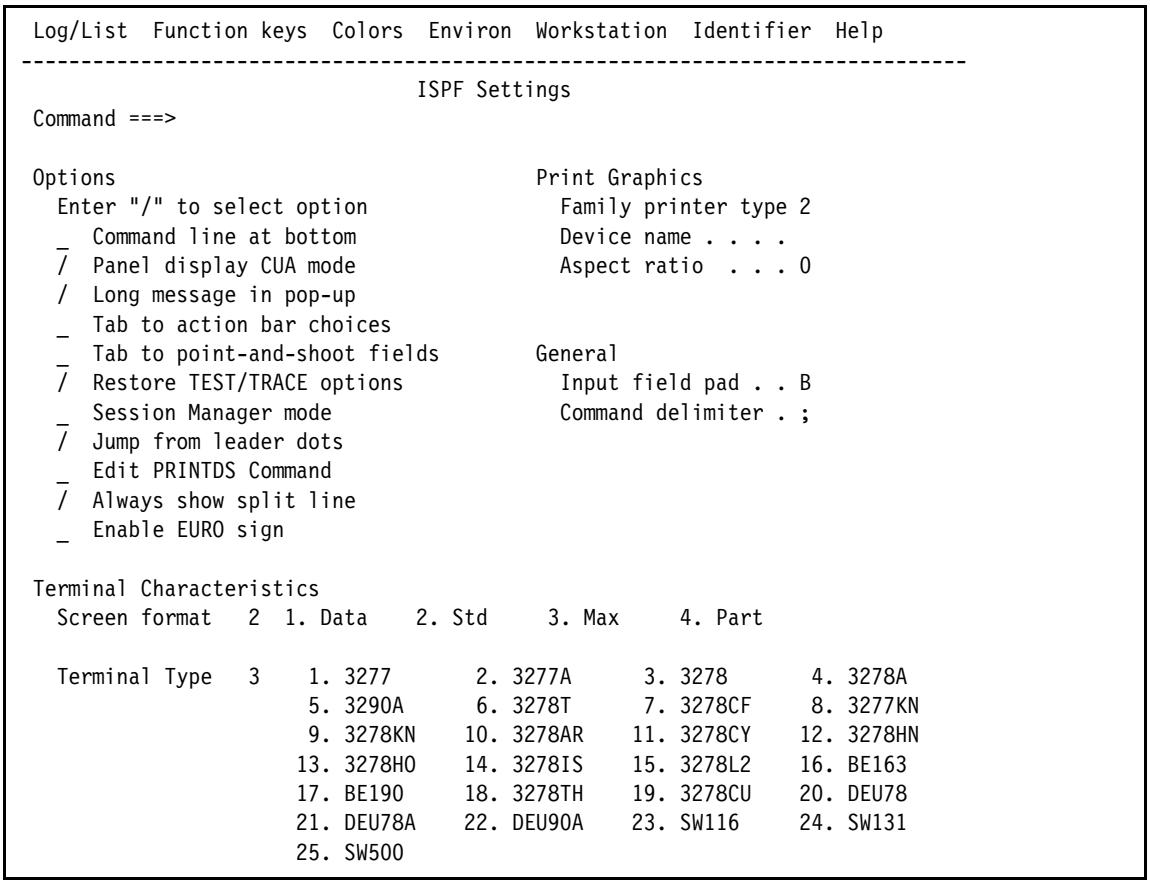


Figure 3-17 ISPF settings

While in this menu, you can change some other parameters that you will need later:

- Remove the "/" from `Panel display CUA mode`.
- Change the `Terminal Type` to 4. This provides 3270 support for symbols used by the C language.
- Move the cursor to the `Log/List` option in the top line and press Enter.
  - Select **1** (Log Data set defaults).
  - Enter Process Option 2 (to delete the data set without printing).
  - Press PF3 to exit.

- Move the cursor to the `Log/List` option again.

- Select **2** (List Data set defaults).
- Enter Process Option 2 to delete the data set without printing.
- PF3 to exit.

► Press PF3 again to exit to the primary menu.

The actions in the bar across the top usually vary from site to site.

Another way to customize ISPF panels is with the **hilite** command, as shown in Figure 3-18. This command allows you to tailor various ISPF options to suit the needs of your environment.

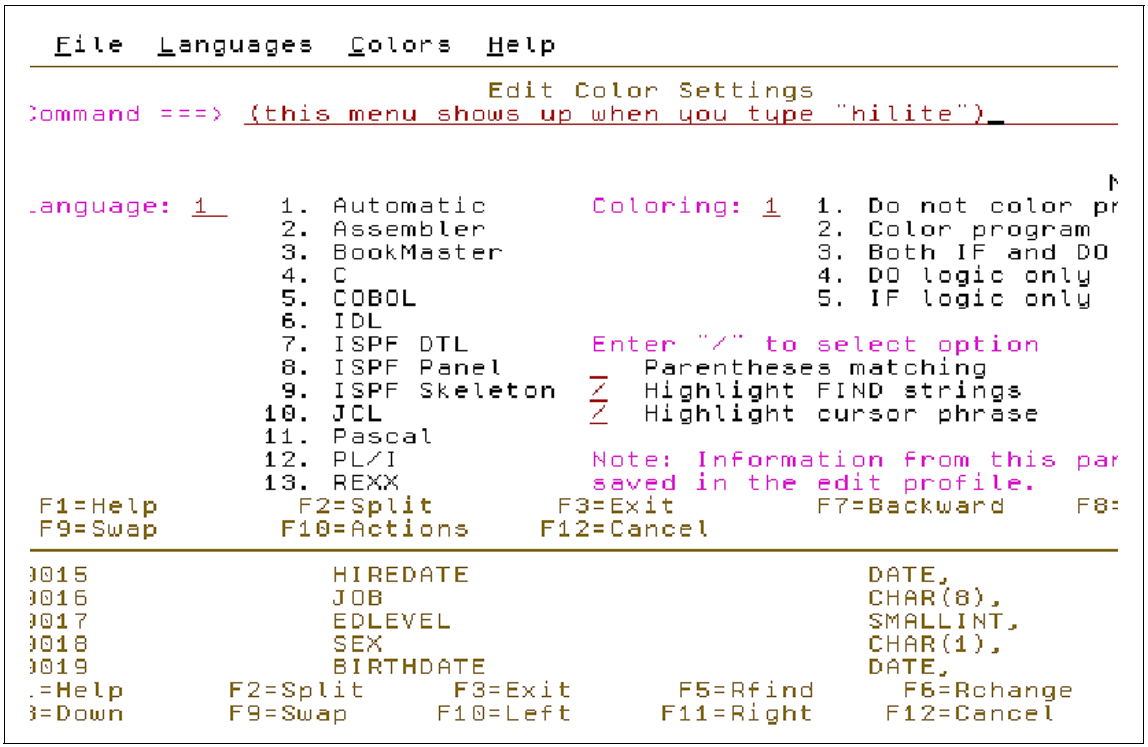


Figure 3-18 Using the hilite command

3.3.8 Adding a GUI to ISPF

ISPF is a full panel application navigated by keyboard. You can, however, download and install a variety of ISPF graphical user interface (GUI) clients to include with a z/OS system. After installing the ISPF GUI client, it is possible to use the mouse.

Figure 3-19 shows an example of an ISPF GUI.

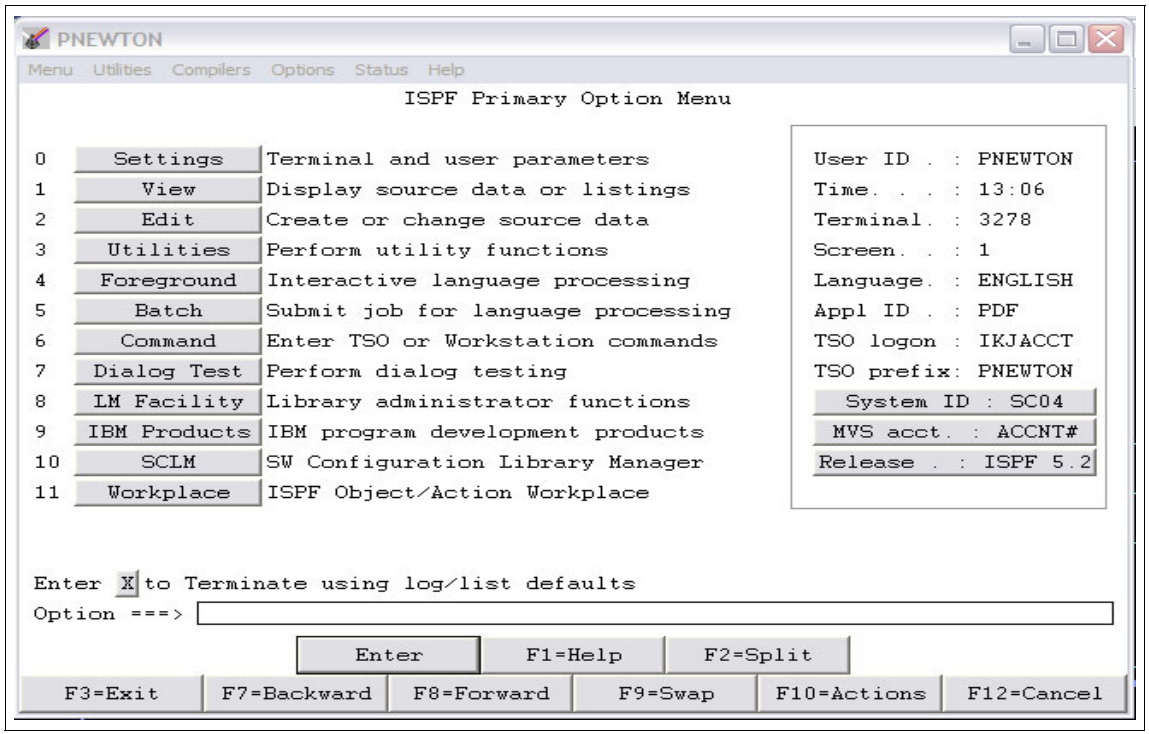


Figure 3-19 ISPF GUI

The drop-down entries at the top of the ISPF panels require you to place the cursor on the selection and press Enter. Move the ISPF GUI client mouse pointer across the drop-down selections to display the respective sub-selections. Also available in the GUI are Enter and PF key boxes.

3.4 z/OS UNIX interactive interfaces

The z/OS UNIX shell and utilities provide an interactive interface to z/OS. The shell and utilities can be compared to the TSO function in z/OS.

To perform some command requests, the shell calls other programs, known as *utilities*. The shell can be used to:

- Invoke shell scripts and utilities.
- Write shell scripts (a named list of shell commands, using the shell programming language).
- Run shell scripts and C language programs interactively, in the TSO background or in batch.

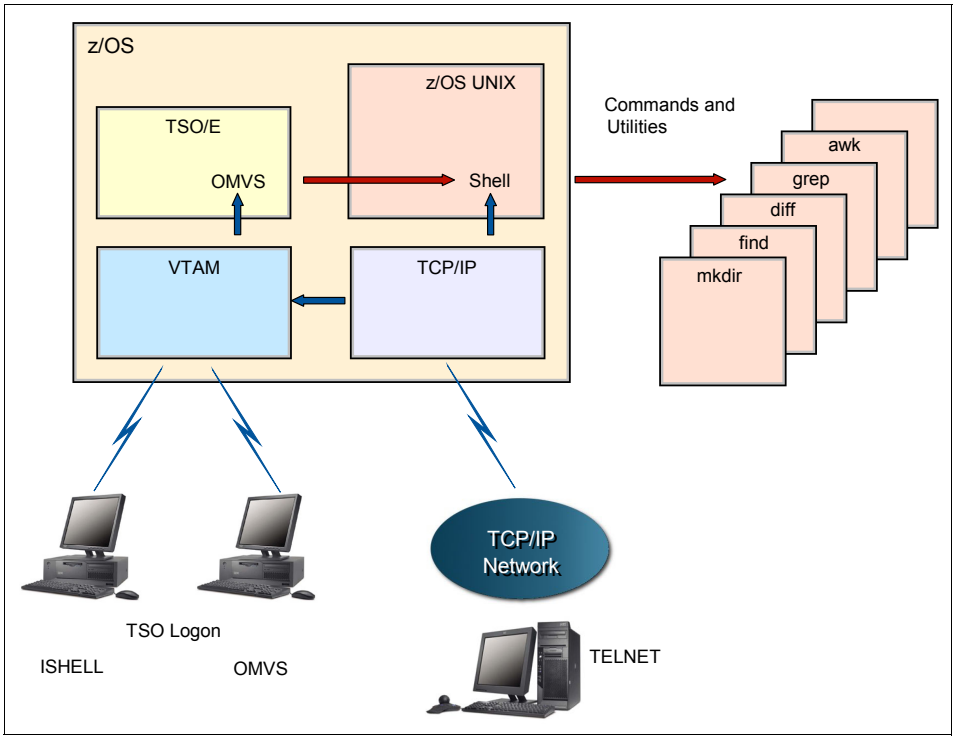


Figure 3-20 Shell and utilities

A user can invoke the sz/OS UNIX shell in the following ways:

- From a 3270 display or a workstation running a 3270 emulator
- From a TCP/IP-attached terminal, using the **`rlogin`** and **`telnet`** commands
- From a TSO session, using the OMVS command.

As an alternative to invoking the shell directly, a user can use ISHELL by entering the command ISHELL from TSO. ISHELL provides an ISPF panel interface to perform many actions for z/OS UNIX operations.

Figure 3-21 shows an overview of these interactive interfaces, the z/OS UNIX shell and ISHELL. Also, there are some TSO/E commands that support z/OS UNIX, but they are limited to functions such as copying files and creating directories.

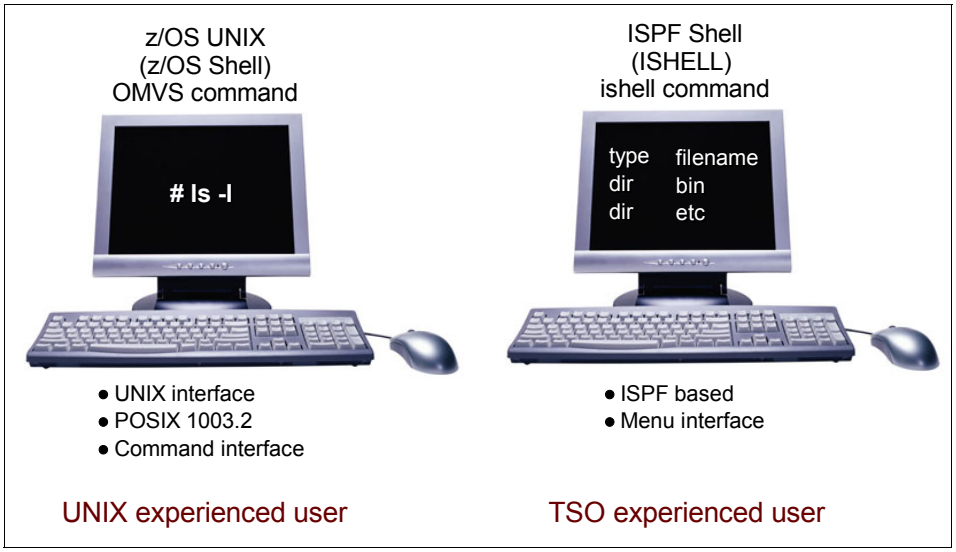


Figure 3-21 z/OS UNIX interactive interfaces

The z/OS UNIX shell is based on the UNIX System V shell and has some of the features from the UNIX Korn shell. The POSIX standard distinguishes between a *command*, which is a directive to the shell to perform a specific task, and a *utility*, which is the name of a program callable by name from the shell. To the user, there is no difference between a command and a utility.

The z/OS UNIX shell provides the environment that has the most functions and capabilities. Shell commands can easily be combined in pipes or shell scripts, and thereby become powerful new functions. A sequence of shell commands can be stored in a course file that can be executed. This is called a *shell script*. The shell supports many of the features of a regular programming language.

The TSO commands used with z/OS UNIX are:

ISHELL The ISHELL command invokes the ISPF panel interface to z/OS UNIX system Services. ISHELL is a good starting point for users familiar with TSO and ISPF who want to use z/OS UNIX. These users can do much of their work with ISHELL, which provides panels for working with the

z/OS UNIX file system, including panels for mounting and unmounting file systems and for doing some z/OS UNIX administration.

ISHELL is often good for system programmers, familiar with z/OS, who need to set up UNIX resources for the users.

OMVS The OMVS command is used to invoke the z/OS UNIX shell.

Users whose primary interactive computing environment is a UNIX system should find the z/OS UNIX shell environment familiar.

3.4.1 ISHELL command (ish)

Figure 3-22 shows the ISHELL or ISPF Shell panel displayed as a result of the ISHELL or ISH command being entered from ISPF Option 6.

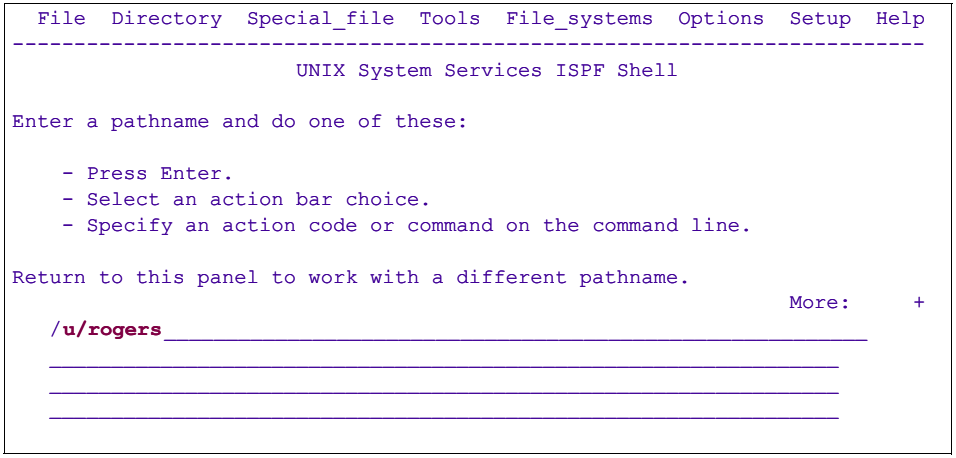


Figure 3-22 Panel displayed after issuing the ISH command

3.4.2 ISHELL - user files and directories

To search a user's files and directories, type the following and then press Enter:

```
/u/userid
```

For example, Figure 3-23 shows the files and directories of user `rogers`.

```
                              Directory List

Select one or more files with / or action codes.  If / is used also select an
action from the action bar otherwise your default action will be used.  Select
with S to use your default action.  Cursor select can also be used for quick
navigation.  See help for details.
EUID=0   /u/rogers/
  Type  Perm  Changed-EST5EDT   Owner      ------Size  Filename    Row 1 of 9
_ Dir    700  2002-08-01 10:51  ADMIN            8192  .
_ Dir    555  2003-02-13 11:14  AAAAAAA             0  ..
_ File   755  1996-02-29 18:02  ADMIN              979  .profile
_ File   600  1996-03-01 10:29  ADMIN               29  .sh_history
_ Dir    755  2001-06-25 17:43  AAAAAAA          8192  data
_ File   644  2001-06-26 11:27  AAAAAAA         47848  inventory.export
_ File   700  2002-08-01 10:51  AAAAAAA            16  myfile
_ File   644  2001-06-22 17:53  AAAAAAA         43387  print.export
_ File   644  2001-02-22 18:03  AAAAAAA         84543  Sc.pdf
```

Figure 3-23 Display of a user's files and directories

From here, you use action codes to do any of the following:

b Browse a file or directory
e Edit a file or directory
d Delete a file or directory
r Rename a file or directory
a Show the attributes of a file or directory
c Copy a file or directory

3.4.3 OMVS command shell session

You use the OMVS command to invoke the z/OS UNIX shell.

