# *z/OS JOB CONTROL LANGUAGE*

FIFTH EDITION

*Gary DeWard Brown*

WILEY COMPUTER PUBLISHING

*John Wiley & Sons, Inc.*

# *z/OS JOB CONTROL LANGUAGE*

FIFTH EDITION

*Gary DeWard Brown*

WILEY COMPUTER PUBLISHING

*John Wiley & Sons, Inc.*

Publisher: Robert Ipsen
Editor: Margaret Eldridge
Developmental Editor: Kathryn A. Malm
Associate Managing Editor: Penny Linskey
New Media Editor: Brian Snapp
Text Design & Composition: North Market Street Graphics



Published by John Wiley & Sons, Inc.,
Published simultaneously in Canada.



Library of Congress Cataloging-in-Publication Data:
ISBN 0471-236357

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

# CONTENTS

# PREFACE

The IBM mainframe continues to be alive and well, despite all the attention received by PCs. It has adapted to the changing environment, where issues such as security and reliability make it an ideal platform for the electronic economy. JCL (Job Control Language) is necessary to run batch jobs on IBM large mainframe computers, and batch jobs are an inherent part of programming on the mainframes. z/OS is the next generation of the OS/390 operating system. The information for it in this book has been extracted from close to 40 IBM manuals. There are so many manuals for z/OS that if programmers were each to have a personal copy of all the manuals they might need, they would risk crushing the continental plate under their weight. Since manuals are expensive, this book saves you and your company money.

This is the fifth edition of the JCL book. I have updated it to incorporate recent changes to JCL and the z/OS features.

My goal for this book continues to be to explain the operating system and provide readers with most of the information they need to use it. For this reason, chapters on such subjects as the linkage editor, the IBM utilities, the Sort/Merge utility, VSAM, TSO/E, ISPF, and REXX are included.

Please visit the book's Web site at www.wiley.com/compbooks/brown for solutions to the exercises and a summary of obsolete JCL features.

Gary DeWard Brown
Los Angeles, California
April 2002

# JOB CONTROL LANGUAGE PARAMETERS

# CHAPTER 1

# *INTRODUCTION*

## 1.1 THE SHOCK OF JCL

Your first use of JCL (*Job Control Language*) will be a shock. No doubt you have used personal computers costing $500 or $1,000 that had wonderfully human-engineered software, giving you an expectation of how easy it is to use a computer. Now, as you use a computer costing several million dollars, you may feel like a waif in a Dickens story standing in the shadow of a massive mainframe computer saying meekly, "Please, sir, may I run my job?" It will come as a shock that its software is not wonderfully human engineered.

The hardware and software design of large IBM mainframe computers date back to the days when Kennedy was president. JCL is a language that may be older than you are. It was designed at a time when user-friendliness was not even a gleam in the eye of its designers. This is easily demonstrated by taking the simple task of copying a file and contrasting how it is done through JCL with how it is done on the most popular personal computer system, Windows. To copy a file with Windows, you left-click twice on the MY COMPUTER icon, left-click on the C: drive icon, left-click twice on the folder containing the file, and right-click on the file to copy. On the resulting menu, you click on COPY and then left-click twice on the folder into which you want the file copied. Finally, you right-click inside the folder and select PASTE from the menu and you are done. It is wonderfully intuitive and simple. To do the same thing with JCL, you might write the following, in which *old-name* names the original file and *new-name* is the name you select for the copy.

```
//RT452216 JOB (45992,335),'SAMPLE JOB',CLASS=A
//STEP1  EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=A
//SYSUT1   DD DSN=old-name,DISP=SHR
//SYSUT2   DD DSN=new-name,DISP=(NEW,CATLG),
```

```
//            UNIT=SYSDA,RECFM=FB,LRECL=80,
//            SPACE=(0,(100,20))
//SYSIN    DD *
 REPRO INFILE(SYSUT1) OUTFILE(SYSUT2)
/*
```

This is far more complicated and not remotely intuitive. But what if you must copy 50 files? With JCL, you could use your text editor to quickly make 50 copies of the JCL statements, and overtype all the *old-names* and *new-names.* Then you simply submit the job to the computer, which is easier than going through the Windows clicking and pointing procedure 50 times. Now suppose you must copy the same 50 files on a daily basis. Windows would be a nightmare, but with JCL, a single command lets you resubmit the lines of JCL that copy the 50 files.

Then the inevitable happens. Your boss comes in and, with an accusing eye, asks if you inadvertently forgot to copy a file the previous day. With the pointing and clicking of Windows, you would have no answer. By contrast, JCL produces a listing—an audit trail—that you can wave in your boss's face to prove that the problem was not yours. The next day your boss informs you that the copy procedure must be done during the graveyard shift. With Windows, your nights will be sleepless. But with JCL, you can specify a job class that starts during the graveyard shift, leaving your nights free for more enjoyable activities.

What has just been described for JCL is the essence of *batch processing* (submitting one or many jobs or procedures in one fell swoop) and *production computing.* Interactive computing, such as that provided by Windows, Linux, Macintosh, and UNIX is wonderful for many things, but once those wonderful things are completed and the daily grind of running them begins, JCL comes into its own.

JCL is neither lovable nor simple. It was designed long ago and shows its age in ways such as having to code information in specific columns of a line. However, IBM has made changes to both JCL and the operating system that eliminate some of its worst aspects.

The large mainframe computer is an extremely conservative environment. The basic hardware architecture and operating system appear to the user much as they did over a third of a century ago. The benefit of this is that programs written back then can still run unchanged today, and knowledge gained back then is still valid. The greatest strength of large IBM mainframe computers, indeed, the strongest force in the computing universe, is *compatibility.* Billions of dollars are tied up in application software, and many companies would not accept an incompatible computer with modern design and software features even if it were free, because software is the dominating cost in computing, not hardware.

## 1.2 THE ROLE OF JCL

You do not use JCL to write computer programs. Instead, it consists of *control statements* that introduce a computer job to the operating system, provide accounting information, direct the operating system on what is to be done, request hardware devices, and execute the job. JCL tells the operating system everything it needs to know about a job's input/output (I/O) requirements. Sitting above JCL in many installations is a *job entry system* (*JES*) with a *job entry control language* (*JECL*). You code the JECL statements to specify on which network computer to run the job, when to run the job, and where to send the resulting output. IBM provides two job entry systems for z/OS: JES2 for decentralized control, and JES3 for highly centralized control of several computers. JCL and JES go hand in hand, and this book describes both.

## 1.3 THE DIFFICULTY OF JCL

JCL provides the means of communicating between an application program and the operating system and computer hardware. Measured by the number of moving parts, the z/OS operating system is one of humankind's most complex creations. The computer hardware is less complex, but complex nonetheless.

JCL is difficult because of the way it is used. A normal programming language, however difficult, soon becomes familiar through constant usage. In contrast, JCL has language features used so infrequently that many never become familiar. JCL is also difficult because of its design. It is not a procedural language like COBOL or C/C++ in which you build up complex applications step by step from simple statements. JCL consists of individual parameters, each of which has an effect that may take pages to describe. JCL makes few useful assumptions for you—you must tell it exactly what to do. For example, virtually every batch computer program prints some output. However, the system doesn't assume this. You must supply a JCL statement to print output.

The z/OS operating system demands an extraordinary amount of information, most of it supplied by JCL, to run a job. For example, to save a file on disk storage, the system wants to know the record type, the record length, the block size, the type of I/O unit, the volume serial number of the disk volume, and more. (The recent changes to simplify JCL have been in this area.)

JCL was designed at a time when people were relatively cheap and computers expensive. In 1965, one million computer instructions were roughly

equivalent in cost to 15 minutes of a programmer's time. Today the opposite is true. People are expensive and computers are cheap. A million instructions today buys less than a fraction of a millisecond of a programmer's time. Consequently, JCL, designed to be efficient in computer time, is operating today in an environment where the cost of a person's time is the dominant expense. This book attempts to save *you* time—not just save *computer* time. Some of the worse things done in computing have been for efficiency. Saving two bytes in the year field is an extreme example that made the new millennium stressful for many. The cost performance of computer chips is still doubling roughly every 18 months, and while this can't last forever, it would be foolhardy to predict when it will end. This has a profound effect on how much effort a company wants to invest in optimizing programs.

## 1.4 THE APPROACH TO JCL

The first several chapters of this book describe the individual language statements, tell how to write them, explain what they do, and suggest how to use them. With this as background, the book shifts to functional descriptions of the hardware devices, access methods, and other topics. Because you rarely need many JCL features, the book notes whether a feature is **ESSENTIAL, SOMETIMES USED,** or **RARELY USED.** You should pick and choose.

The goal of this book is to give you all the information you need to program on z/OS, except for the programming language you use. To accomplish this, the book goes far beyond JCL. The book introduces the concepts and facilities of the operating system from the application programmer's point of view. Several non-JCL operating system facilities are also described, including IBM utility programs, the Sort/Merge program, the linkage editor, VSAM (Virtual Sequential Access Method), TSO/E (Time Sharing Option), and REXX (REstructured eXtended eXecutor).

This book explains JCL and shows you how to use it, but it won't try to make you like it because JCL is not a likable language. Almost no one, even programmers with years of experience, can sit down and write JCL statements the way they could COBOL or C++ statements. Consequently, people who write JCL generally know what they want to do and then consult some existing JCL to use as a prototype to write the new JCL statements. The book gives many short examples to serve as prototypes with the assumption that short, concise examples are preferable to lengthy explanations.

The book gives special attention to the use of JCL with COBOL, FORTRAN, PL/I, C/C++, and Assembler Language. Where appropriate, the book describes their interfaces to JCL. The book is based on the z/OS ver-

sion of the operating system. Although IBM occasionally releases new versions of z/OS, the new releases seldom change existing JCL.

If you are just learning z/OS, you can use this book as an introduction to the operating system and JCL, skipping over the **RARELY USED,** being selective with the **SOMETIMES USED,** and concentrating on the **ESSENTIAL** features. The book presumes you have some familiarity with a higher-level language. If you are an experienced z/OS programmer, you can use this book to learn unfamiliar JCL features or to refresh your memory on seldom-used features. Finally, the book serves as a reference for all who program in z/OS.

For classroom usage, read Chapters 1 to 14 in sequence, working in the installation's particular requirements. Then select topics from the remaining chapters as needed. Exercises at the end of many chapters consist of short, simple problems that can be run on a computer. They are designed as much to teach you about your installation and the problems of actually running jobs as they are about JCL as a language.

JCL provides many abbreviations and alternate names for coding. For simplicity and to reduce errors, the book shows only one form, the shortest, except when the short form conveys no meaning (CANCEL rather than C) or the long form is very short (NO rather than N). The book displays JCL statements and parameters in capital letters, with italics denoting items for which you select values. Text comments about a JCL statement are set off in brackets beneath the statement:

```
//SYSUT1   DD DSN=new-name,DISP=NEW
```

[The *new-name* represents a name you choose for the file. Code uppercase characters exactly as shown.]

# CHAPTER 2

# *INTRODUCTION TO JCL AND z/OS*

**Essential**

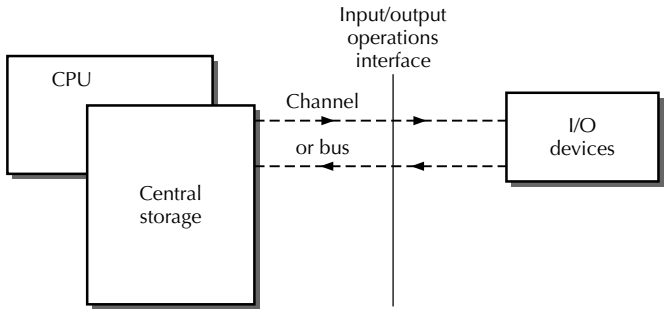
This chapter introduces z/OS concepts and vocabulary. You may be familiar with many of the concepts, but perhaps not in the z/OS context. While large mainframe computers have essentially the same hardware components as all other computers, including personal computers (PCs), the terminology differs. The formidable-sounding *direct-access storage device* (or *DASD*—pronounced *daz-dee*) on a mainframe becomes the more unassuming *hard disk* on a PC. Likewise, the impressive term *data set* on a mainframe turns out to be just a *file* on the PC.

The computer is essentially a device that reads some data, does some computing or processing, and writes some data. The main computer hardware that concerns you is illustrated in Figure 2.1 and consists of

- The CPU, or central processing unit. (Corresponding to a Pentium chip on a PC.)
- Central storage—formerly called real storage, computer memory, and core storage. (Corresponding to RAM, random access memory, on a PC.)
- Input/output (I/O) devices, often called peripherals, such as disks, tapes, and printers.

## 2.1 z/OS CONCEPTS AND VOCABULARY

The IBM Mainframe consists of a computer (the zSeries 900), an operating system (z/OS), and a vocabulary. People who work on z/OS form a separate culture, and you need to understand the concepts and speak the language to be accepted within the culture. Acronyms are the key to acceptance because everyone uses them, quickly forgetting what they represent.




CPU — The central processing unit executes instructions to perform computations, initiates input/output and directs the operation of the computer. It is the part of the computer that does the computing.

Central storage — The computer memory holds both the instructions and data during computations and is termed *central storage* or *real storage*. It corresponds to random access memory (RAM) on a PC.

I/O device — The input/output devices contain external data. A collection of data stored on an external device is called a *data set* or *file*. I/O devices consist of a recording device and access mechanism, a control unit (actually a small computer itself), and a channel or bus to transmit the data between the I/O device and the computer's storage. The most popular I/O devices are disk, tape, and printer.

**FIGURE 2.1** A typical computer.

### 2.1.1 The People

One difference between large IBM mainframes and smaller computers is the number of people required to support them. Computer operators run the mainframe, schedule jobs, mount tapes, and distribute output. One of the advantages of a large computer is that you don't have to worry about hardware problems or paper jams—the operators do this.

System programmers on large IBM mainframe computers install, maintain, and tune the operating system software. The software on large IBM mainframe computers is extremely complicated to install, and once installed, is even more difficult to maintain and tune for good performance. System programmers are highly skilled and valuable—and aware of this fact.