The shell is a command processor that you use to:

- Invoke shell commands or utilities that request services from the system.
- Write shell scripts using the shell programming language.
- Run shell scripts and C-language programs interactively (in the foreground), in the background, or in batch.

Shell commands often have options (also known as *flags*) that you can specify, and they usually take an argument, such as the name of a file or directory. The format for specifying the command begins with the command name, then the option or options, and finally the argument, if any.

For example, in Figure 3-24 on page 103 the following command is shown:

```
ls -al /u/rogers
```

where `ls` is the command name, and `-al` are the options.

```
ROGERS @ SC43:/>ls -al /u/rogers
total 408
drwx------   3 ADMIN    SYS1        8192 Aug  1  2005 .
dr-xr-xr-x  93 AAAAAAA  TTY            0 Feb 13 11:14 ..
-rwxr-xr-x   1 ADMIN    SYS1         979 Feb 29  1996 .profile
-rw-------   1 ADMIN    SYS1          29 Mar  1  1996 .sh_history
-rw-r--r--   1 AAAAAAA  SYS1       84543 Feb 22  2001 Sc.pdf
drwxr-xr-x   2 AAAAAAA  SYS1        8192 Jun 25  2001 data
-rw-r--r--   1 AAAAAAA  SYS1       47848 Jun 26  2001 inventory.export
-rwx------   1 AAAAAAA  SYS1          16 Aug  1  2005 myfile
-rw-r--r--   1 AAAAAAA  SYS1       43387 Jun 22  2001 print.export
```

Figure 3-24 OMVS shell session display after issuing the OMVS command

This command lists the files and directories of the user. If the pathname is a file, `ls` displays information on the file according to the requested options. If it is a directory, `ls` displays information on the files and subdirectories therein. You can get information on a directory itself by using the `-d` option.

If you do not specify any options, `ls` displays only the file names. When `ls` sends output to a pipe or file, it writes one name per line; when it sends output to the terminal, it uses the `-C` (multi-column) format.

Terminology note: z/OS users tend to use the terms *data set* and *file* synonymously, but not when it comes to z/OS UNIX System Services. With the UNIX support in z/OS, the file system is a data set that contains directories and files. So *file* has a very specific definition. z/OS UNIX files are different from other z/OS data sets because they are byte-oriented rather than record-oriented.

3.4.4 Direct login to the shell

You can login directly to the z/OS UNIX shell from a system that is connected to z/OS through TCP/IP. Use one of the following methods:

rlogin You can rlogin (remote log in) to the shell from a system that has rlogin client support. To log in, use the **`rlogin`** command syntax supported at your site.

telnet You can telnet into the shell. To log in, use the telnet command from your workstation or from another system with telnet client support.

As shown in Figure 3-25 on page 104, each of these methods requires the inetd daemon to be set-up and active on the z/OS system.

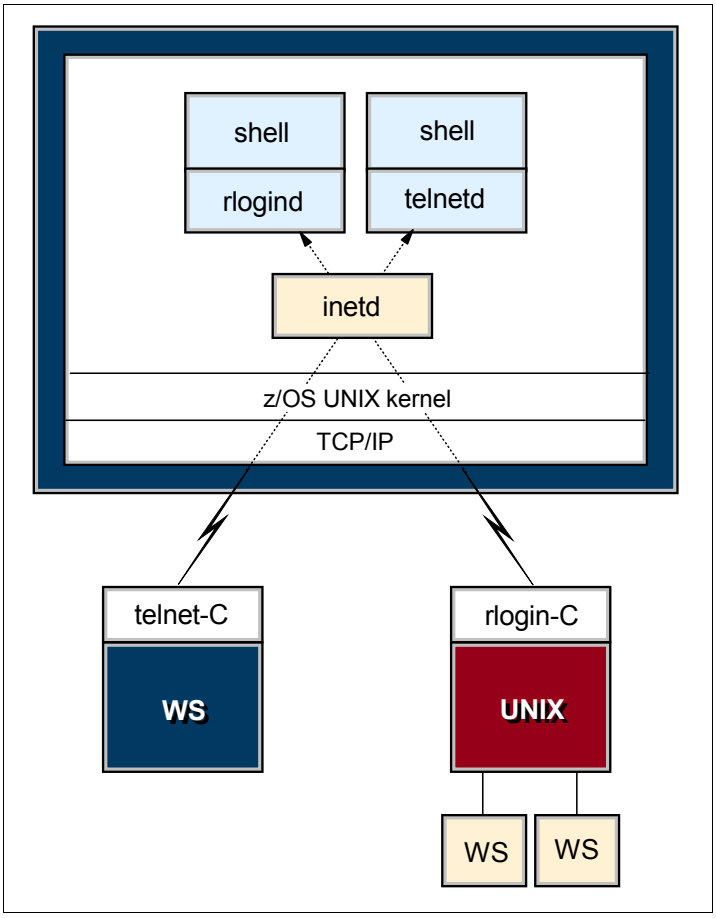


Figure 3-25 Diagram of a login to the shell from a terminal

Figure 3-26 shows the z/OS shell after login through telnet.

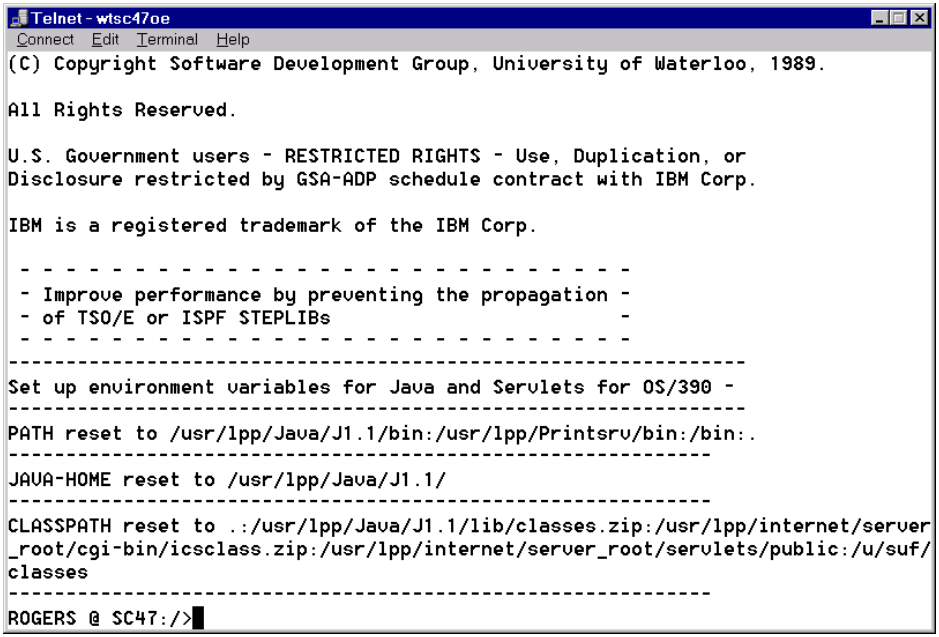


Figure 3-26 Telnet login to the shell screen

There are some differences between the asynchronous terminal support (direct shell login) and the 3270 terminal support (OMVS command):

- You cannot switch to TSO/E. However, you can use the TSO SHELL command to run a TSO/E command from your shell session.
- You cannot use the ISPF editor (this includes the **oedit** command, which invokes ISPF edit).
- You can use the UNIX vi editor, and other interactive utilities that depend on receiving each keystroke, without hitting the Enter key.
- You can use UNIX-style command-line editing.

3.5 Summary

TSO allows users to log on to z/OS and use a limited set of basic commands. This is sometimes called using TSO in its native mode.

ISPF is a menu-driven interface for user interaction with a z/OS system. The ISPF environment is executed from native TSO.

ISPF provides utilities, an editor and ISPF applications to the user. To the extent permitted by various security controls an ISPF user has full access to most z/OS system functions.

TSO/ISPF should be viewed as a system management interface and a development interface for traditional z/OS programming.

The z/OS UNIX shell and utilities provide a command interface to the z/OS UNIX environment. You can access the shell either by logging on to TSO/E or by using the remote login facilities of TCP/IP (rlogin).

If you use TSO/E, a command called OMVS creates a shell for you. You can work in the shell environment until exiting or temporarily switching back to the TSO/E environment.

Key terms in this chapter		
3270 and 3270 emulator	CLIST	ISHELL
ISPF	logon	native mode
OMVS command	path or pathname	READY prompt
Restructured Extended Executor (REXX)	shell	Time Sharing Option / Extensions (TSO/E)

4

Working with data sets

Objective: In working with the z/OS operating system, you must understand *data sets*, the files that contain programs and data. The characteristics of traditional z/OS data sets differ considerably from the file systems used in UNIX and PC systems. To matters even more interesting, you can also create UNIX file systems on z/OS, with the common characteristics of UNIX systems.

After completing this chapter, you will be able to:

- Explain what a data set is
- Describe data set naming conventions and record formats
- List some access methods for managing data and programs
- Explain what catalogs and VTOCs are used for
- Create, delete and modify data sets
- Explain the differences between UNIX file systems and z/OS data sets
- Describe the z/OS UNIX file systems' use of data sets

© Copyright IBM Corp. 2005, 2006. All rights reserved.

4.1 What is a data set?

Nearly all work in the system involves data input or data output. In a mainframe system, the channel subsystem manages the use of I/O devices, such as disks, tapes, and printers, while z/OS associates the data for a given task with a device.

z/OS manages data by means of *data sets*. The term *data set* refers to a file that contains one or more records. Any named group of records is called a data set. Data sets can hold information such as medical records or insurance records, to be used by a program running on the system. Data sets are also used to store information needed by applications or the operating system itself, such as source programs, macro libraries, or system variables or parameters. For data sets that contain readable course, you can print them or display them on a console (many data sets contain load modules or other binary data that is not really printable). Data sets can be *cataloged*, which permits the data set to be referred to by name without specifying where the data set is stored.

In simplest terms, a *record* is a fixed number of bytes containing data. Often, a record collects related information that we treat as a unit, such as one item in a database or personnel data about one member of a department. The term *field* refers to a specific portion of a record used for a particular category of data, such as an employee's name or department.

The record is the basic unit of information used by a program running on z/OS[1]. The records in a data set can be organized in various ways, depending on how we plan to access the information. If you write an application program that processes things like personnel data, for example, your program can define a record format for each person's data.

There are many different types of data sets in z/OS, and different methods for accessing them. This chapter discusses three types of data sets: sequential, partitioned, and VSAM data sets.

In a *sequential data set*, records are data items that are stored consecutively. To retrieve the tenth item in the data set, for example, the system must first pass the preceding nine items. Data items that must all be used in sequence, like the alphabetical list of names in a classroom roster, are best stored in a sequential data set.

A *partitioned data set* or *PDS* consists of a directory and *members*. The directory holds the address of each member and thus makes it possible for programs or the operating system to access each member directly. Each member, however, consists of sequentially stored records. Partitioned data sets are often called *libraries*. Programs are stored as members of partitioned data sets. Generally, the operating system loads a PDS's members

[1] z/OS UNIX files are different from the typical z/OS data set because they are byte-oriented rather than record-oriented.

into storage sequentially, but it can access members directly when selecting a program for execution.

In a *Virtual Storage Access Method* (*VSAM*) *key sequenced data set* (*KSDS*), records are data items that are stored with control information (keys) so that the system can retrieve an item without searching all preceding items in the data set. VSAM KSDS data sets are ideal for data items that are used frequently and in an unpredictable order.

We discuss the different types of data sets and the use of catalogs later in this chapter.

4.2 Where are data sets stored?

z/OS supports many different devices for data storage. Disks or tape are most frequently used for storing data sets on a long term basis. Disk drives are known as *direct access storage devices* (*DASDs*) because, although some data sets on them might be stored sequentially, these devices can handle direct access. Tape drives are known as sequential access devices because data sets on tape must be accessed sequentially.

The term *DASD* applies to disks or simulated equivalents of disks. All types of data sets can be stored on DASD (only sequential data sets can be stored on magnetic tape). You use DASD volumes for storing data and executable programs, including the operating system itself, and for temporary working storage. You can use one DASD volume for many different data sets, and reallocate or reuse space on the volume.

To allow the system to locate a specific data set quickly, z/OS includes a data set known as the master catalog that permits access to any of the data sets in the computer system or to other catalogs of data sets. z/OS requires that the master catalog reside on a DASD that is always mounted on a drive that is online to the system. We discuss catalogs further in 4.12, “Catalogs and VTOCs” on page 125.

4.3 Role of DFSMS in managing space

In a z/OS system, space management involves the allocation, placement, monitoring, migration, backup, recall, recovery, and deletion of data sets. These activities can be done either manually or through the use of automated processes. When data management is automated, the operating system determines object placement, and automatically manages data set backup, movement, space, and security. A typical z/OS production system includes both manual and automated processes for managing data sets.

Depending on how a z/OS system and its storage devices are configured, a user or program can directly control many aspects of data set use, and in the early days of the operating system, users were required to do so. Increasingly, however, z/OS customers rely on installation-specific settings for data and resource management, and additional

space management products, such as DFSMS, to automate the use of storage for data sets.

Data management includes these main tasks:

- Sets aside (allocates) space on DASD volumes
- Automatically retrieves cataloged data sets by name
- Mounts magnetic tape volumes in the drive
- Establishes a logical connection between the application program and the medium
- Controls access to data
- Transfers data between the application program and the medium

The primary means of managing space in z/OS is through the DFSMS component of the operating system. DFSMS performs the essential data, storage, program, and device management functions of the system. DFSMS is a set of products, and one of these products, DSFMSdfp, is required for running z/OS. DFSMS, together with hardware products and installation-specific settings for data and resource management, provides system-managed storage in a z/OS environment.

The heart of DFSMS is the Storage Management Subsystem (SMS). Using SMS, the system programmer or storage administrator defines policies that automate the management of storage and hardware devices. These policies describe data allocation characteristics, performance and availability goals, backup and retention requirements, and storage requirements for the system. SMS governs these policies for the system and the Interactive Storage Management Facility (ISMF) provides the user interface for defining and maintaining the policies.

The data sets allocated through SMS are called system-managed data sets or SMS-managed data sets. When you allocate or define a data set to use SMS, you specify your data set requirements by using a data class, a storage class, and a management class. Typically, you do not need to specify these classes because a storage administrator has set up automatic class selection (ACS) routines to determine which classes to use for a data set.

DFSMS uses a set of constructs, user interfaces, and routines (using the DFSMS products) that allow the storage administrator to better manage the storage system. The core logic of DFSMS, such as the ACS routines, ISMF code, and constructs, is located in DFSMSdfp™. DFSMShsm™ and DFSMSdss™ are involved in the management class construct.

With DFSMS, the z/OS system programmer or storage administrator can define performance goals and data availability requirements, create model data definitions for typical data sets, and automate data backup. DFSMS can automatically assign, based on installation policy, those services and data definition attributes to data sets when they are created. IBM storage management-related products determine data placement, manage data backup, control space usage, and provide data security.

4.4 What are access methods?

An access method defines the technique that is used to store and retrieve data. Access methods have their own data set structures to organize data, system-provided programs (or *macros*) to define data sets, and utility programs to process data sets.

Access methods are identified primarily by the data set organization. z/OS users, for example, use the basic sequential access method (BSAM) or queued sequential access method (QSAM) with sequential data sets.

However, there are times when an access method identified with one organization can be used to process a data set organized in a different manner. For example, a sequential data set (not extended-format data set) created using BSAM can be processed by the basic direct access method (BDAM), and vice versa. Another example is UNIX files, which you can process using BSAM, QSAM, basic partitioned access method (BPAM), or virtual storage access method (‘).

This course does not describe all of the access methods available on z/OS. Commonly used access methods include the following:

QSAM	Queued Sequential Access Method (heavily used)
BSAM	Basic Sequential Access Method (for special cases)
BDAM	Basic Direct Access Method (becoming obsolete)
BPAM	Basic Partitioned Access Method (for libraries)
VSAM	Virtual Sequential Access Method (used for more complex applications)

4.5 How are DASD volumes used?

DASD volumes are used for storing data and executable programs (including the operating system itself), and for temporary working storage. One DASD volume can be used for many different data sets, and space on it can be reallocated and reused.

On a volume, the name of a data set must be unique. A data set can be located by device type, volume serial number, and data set name. This is unlike the file tree of a UNIX system. The basic z/OS file structure is not hierarchical. z/OS data sets have no equivalent to a path name.

Although DASD volumes differ in physical appearance, capacity, and speed, they are similar in data recording, data checking, data format, and programming. The recording surface of each volume is divided into many concentric *tracks*. The number of tracks and their capacity vary with the device. Each device has an access mechanism that contains read/write heads to transfer data as the recording surface rotates past them.

4.5.1 DASD terminology for UNIX and PC users

The disk and data set characteristics of mainframe hardware and software differ considerably from UNIX and PC systems, and carry their own specialized terminology. Throughout this course, the following terms are used to describe various aspects of storage management on z/OS

- *Direct Access Storage Device* (DASD) is another name for a disk drive.
- A disk drive is also known as a disk volume, a disk pack, or a *Head Disk Assembly* (HDA). We use the term *volume* in this course except when discussing physical characteristics of devices.
- A disk drive contains cylinders.
- Cylinders contain tracks.
- Tracks contain data records and are in *Count Key Data* (CKD) format.[2]
- Data blocks are the units of recording on disk.

4.5.2 What are DASD labels?

The operating system uses groups of labels to identify DASD volumes and the data sets they contain. Customer application programs generally do not use these labels directly. DASD volumes must use standard labels. Standard labels include a volume label, a data set label for each data set, and optional user labels. A volume label, stored at track 0 of cylinder 0, identifies each DASD volume.

The z/OS system programmer or storage administrator uses the ICKDSF utility program to initialize each DASD volume before it is used on the system. ICKDSF generates the volume label and builds the volume table of contents (VTOC), a structure that contains the data set labels (we discuss VTOCs in “What is a VTOC?” on page 126). The system programmer can also use ICKDSF to scan a volume to ensure that it is usable and to reformat all of the tracks.

4.6 Allocating a data set

To use a data set, you first allocate it (establish a link to it), then access the data using macros for the access method that you have chosen.

The *allocation* of a data set means either or both of two things:

- To set aside (create) space for a new data set on a disk.
- To establish a logical link between a job step and any data set.

At the end of this chapter, we follow an exercise to allocate a data set using ISPF panel option 3.2. Other ways to allocate a data set include the following methods:

[2] Current devices actually use *Extended Count Key Data* (ECKD™) protocols, but we use CKD as a collective name in the course.

Access Method Services	You can allocate data sets through a multifunction services program called access method services. Access method services include commonly used commands for working with data sets, as ALLOCATE, ALTER, DELETE, and PRINT.
ALLOCATE command	You can use the TSO ALLOCATE command to create data sets. The command actually guides you through the allocation values that you must specify
Using JCL	You can use job control language to allocate data sets (we discuss the use of JCL in Chapter 6, "Using JCL and SDSF" on page 153

4.7 Allocating space on DASD volumes through JCL

This section describes allocating a data set as you would do it using job control language (JCL). We discuss the use of JCL later in this book; this section previews some of the data set space allocation settings you will use in Chapter 6, "Using JCL and SDSF" on page 153. Besides JCL, other common methods for allocating data sets include the IDCAMS utility program, or using DFSMS to automate the allocation of data sets.

In JCL, you can specify the amount of space required in blocks, records, tracks, or cylinders. When creating a DASD data set, you specify the amount of space needed explicitly through the SPACE parameter, or implicitly by using the information available in a data class.[3]

The system can use a data class if SMS is active even if the data set is not SMS managed. For system-managed data sets, the system selects the volumes, saving you from having to specify a volume when you allocate a data set.

If you specify your space request by average record length, space allocation is independent of device type. Device independence is especially important to system-managed storage.