Installations usually have support staff to establish and enforce standards and procedures for using the computer. The support staff must also provide security, issue passwords and userids, and control access to the computer. Many installations also have a hot line or a help desk for answering questions. The first thing to do in any installation is to find out where to go for help.

### 2.1.2 The Operating System (z/OS)

The z/OS operating system introduces programs to the computer, initiates their execution, and schedules all the resources and services they require, such as printers, central storage, and disk storage space. The operating system is made up of a general library of programs that can be tailored to accommodate a variety of applications on a wide range of hardware configurations. The system programmers select the portions that an installation needs through a *system-generating process* (SYSGEN), add their own procedures, and update the procedures as the needs change.

The programs and routines that comprise the operating system are classified as either control or processing programs. *Control programs* perform the tasks of the operating system: reading, scheduling, initiating, allocating, executing, and terminating jobs in a continuous flow; supervising the dispatching and service requests of all work in the system; storing and retrieving all the data; controlling the user of virtual, central, and expanded storage; allocating the computer's resources; and finally, recovery from system and hardware failures.

*Processing programs* consist of *language translators* (such as the COBOL and C++ compilers), *service programs* (such as the linkage editor and sort programs), and *application programs* (such as the programs that you write.)

#### *Batch versus On-Line*

The original System/360 operating system was designed as a *batch system that could run on-line jobs.* This is in contrast to the PC, UNIX, and VM/ESA (the other popular system on large IBM mainframe computers), which were designed as *on-line systems that could run batch jobs.* With a *batch* system, you prepare a complete job and submit it to the computer. The computer's operating system schedules the job and executes it at its convenience, which may be hours later. You have no control over the job once you submit it. Often you submit a job and then hours later, when you get your output, find that a minor JCL error caused the entire job not to run. (With JCL, there is no such thing as a *minor* error.) *On-line* or *interactive* jobs are submitted from your computer terminal and you stay at the terminal while the job runs, usually interacting with the job to supply necessary information. You can quickly correct any errors.

Batch execution is great for long-running jobs, especially those run on a routine basis—*production* jobs. They have several advantages over interactive jobs:

You needn't wait at your terminal for your job to complete. You can go away and come back later.

You can run the job off shift, when computer rates are usually lower.

You can set up a complex job once and then keep resubmitting it rather than having to retype all the run commands each time.

The job may run more efficiently because it doesn't require the computer resources of an on-line job.

The computer can schedule the jobs at its convenience to make the most effective use of its hardware to maximize throughput.

Your program will produce an audit trail so you can reconstruct what happened during the run.

Of course, there are also disadvantages to running jobs in batch with JCL:

You don't get immediate turnaround, as a rule.

You must learn to use JCL.

If something goes wrong while a job is running, you don't get a chance to correct it and continue.

You must set the entire job up in advance, anticipating all the contingencies. You can't play it by ear.

You don't get to interact with the job and enter data and directions while it is running.

On-line is great for jobs requiring interaction while the job runs, jobs that change frequently, and jobs requiring quick turnaround. In practice, on-line is usually used for program development, data entry, inquiry, and quick response, whereas batch is used for production runs once the programs are developed.

Since many jobs are run concurrently on a mainframe computer, the resources of the computer must be shared so that each job eventually gets the resources it needs. This is done by multiprogramming and time-sharing.

### *Time-Sharing and Multiprogramming*

*Time-sharing* allows many people to use a computer simultaneously in such a way that each is unaware that others are using the computer. The usual case is an on-line system with several consoles using the main computer at the same time. Time-sharing attempts to maximize an individual's use of the computer, not the efficiency of the computer itself. Time-sharing is supported on z/OS by such systems as the TSO/E (Time-Sharing Option), CICS (Customer Information Control System), IMS (Information Management System/Virtual Storage), VTAM (Virtual Telecommunications Access Method), and UNIX and Linux.

*Multiprogramming* is just the opposite of time-sharing in concept. It attempts to maximize the efficiency of the computer by keeping all the major components busy, such as the CPU, I/O devices, and central storage. Most jobs running on a large general-purpose computer do not use all the I/O devices or storage. Moreover, not all of the CPU is used since time is spent waiting for some I/O action to complete. Rewinding a tape is an extreme case in point.

Since most jobs do not use all of the storage, all of the I/O devices, or all of the CPU, a multiprogramming system can keep several jobs inside the computer at the same time and switch back and forth between them. Several jobs are loaded into storage and the operating system gives control to one job. It then switches control to another whenever one becomes idle. By balancing I/O-bound jobs with the compute-bound jobs, several jobs can be completed in little more time than it would take to complete a single job.

## 2.2 z/OS HARDWARE ARCHITECTURE

Perhaps the best way to understand the architecture is to trace its individual components to see why they were added. z/OS has evolved over the past third of century, and while not elegant or simple, it is eminently practical.

### 2.2.1 The 1960s and the Quest for Multiprogramming: MVT

Before the introduction of System/360 (z/OS's ancestor) in 1964, IBM computers ran a single program at a time. Business and scientific applications used different computers with different instruction sets and operating systems. Different-sized computers also had their own instruction sets and operating systems. System/360 gave all IBM mainframes the same hardware architecture so that applications could run independent of a particular computer model. The System/360 ancestor of today's operating system, MVT (Multiprogramming with Variable Tasks), could run a maximum of 15 jobs concurrently in the same central storage. The programs were placed in variable-sized areas called *regions* as shown in Figure 2.2. The computer used 24 bits for addressing, and so could address 16MB of memory or central storage. (The amount of storage that a computer can address is termed its *address space,* which is determined by the number of bits used to form an address. The address space size is calculated by the number of bits raised to the power of 2, so that 15 bits gives a $2^{15} = 32{,}768$ address space.) However, memory was so expensive that a 512K machine was considered large.

Multiprogramming added immensely to the complexity of the operating system. The system had to protect each job in storage from other jobs. This was implemented through a hardware feature called *storage protection* in

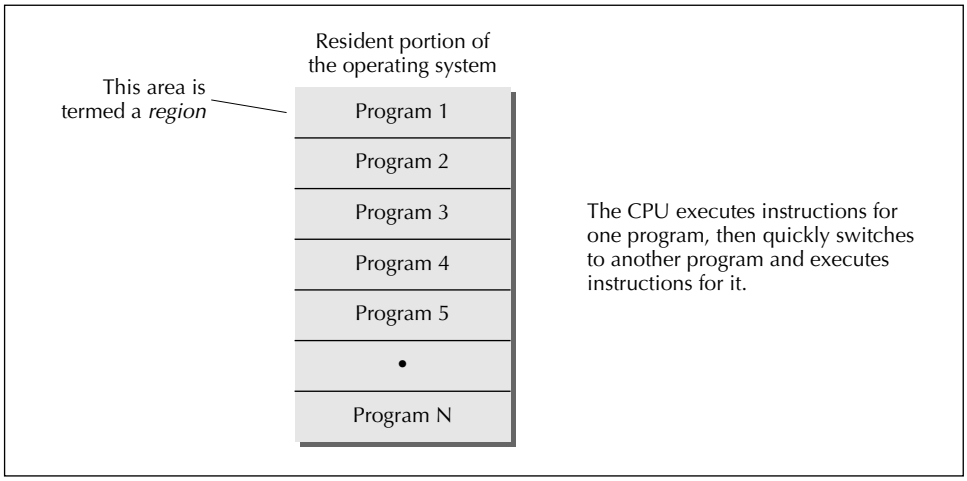



**FIGURE 2.2** Central storage in MVT.

which each region could change data only in its own region, protecting other regions and the operating system. The system also had to dole out the computer's resources so that the mixture of jobs in storage did not contend for nonshareable resources, such as tape drives. (PCs went through this barrier when Windows replaced MS-DOS.) The system had to hold back a job if it needed an unavailable resource and schedule another job whose resource requirements could be met.

The fact that the large mainframe computer was shared was a major reason it became so complex and difficult to use. Files and applications were potentially accessed by hundreds of users, making security a critical issue. If the mainframe crashed, hundreds of people could be affected, and in many cases, prolonged downtime could destroy a company. The mainframe, by necessity, became a serious environment.

The system, since it scheduled jobs based on their resource requirements, had to be told what resources each job needed. This was (and still is) done with JCL statements. For example, if a job needed a tape, a JCL statement had to describe the tape unit needed. The system would not schedule the job until such a tape unit became available. This prevented the job from sitting idle in central storage waiting for an available tape unit.

Figure 2.3 illustrates the original MVT multiprogramming system. Each job occupied a contiguous region in central storage, and the jobs remained there until they were complete. Users specified their region size with JCL statements. Some regions, such as the readers or writers, never completed and were always resident. The system decided which region to run and for how long and was made as crashproof as possible so that, although a particular job might fail, the system was not disturbed. This continues today,

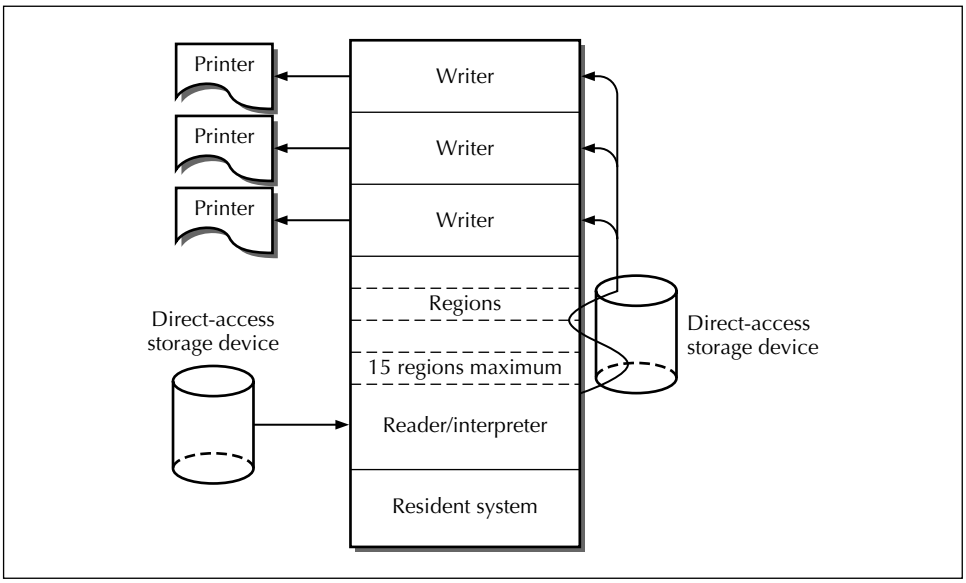



**FIGURE 2.3** MVT multiprogramming system.

and much of the expense of the mainframe is in hardware and software designed to isolate problems and speed recovery. One of the advantages of z/OS as an old system is that most of the bugs have been shaken out and it is extremely reliable. (But not bugfree. Nothing as complex as an operating system is without bugs.)

The first region contained the *nucleus* or resident portion of the operating system (those portions of the system not kept on a direct-access storage device). The *reader/interpreter* (or *reader*) read in jobs and queued them on a direct-access volume. The term *volume* referred to a specific storage unit such as a direct-access storage volume or a tape cartridge. A direct-access storage volume was usually a disk pack—a large version of the hard disk on a PC. The *writers* wrote output from the queue on the direct-access volume onto the proper output device. Queuing the input and output on direct-access volumes was called *spooling,* which you might equate to the take-up reel of a movie projector. Spooling was an acronym for *simultaneous peripheral operation on-line;* one of those rare cases where an acronym conveys more meaning than that for which it stands. The sequence of lines read by the reader was called the *input stream,* and the sequence of output written by the writers was called the *output stream.*

The *unit record* devices (printers, and, back then but never today, card readers and cardpunches) were normally assigned to the readers and writers. All other I/O units except tapes could be concurrently used by any of the regions. The system could assign a tape unit to only one region at a time.

This was one reason for using direct-access storage devices—the jobs were not kept waiting for a tape unit to become free.

To capsulate the operating system: The reader/interpreter read jobs from the input stream and queued them on a direct-access volume. When a job came to the top of its queue, a system program called the *initiator/terminator* loaded it into a region and executed it. If the job read data from the input stream, the system fetched the lines from a direct-access volume where they had been stored. If the job printed output, this output was again queued on a direct-access volume. A job never printed directly—it was all done by queuing on a direct-access volume. The job was not aware of this because the system did it automatically. After the output was queued on a direct-access volume, the writers wrote it out to the appropriate output device. Figure 2.4 illustrates how spooling worked. All of this holds true today, except that users often examine their output on-line through a terminal while the output is in the output queue, and then decide whether to discard or print it. Although the running of a single region has been described here, several regions could be kept running concurrently in a similar manner.

Multiprogramming, which divided the computer's central storage into regions, presented several problems. A job was limited to the size of the largest region, and inevitably, some applications needed more central storage to run than the computer had. Central storage was often wasted when it became fragmented because jobs could only be run in contiguous storage.

Two applications aggravated these problems, time-sharing and teleprocessing. Time-sharing gave many people access to a computer at the same time with fast response required for each. *Teleprocessing* connected remote devices to the main computer through communication lines. The applica-

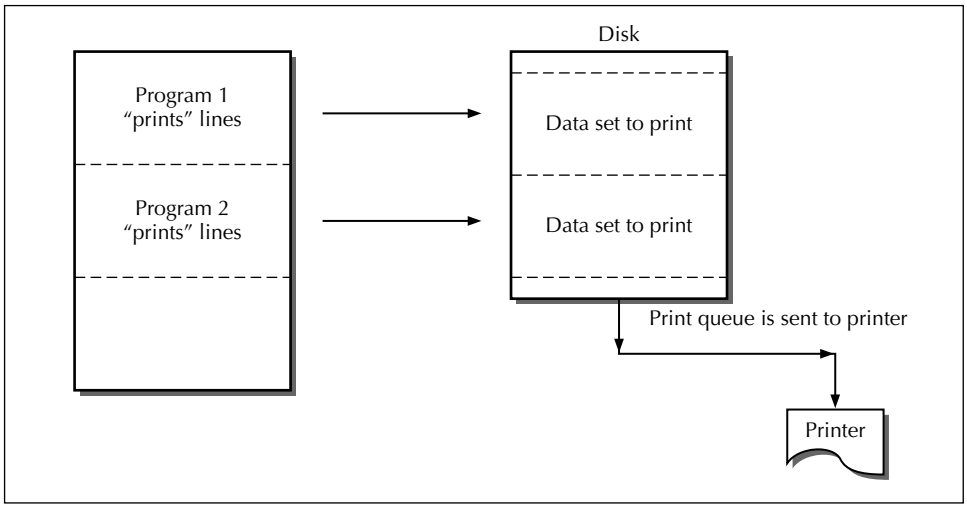



**FIGURE 2.4** Spooling.

tions were often real time as typified by an airline reservations system in which an agent at a remote location reserved a seat on an airliner by communicating with the central computer. Time-sharing and teleprocessing applications were both characterized by long periods of inactivity and then brief spurts of frenzied activity requiring fast response.