4.7.1 Logical records and blocks

A logical record length (LRECL) is a unit of information about a unit of processing (for example, a customer, an account, a payroll employee, and so on). It is the smallest amount of data to be processed, and it is comprised of fields which contain information recognized by the processing application.

Logical records, when located on DASD, tape, or optical devices, are grouped within physical records named blocks. BLKSIZE indicates the length of those blocks. Each

[3] When allocating a data set through DFSMS or the IDCAMS utility program, you can specify space allocations in kilobytes or megabytes, rather than blocks, records, tracks, or cylinders.

block of data on a DASD volume has a distinct location and a unique address, thus making it possible to find any block without extensive searching. Logical records can be stored and retrieved either directly or sequentially.

The maximum length of a logical record (LRECL) is limited by the physical size of the used media.

When the amount of space required is expressed in blocks, you must specify the number and average length of the blocks within the data set.

We have not yet discussed JCL, but here is a typical example of how space is allocated on a DASD volume through JCL:

```
//  DD SPACE=(300,(5000,100)), ...
```

In this example, the values indicate a storage request as follows:

- 300 = average block length in bytes
- 5000 = primary quantity (number of blocks)
- 100 = secondary quantity, to be allocated if the primary quantity is not enough (in blocks)

From this information, the operating system estimates and allocates the number of tracks required.

4.7.2 Data set extents

Space for a disk data set is assigned in *extents*. An extent is a *contiguous* number of disk drive tracks, cylinders, or blocks. Data sets can increase in extents as they grow. Older types of data sets can have up to 16 extents per volume. Newer types of data sets can have up to 128 extents per volume or 255 extents total on multiple volumes.

Extents are relevant when you are not using PDSEs and have to manage the space yourself, rather than through DFSMS. Here, you want the data set to fit into a single extent to maximize disk performance. Reading or writing contiguous tracks is faster than reading or writing tracks scattered over the disk, as might be the case if tracks were allocated dynamically. But if there is not sufficient contiguous space, a data set goes into extents.

4.8 Data set record formats

Traditional z/OS data sets are *record oriented*. In normal usage, there are no byte stream files such as are found in PC and UNIX systems. (z/OS UNIX has byte stream files, and byte stream functions exist in other specialized areas. These are not considered to be traditional data sets.)

In z/OS, there are no new line (NL) or carriage return and line feed (CR+LF) characters to denote the end of a record. Records are either fixed length or variable length in a given data set. When editing a data set with ISPF, for example, each line is a record.

Traditional z/OS data sets have one of five record formats, as follows:

F - Fixed	This means that one physical block on disk is one logical record and all the blocks/records are the same size. This format is seldom used.
FB - Fixed Blocked	This means that several logical records are combined into one physical block. This can provide efficient space utilization and operation. This format is commonly used for fixed-length records.
V - Variable	This format has one logical record as one physical block. The application is required to insert a four-byte Record Descriptor Word (RDW) at the beginning of the record. The RDW contains the length of the record plus the four bytes for the RDW. This format is seldom used.
VB - Variable Blocked	This format places several variable-length logical records (each with an RDW) in one physical block. The software must place an additional Block Descriptor Word (BDW) at the beginning of the block, containing the total length of the block.
U - Undefined	This format consists of variable-length physical records/blocks with no predefined structure. Although this format may appear attractive for many unusual applications, it is normally used only for executable modules.

We must stress the difference between a block and a record. A block is what is written on disk, while a record is a logical entity.

The terminology here is pervasive throughout z/OS literature. The key terms are:

- Block Size (BLKSIZE) is the physical block size written on the disk for F and FB records. For V, VB, and U records it is the maximum physical block size that can be used for the data set.
- Logical Record Size (LRECL) is the logical record size (F, FB) or the maximum allowed logical record size (V, VB) for the data set. Format U records have no LRECL.
- Record Format (RECFM) is F, FB, V, VB, or U as just described.

These terms are known as data control block (DCB) characteristics, named for the control block where they may be defined in an assembly language program. The user is often expected to specify these parameters when creating a new data set. The type and

length of a data set is defined by its record format (RECFM) and logical record length (LRECL). Fixed-length data sets have a RECFM of F, FB, FBS, and so on. Variable-length data sets have a RECFM of V, VB, VBS, and so on.

A data set with RECFM=FB and LRECL=25 is a fixed-length (FB) data set with a record length of 25-bytes (the B is for blocked). For an FB data set, the LRECL tells you the length of each record in the data set; all of the records are the same length. The first data byte of an FB record is in position 1. A record in an FB data set with LRECL=25 might look like this:

```
Positions 1-3:  Country Code = 'USA'
Positions 4-5:  State Code = 'CA'
Positions 6-25:  City = 'San Jose' padded with 12 blanks on the right
```

A data set with RECFM=VB and LRECL=25 is a variable-length (VB) data set with a maximum record length of 25-bytes. For a VB data set, different records can have different lengths. The first four bytes of each record contain the Record Descriptor Word or RDW, and the first two bytes of the RDW contain the length of that record (in binary). The first data byte of a VB record is in position 5, after the 4-byte RDW in positions 1-4. A record in a VB data set with LRECL=25 might look like this:

```
Positions 1-2:    Length in RDW = hex 000E = decimal 14
Positions 3-4:    Zeros in RDW = hex 0000 = decimal 0
Positions 5-7:    Country Code = 'USA'
Positions 8-9:    State Code =  'CA'
Positions 10-17:  City = 'San Jose'
```

Figure 4-1 on page 117 shows the relationship between records and blocks for each of the five record formats.

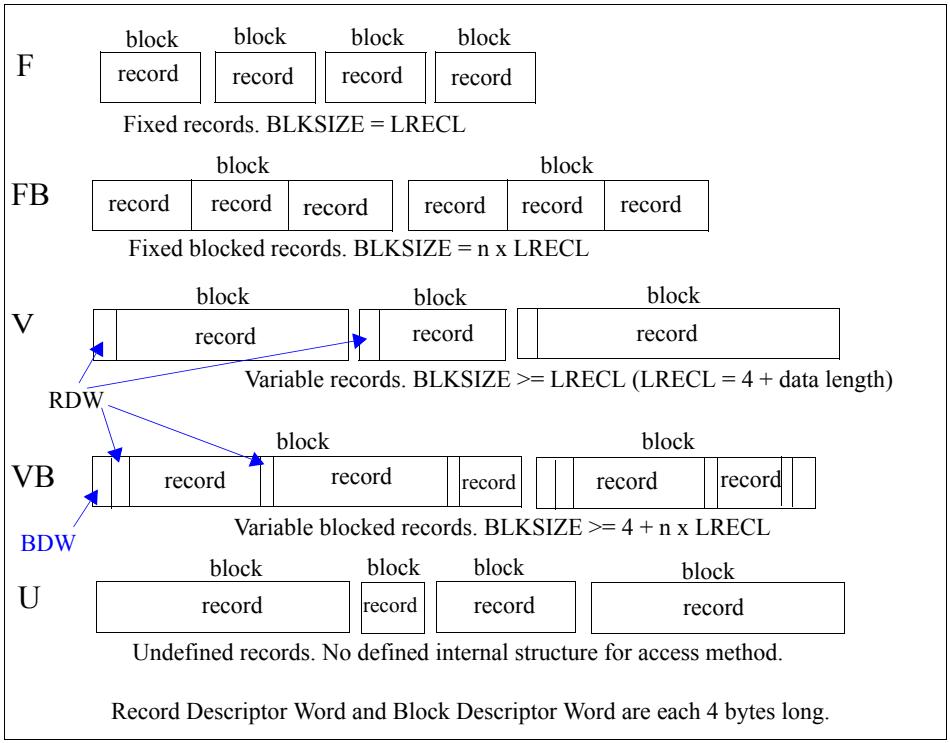


Figure 4-1 Basic record formats

4.9 Types of data sets

There are many different types of data sets in z/OS, and different methods for managing them. This chapter discusses three types: sequential, partitioned, and VSAM. These are all used for disk storage; we mention tape data sets briefly as well.

4.9.1 What is a sequential data set?

The simplest data structure in a z/OS system is a sequential data set. It consists of one or more records that are stored in physical order and processed in sequence. New records are appended to the end of the data set.

An example of a sequential data sets might be an output data set for a line printer or a deck of punch cards.

A z/OS user defines sequential data sets through job control language (JCL) with a data set organization of PS (DSORG=PS), which stands for physical sequential. In other words, the records in the data set are physically arranged one after another.

This chapter covers mainly disk data sets, but mainframe applications might also use tape data sets for many purposes. Tapes store sequential data sets. Mainframe tape drives have variable-length records (blocks). The maximum block size for routine programming methods is 65K bytes. Specialized programming can produce longer blocks. There are a number of tape drive products with different characteristics.

4.9.2 What is a PDS?

A *partitioned data set* adds a layer of organization to the simple structure of sequential data sets. A PDS is a collection of sequential data sets, called *members.* Each member is like a sequential data set and has a simple name, which can be up to eight characters long.

PDS also contains a directory. The directory contains an entry for each member in the PDS with a reference (or pointer) to the member. Member names are listed alphabetically in the directory, but members themselves can appear in any order in the library. The directory allows the system to retrieve a particular member in the data set.

A partitioned data set is commonly referred to as a *library.* In z/OS, libraries are used for storing source programs, system and application control parameters, JCL, and executable modules. There are very few system data sets that are not libraries.

A PDS loses space whenever a member is updated or added. As a result, z/OS users regularly need to compress a PDS to recover the lost space.

A z/OS user defines a PDS through JCL with a data set organization of PO (DSORG=PO), which stands for partitioned organization.

Why is a PDS structured like that?

The PDS structure was designed to provide efficient access to libraries of related members, whether they be load modules, program source modules, JCL or many other types of content.

Many system data sets are also kept in PDS data sets, especially when they consist of many small, related files. For example, the definitions for ISPF panels are kept in PDS data sets.

A primary use of ISPF is to create and manipulate PDS data sets. These data sets typically consist of source code for programs, course for manuals or help screens, or JCL to allocate data sets and run programs.

Advantages of a PDS

A PDS data set offers a simple and efficient way to organize related groups of sequential files. A PDS has the following advantages for z/OS users:

- Grouping of related data sets under a single name makes z/OS data management easier. Files stored as members of a PDS can be processed either individually or all the members can be processed as a unit.
- Because the space allocated for z/OS data sets always starts at a track boundary on disk, using a PDS is a way to store more than one small data set on a track. This saves you disk space if you have many data sets that are much smaller than a track. A track is 56,664 bytes for 3390 disk device.
- Members of a PDS can be used as sequential data sets, and they can be appended (or *concatenated*) to sequential data sets.
- Multiple PDS data sets can be concatenated to form large libraries.
- PDS data sets are easy to create with JCL or ISPF; they are easy to manipulate with ISPF utilities or TSO commands.

Disadvantages of a PDS

PDS data sets are simple, flexible, and widely used. However, some aspects of the PDS design affect both performance and the efficient use of disk storage, as follows:

- **Wasted space.** When a member in a PDS is replaced, the new data area is written to a new section within the storage allocated to the PDS. When a member is deleted, its pointer is deleted too, so there is no mechanism to reuse its space. This wasted space is often called *gas* and must be periodically removed by reorganizing the PDS, for example, by using the utility IEBCOPY to compress it.
- **Limited directory size.** The size of a PDS directory is set at allocation time. As the data set grows, it can acquire more space in units of the amount you specified as its secondary space. These extra units are called *secondary extents*.

  However, you can only store a fixed number of member entries in the PDS directory because its size is fixed when the data set is allocated. If you need to store more entries than there is space for, you have to allocate a new PDS with more directory blocks and copy the members from the old data set into it. This means that when you allocate a PDS, you must calculate the amount of directory space you need.
- **Lengthy directory searches.** As mentioned earlier, an entry in a PDS directory consists of a name and a pointer to the location of the member. Entries are stored in alphabetical order of the member names. Inserting an entry near the front of a large directory can cause a large amount of I/O activity, as all the entries behind the new one are moved along to make room for it.

  Entries are also searched sequentially in alphabetical order. If the directory is very large and the members small, it might take longer to search the directory than to retrieve the member when its location is found.

4.9.3 What is a PDSE?

A PDSE is a partitioned data set extended. It consists of a directory and zero or more members, just like a PDS. It can be created with JCL, TSO/E, and ISPF, just like a PDS, and can be processed with the same access methods. PDSE data sets are stored only on DASD, not tape.

The directory can expand automatically as needed, up to the addressing limit of 522,236 members. It also has an index, which provides a fast search for member names. Space from deleted or moved members is automatically reused for new members, so you do not have to compress a PDSE to remove wasted space. Each member of a PDSE can have up to 15,728,639 records. A PDSE can have a maximum of 123 extents, but it cannot extend beyond one volume. When a directory of a PDSE is in use, it is kept in processor storage for fast access.

PDSE data sets can be used in place of nearly all PDS data sets that are used to store data. But PDSE format is not intended as a PDS replacement. When a PDSE is used to store load modules, it stores them in structures called *program objects*.

PDS versus PDSE

In many ways, a PDSE is similar to a PDS. Each member name can be eight bytes long. For accessing a PDS directory or member, most PDSE interfaces are indistinguishable from PDS interfaces. Both PDS and PDSE data sets are processed using the same access methods (BSAM, QSAM, BPAM). And, in case you were wondering, within a given PDS or PDSE, the members must use the same access method.

However, PDSE data sets have a different internal format, which gives them increased usability. You can use a PDSE in place of a PDS to store data, or to store programs in the form of program objects. A program object is similar to a load module in a PDS. A load module cannot reside in a PDSE and be used as a load module. One PDSE cannot contain a mixture of program objects and data members.

PDSE data sets have several features that can improve user productivity and system performance. The main advantage of using a PDSE over a PDS is that a PDSE automatically reuses space within the data set without the need for anyone to periodically run a utility to reorganize it.

Also, the size of a PDS directory is fixed regardless of the number of members in it, while the size of a PDSE directory is flexible and expands to fit the members stored in it.

Further, the system reclaims space automatically whenever a member is deleted or replaced, and returns it to the pool of space available for allocation to other members of the same PDSE. The space can be reused without having to do an IEBCOPY compress.

Other advantages of PDSE data sets follow:

- PDSE members can be shared. This makes it easier to maintain the integrity of the PDSE when modifying separate members of the PDSE at the same time.
- Reduced directory search time. The PDSE directory, which is indexed, is searched using that index. The PDS directory, which is organized alphabetically, is searched sequentially. The system might cache in storage directories of frequently used PDSE data sets.
- Creation of multiple members at the same time. For example, you can open two DCBs to the same PDSE and write two members at the same time.
- PDSE data sets contain up to 123 extents. An extent is a continuous area of space on a DASD storage volume, occupied by or reserved for a specific data set.
- When written to DASD, logical records are extracted from the user's blocks and reblocked. When read, records in a PDSE are reblocked into the block size specified in the DCB. The block size used for the reblocking can differ from the original block size.

4.9.4 When a data set runs out of space

As mentioned earlier, when you allocate a data set, you reserve a certain amount of space in units of blocks, tracks, or cylinders on a storage disk. If you use up that space, you see a message that says: SYSTEM ABEND '0D37' or possibly B37 or E37.

We haven't discussed abend codes in this course, but this problem is something you will need to deal with. If you are in an edit session, you are not able to leave the session until you resolve the problem. Some of the things you can do are:

1. Compress the data set, if it is a PDS, by doing the following:
   a. Split (PF 2) the screen and select the UTILITIES option (option 3).
   b. Select the LIBRARIES option (option 1) on the Utility Selection Menu.
   c. Specify the name of the data set and enter C on the option line.
   d. When the data set is compressed, you see the message: COMPRESS SUCCESSFUL.
   e. You should then be able to swap (PF 9) to the edit session and save the new material.
2. Allocate a larger data set and copy into it by doing the following:
   a. Split (PF 2) the screen and select UTILITIES (option 3), then DATASET (option 2) from the other side of the split.
   b. Specify the name of the data set that received the abend to display its characteristics.
   c. Allocate another data set with more space.
   d. Select the MOVE/COPY option (option 3) on the Utility Selection Menu to copy members from the old data set to the new larger data set.
   e. Browse (option 1) the new data set to make sure everything was copied correctly.

f. Swap (PF 9) back to the abending edit session, type CC on the top line of input, type CC on the bottom line of input, type CREATE on the command line, and press the Enter key.
g. Enter the new, larger data set name and a member name to receive the copied information.
h. You again see the abending edit session. Enter cancel on the command line. Press the RETURN PF key (PF 4) key to exit the edit session.
i. Select the DATASET option (option 2) from the Utility Selection Menu to delete the old data set.
j. Rename the new data set to the old name.

3. Cancel the new material entered in the edit session by entering cancel on the command line. You should then be able to exit without abending, however, all information that was not previously saved is lost.

4.10 What is VSAM?

The term *Virtual Storage Access Method* (VSAM) applies to both a data set type and the access method used to manage various user data types. As an access method, VSAM provides much more complex functions than other disk access methods. VSAM keeps disk records in a unique format that is not understandable by other access methods.

VSAM is used primarily for applications. It is not used for source programs, JCL, or executable modules. VSAM files cannot be routinely displayed or edited with ISPF.

You can use VSAM to organize records into four types of data sets: key-sequenced, entry-sequenced, linear, or relative record. The primary difference among these types of data sets is the way their records are stored and accessed.

VSAM data sets are briefly described as follows:

- *Key Sequence Data Set* (KSDS). This is the most common use for VSAM. Each record has one or more key fields and a record can be retrieved (or inserted) by key value. This provides random access of data. Records are variable length.
- *Entry Sequence Data Set* (ESDS). This form of VSAM keeps records in sequential order. Records can be accessed sequentially. It is used by IMS, DB2, and z/OS UNIX.
- *Relative Record Data Set* (RRDS). This VSAM format allows retrieval of records by number; record 1, record 2, and so forth. This provides random access and assumes the application program has a way to derive the desired record numbers.
- *Linear Data Set* (LDS). This is, in effect, a byte-stream data set and is the only form of a byte-stream data set in traditional z/OS files (as opposed to z/OS UNIX files). A number of z/OS system functions use this format heavily, but it is rarely used by application programs.

There are several additional methods of accessing data in VSAM that are not listed here. Most application use of VSAM is for keyed data.

VSAM works with a logical data area known as a Control Interval (CI) that is diagrammed in Figure 4-2. The default CI size is 4K bytes, but it can be up to 32K bytes. The CI contains data records, unused space, Record Descriptor Fields (RDF), and a CI Descriptor Field.

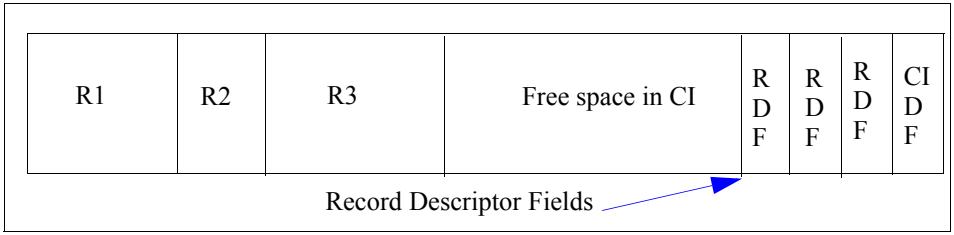


Figure 4-2 Simple VSAM control interval

A control interval may be constructed from smaller disk blocks, but this level of detail is internal to VSAM. Multiple control intervals are placed in a Control Area (CA). A VSAM data set consists of control areas and index records. One form of index record is the sequence set, which is the lowest-level index pointing to a control interval.

Typical use of VSAM permits an application to insert new records in a data set. The structure that permits this is interesting:

- KSDS records in a control interval are in ascending order, based on the primary key of the records. A given control interval contains all the data set records between the record with the lowest key and the record with the highest key in that control interval.
- The RDF fields contain the length of each logical record and the offset into the CI for that record. The CIDF contains information about the free space in the CI.
- When the VSAM data set is first allocated, the user can specify the amount of free space to leave in each control interval. A percentage of free (empty) control intervals in each control area can also be specified.
- When a new record is to be inserted (in key order) VSAM uses the key records to determine the control interval holding records in the appropriate key range.
- If the control interval has sufficient free space the logical records in the control interval are shifted (if necessary) and the new record is inserted.
- If the control interval does not have sufficient space, a control interval split occurs. A free control interval (in the same control area) is located and approximately half the records in the data control interval are moved to the empty control interval and the new record is inserted into the original control interval.
- Index pointers are adjusted as necessary to include the newly used control interval.

- If no free control interval is found in the control area, a control area split occurs. This follows roughly the same logic as a control interval split. If a free control area does not exist (or cannot be added to the data set), the data insert fails and the application most likely fails.

VSAM data is always variable length and records are automatically blocked in control intervals. The RECFM attributes (F, FB, V, VB, U) do not apply to VSAM, nor does the BLKSIZE attribute. The Access Method Services (AMS) utility is used to define and delete VSAM structures, such as files and indexes. Example 4-1 shows an example.

Example 4-1 Defining a VSAM KSDS using AMS

```
DEFINE CLUSTER -
(NAME(VWX.MYDATA) -
VOLUMES(VSER02) -
RECORDS(1000 500)) -
 DATA -
(NAME(VWX.KSDATA) -
 KEYS(15 0) -
RECORDSIZE(250 250) -
BUFFERSPACE(25000) ) -
INDEX -
(NAME(VWX.KSINDEX) -
CATALOG (UCAT1)
```

There are many details of VSAM processing that are not included in this brief description. Most processing is handled transparently by VSAM; the application program merely retrieves, updates, deletes or adds records based on key values.

4.11 How data sets are named

When you allocate a new data set (or when the operating system does), you must give the data set a unique name.

A data set name can be one name segment, or a series of joined name segments. Each name segment represents a level of qualification. For example, the data set name VERA.LUZ.DATA is composed of three name segments. The first name on the left is called the high-level qualifier (HLQ), the last name on the right is the lowest-level qualifier (LLQ).

Segments or *qualifiers* are limited to eight characters, the first of which must be alphabetic (A to Z) or *special* (# @ $). The remaining seven characters are either alphabetic, numeric (0 - 9), special, a hyphen (-). Name segments are separated by a period (.).

Including all name segments and periods, the length of the data set name must not exceed 44 characters. Thus, a maximum of 22 name segments can make up a data set name.

For example, the following names are not valid data set names:

- Name with a qualifier that is longer than eight characters (HLQ.ABCDEFGHI.XYZ)
- Name containing two successive periods (HLQ..ABC)
- Name that ends with a period (HLQ.ABC.)
- Name that contains a qualifier that does not start with an alphabetic or special character (HLQ.123.XYZ)

The HLQ for a user's data sets is typically controlled by the security system. There are a number of conventions for the remainder of the name. These are *conventions*, not rules, but are widely used. They include the following:

- The letters LIB somewhere in the name indicate that the data set is a library. The letters PDS are a lesser-used alternative for this.
- The letters CNTL, JCL, or JOB somewhere in the name typically indicate the data set contains JCL (but might not be exclusively devoted to JCL.)
- The letters LOAD, LOADLIB, or LINKLIB in the name indicate that the data set contains executables. (A library with z/OS executable modules must be devoted solely to executable modules.)
- The letters PROC, PRC, or PROCLIB indicate a library of JCL procedures. These are described in 6.7, "JCL procedures (PROCs)" on page 163.
- Various combinations are used to indicate source code for a specific language, for example COBOL, Assembler, FORTRAN, PL/I, JAVA, C, or C++.
- A portion of a data set name may indicate a specific project, such as PAYROLL.
- Using too many qualifiers is considered poor practice. For example,

  `P390A.A.B.C.D.E.F.G.H.I.J.K.L.M.N.O.P.Q.R.S`

  is a valid data set name (upper case, does not exceed 44 bytes, no special characters) but it is not very meaningful. A good practice is for a data set name to contain three or four qualifiers.
- Again, the periods count toward the 44-character limit.

4.12 Catalogs and VTOCs

z/OS uses a catalog and a volume table of contents (VTOC) on each DASD to manage the storage and placement of data sets; these are described in the sections that follow:

- "What is a VTOC?" on page 126
- "What is a catalog?" on page 127

z/OS also makes it possible to group data sets based on historically related data, as described in "What is a generation data group?" on page 129.

4.12.1 What is a VTOC?

z/OS requires a particular format for disks, which is shown in Figure 4-3 on page 126. Record 1 on the first track of the first cylinder provides the label for the disk. It contains the 6-character volume serial number (volser) and a pointer to the *volume table of contents* (VTOC), which can be located anywhere on the disk.

The VTOC lists the data sets that reside on its volume, along with information about the location and size of each data set, and other data set attributes. A standard z/OS utility program, ICKDSF, is used to create the label and VTOC.

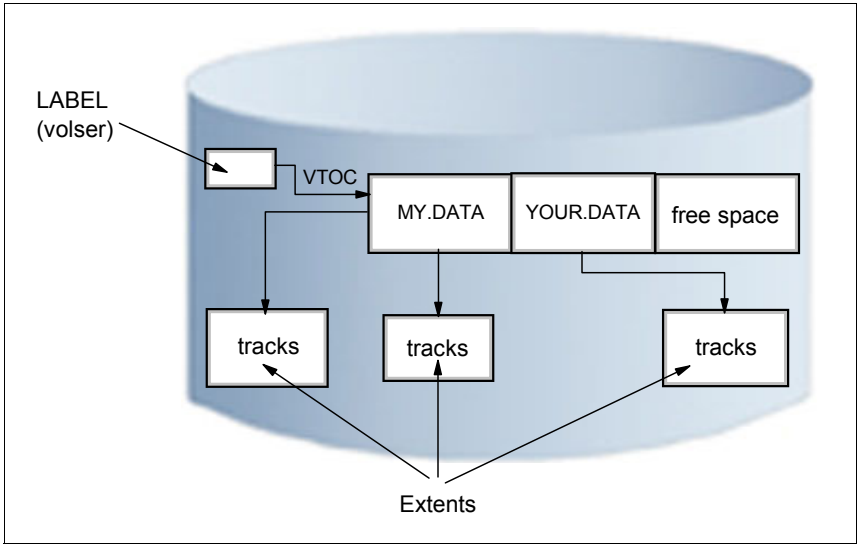


Figure 4-3 Disk label, VTOC, and extents

When a disk volume is initialized with ICKDSF, the owner can specify the location and size of the VTOC. The size can be quite variable, ranging from a few tracks to perhaps 100 tracks, depending on the expected use of the volume. More data sets on the disk volume require more space in the VTOC.

The VTOC also has entries for all the free space on the volume. Allocating space for a data set causes system routines to examine the free space records, update them, and create a new VTOC entry. Data sets are always an integral number of tracks (or cylinders) and start at the beginning of a track (or cylinder).

You can also create a VTOC with an index. The VTOC index is actually a data set with the name SYS1.VTOCIX.*volser,* which has entries arranged alpahabetically by data set name with pointers to the VTOC entries. It also has bitmaps of the free space on the volume. A VTOC index allows the user to find the data set much faster.

4.12.2 What is a catalog?

A catalog describes data set attributes and indicates the volumes on which a data set is located. When a data set is cataloged, it can be referred to by name without the user needing to specify where the data set is stored. Data sets can be cataloged, uncataloged, or recataloged. All system-managed DASD data sets are cataloged automatically in a catalog. Cataloging of data sets on magnetic tape is not required but usually it simplifies users jobs.

In z/OS, the master catalog and user catalogs store the locations of data sets. Both disk and tape data sets can be cataloged.

To find a data set that you have requested, z/OS must know three pieces of information:

- Data set name
- Volume name
- Unit (the volume device type, such as a 3390 disk or 3590 tape)

You can specify all three values on ISPF panels or in JCL. However, the unit device type and the volume are often not relevant to an end user or application program. A system catalog is used to store and retrieve UNIT and VOLUME location of a data set. In its most basic form a catalog can provide the unit device type and volume name for any data set that is cataloged. A system catalog provides a simple look up function. With this facility the user need only provide a data set name.

Master catalogs and user catalogs

A z/OS system always has at least one master catalog. If a z/OS system has a single catalog, this catalog would be the master catalog and the location entries for all data sets would be stored in it. A single catalog, however, would be neither efficient nor flexible, so a typical z/OS system uses a master catalog and numerous *user catalogs* connected to it as shown in Figure 4-4.

A user catalog stores the name and location of a data set (dsn/volume/unit). The master catalog usually stores only a data set HLQ with the name of the user catalog, which contains the location of all data sets prefixed by this HLQ. The HLQ is called an alias.

In Figure 4-4, the data set name of the master catalog is `SYSTEM.MASTER.CATALOG.` This master catalog stores the full data set name and location of all data sets with a SYS1 prefix such as SYS1.A1. Two HLQ (alias) entries were defined to the master catalog, IBMUSER and USER. The statement that defined that IBMUSER included the data set name of the user catalog containing all the fully qualified IBMUSER data sets with their respective location. The same is true for USER HLQ (alias).

When SYS1.A1 is requested, the master catalog returns the location information, volume(WRK001) and unit(3390), to the requestor. When IBMUSER.A1 is requested,

the master catalog redirects the request to USERCAT.IBM, then USERCAT.IBM returns the location information to the requestor.

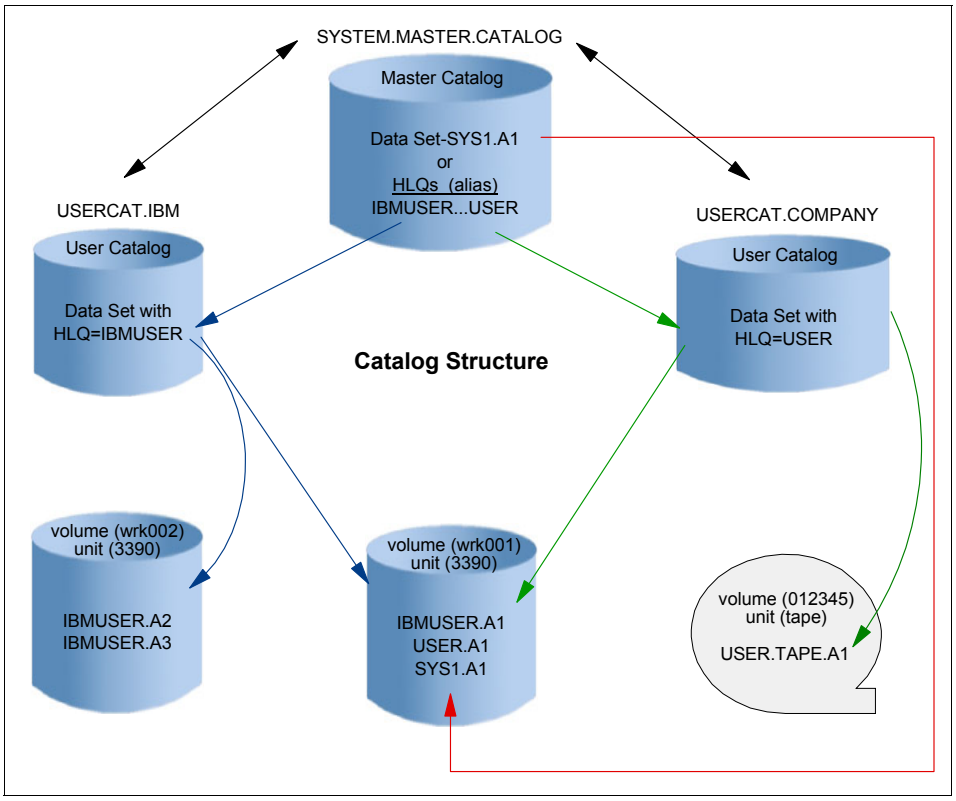


Figure 4-4 Catalog concept

Take, as a further example, the define statements below.

```
DEFINE  ALIAS ( NAME ( IBMUSER ) RELATE ( USERCAT.IBM ) )
DEFINE  ALIAS ( NAME ( USER ) RELATE ( USERCAT.COMPANY ) )
```

These are used to place IBMUSER and USER alias names in the master catalog with the name of the user catalog that will store the fully qualified data set names and location information. If IBMUSER.A1 is cataloged, a JCL statement to allocate it to the job would be:

```
//INPUT DD DSN=IBMUSER.A1,DISP=SHR
```

If IBMUSER.A1 is not cataloged, a JCL statement to allocate it to the job would be:

```
//INPUT DD DSN=IBMUSER.A1,DISP=SHR,VOL=SER=WRK001,UNIT=3390
```

As a general rule, all user data sets in a z/OS installation are cataloged. Uncataloged data sets are rarely needed and their use is often related to recovery problems or installation of new software. Data sets created through ISPF are automatically cataloged.

Using an alternate master catalog

So, what happens if an installation loses its master catalog, or the master catalog somehow becomes corrupted? Such an occurrence would pose a serious problem and require swift recovery actions.

To save this potential headache, most system programmers define a back-up for the master catalog. The system programmer specifies this alternate master catalog during system start-up. In this case, it's recommended that the system programmer keep the alternate on a volume separate from that of the master catalog (to protect against a situation in which the volume becomes unavailable).

4.12.3 What is a generation data group?

In z/OS, it is possible to catalog successive updates or generations of related data. They are called generation data groups (GDGs).

Each data set within a GDG is called a generation data set (GDS) or generation. A generation data group (GDG) is a collection of historically related non-VSAM data sets that are arranged in chronological order. That is, each data set is historically related to the others in the group.

Within a GDG, the generations can have like or unlike DCB attributes and data set organizations. If the attributes and organizations of all generations in a group are identical, the generations can be retrieved together as a single data set.

There are advantages to grouping related data sets. For example:

- All of the data sets in the group can be referred to by a common name.
- The operating system is able to keep the generations in chronological order.
- Outdated or obsolete generations can be automatically deleted by the operating system.

Generation data sets have sequentially ordered absolute and relative names that represent their age. The operating system's catalog management routines use the absolute generation name. Older data sets have smaller absolute numbers. The relative name is a signed integer used to refer to the latest (0), the next to the latest (-1), and so forth, generation.

For example, a data set name LAB.PAYROLL(0) refers to the most recent data set of the group; LAB.PAYROLL(-1) refers to the second most recent data set; and so forth. The relative number can also be used to catalog a new generation (+1). A generation data

group (GDG) base is allocated in a catalog before the generation data sets are cataloged. Each GDG is represented by a GDG base entry.

For new non-system-managed data sets, if you do not specify a volume and the data set is not opened, the system does not catalog the data set. New system-managed data sets are always cataloged when allocated, with the volume assigned from a storage group.

4.13 z/OS UNIX file systems

Think of a UNIX file system as a container that holds a portion of the entire UNIX directory tree. Unlike a traditional z/OS library, a UNIX file system is hierarchical and byte-oriented. To find a file in a UNIX file system, you must search a directory or a series of directories (see Figure 4-5). There is no concept of a z/OS catalog that points directly to a file.

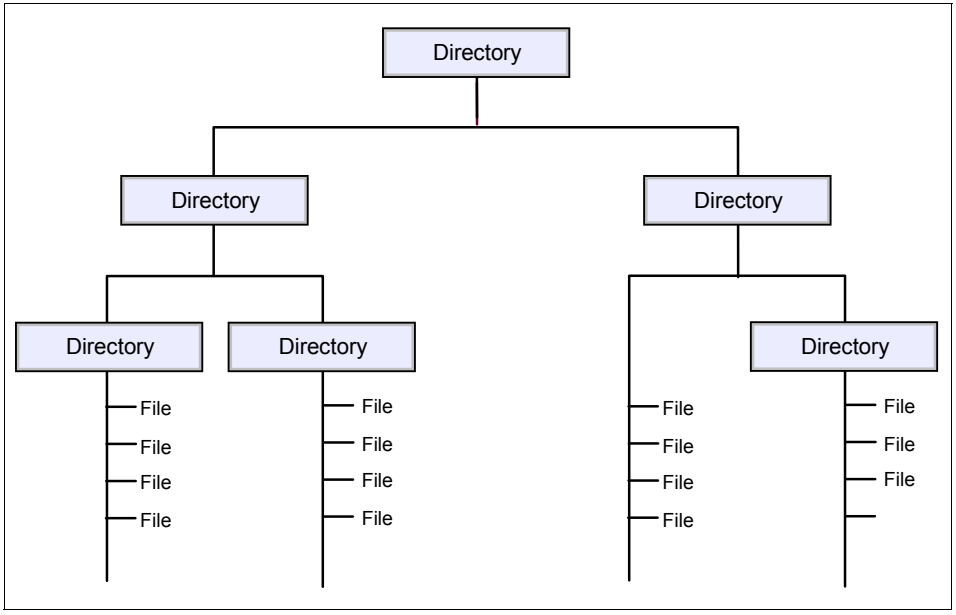


Figure 4-5 Hierarchical file system structure

z/OS UNIX System Services (z/OS UNIX) allows z/OS users to create UNIX file systems and file system directory trees on z/OS, and to access UNIX files on z/OS and other systems. In z/OS, a UNIX file system is mounted over an empty directory by the system programmer (or a user with mount authority).

You can use the following file system types with z/OS UNIX:

- zSeries File System (zFS), which is a file system that stores files in VSAM linear data sets.
- hierarchical file system (HFS), which is a mountable file system.
- z/OS Network File System (z/OS NFS), which allows a z/OS system to access a remote UNIX (z/OS or non-z/OS) file system over TCP/IP, as if it were part of the local z/OS directory tree.
- Temporary file system (TFS), which is a temporary, in-memory physical file system that supports in-storage mountable file systems.

As with other UNIX file systems, a path name identifies a file and consists of directory names and a file name. A fully qualified file name, which consists of the name of each directory in the path to a file plus the file name itself, can be up to 1023 bytes long.

The path name is constructed of individual directory names and a file name separated by the forward-slash character, for example:

/dir1/dir2/dir3/MyFile

Like UNIX, z/OS UNIX is case-sensitive for file and directory names. For example, in the same directory, the file MYFILE is a different file than MyFile.

The files in the hierarchical file system are sequential files, and are accessed as byte streams. A record concept does not exist with these files other than the structure defined by an application.

The zFS data set that contains the UNIX file system is a z/OS data set type (a VSAM linear data set). zFS data sets and z/OS data sets can reside on the same DASD volume. Example 4-2 shows the definition of a zFS data set. Also, z/OS provides commands for managing zFS space utilization.

Example 4-2 Creating an HFS data set using JCL

```
//FSJOB JOB
//STEP1 EXEC PGM=IEFBR14
//MKFS1 DD DSNAME=FILE.SYSTEM.FS001,
//     DISP=(NEW,KEEP),
//     DSNTYPE=HFS,
//     SPACE=(CYL(100,100,1)),
//     DATACLAS=FILESYS,
// MGMTCLAS=NEVER,STORCLAS=SECURE
```

The integration of the zFS file system with existing z/OS file system management services provides automated file system management capabilities that might not be available on other UNIX platforms. This integration allows file owners to spend less time on tasks such as backup and restore of entire file systems.

4.13.1 z/OS data sets versus file system files

Many elements of UNIX have analogs in the z/OS operating system. Consider, for example, that the organization of a user catalog is analogous to a user directory (/u/ibmuser) in the file system.

In z/OS, the user prefix assigned to z/OS data sets points to a user catalog. Typically, one user owns all the data sets whose names begin with his user prefix. For example, the data sets belonging to the TSO/E user ID IBMUSER all begin with the high-level qualifier (prefix) IBMUSER. There could be different data sets named IBMUSER.C, IBMUSER.C.OTHER and IBMUSER.TEST.