Because the MVT was a batch system, on-line systems had to be developed that would run under its control in a region. Even today, such on-line systems as TSO/E, CICS, and VTAM are executed as batch jobs through JCL statements, and once running in their region, they begin supporting the many on-line users as illustrated in Figure 2.5.

### 2.2.2 The 1970s and the Quest for Central Storage: OS/VS2 and MVS/370

By the 1970s, the amount of central storage became a critical bottleneck. The operating system provided for 15 concurrent users, but there wasn't enough central storage for more than a few to reside in the computer at the same time. Central storage was still extremely expensive.

In 1972, IBM introduced the 370 family of computers with new circuitry that included a major architectural component called virtual storage. With *virtual storage,* the storage addresses of an application program were made independent of the addresses of the computer's central storage. The

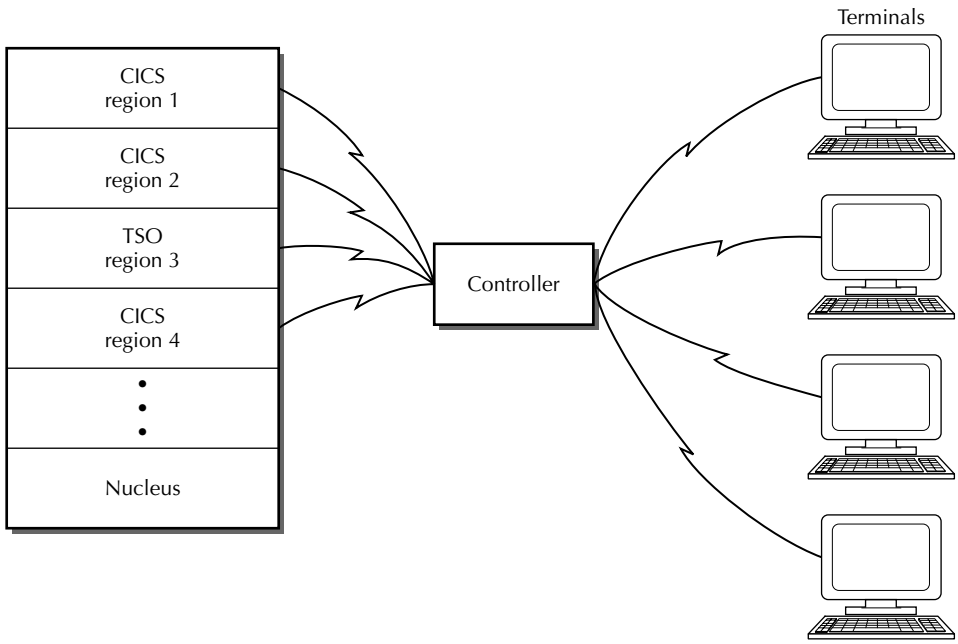



**FIGURE 2.5** On-line system supporting many users.

program, still limited to 16MB, was stored on disk (virtual storage), and the system brought small portions of the program into central storage on demand. A hardware feature was added to translate the user's virtual storage addresses to the computer's central storage addresses during execution. The MVT operating system was renamed OS/VS2 (Operating System/Virtual Storage 2), and it could operate as if it had the maximum of 16MB of central storage when in fact the actual amount might be (and generally was) considerably smaller. Each region residing in virtual storage could also be a maximum of 16MB, including space occupied by the operating system, but the total for all regions was limited to 16MB as illustrated in Figure 2.6.

One of the limitations in any computer is the amount of storage it can address—its *address space.* The IBM mainframe was a 32-bit computer. However, it used only 24 bits to contain addresses, and the maximum size of the address space was $2^{24}$ or roughly 16 million (MB) bytes.

Sixteen million bytes of address space seemed like a lot of memory back when the typical computer had only 512K of memory. A thousand bytes of memory—actually 1,024 bytes—is called a *kilobyte* and abbreviated as *K.* The 1,024 is used because it is a power of two. A million bytes is termed a

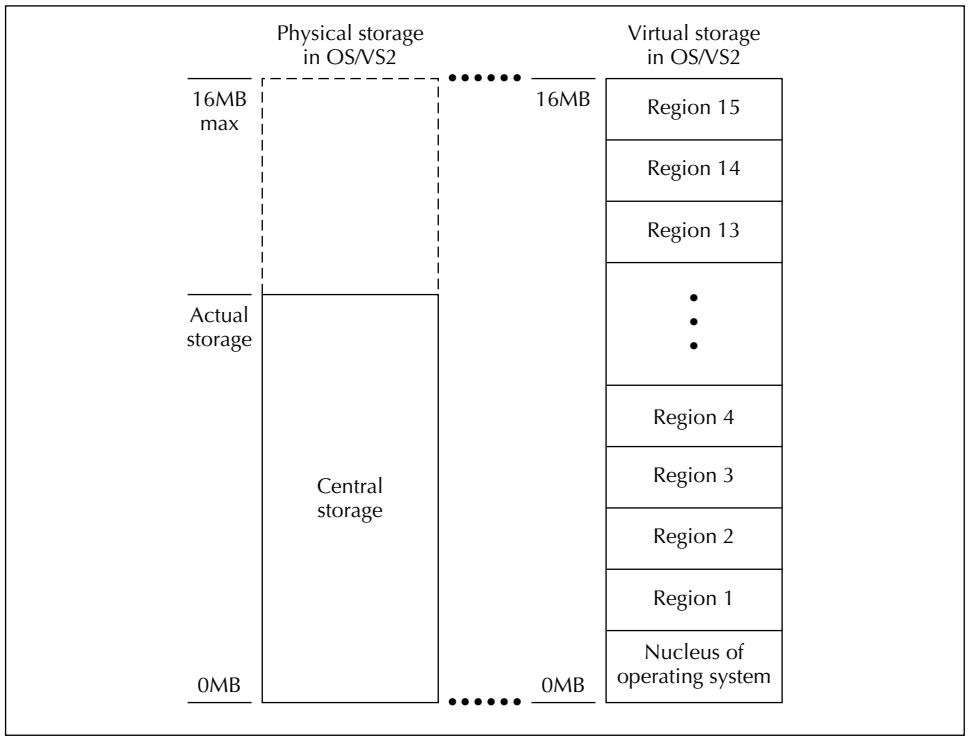



**FIGURE 2.6** OS/VS2 with virtual storage.

*megabyte* or just a *meg* and abbreviated as *MB.* A billion bytes are termed a *gigabyte* or just a *gig,* abbreviated as *GB.* But application programs kept growing. Also, the operating system required a portion of the application's program address space, and less address space became available as the operating system grew to where little more than 8MB was available to an application program. Programs also grew larger. Finally, applications such as CICS, IMS, VTAM, and DB2 required huge amounts of address space for buffers. Even worse, some, such as CICS, were designed to run all of their on-line applications in the same address space.