In the UNIX file system, ibmuser would have a user directory named /u/ibmuser. Under that directory there could be a subdirectory named /u/ibmuser/c, and /u/ibmuser/c/pgma would point to the file pgma (see Figure 4-6).

Of the various types of z/OS data sets, a partitioned data set (PDS) is most like a user directory in the file system. In a partitioned data set such as IBMUSER.C, you could have members (files) PGMA, PGMB, and so on. For example, you might have IBMUSER.C(PGMA) and IBMUSER.C(PGMB). Along the same lines, a subdirectory such as /u/ibmuser/c can hold many files, such as pgma, pgmb, and so on.

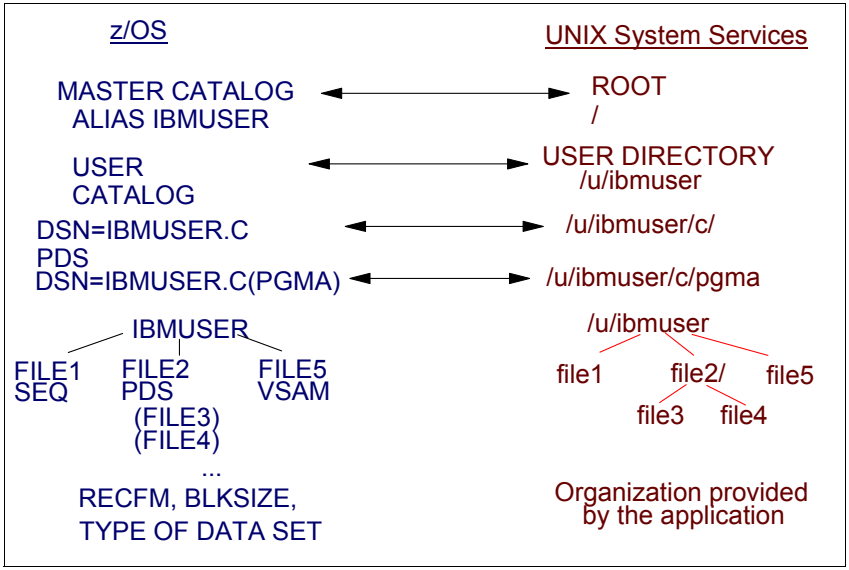


Figure 4-6 Comparison of z/OS data sets and file system files

All data written to the hierarchical file system can be read by all programs as soon as it is written. Data is written to a disk when a program issues an **`fsync()`**.

4.14 Summary

A data set is a collection of logically related data; it can be a source program, a library of programs, or a file of data records used by a processing program. Data set *records* are the basic unit of information used by a processing program.

Users must define the amount of space to be allocated for a data set (before it is used), or these allocations must be automated through the use of DFSMS. With DFSMS, the z/OS system programmer or storage administrator can define performance goals and data availability requirements, create model data definitions for typical data sets, and automate data backup. DFSMS can automatically assign, based on installation policy, those services and data definition attributes to data sets when they are created. Other storage management-related products can be used to determine data placement, manage data backup, control space usage, and provide data security.

Almost all z/OS data processing is record-oriented. Byte stream files are not present in traditional processing, although they are a standard part of z/OS UNIX. z/OS records(and physical blocks follow one of several well-defined formats. Most data sets have DCB attributes that include the record format (RECFM—F, FB, V, VB, U), the maximum logical record length (LRECL), and the maximum block size (BLKSIZE).

z/OS libraries are known as partitioned data sets (PDS or PDSE) and contain members. Source programs, system and application control parameters, JCL, and executable modules are almost always contained in libraries. Virtual storage access method (VSAM) is an access method that provides much more complex functions than other disk access methods. VSAM is primarily for applications and cannot be edited through ISPF.

z/OS data sets have names with a maximum of 44 uppercase characters, divided by periods into qualifiers with a maximum of 8 bytes per qualifier name. The high-level qualifier (HLQ) may be fixed by system security controls, but the rest of a data set name is assigned by the user. A number of conventions exist for these names.

An existing data set can be located when the data set name, volume and device type are known. These requirements can be shortened to knowing only the data set name if the data set is cataloged. The system catalog is a single logical function, although its data may be spread across the master catalog and many user catalogs. In practice, almost all disk data sets are cataloged. One side effect of this is that all (cataloged) data sets must have unique names.

A file in the UNIX file system can be either a course file or a binary file. In a course file each line of course is separated by a newline delimiter. A binary file consists of sequences of binary words (byte stream), and no record concept other than the structure defined by an application exists. An application reading the file is responsible for interpreting the format of the data. z/OS treats an entire UNIX file system hierarchy as a collection of data sets. Each data set is a mountable file system.

Key terms in this chapter		
block size	catalog	data set
high level qualifier or HLQ	library	logical record length or LRECL
member	PDS and PDSE	record format or RECFM
system managed storage or SMS	virtual storage access method or VSAM	VTOC

5

Batch processing and JES

Objective: As a mainframe professional, you will need to understand the ways in which the system processes your company's core applications, such as payroll. Such workloads are usually performed through *batch processing,* which involves executing one or more *batch jobs* in a sequential flow.

Further, you will need to understand how the *job entry subsystem* (JES) enables batch processing. JES helps z/OS receive jobs, schedule them for processing, and determine how job output is processed.

After completing this chapter, you will be able to:

- Give an overview of batch processing and how work is initiated and managed in the system.
- Explain how JES governs the flow of work through a z/OS system.

© Copyright IBM Corp. 2005, 2006. All rights reserved.

5.1 What is batch processing?

The term *batch job* originated in the days when punched cards contained the directions for a computer to follow when running one or more programs. Multiple card decks representing multiple jobs would often be stacked on top of one another in the hopper of a card reader, and be run in batches.

Today, jobs that can run without end user interaction, or can be scheduled to run as resources permit, are called batch jobs. A program that reads a large file and generates a report, for example, is considered to be a batch job.

There is no direct counterpart to z/OS batch processing in PC or UNIX systems. Batch processing is for those frequently used programs that can be executed with minimal human interaction. They are typically executed at a scheduled time or on an as-needed basis. Perhaps the closest comparison is with processes run by an AT or CRON command in UNIX, although the differences are significant.

You might also consider batch processing as being somewhat analogous to the printer queue as it typically managed on an Intel-based operating system. Users submit jobs to be printed, and the print jobs wait to be processed until each is selected by priority from a print spool.

To enable the processing of a batch job, z/OS professionals use job control language or JCL to tell z/OS which programs are to be executed and which files will be needed by the executing programs. JCL allows the user to describe certain attributes of a batch job to z/OS, such as:

- Who you are (the submitter of the batch job)
- What program to run
- Where input and output is located
- When a job is to run.

After the user submits the job to the system, there is normally no further human interaction with the job until it is complete.

The use of JCL is covered in detail in Chapter 6, “Using JCL and SDSF” on page 153, but for now understand that JCL is the means by a batch job requests resources and services from the operating system.

5.2 What is JES?

z/OS uses a *job entry subsystem* or *JES* to receive jobs into the operating system, to schedule them for processing by z/OS, and to control their output processing. JES is the component of the operating system that provides supplementary job management, data

management, and task management functions such as scheduling, control of job flow, and spooling.

z/OS manages work as tasks and subtasks. Both transactions and batch jobs are associated with an internal task queue that is managed on a priority basis. JES is a component of z/OS that works on the front end of program execution to prepare work to be executed. JES is also active on the back end of program execution to help clean up after work is performed. This includes managing the printing of output generated by active programs.

More specifically, JES manages the input and output job queues and data.

For example, JES handles the following aspects of batch processing for z/OS:

- Receive jobs into the operating system
- Schedule them for processing by z/OS
- Control their output processing

z/OS has two versions of job entry systems: JES2 and JES3. Of these, JES2 is the most common by far and is the JES used in examples in this course. JES2 and JES3 have many functions and features, but their most basic functions are as follows:

- Accept jobs submitted in various ways:
  - From ISPF through the SUBMIT command
  - Over a network
  - From a running program, which can submit other jobs through the JES internal reader
  - From a card reader (very rare!)
- Queue jobs waiting to be executed. Multiple queues can be defined for various purposes.
- Queue jobs for an *initiator,* which is a system program that requests the next job in the appropriate queue.
- Accept printed output from the job while it is running and queue the output.
- Optionally, send output to a printer, or save it on *spool* for PSF, InfoPrint, or another output manager to retrieve.

JES uses one or more disk data sets for *spooling*. JES combines multiple spool data sets (if present) into a single conceptual data set. The internal format is not in a standard access-method format and is not written or read directly by applications. Input jobs and printed output from many jobs are stored in the single (conceptual) spool data set. In a small z/OS system the spool data sets might be a few hundred cylinders of disk space; in a large installation they might be many complete volumes of disk space.

Spool simply means to queue and hold data in card-image format (for input) or printed format (for output). JES2 is described more fully in 5.5, “Job and output management with JES” on page 141.

The basic elements of batch processing are shown in Figure 5-1.

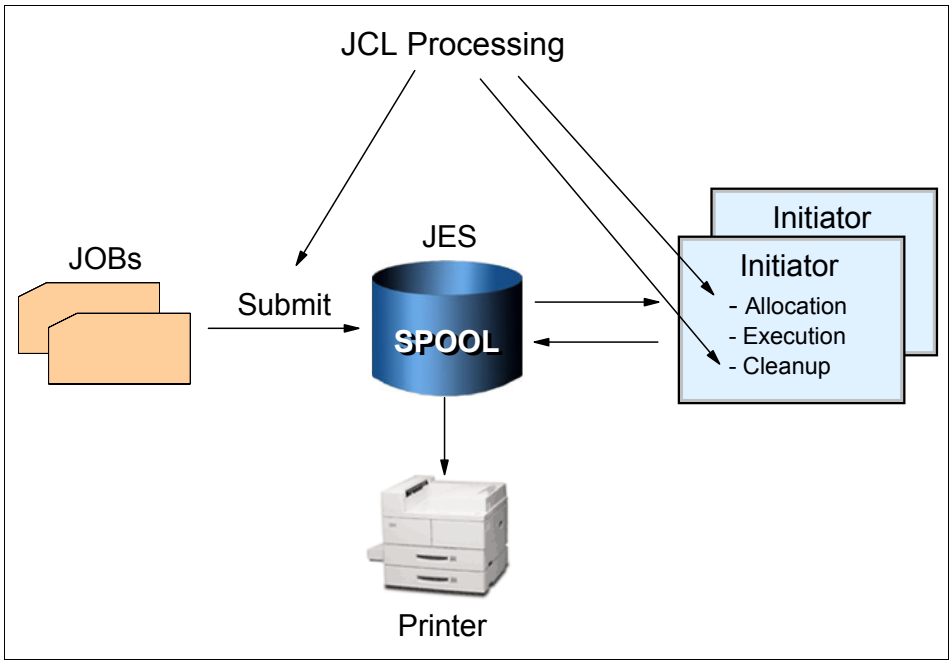


Figure 5-1 Basic batch flow

The *initiator* is an integral part of z/OS that reads, interprets, and executes the JCL. It is normally running in several address spaces (as *multiple initiators*). An initiator manages the running of batch jobs, one at a time, in the same address space. If ten initiators are active (in ten address spaces), then ten batch jobs can run at the same time. JES does some JCL processing, but the initiator does the key JCL work

The jobs in Figure 5-1 on page 138 represent JCL and perhaps data intermixed with the JCL. Source code input for a compiler is an example of data (the source statements) that might be intermixed with JCL. Another example is an accounting job that prepares the weekly payroll for different divisions of a firm (presumably, the payroll application program is the same for all divisions, but the input and master summary files may differ).

The diagram represents the jobs as punched cards (using the conventional symbol for punched cards) although real punched card input is very rare now. Typically, a job consists of card images (80-byte fixed-length records) in a member of a partitioned data set.

5.3 What an initiator does

To run multiple jobs asynchronously, the system must perform a number of functions:

- Select jobs from the input queues (JES does this).
- Ensure that multiple jobs (including TSO users and other interactive applications) do not conflict in data set usage.
- Ensure that single-user devices, such as tape drives, are allocated correctly.
- Find the executable programs requested for the job.
- Clean up after the job ends and then request the next job.

Most of this work is done by the initiator, based on JCL information for each job. The most complex function is to ensure there are no conflicts due to data set utilization. For example, if two jobs try to write in the same data set at the same time (or one reads while the other writes), there is a conflict.[1] This event would normally result in corrupted data. The primary purpose of JCL is to tell an initiator what is needed for the job.

The prevention of conflicting data set usage is critical to z/OS and is one of the defining characteristics of the operating system. When the JCL is properly constructed (which is the usual case) the prevention of conflicts is automatic. For example, if job A and job B must both write to a particular data set, the system (through the initiator) does not permit both jobs to run at the same time. Instead, whichever job starts first causes an initiator attempting to run the other job to wait until the first job completes.

5.4 Why z/OS uses symbolic file names

z/OS normally uses symbolic file names,[2] and this is another defining characteristic of this operating system. It applies a naming redirection between a data set-related name used in a program and the actual data set used during execution of that program. This is illustrated in Figure 5-2 on page 140.

[1] There are cases where such usage is correct and JCL can be constructed for these cases. In the case of simple batch jobs, such conflicts are normally unacceptable.

[2] This applies to normal traditional processing. Some languages, such as C, have defined interfaces that bypass this function.

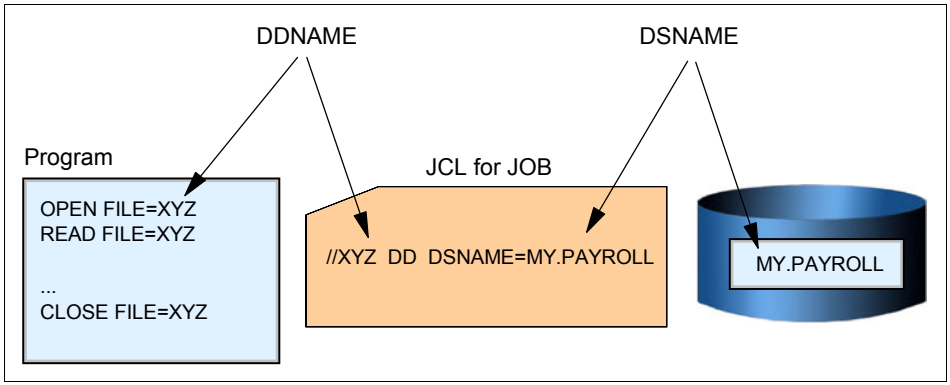


Figure 5-2 DDNAME and DSNAME

In this illustration we have a program, in some arbitrary language, that needs to open and read a data set.[3] When the program is written, the name XYZ is arbitrarily selected to reference the data set. The program can be compiled and stored as an executable. When someone wants to run the executable program, a JCL statement must be supplied that relates the name XYZ to an actual data set name. This JCL statement is a DD statement. The symbolic name used in the program is a DDNAME and the real name of the data set is a DSNAME.

The program can be used to process different input data sets simply by changing the DSNAME in the JCL. This becomes significant for large commercial applications that might use dozens of data sets in a single execution of the program. A payroll program for a large corporation is a good example. This can be an exceptionally complex application that might use hundreds of data sets. The same program might be used for different divisions in the corporation by running it with different JCL. Likewise, it can be tested against special test data sets by using a different set of JCL.

[3] The pseudo-program uses the term *file*, as is common in most computer languages.

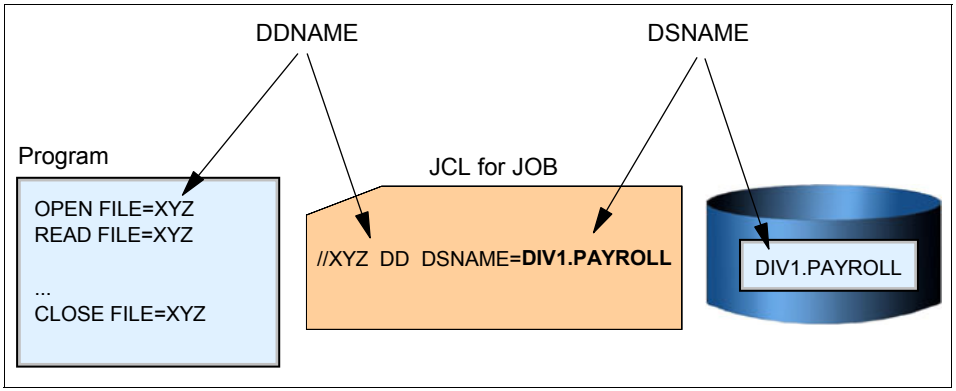


Figure 5-3 Symbolic file name - same program, but another data set

The firm could use the same company-wide payroll application program for different divisions and only change a single parameter in the JCL card (the `DD DSN=DIV1.PAYROLL`). The parameter value DIV1.PAYROLL would cause the program to access the data set for Division 1. This example demonstrates the power and flexibility afforded by JCL and symbolic file names.

This DDNAME--JCL--DSNAME processing applies to all traditional z/OS work although it might not always be apparent. For example, when ISPF is used to edit a data set, ISPF builds the internal equivalent of a DD statement and then opens the requested data set with the DD statement. The ISPF user does not see this processing—it takes place "transparently."[4]

5.5 Job and output management with JES

Let us further explore the JES concept through two scenarios.

5.5.1 *Scenario 1*

Imagine that you are a z/OS application programmer and you are developing a program for non-skilled users. Your program is supposed to read a couple of files, write to another couple of files, and produce a printed report. This program will run as a batch job on z/OS.

What sorts of functions are needed in the operating system to fulfill the requirements of your program? And, how will your program access those functions?

[4] Again, we are ignoring some of the operational characteristics of the z/OS UNIX interfaces of z/OS. The discussion here applies to traditional z/OS usage.

First, you need a sort of special language to inform the operating system about your needs. On z/OS, this is *Job Control Language* (JCL). The use of JCL is covered in detail in Chapter 6, "Using JCL and SDSF" on page 153, but for now assume that JCL provides the means for you to request resources and services from the operating system for a batch job.

Specifications and requests you might make for a batch job include the functions you need to compile and execute the program, and allocate storage for the program to use as it runs.

With JCL, you can specify the following:

- Who you are (important for security reasons).
- Which resources (programs, files, memory) and services are needed from the system to process your program. You might, for example, need to do the following:
  - Load the compiler code in memory.
  - Make accessible to the compiler your source code, that is, when the compiler asks for a read, your source statements are brought to the compiler memory.
  - Allocate some amount of memory to accommodate the compiler code, I/O buffers, and working areas.
  - Make accessible to the compiler an output disk data set to receive the object code, which is usually refered to as the *object deck* or simply *OBJ*.
  - Make accessible to the compiler a print file where it will tell you your eventual mistakes.
  - Conditionally, have z/OS load the newly created object deck into memory (but skip this step if the compilation failed).
  - Allocate some amount of memory for your program to use.
  - Make accessible to your program all the input and output files.
  - Make accessible to your program a printer for eventual messages.

In turn, you require the operating system to:

- Understand JCL (correcting eventual errors).
- Convert JCL to control blocks that describe the required resources.
- Allocate the required resources (programs, memory, files).
- Schedule the execution on a timely basis, for example, your program only runs if the compilation succeeds.
- Free the resources when the program is done.

The parts of z/OS that perform these tasks are JES and a batch initiator program.

Think of JES as the manager of the jobs waiting in a queue. It manages the priority of the set of jobs and their associated input data and output results. The initiator uses the statements on the JCL cards to specify the resources required of each individual job once it has been released (dispatched) by JES.

Your JCL as described is called a job—in this case formed by two sequential steps, the compilation and execution. The steps in a job are always executed sequentially. The job must be submitted to JES in order to be executed. In order to make your task easier, z/OS provides a set of procedures in a data set called SYS1.PROCLIB. A procedure is a set of JCL statements that are ready to be executed.