Virtual storage relieved much of this problem. When a program began execution, the system first *loaded* it onto a direct-access volume—the *virtual storage*—rather than directly into the computer's memory, or *central storage.* The system divided the application program into small parts called *pages.* Central storage was likewise divided into parts termed *page frames* to contain the pages on the direct-access volume. A *paging supervisor* in the operating system loaded each page from the direct-access volume into central storage on demand as illustrated in Figure 2.7. Thus, large portions of an application program might reside on a direct-access volume, the *external paging storage,* rather than in central storage at any given time during execution. The implications of this were twofold: the size of the program could exceed that of central storage, and inactive portions of programs wasted little central storage.

When the hardware detected a reference to a virtual storage address of a page not in central storage, it notified the paging supervisor, an event

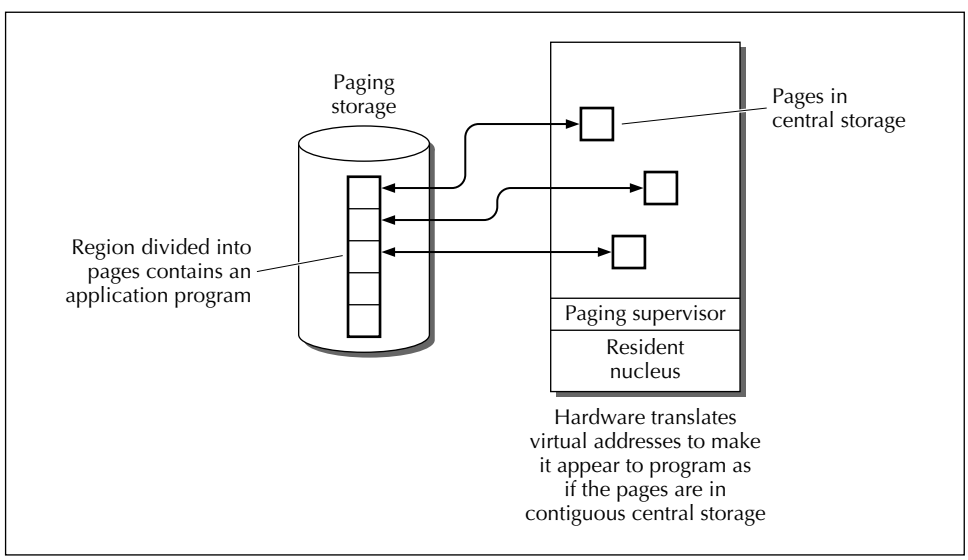



**FIGURE 2.7** Virtual paging hardware.

termed a *page break.* The paging supervisor looked around in central storage for a free page frame in which to load the page. If it found no free page frame, it looked for an inactive page in central storage to swap out.

With virtual storage, a program need occupy only a relatively small amount of central storage. This let programs run whose size exceeded the central storage available on the computer. It had another important benefit in that it allowed a larger number of programs to be run on the computer at the same time, although there was still a limit of 15. Only the virtual storage (disk) needed to contain an entire program while it was executing. Virtual storage remains an inherent part of z/OS today.

In 1974, IBM changed the name of the operating system to MVS/370 (Multiple Virtual Storage/370). It also scraped together two additional bits for addressing. With 26-bit addressing, the address space increased to 64MB. Central storage could be 64MB, the sum total of all concurrent programs in virtual storage could be 128MB, but individual programs were still limited to 16MB. MVS/370 also eliminated the restriction of 15 concurrent regions maximum. Figure 2.8 illustrates MVS/370.

By the end of the 1970s, another bottleneck appeared. Virtual storage allowed the computer to think it had a very large storage, but the 16MB

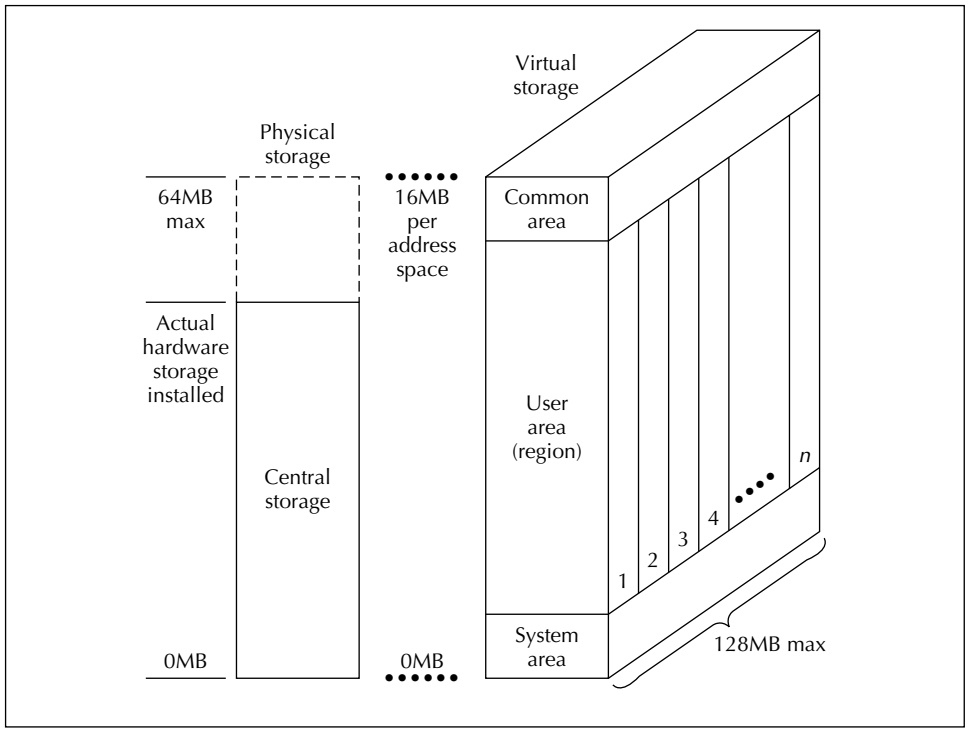



**FIGURE 2.8** MVS/370 Virtual storage and regions.

maximum for individual programs and 128MB limit for all programs in virtual storage became a problem. Programs were now sharing data, and the shared data was placed in a common area that came out of a program's address space, as did the nucleus of the operating system. (Considering that 16MB today is insufficient for one user in Windows, this was a severe limitation indeed.)

### 2.2.3 The 1980s and the Quest for Address Space: MVS/XA and MVS/ESA

In 1983, IBM introduced the successor to MVS/370 and called it System/370-XA (System/370-Extended Address) to reflect the computer's ability to now use 31 bits for addressing, which provided 2GB of address space. Although central storage could in theory be 2GB, XA supported only a 256MB central storage. However, applications residing in virtual storage could address 2GB as shown in Figure 2.9.

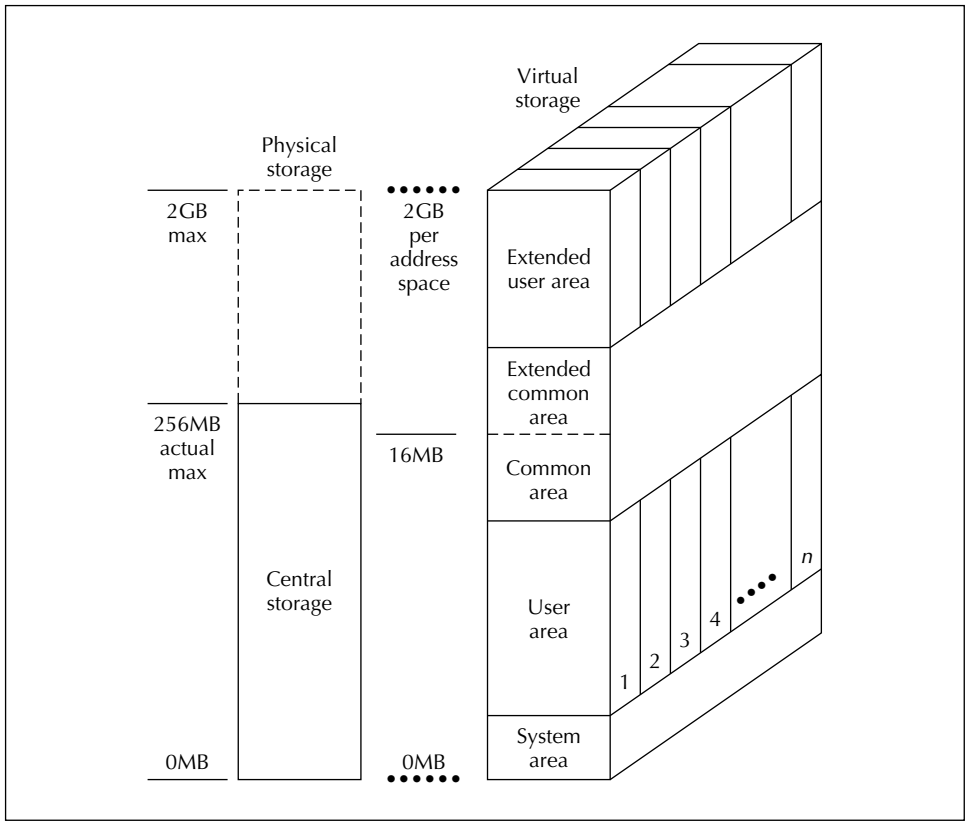



**FIGURE 2.9** MVS/XA storage.

Increasing the address space from 26 to 31 bits required changing both the hardware and the format of the computer's instructions. For compatibility, old programs could be marked to run in the 16MB address space mode, whereas new programs could use the entire 2GB address space. Programs that exceeded the 16MB address space were termed *above the line.* This could occur in two ways. First, a program could require more than 16MB in which to run. This would be a very large program and was rare. Second, and far more common, were CICS applications. In CICS, all the applications had to run in the same address space, and so even a small program could be forced above the 16MB line because of other applications loaded into storage below it, as depicted in Figure 2.10.

Users had to set special parameters during compilation and linkage editing for a program to run above the 16MB line. But if one didn't write very large programs or CICS programs, one generally didn't have to worry about it.

MVS/XA fulfilled the address space need for the large IBM mainframe for only a few years. As very large database applications were developed, address space again became a bottleneck. In response to this, IBM enhanced the operating system again in 1985 and renamed it MVS/ESA (Enterprise Systems Architecture). MVS/ESA allowed central storage to be as large as 2GB. Like MVS/XA, it also permitted an application to have a 2GB address space, but additionally permitted the same application to have multiple 2GB address spaces as illustrated in Figure 2.11.

MVS/ESA not only provided a larger effective address space, it also allowed huge applications to be segregated into functional parts. This especially benefited CICS because now each on-line application's data could be placed in a separate segment so that one application could no longer clobber another's data. Before this, CICS couldn't be impacted by a non-CICS job