Example 5-1 on page 143 shows a JCL procedure that can compile, link-edit and execute a program. The first step identifies the COBOL compiler, as declared in `//COBOL  EXEC PGM=IGYCRCTL`. The statement `//SYSLIN  DD` describes the output of the compiler (the object deck).

The object deck is the input for the second step, which performs link-editing (through program IEWL). Link-editing is needed to resolve external references and *bring in* or *link* the previously developed common routines (a type of code re-use).

In the third step, the program is executed.

Example 5-1 Procedure to compile, linkedit, and execute programs

```
000010 //IGYWCLG PROC LNGPRFX='IGY.V3R2M0',SYSLBLK=3200,
000020 //             LIBPRFX='CEE',GOPGM=GO
000030 //*
000040 //*****************************************************************
000050 //*                                                               *
000060 //*  Enterprise COBOL for z/OS and OS/390                         *
000070 //*               Version 3 Release 2 Modification 0              *
000080 //*                                                               *
000090 //*  LICENSED MATERIALS - PROPERTY OF IBM.                        *
000100 //*                                                               *
000110 //*  5655-G53 5648-A25 (C) COPYRIGHT IBM CORP. 1991, 2002         *
000120 //*  ALL RIGHTS RESERVED                                          *
000130 //*                                                               *
000140 //*  US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                 *
000150 //*  DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                  *
000160 //*  ADP SCHEDULE CONTRACT WITH IBM CORP.                         *
000170 //*                                                               *
000180 //*****************************************************************
000190 //*
000300 //COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
000310 //STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,
000320 //             DISP=SHR
000330 //SYSPRINT DD  SYSOUT=*
000340 //SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
000350 //             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
000360 //             DCB=(BLKSIZE=&SYSLBLK)
000370 //SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
000440 //LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
000450 //SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,
```

```
000460 //             DISP=SHR
000470 //SYSPRINT DD  SYSOUT=*
000480 //SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
000490 //         DD  DDNAME=SYSIN
000500 //SYSLMOD  DD  DSNAME=&&GOSET(&GOPGM),SPACE=(TRK,(10,10,1)),
000510 //             UNIT=SYSDA,DISP=(MOD,PASS)
000520 //SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
000530 //GO     EXEC PGM=*.LKED.SYSLMOD,COND=((8,LT,COBOL),(4,LT,LKED)),
000540 //             REGION=2048K
000550 //STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,
000560 //             DISP=SHR
000570 //SYSPRINT DD  SYSOUT=*
000580 //CEEDUMP  DD  SYSOUT=*
000590 //SYSUDUMP DD  SYSOUT=*
```

To invoke a procedure, you need to write some simple JCL as shown in Example 5-2. In this JCL we are adding other DD statements, one of which is:

```
//COBOL.SYSIN DD *
```

It contains the COBOL source code.

Example 5-2 COBOL program

```
000001 //COBOL1 JOB (POK,999),MGELINSKI,MSGLEVEL=(1,1),MSGCLASS=X,
000002 //  CLASS=A,NOTIFY=&SYSUID
000003 /*JOBPARM SYSAFF=*
000004 // JCLLIB   ORDER=(IGY.SIGYPROC)
000005 //*
000006 //RUNIVP EXEC IGYWCLG,PARM.COBOL=RENT,REGION=1400K,
000007 //             PARM.LKED='LIST,XREF,LET,MAP'
000008 //COBOL.STEPLIB DD DSN=IGY.SIGYCOMP,
000009 //             DISP=SHR
000010 //COBOL.SYSIN DD *
000011        IDENTIFICATION DIVISION.
000012        PROGRAM-ID.    CALLIVP1.
000013        AUTHOR.        STUDENT PROGRAMMER.
000014        INSTALLATION.  MY UNIVERSITY
000015        DATE-WRITTEN.  JUL 27, 2004.
000016        DATE-COMPILED.
000017        /
000018        ENVIRONMENT DIVISION.
000019        CONFIGURATION SECTION.
000020        SOURCE-COMPUTER.  IBM-390.
000021        OBJECT-COMPUTER.  IBM-390.
000022
000023        PROCEDURE DIVISION.
000024            DISPLAY "***** HELLO WORLD *****" UPON CONSOLE.
000025            STOP RUN.
```

```
000026
000027 //GO.SYSOUT DD SYSOUT=*
000028 //
```

During the execution of a step, the program is controlled by z/OS, not by JES (Figure 5-4). Also, a spooling function is needed at this point in the process.

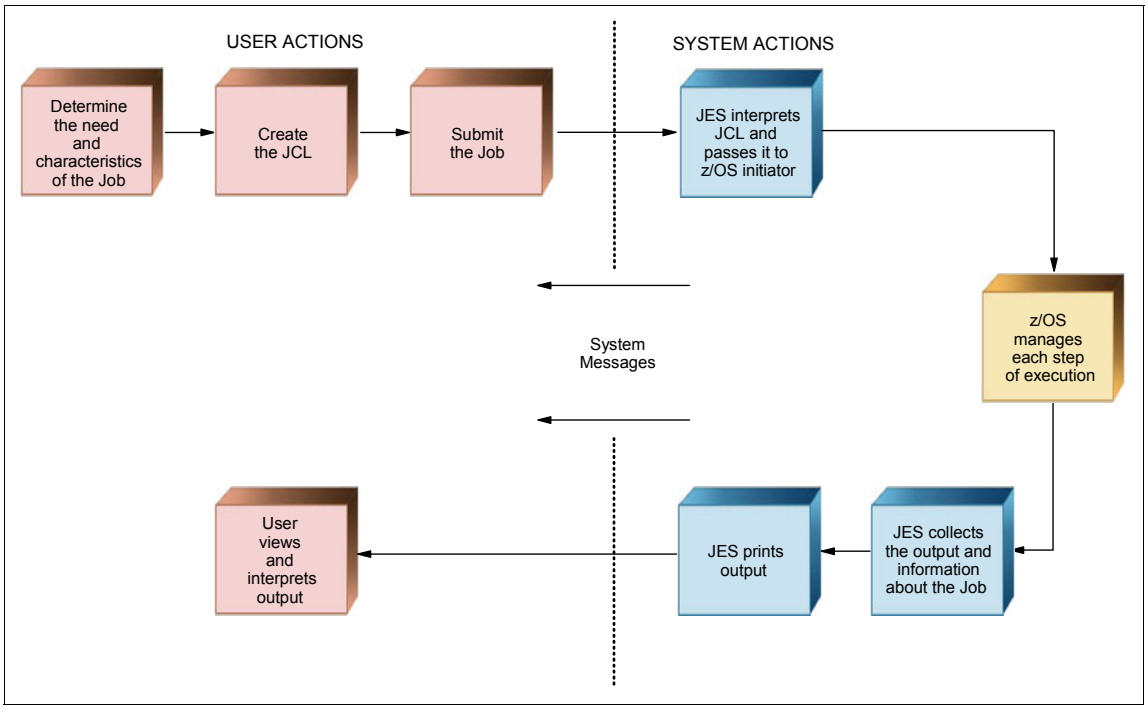


Figure 5-4 Related actions with JCL

Spooling is the means by which the system manipulates its work, including:

- Using storage on *direct access storage devices* (DASD) as buffer storage to reduce processing delays when transferring data between peripheral equipment and a program to be run.
- Reading and writing input and output streams on an intermediate device for later processing or output.
- Performing an operation such as printing while the computer is busy with other work.

There are two sorts of spooling: input and output. Both improve the performance of the program reading the input and writing the output.

To implement input spooling in JCL, you declare `// DD *`, which defines one file whose content records are in JCL between the `// DD *` statement and the `/*` statements. All the

logical records must have 80 characters. In this case this file is read and stored in a specific JES2 spool area (a huge JES file on disk) as shown in Figure 5-5.

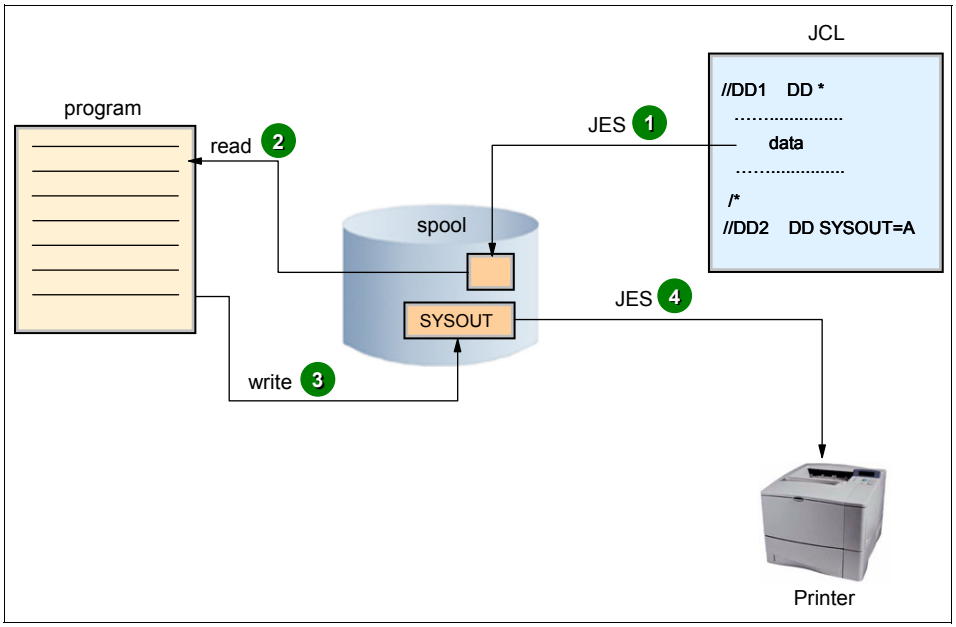


Figure 5-5 Spooling

Later, when the program is executed and asks to read this data, JES2 picks up the records in the spool and delivers them to the program (at disk speed).

To implement output spooling in JCL, you specify the keyword `SYSOUT` on the DD statement. `SYSOUT` defines an empty file in the spool, allocated with logical records of 132 characters in a printed format (EBCDIC/ASCII/UNICODE). This file is allocated by JES when interpreting a DD card with the `SYSOUT` keyword, and used later for the step program. Generally, after the end of the job, this file is printed by a JES function.

5.5.2 *Scenario 2*

Suppose now that you want to make a backup of one master file and then update the master file with records read-in from another file (the *update* file). If so, you need a job with two steps. In Step 1, your job reads the master file, and writes it to tape. In Step 2, another program (which can be written in COBOL) is executed to read a record from the update file and searches for its match in the master file. The program updates the existing record (if it finds a match) or adds a new record if needed.

In this scenario, what kind of functions are needed in the operating system to meet your requirements?

Build a job with two steps that specify the following:

- Who you are
- What resources are needed by the job, such as the following:
  - Load the backup program (that you already have compiled).
  - How much memory the system needs to allocate to accommodate the backup program, I/O buffers, and working areas.
  - Make accessible to the backup program an output tape data set to receive the backup, a copy, and the master file data set itself.
  - At program end indicate to the operating system that now your update program needs to be loaded into memory (however, this should not be done if the backup program failed).
  - Make accessible to the update program the update file and master file.
  - Make accessible to your program a printer for eventual messages.

Your JCL must have two steps, the first one indicating the resources for the backup program, and the second for the update program (Figure 5-6).

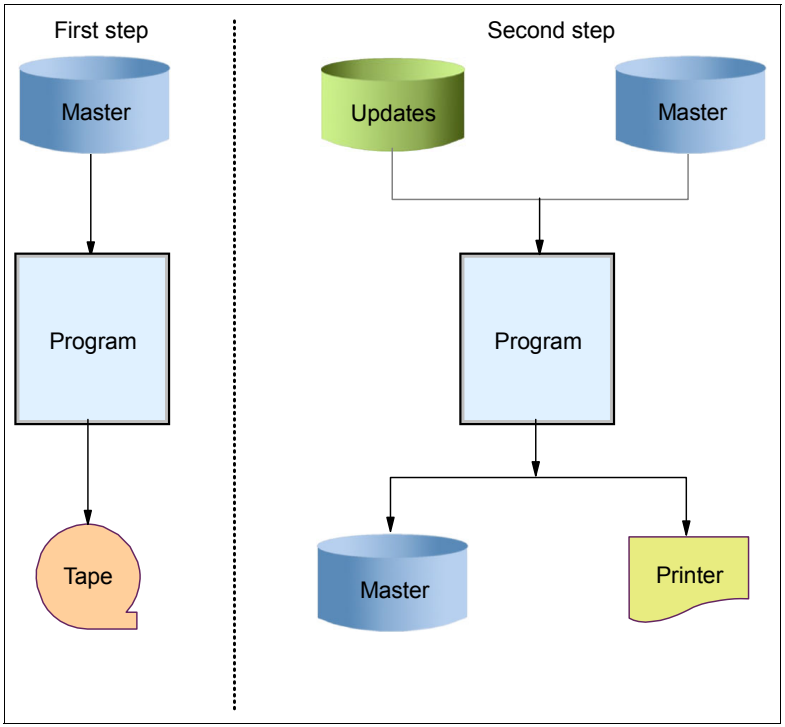


Figure 5-6 Scenario 2

Logically, the second step will not be executed if the first one fails for any reason. The second step will have a `// DD SYSOUT` statement to indicate the need for output spooling.

The jobs are only allowed to start when there are enough resources available. In this way, the system is made more efficient: JES manages jobs before and after running the program; the base control program manages jobs during processing.

Two types of job entry subsystems are offered with z/OS: JES2 and JES3. This section discusses JES2. For a brief comparison of JES2 and JES3, see 5.7, “JES2 compared to JES3” on page 151.

5.6 Job flow through the system

Let us look in more detail at how a job is processed through the combination of JES and a batch initiator program.

During the life of a job, JES2 and the base control program of z/OS control different phases of the overall processing. The job queues contain jobs that are waiting to run, currently running, waiting for their output to be produced, having their output produced, and waiting to be purged from the system.

Generally speaking, a job goes through the following phases:

- Input
- Conversion
- Processing
- Output
- Print/punch (hard copy)
- Purge

Figure 5-7 on page 149 shows the different phases of a job during batch processing.

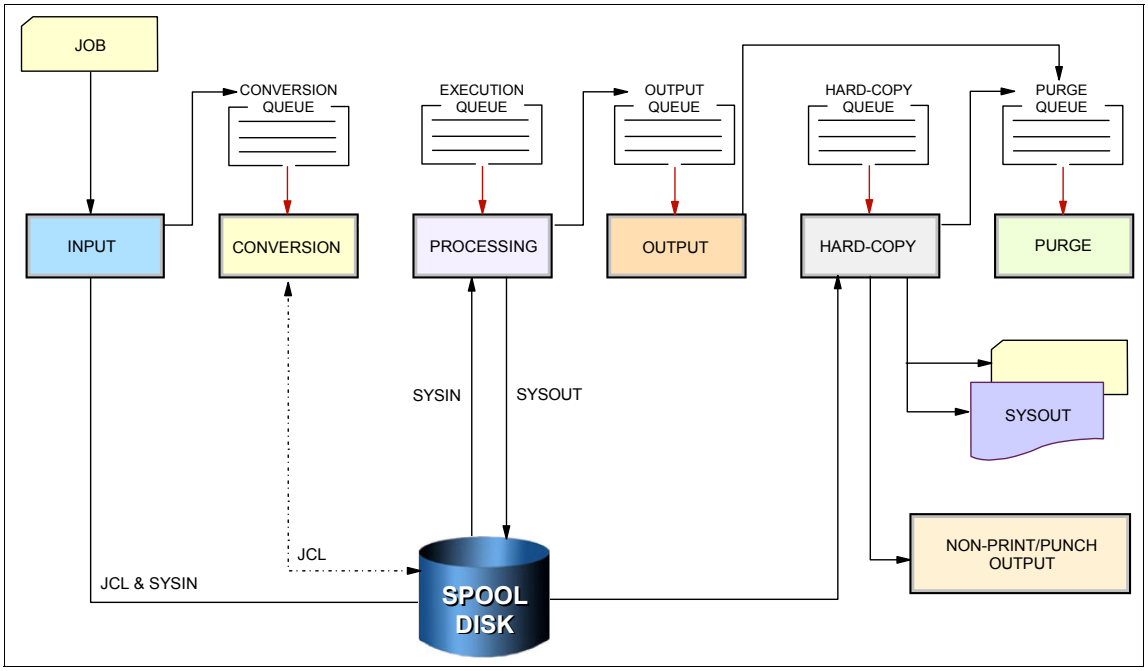


Figure 5-7 Job flow through the system

1. Input phase

   JES2 accepts jobs, in the form of an input stream, from input devices, from other programs through internal readers, and from other nodes in a job entry network.

   The internal reader is a program that other programs can use to submit jobs, control statements, and commands to JES2. Any job running in z/OS can use an internal reader to pass an input stream to JES2. JES2 can receive multiple jobs simultaneously through multiple internal readers.

   The system programmer defines internal readers to be used to process all batch jobs other than s*tarted Tasks* (STCs) and TSO requests.

   JES2 reads the input stream and assigns a job identifier to each JOB JCL statement. JES2 places the job's JCL, optional JES2 control statements, and SYSIN data onto DASD data sets called spool data sets. JES2 then selects jobs from the spool data sets for processing and subsequent running.

2. Conversion phase

   JES2 uses a converter program to analyze a job's JCL statements. The converter takes the job's JCL and merges it with JCL from a procedure library. The procedure library can be defined in the JCLLIB JCL statement, or system/user procedure libraries can be defined in the PROCxx DD statement of the JES2 startup procedure. Then, JES2

converts the composite JCL into converter/interpreter course that both JES2 and the initiator can recognize. Next, JES2 stores the converter/interpreter course on the spool data set. If JES2 detects any JCL errors, it issues messages, and the job is queued for output processing rather than execution. If there are no errors, JES2 queues the job for execution.

3. Processing phase

   In the processing phase, JES2 responds to requests for jobs from the initiators. JES2 selects jobs that are waiting to run from a job queue and sends them to initiators.

   An initiator is a system program belonging to z/OS, but controlled by JES or by the workload management (WLM) component of z/OS), which starts a job allocating the required resources to allow it to compete with other jobs that are already running (we discussed WLM in 2.5, “What is workload management?” on page 57).

   JES2 initiators are initiators that are started by the operator or by JES2 automatically when the system initializes. They are defined to JES2 through JES2 initialization statements. The installation associates each initiator with one or more job classes in order to obtain an efficient use of available system resources. initiators select jobs whose classes match the initiator-assigned class, obeying the priority of the queued jobs.

   WLM initiators are started by the system automatically based on performance goals, relative importance of the batch workload, and the capacity of the system to do more work. The initiators select jobs based on their service class and the order they were made available for execution. Jobs are routed to WLM initiators through a JOBCLASS JES2 initialization statement.

4. Output phase

   JES2 controls all SYSOUT processing. SYSOUT is system-produced output; that is, all output produced by, or for, a job. This output includes system messages that must be printed, as well as data sets requested by the user that must be printed or punched. After a job finishes, JES2 analyzes the characteristics of the job’s output in terms of its output class and device setup requirements; then JES2 groups data sets with similar characteristics. JES2 queues the output for print or punch processing.

5. Hardcopy phase

   JES2 selects output for processing from the output queues by output class, route code, priority, and other criteria. The output queue can have output that is to be processed locally or at a remote location. After processing all the output for a particular job, JES2 puts the job on the purge queue.

6. Purge phase

   When all processing for a job completes, JES2 releases the spool space assigned to the job, making the space available for allocation to subsequent jobs. JES2 then issues a message to the operator indicating that the job has been purged from the system.

5.7 JES2 compared to JES3

As previously mentioned, IBM provides two kinds of job entry subsystems: JES2 and JES3. In many cases, JES2 and JES3 perform similar functions. They read jobs into the system, convert them to internal machine-readable form, select them for processing, process their output, and purge them from the system.

In a mainframe installation that has only one processor, JES3 provides tape setup, dependent job control, and deadline scheduling for users of the system, while JES2 in the same system would require its users to manage these activities through other means.

In an installation with a multi-processor configuration, there are noticeable differences between the two, mainly in how JES2 exercises independent control over its job processing functions. That is, within the configuration, each JES2 processor controls its own job input, job scheduling, and job output processing.

In cases where multiple z/OS systems are clustered (a *sysplex*), it is possible to configure JES2 to share spool and checkpoint data sets with other JES2 systems in the same sysplex. This configuration is called *Multi-Access Spool* (MAS).

In contrast, JES3 exercises centralized control over its processing functions through a single global JES3 processor. This global processor provides all job selection, scheduling, and device allocation functions for all the other JES3 systems. The centralized control that JES3 exercises provides increased job scheduling control, deadline scheduling capabilities, and increased control by providing its own device allocation.

Most installations use JES2, as do the examples in this course.

5.8 Summary

Batch processing is the most fundamental function of z/OS. Many batch jobs are run in parallel and JCL is used to control the operation of each job. Correct use of JCL parameters (especially the DISP parameter in DD statements) allows parallel, asynchronous execution of jobs that may need access to the same data sets.

An *initiator* is a system program that processes JCL, sets up the necessary environment in an address space, and runs a batch job in the same address space. Multiple initiators (each in an address space) permit the parallel execution of batch jobs.

A goal of an operating system is to process work while making the best use of system resources. To achieve this goal, resource management is needed during key phases:

- Before job processing, reserve input and output resources for jobs
- During job processing, manage spooled SYSIN and SYSOUT data

- After job processing, free all resources used by the completed jobs, making the resources available to other jobs.

z/OS shares with JES the management of jobs and resources. JES receives jobs into the system, schedules them for processing by z/OS, and controls their output processing. JES is the manager of the jobs waiting in a queue. It manages the priority of the jobs and their associated input data and output results. The initiator uses the statements in the JCL records to specify the resources required of each individual job after it is released (dispatched) by JES.

During the life of a job, both JES and the z/OS base control program control different phases of the overall processing. Jobs are managed in queues: Jobs that are waiting to run (conversion queue), currently running (execution queue), waiting for their output to be produced (output queue), having their output produced (hard-copy queue), and waiting to be purged from the system (purge queue).

Key terms in this chapter		
batch job	execution	initiator
job entry subsystem or JES	output	procedure
purge	queue	spool
symbolic file name		

6

Using JCL and SDSF

Objective: As a technical professional in the world of mainframe computing, you will need to know JCL, the language that tells z/OS which resources will be needed to process a batch job or start a system task.

After completing this chapter, you will be able to:

- Explain how JCL works with the system, an overview of JCL coding techniques, and a few of the more important statements and keywords
- Create a simple job and submit it for execution
- Check the output of your job through SDSF

© Copyright IBM Corp. 2005, 2006. All rights reserved.

6.1 What is JCL?

For every batch job that you submit, you need to tell z/OS where to find the appropriate input, how to process that input (that is, what program or programs to run), and what to do with the resulting output. You use *job control language* or *JCL* to convey this information to z/OS through a set of statements known as job control statements.

The set of job control statements is quite large, which allows you to provide a great deal of information to z/OS. Most jobs, however, can be run using a very small subset of these control statements. Once you become familiar with the characteristics of the jobs you typically run, you may find that you need to know the details of only some of the control statements. This chapter discusses just a selected few.

Who uses JCL? While application programmers need some knowledge of JCL, the production control analyst responsible must be *highly* proficient with JCL, to create, monitor, correct and rerun the company's daily batch workload. System programmers also tend to be expert users of JCL.

Related Reading: For descriptions of the full set of JCL statements, see the IBM publications *z/OS MVS JCL User's Guide* and *z/OS MVS JCL Reference*, which are available at the z/OS Internet Library Web site:

http://www.ibm.com/servers/eserver/zseries/zos/bkserv/

6.2 Basic JCL statements

Within each job, the control statements are grouped into job steps. A job step consists of all the control statements needed to run one program. If a job needs to run more than one program, the job would contain a different job step for each of those programs.

There are three basic JCL statements:

JOB Provides a name for the job (jobname to the system) for control purposes, as well as providing user accounting information.

EXEC Provides the name of a program to execute. There can be multiple EXEC statements in a job. Each EXEC statement within the same job is a *job step*.

DD Provides the inputs and outputs to the execution program on the EXEC statement. This statement links a data set or other I/O device or function to a DDNAME coded in the program. DD statements are associated with a particular job step.

Figure 6-1 shows the basic JCL coding syntax.

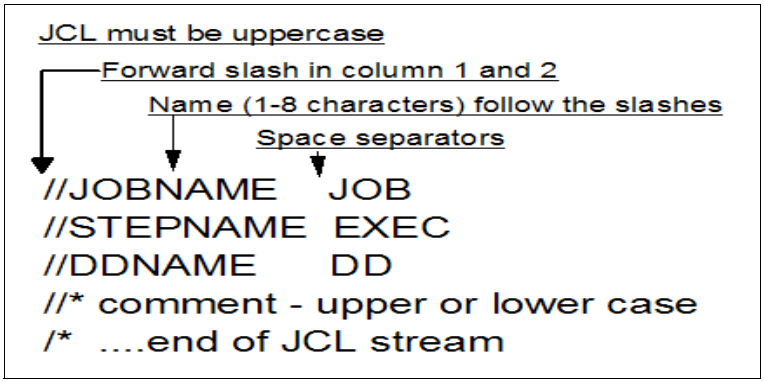


Figure 6-1 Basic JCL coding syntax

Example 6-1 shows some sample JCL.

Example 6-1 JCL example

```
//MYJOB     JOB 1
//MYSORT    EXEC PGM=SORT
//SORTIN    DD DISP=SHR,DSN=ZPROF.AREA.CODES
//SORTOUT   DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//SYSIN     DD *
  SORT FIELDS=(1,3,CH,A)
/*
```

In Chapter 3, “TSO/E, ISPF, and UNIX: Interactive facilities of z/OS” on page 75, we executed the same routine from the TSO READY prompt. Each JCL DD statement is equivalent to the TSO ALLOCATE command. Both are used to associate a z/OS data set with a *ddname,* which is recognized by the program as an input or output. The difference in method of execution is that TSO executes the sort in the foreground while JCL is used to execute the sort in the background.

When submitted for execution:

MYJOB Is a jobname the system associates with this workload.

MYSORT Is the stepname, which instructs the system to execute the SORT program.

SORTIN On the DD statement, this is the ddname. The SORTIN ddname is coded in the SORT program as a program input. The data set name (DSN) on this DD statement is *ZPROF*.AREA.CODES. The data set can be shared (DISP=SHR) with other system processes. The data content of *ZPROF*.AREA.CODES is the SORT program input.

SORTOUT This ddname is the SORT program output.

SYSOUT SYSOUT=* specifies to send system output messages to the Job Entry Subsystem (JES) print output area. It is possible to send the output to a data set.

SYSIN DD * is another input statement. It specifies that what follows is data or control statements. In this case, it is the sort instruction telling the SORT program which fields of the SORTIN data records are to be sorted.

We use *JCL statements* in this course; some z/OS users use the older term *JCL card,* even though the JCL resides in a disk library.

6.3 JOB, EXEC, and DD operands

The JOB, EXEC and DD statements have many operands to allow the user to specify instructions and information. Describing them all would fill an entire book, such as the IBM publication, *z/OS JCL Reference*.

This section provides only a brief description of a few of the more commonly used operands for the JOB, EXEC, and DD statements.

6.3.1 JOB operands

In JCL, the JOB statement is used to identify the job (the *job name*) and, optionally, specify the job's accounting information, which is sometimes used for charging system users. The information in the JOB statement applies to all job steps within the job.

For example, the following statement:

```
//MYJOB JOB 1
```

is a JOB statement with name `MYJOB` and an accounting field set to `1`.

Some common JOB statement operands include:

REGION= Requests specific memory resources to be allocated to the job. REGION prevents a program from using excessive resources if it breaks or misbehaves. (For 64-bit virtual storage, there is a MEMLIMIT keyword that works similar to REGION.)

NOTIFY= Sends notification of job completion to a particular user, such as the submitter of the job

USER= Specifies that the job is to assume the authority of the user ID specified

TIME=	Limits the amount of time a program can run. TIME prevents a program from running longer than necessary, for example, due to a programming error.
TYPRUN=	Delays or holds the job from running, to be released later
CLASS=	Directs a JCL statement to execute on a particular input queue
MSGCLASS=	Directs job output to a particular output queue
MSGLEVEL=	Controls the number of system messages to be received

Example:

```
//MYJOB JOB 1,NOTIFY=&SYSUID,REGION=6M
```

6.3.2 EXEC operands

In JCL, the EXEC statement is used to specify the name of a program to run. A job often runs multiple programs, and therefore contains multiple EXEC statements. Each EXEC statement within the same job is a job step with its own *step name*.

For example, the following EXEC statement

```
//MYSTEP EXEC
```

has a step name of `MYSTEP`.

The EXEC statement is followed by `PGM=`(executable program name) or a JCL PROC name. When you code the PGM parameter, z/OS locates the program by searching system libraries in the following order:

1. STEPLIB, if present
2. JOBLIB, if present
3. LPA data sets
4. LINKLIST data sets.

When a JCL PROC is present, the operands are the variable substitutions required by the JCL PROC. Common operands found on the `EXEC PGM=` statement are:

PARM=	Parameters known by and passed to the program.
COND=	Boolean logic for controlling execution of other EXEC steps in this job....IF, THEN, ELSE JCL statements exist that are superior to using COND; however, lots of old JCL may exist in production environments using this statement.
TIME=	Imposes a time limit.

Example:

```
//MYSTEP EXEC PGM=SORT
```

6.3.3 DD operands

In JCL, the *data definition* or DD statement is used to specify the data inputs and data outputs to the program that is specified on the EXEC statement. The DD statement links a data set or other I/O device or function to a *ddname* that is coded in the program. DD statements are associated with a particular job step.

For example, the following DD statement:

```
//MYDATA DD
```

has a ddname of `MYDATA`.

The DD statement is related to many aspects of defining or describing attributes of the program inputs or outputs. As a result, the DD statement has significantly more operands than the JOB or EXEC statements.

Some common DD statement operands are:

DSN= The name of the data set; this can include creation of temporary data sets or a reference back to the data set name.

DISP= Data set disposition, such as whether the data set needs to be created or already exists, and whether the data set can be shared by more than one job. DISP= is so important, in fact, that we devote the next section to it: 6.4, “Data set disposition, DISP operand” on page 159

SPACE= Amount of disk storage requested for a new data set.

SYSOUT= Defines a print location (and the output queue or data set).

VOL=SER= Volume name, disk name or tape name

UNIT= System disk, tape, special device type, or esoteric (local name).

DEST= Routes output to a remote destination.

DCB= Data set control block, numerous sub-operands, the most common being:

- LRECL — Logical record length. This is the number of bytes per record.
- RECFM — Record format. This can be fixed, blocked or variable.
- BLKSIZE — Block size. Store records in a block of this size, typically a multiple of LRECL. A value of 0 will let the system pick the best value.
- DSORG — Data set organization. This can be sequential, partitioned, and so on.

LABEL= Tape label expected (No Label or Standard Label followed by data set location). A tape can store multiple data sets; each data set on the tape is in a file position. The first data set on tape is file 1.

DUMMY Results in a null input or throwing away data written to this ddname.

* Input data or control statements follow—a method of passing data to a program from the JCL stream.

*,DLM= Everything following is data input (even //) until the two alphanumeric or special characters specified are encountered in column 1.

6.4 Data set disposition, DISP operand

All JCL parameters are important, but the DISP function is perhaps the most important for DD statements. Among its other uses, the DISP parameter advises the system about data set enqueuing needed for this job to prevent conflicting use of the data set by other jobs.

The complete parameter has these fields:

```
DISP=(status,normal end,abnormal end)
DISP=(status,normal end)
DISP=status
```

where status can be NEW, OLD, SHR, or MOD:

NEW Indicates that a new data set is to be created. This job has exclusive access to the data set while it is running. The data set must not already exist.

OLD Indicates that the data set already exists and that this job is to have exclusive access to it while it is running.

SHR Indicates that the data set already exists and that several concurrent jobs can share access while they are running. All the concurrent jobs must specify SHR.

MOD Indicates that the data set already exists and the current job must have exclusive access while it is running. If the current job opens the data set for output, the output will be appended to the current end of the data set.

The *normal end* parameter indicates what to do with the data set (the *disposition*) if the current job step ends normally. Likewise, the abnormal end parameter indicates what to do with the data set if the current job step abnormally ends.

The options are the same for both parameters:

DELETE Delete (and uncatalog) the data set at the end of the job step

KEEP Keep (but not catalog) the data set at the end of the job step

CATLG Keep and catalog the data set at the end of the job step

UNCATLG Keep the data set but uncatalog it at the end of the job step

PASS Allow a later job step to specify a final disposition.

The default disposition parameters (for normal and abnormal end) are to leave the data set as it was before the job step started. (We discussed catalogs in 4.12.2, “What is a catalog?” on page 127.)

You might wonder, what would happen if you specified DISP=NEW for a data set that already exists? Very little, actually! To guard against the inadvertent erasure of files, z/OS rejects a DISP=NEW request for an existing data set. You get a JCL error message instead of a new data set.

6.4.1 Creating new data sets

If the DISP parameter for a data set is NEW, you must provide more information, including:

- A data set name.
- The type of device for the data set.
- A volser if it is a disk or labeled tape.
- If a disk is used, the amount of space to be allocated for the primary extent must be specified.
- If it is a partitioned data set, the size of the directory must be specified.
- Optionally, DCB parameters can be specified. Alternately, the program that will write the data set can provide these parameters.

The DISP and data set names have already been described. Briefly, the other parameters are:

Volser The format for this in a DD statement is VOL=SER=xxxxxx, where xxxxxx is the volser. The VOL parameter can specify other details, which is the reason for the format.

Device type There are a number of ways to do this, but UNIT=xxxx is the most common. The xxxx can be an IBM device type (such as 3390), or a specific device address (such as 300), or an *esoteric name* defined by the installation (such as SYSDA). Typically, you code SYSDA to tell the system to choose any available disk volume from a pool of available devices.

Member name Remember that a library (or partitioned data set, PDS) member can be treated as a data set by many applications and utilities. The format DSNAME=*ZPROF*.LIB.CNTL(TEST) is used to reference a specific member. If the application or utility program is expecting a sequential data set, then either a sequential data set or a member of a library must be specified. A whole library name (without a specific member name) can be used only if the program/utility is expecting a library name.

Space:

The SPACE DD parameter identifies the space required for your data set. Before a data set can be created on disk, the system must know how much space the data set requires and how the space is to be measured.

There are a number of different formats and variations for this. Common examples are:

`SPACE=(TRK,10)`	10 tracks, no secondary extents
`SPACE=(TRK,(10,5))`	10 tracks primary, 5 tracks for each secondary extent
`SPACE=(CYL,5)`	Can use CYL (cylinders) instead of TRK
`SPACE=(TRK,(10,5,8))`	PDS with 8 directory blocks
`SPACE=(1000,(50000,10000))`	Primary 50000 records@1000 byte each

In the basic case, SPACE has two parameters. These are the unit of measure and the amount of space. The unit of measure can be tracks, cylinders, or the average block size.[1]

The amount of space typically has up to three subparameters:

- The first parameter is the primary extent size, expressed in terms of the unit of measure. The system will attempt to obtain a single extent (contiguous space) with this much space. If the system cannot obtain this space in not more than five extents (on a single volume) before the job starts, the job is failed.
- The second parameter, if used, is the size of each secondary extent. The system does not obtain this much space before the job starts and does not guarantee that this space is available. The system obtains secondary extents dynamically, while the job is executing. In the basic examples shown here the secondary extents are on the same volume as the primary extent.
- The third parameter, if it exists, indicates that a partitioned data set (library) is being created. (Another way to create a PDS is to specify DSORG=PO and optionally DSNTYPE=PDS.) The numeric value is the number of directory blocks (255 bytes each) that are assigned for the PDS directory. Another JCL parameter might be needed to create a PDSE instead of a PDS, if this decision is not automated through the use of DFSMS.

If the space parameter contains more than one subparameter, the whole space parameter must be inclosed in parentheses.

6.5 Continuation and concatenation

As a consequence of the limitations of the number of characters that could be contained in single 80-column punched cards used in earlier systems, early mainframe operating systems introduced the concepts of continuation and concatenation. z/OS retains these conventions in order to minimize the impact on previous applications and operations.

Continuation of JCL syntax usually involves specifying a comma at the end of the last complete operand. The next JCL line would include // followed by at least one space, then the additional operands. JCL operand syntax on a continuation line must begin on or before column sixteen and should not extend beyond column 72[2].

[1] The unit of measure can also be KB and MB but these are not as commonly used.
[2] Columns 73 through 80 are reserved for something called card sequence numbers.

```
//JOBCARD JOB 1,REGION=8M,NOTIFY=ZPROF
```

The JCL statement above would have the same result as the following continuation JCL:

```
//JOBCARD JOB 1,
//        REGION=8M,
//        NOTIFY=ZPROF
```

An important feature of DD statements is the fact that a single ddname can have multiple DD statements. This is called *concatenation*.

The following JCL indicates that data sets are concatenated:

```
//DATAIN DD DISP=OLD,DSN=MY.INPUT1
//       DD DISP=OLD,DSN=MY.INPUT2
//       DD DISP=SHR,DSN=YOUR.DATA
```

Concatenation applies only to input data sets. The data sets are automatically processed in sequence. In the example, when the application program reads to the end of `MY.INPUT1`, the system automatically opens `MY.INPUT2` and starts reading it. The application program is not aware that it is now reading a second data set. This continues until the last data in the concatenation is read; at that time the application receives an end-of -file indication.

6.6 Reserved DDNAMES

A programmer can select *almost* any name for a DD name, however, using a meaningful name (within the 8-character limit) is recommended.

There are a few reserved DD names that a programmer cannot use (all of these are optional DD statements):

```
//JOBLIB DD ...
//STEPLIB DD ...
//JOBCAT DD ...
//STEPCAT DD ...
//SYSABEND DD ...
//SYSUDUMP DD ...
//SYSMDUMP DD ...
//CEEDUMP DD ...
```

A JOBLIB DD statement, placed just after a JOB statement, specifies a library that should be searched first for the programs executed by this job. A STEPLIB DD statement, placed just after an EXEC statement, specifies a library that should be searched first for the program executed by the EXEC statement. A STEPLIB overrides a JOBLIB if both are used.

The JOBCAT and STEPCAT are used to specify private catalogs, but these are rarely used (the most recent z/OS releases no longer support private catalogs). Nevertheless, these DD names should be treated as reserved names.

The SYSABEND, SYSUDUMP, SYSMDUMP, and CEEDUMP DD statements are used for various types of memory dumps that are generated when a program abnormally ends (or *ABENDs*.)

6.7 JCL procedures (PROCs)

Some programs and tasks require a larger amount of JCL than a user can easily enter. JCL for these functions can be kept in procedure libraries. A procedure library member contains *part* of the JCL for a given task -- usually the fixed, unchanging part of JCL. The user of the procedure supplies the variable part of the JCL for a specific job. In other words, a JCL procedure is like a macro.

Such a procedure is sometimes known as a *cataloged procedure*. Do not confuse a cataloged procedure with a cataloged data set. A cataloged procedure is a named collection of JCL stored in a data set, and a cataloged data set is a data set whose name is recorded by the system (as we discussed in

Example 6-2 shows an example of a JCL PROC or JCL procedure.

Example 6-2 Example JCL procedure

```
//MYPROC    PROC
//MYSORT    EXEC PGM=SORT
//SORTIN    DD DISP=SHR,DSN=&SORTDSN
//SORTOUT   DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//          PEND
```

Much of this JCL should be recognizable now. New JCL functions presented here include:

- `PROC` and `PEND` statements are unique to procedures. They are used to identify the beginning and end of the JCL procedure.
- The `PROC` is preceded by a label or name; the name defined in Example 6-2 is `MYPROC`.
- JCL variable substitution is the reason JCL PROCs are used. `&SORTDSN` is the only variable in Example 6-2.

In Example 6-3 we include the inline procedure in Example 6-2 in our job stream.

Example 6-3 Sample inline procedure

```
//MYJOB     JOB 1
```

```
//*---------------------------------*
//MYPROC    PROC
//MYSORT    EXEC PGM=SORT
//SORTIN    DD DISP=SHR,DSN=&SORTDSN
//SORTOUT   DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//          PEND
//*---------------------------------*
//STEP1     EXEC MYPROC,SORTDSN=ZPROF.AREA.CODES
//SYSIN     DD *
  SORT FIELDS=(1,3,CH,A)
```

- When MYJOB is submitted, the JCL from Example 6-2 on page 163 is effectively substituted for EXEC MYPROC. The value for &SORTDSN must be provided.
- SORTDSN and its value were placed on a separate line, a continuation of the EXEC statement. Notice the comma after MYPROC.
- //SYSIN DD * followed by the SORT control statement will be appended to the substituted JCL.

6.7.1 JCL PROC statement override

When an entire JCL PROC statement needs to be replaced, then a JCL PROC override statement can be used. An override statement has the following form:

```
//stepname.ddname DD ...
```

Example 6-4 shows an example of overriding the SORTOUT DD statement in MYPROC. Here, SORTOUT is directed to a newly created sequential data set.

Example 6-4 Sample procedure with statement override

```
//MYJOB     JOB 1
//*---------------------------------*
//MYPROC    PROC
//MYSORT    EXEC PGM=SORT
//SORTIN    DD DISP=SHR,DSN=&SORTDSN
//SORTOUT   DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//          PEND
//*---------------------------------*
//STEP1     EXEC MYPROC,SORTDSN=ZPROF.AREA.CODES
//MYSORT.SORTOUT DD DSN=ZPROF.MYSORT.OUTPUT,
//          DISP=(NEW,CATLG),SPACE=(CYL,(1,1)),
//          UNIT=SYSDA,VOL=SER=SHARED,
//          DCB=(LRECL=20,BLKSIZE=0,RECFM=FB,DSORG=PS)
//SYSIN     DD *
  SORT FIELDS=(1,3,CH,A)
```

6.7.2 How is a job submitted for batch processing?

Using UNIX and AIX® as an analogy, a UNIX process can be processed in the background by appending an ampersand (&) to the end of a command or script. Pressing Enter then submits the work as a background process.

In z/OS terminology, work (a job) is submitted for batch processing. Batch processing is a rough equivalent to UNIX background processing. The job runs independently of the interactive session. The term batch is used because it is a large collection of jobs that can be queued, waiting their turn to be executed when the needed resources are available.

Commands to submit jobs might take any of the following forms:

ISPF editor command line	SUBmit and press Enter.
ISPF command shell	SUBmit ‘USER.JCL’ where the data set is sequential.
ISPF command line	TSO SUBmit 'USER.JCL' where the data set is sequential.
ISPF command line	TSO SUBmit ‘USER.JCL(MYJOB)’ where the data set is a library or partitioned data set containing member `MYJOB`.

6.8 Using SDSF

After submitting a job, it is common to use *System Display and Search Facility* (SDSF) a utility that lets you look at output data sets. You can use SDSF to review the job output for successful completion, or to find and fix JCL errors. Specifically, SDSF allows you to display printed output held in the JES spool area. Much of the printed output sent to JES by batch jobs (and other jobs) is never actually printed. Instead it is inspected using SDSF and deleted or used as needed.

SDSF provides a number of additional functions, including:

- Viewing the system log and searching for any literal string
- Entering system commands (in earlier forms of the operating system, only the operator could enter commands)
- Controlling job processing (hold, release, cancel, and purge jobs)
- Monitoring jobs while they are being processed
- Displaying job output before deciding to print it
- Controlling the order in which jobs are processed
- Controlling the order in which output is printed
- Controlling printers and initiators

From the ISPF Primary Option Menu, locate and select SDSF. The ISPF Primary Option Menu typically includes more selections than those listed on first panel, with instructions on how to display the additional selections.

Figure 6-2 on page 166 shows the SDSF primary option menu.

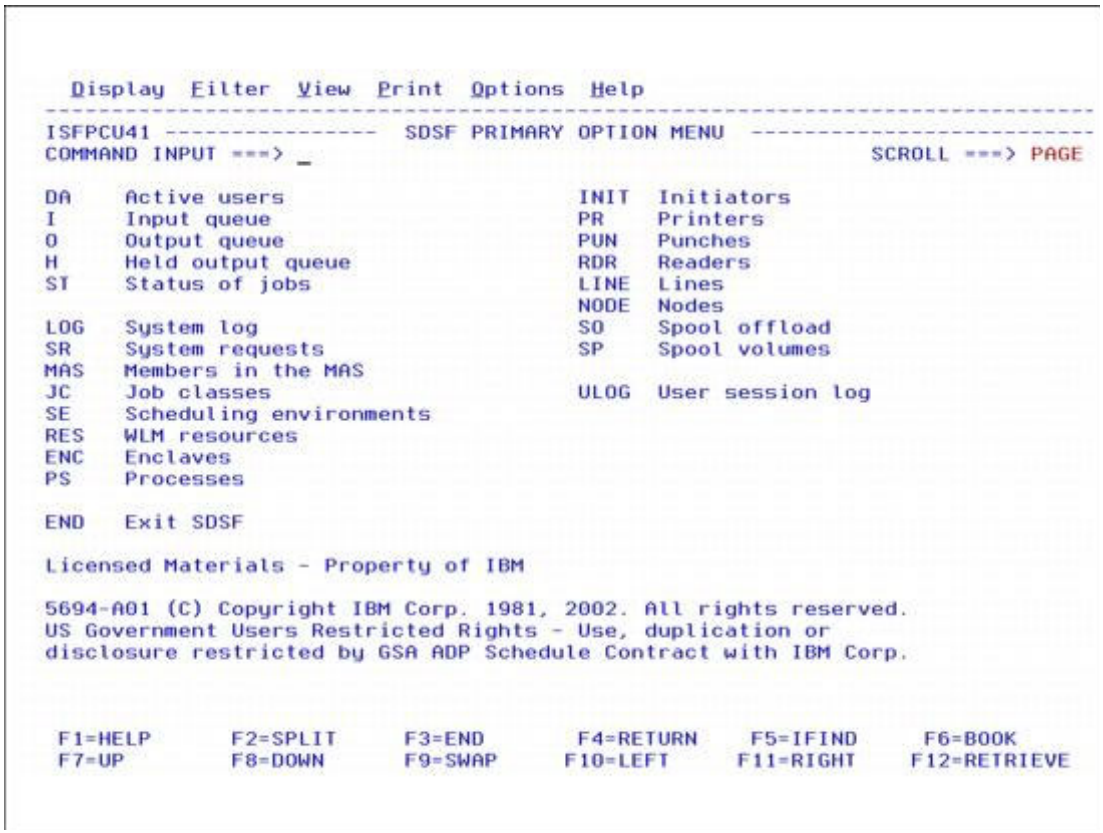


Figure 6-2 SDSF primary option menu

SDSF uses a hierarchy of online panels to guide users through its functions, as shown in Figure 6-3 on page 167.

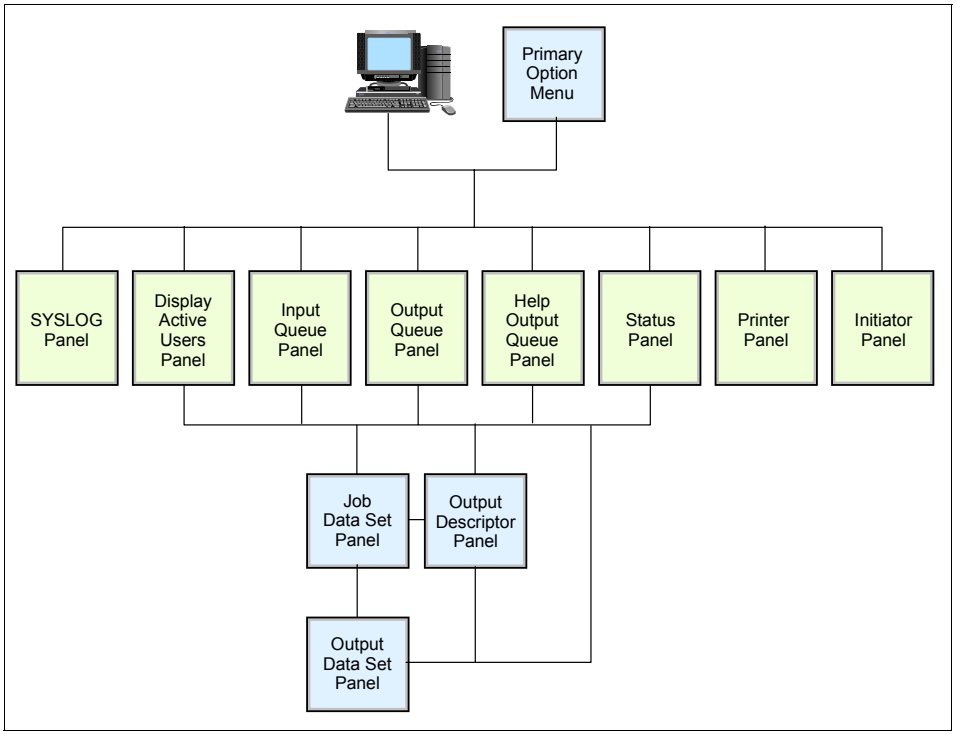


Figure 6-3 SDSF panel hierarchy

This hierarchy shows a subset of the SDSF panels. In actual use, the SDSF menu shows only the functions for which the user is authorized. A user who occasionally runs some jobs might only see a few options, while a system programmer will see more than 20 options.

View the JES2 output files

You can see the JES output data sets created during the execution of your batch job. They are saved on the JES spool data set. You can see the JES data sets in any JES queue:

I Input
DA Execution queue
O Output queue
H Held queue
ST Any queue

For output and held queues, you cannot see those JES data sets you requested to be automatically purged by setting a MSGCLASS out sysout CLASS that has been defined to not save output. Also, depending on the MSGCLASS you chose on the JOB card, the sysouts can be either in the Output queue or in the Held queue.

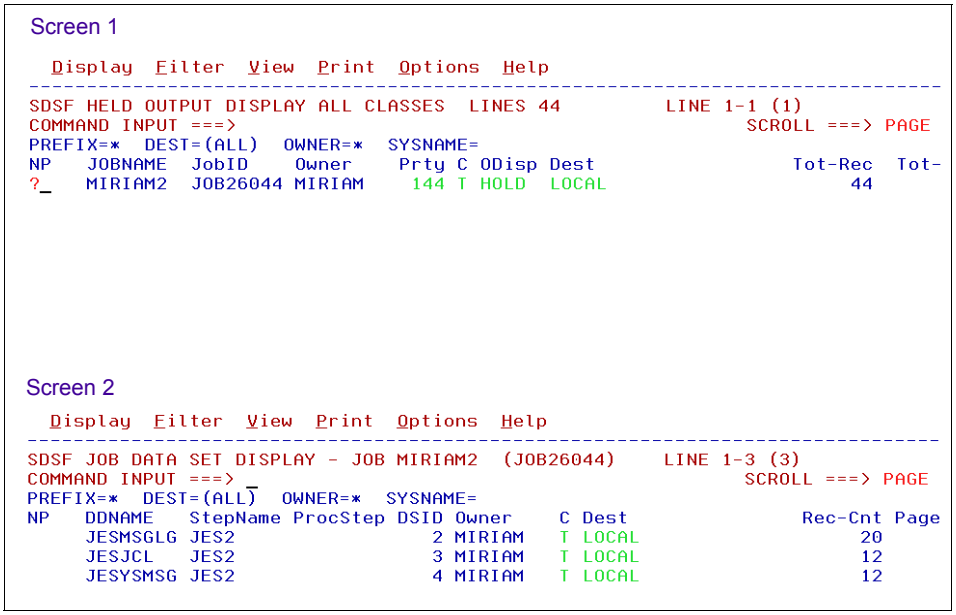


Figure 6-4 SDSF viewing the JES2 Output files

The first screen shown in Figure 6-4 displays a list of the jobs we submitted and whose output we directed to the HELD (Class T) queue, as identified in the `MSGCLASS=T` parameter on the job card. In our case only one job has been submitted and executed. Therefore, we only have one job on the Held queue.

Issuing a ? command in the NP column displays the output files generated by job 7359. The second screen, shown in Figure 6-4, displays three ddnames: the JES2 messages log file, the JES2 JCL file, and the JES2 system messages file. This option is useful when you are seeing jobs with many files directed to SYSOUT and you want to display one associated with a specific step. You issue an S in the NP column to select a file you want.

To see all files, instead of a `?`, type `S` in the NP column; the result is presented in Figure 6-5 on page 168.

JES2 job log

The following is an example of a JES2 job log.

Example 6-5 JES2 job log

```
J E S 2  J O B  L O G  --  S Y S T E M  S C 6 4  --  N O D E

13.19.24 JOB26044 ---- WEDNESDAY, 27 AUG 2003 ----
13.19.24 JOB26044  IRR010I  USERID MIRIAM   IS ASSIGNED TO THIS JOB.
```

```
13.19.24 JOB26044  ICH70001I MIRIAM   LAST ACCESS AT 13:18:53 ON WEDNESDAY,
AUGU
13.19.24 JOB26044  $HASP373 MIRIAM2  STARTED - INIT 1    - CLASS A - SYS SC64
13.19.24 JOB26044  IEF403I MIRIAM2 - STARTED - ASID=0027 - SC64
13.19.24 JOB26044  -                                          --TIMINGS
(MINS.)--
13.19.24 JOB26044  -JOBNAME  STEPNAME PROCSTEP    RC   EXCP    CPU    SRB
CLOCK
13.19.24 JOB26044  -MIRIAM2           STEP1       00      9    .00    .00
.00
13.19.24 JOB26044  IEF404I MIRIAM2 - ENDED - ASID=0027 - SC64
13.19.24 JOB26044  -MIRIAM2  ENDED.  NAME-MIRIAM              TOTAL CPU TIME=
13.19.24 JOB26044  $HASP395 MIRIAM2  ENDED
------ JES2 JOB STATISTICS ------
  27 AUG 2003 JOB EXECUTION DATE
           11 CARDS READ
           44 SYSOUT PRINT RECORDS
            0 SYSOUT PUNCH RECORDS
            3 SYSOUT SPOOL KBYTES
         0.00 MINUTES EXECUTION TIME
        1 //MIRIAM2 JOB 19,MIRIAM,NOTIFY=&SYSUID,MSGCLASS=T,
          // MSGLEVEL=(1,1),CLASS=A
          IEFC653I SUBSTITUTION JCL -
19,MIRIAM,NOTIFY=MIRIAM,MSGCLASS=T,MSGLEVE
        2 //STEP1 EXEC PGM=IEFBR14
          //*-------------------------------------------------*
//* THIS IS AN EXAMPLE OF A NEW DATA SET ALLOCATION
          //*-------------------------------------------------*
        3 //NEWDD DD   DSN=MIRIAM.IEFBR14.TEST1.NEWDD,
          //           DISP=(NEW,CATLG,DELETE),UNIT=SYSDA,
          //           SPACE=(CYL,(10,10,45)),LRECL=80,BLKSIZE=3120
        4 //SYSPRINT  DD  SYSOUT=T
          /*
ICH70001I MIRIAM   LAST ACCESS AT 13:18:53 ON WEDNESDAY, AUGUST 27, 2003
IEF236I ALLOC. FOR MIRIAM2 STEP1
IGD100I 390D ALLOCATED TO DDNAME NEWDD    DATACLAS (        )
IEF237I JES2 ALLOCATED TO SYSPRINT
IEF142I MIRIAM2 STEP1 - STEP WAS EXECUTED - COND CODE 0000
IEF285I   MIRIAM.IEFBR14.TEST1.NEWDD                  CATALOGED
IEF285I   VOL SER NOS= SBOX38.
IEF285I   MIRIAM.MIRIAM2.JOB26044.D0000101.?           SYSOUT
IEF373I STEP/STEP1   /START 2003239.1319
IEF374I STEP/STEP1   /STOP  2003239.1319 CPU    0MIN 00.00SEC SRB    0MIN
00.00S
IEF375I  JOB/MIRIAM2 /START 2003239.1319
IEF376I  JOB/MIRIAM2 /STOP  2003239.1319 CPU    0MIN 00.00SEC SRB    0MIN
00.00S
```

6.9 Utilities

z/OS includes a number of programs useful in batch processing called *utilities*. These programs provide many small, obvious, and useful functions.

Customer sites often add their own customer-written utility programs (although most users refrain from naming them *utilities*) and many of these are widely shared by the user community. Independent software vendors also provide many similar products (for a fee).

6.10 System libraries

z/OS has many standard system libraries. A brief description of several libraries is appropriate here. The traditional libraries include:

- SYS1.PROCLIB. This library contains JCL procedures distributed with z/OS. In practice, there are many other JCL procedure libraries (supplied with various licensed programs) concatenated with it.
- SYS1.PARMLIB. This library contains control parameters for z/OS and for some licensed programs. In practice, there may be other libraries concatenated with it.
- SYS1.LINKLIB. This library contains many of the basic execution modules of the system. In practice, it is one of a large number of execution libraries that are concatenated together.
- SYS1.LPALIB. This library contains system execution modules that are loaded into the link pack area when the system is initialized. There may be several other libraries concatenated with it. Programs stored here are available to other address spaces.
- SYS1.NUCLEUS. This library contains the basic supervisor (“kernel”) modules of z/OS.

These libraries are standard PDS data sets and are found on the system disk volumes.

6.11 Summary

For every job that you submit, you need to tell z/OS where to find the appropriate input, how to process that input (that is, what program or programs to run), and what to do with the resulting output. You use JCL to convey this information to z/OS through a set of statements known as job control statements. JCL's set of job control statements is quite large, enabling you to provide a great deal of information to z/OS. Most jobs, however, can be run using a very small subset of these control statements. Once you become familiar with the characteristics of the jobs you typically run, you may find that you need to know the details of only some of the control statements.

Within each job, the control statements are grouped into job steps. A job step consists of all the control statements needed to run one program. If a job needs to run more than one program, the job would contain a different job step for each of those programs.

Basic JCL contains three types of statements: JOB, EXEC, and DD. A job can contain several EXEC statements (steps) and each step might have several DD statements. JCL provides a wide range of parameters and controls; only a basic subset is described here.

A batch job uses artificial names (DD names) internally to access data sets. A JCL DD statement connects the DD name to a specific data set (DS name) for one execution of the program. A program can access different groups of data sets (in different jobs) by changing the JCL for each job.

The DISP parameters of DD statements help to prevent unwanted simultaneous access to data sets. This is very important for general system operation. The DISP parameter is not a security control, rather it helps manage the integrity of data sets. New data sets can be created through JCL by using the DISP=NEW parameter and specifying the desired amount of space and the desired volume.

System users are expected to write simple JCL, but normally use JCL procedures for more complex jobs. A cataloged procedure is written once and can then be used by many users. z/OS supplies many JCL procedures, and locally-written ones can be added easily. A user must understand how to override or extend statements in a JCL procedure in order to supply the parameters (usually DD statements) needed for a specific job.

Key terms in this chapter		
concatenation	DD statement	cataloged procedure
EXEC statement	JOB statement	jobname
SDSF	stepname	utility