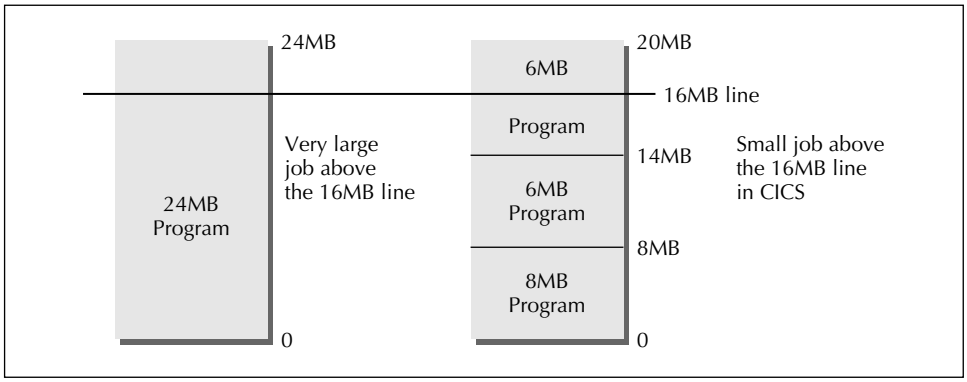



**FIGURE 2.10** Jobs "above the line."

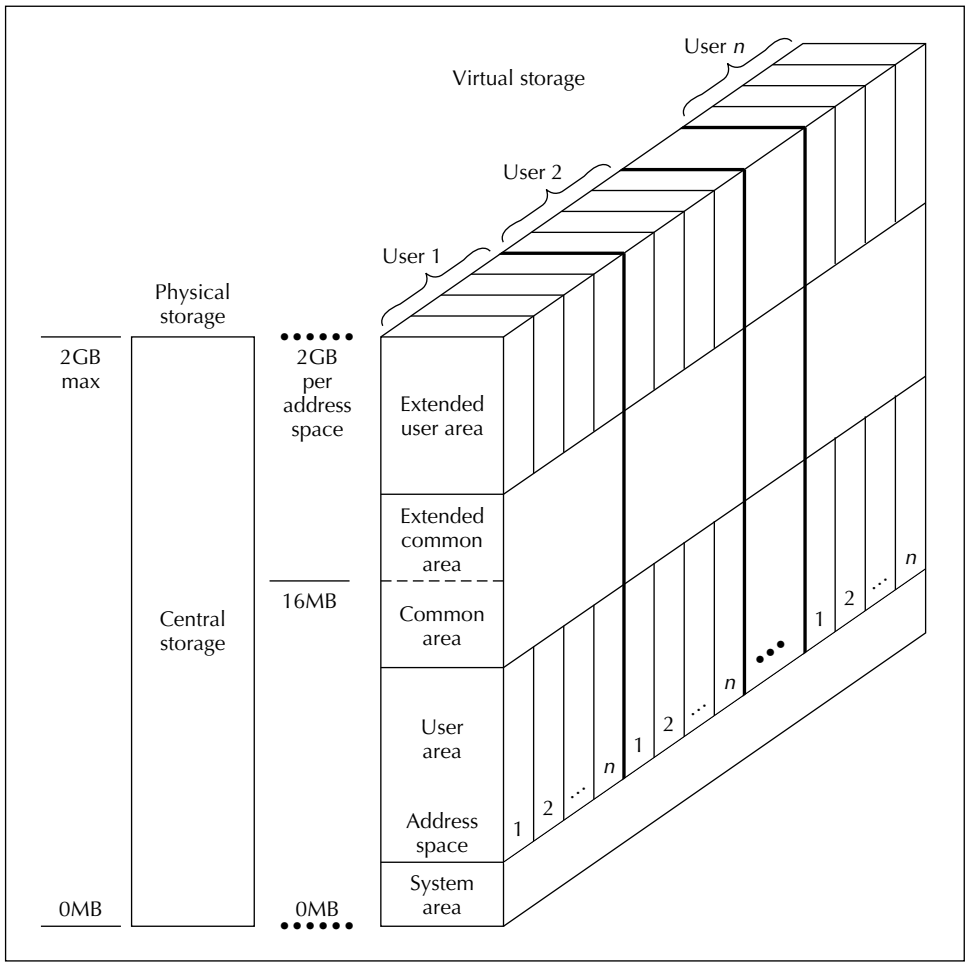



**FIGURE 2.11** MVS/ESA storage.

and vice versa, but one CICS application could take down the entire CICS system. MVS/ESA solved this problem first by allowing each CICS application to have its own data space. It then implemented subsystem storage protection to protect system code, control blocks, and data areas within the same region. With this, one CICS user could not take down another CICS user or the CICS system.

In MVS/ESA, the first address space was called the *application space,* and programs had to execute in it. The extended storage of the other address spaces were called *data spaces* and, as their name implies, could contain only data. They could contain a program as a file of data, but the

program could not be executed in a data space. The system had to copy it to the application space to execute it. The operating system didn't take up any part of the data space, so each data space had the full 2GB available for data. The data space was byte addressable, resided in virtual storage for the duration of a job, and was used as a logical extension of the address space for containing large tables, databases, and buffers.

MVS/ESA also added a facility called *Hiperspaces,* which allowed temporary data to be stored or retrieved in 4K blocks under program control, using the fast paging hardware of the computer. An application program could create a hyperspace as an alternative to storing temporary data sets on disk. The system moved the data from the hyperspace in 4K blocks into the application space for the application program to use the data.

### 2.2.4 The 1990s and the Quest for Capacity: OS/390

With MVS/ESA, the mainframe was well suited for large applications. Meanwhile, the PC had become dominate for small applications. PCs grew in CPU speed, central storage, and disk space. The mainframe had always maintained a large edge in speed over the PC, partly because it used a bipolar technology in its chips in which power was permanently applied to save time in executing a logical operation. But by the early 1990s, bipolar technology hit a wall. The chips generated large quantities of heat, and even with air conditioning and water cooling, heat could not be dissipated fast enough to allow circuits to be packed closer together. Consequently, IBM began switching to CMOS (Complementary Metal Oxide Semiconductor) technology, long used in PCs. In CMOS technology, power was applied only during an actual logical operation, generating less heat and consuming less electricity. This not only saved money, but also made battery backup more practical as a guard against power outages. CMOS technology had been slowly gaining on bipolar technology to where today, it exceeds it.

One of the consequences of the bipolar limits and the switch to CMOS was that the mainframe CPU lost its speed advantage over the PC. Size conveys no speed advantage in computers because with chips, smaller is faster. In fact, OS/390 is at a disadvantage because of its large, robust instruction set, and many smaller computers achieve their high speeds by reducing their instruction sets (RISC, reduced instruction set computers).

Central storage became so inexpensive on the PC that 128MB is considered average. Hard disks grew to where 40GB is common. How was a poor mainframe to compete against such a PC that cost only a few hundred dollars? Why not connect the PCs into a network; write some software to schedule applications, share data, and handle security; and scrap the mainframe? Some installations did this, only to discover that at great time and expense, they had reinvented the mainframe.

The mainframe cannot compete with the PC in interactivity. An individual PC may or may not be as fast as the CPU of mainframe, but it serves only one user whereas the mainframe serves many. Unused CPU cycles on a PC are valueless, so no one begrudges that PC systems devote most of their CPU cycles to screen savers.

What then was the role of the mainframe? It seemed as if the comments made back in the 1950s by T. J. Watson, Sr., the founder of IBM, that there would be a need for no more than a dozen or so large computers to handle the world's needs might indeed be prophetic.

However, there was less to all this than met the eye. Mainframes continued to be as busy as ever and IBM continued to sell as many as ever. The reason for this is threefold. First, many old applications are still running on the mainframe that would be expensive and impractical to migrate to the PC. Second, the mainframe is superb at what it does best: process vast amounts of data. When performance is measured in throughput (processing transactions, large databases), the PC cannot touch the mainframe. When there is no throughput and performance is measured by interactivity (word processing, spreadsheets, small databases, and graphics), the mainframe cannot touch the PC. The mainframe's high-speed channels, high-capacity I/O devices, its large address space, the reliability of the hardware and the software, its backup procedures and standards, and its software tools for handling databases all permit the mainframe to tackle applications involving large amounts of data that would be impractical on a PC. Such applications are also the life stream of many corporations.

The third is, of course, the Internet. Since the Internet provides access to data and most of the data is on a mainframe, mainframes quickly evolved into servers on the Internet. The IBM mainframe also supports UNIX with its z/OS UNIX System Services, and can also support hundreds of Linux images running open source applications. And it runs everything concurrently in one system.

There is another thing that the IBM mainframes do very well that we don't often think about. They are secure systems. Security was an inherent part of the system design, not something recently added. While this has been important in the past, it is likely to become the overriding consideration for future computer systems as we plunge into an information economy. Aside from issues of privacy, trillions of dollars move over the electronic highways in a hostile environment in which many people with many motivations, ranging from teenage hackers to disgruntled employees to terrorists to sophisticated crooks, stand ready to attack our computer systems.

Since individual CPUs were limited in speed, IBM developed the *Parallel Sysplex* (System Complex) to interconnect multiple systems with high-speed fiber optics so that they act as if they were a single logical entity.

Individual processors (one or more CPUs operating on the same storage) could be dynamically connected and disconnected into the Sysplex without stopping systems operations for both hardware and software updates and maintenance. Even new processors could be added without bringing down the system.

The Parallel Sysplex also provided for *shared data* in which separate servers in a client/server environment share the same image of both the operating system and user data. This permitted multiple users connected to multiple servers to operate on the same database. It was the opposite of *distributed data systems* in which different portions of the database were sent to different servers, and different than *partitioned data systems* in which multiple servers could each access different portions of a database. Shared data also allowed work to be balanced across multiple processors. The processors could share the same libraries for ease of maintenance and reduced cost.

As shown in Figure 2.12, OS/390 computers could also have multiple CPUs that operated on the same processor storage, so that when the CPU ran out of gas, the installation could add another. In addition, multiple processors could be configured as n-way multiprocessors in which two or more individual processors could either act as one large computer system

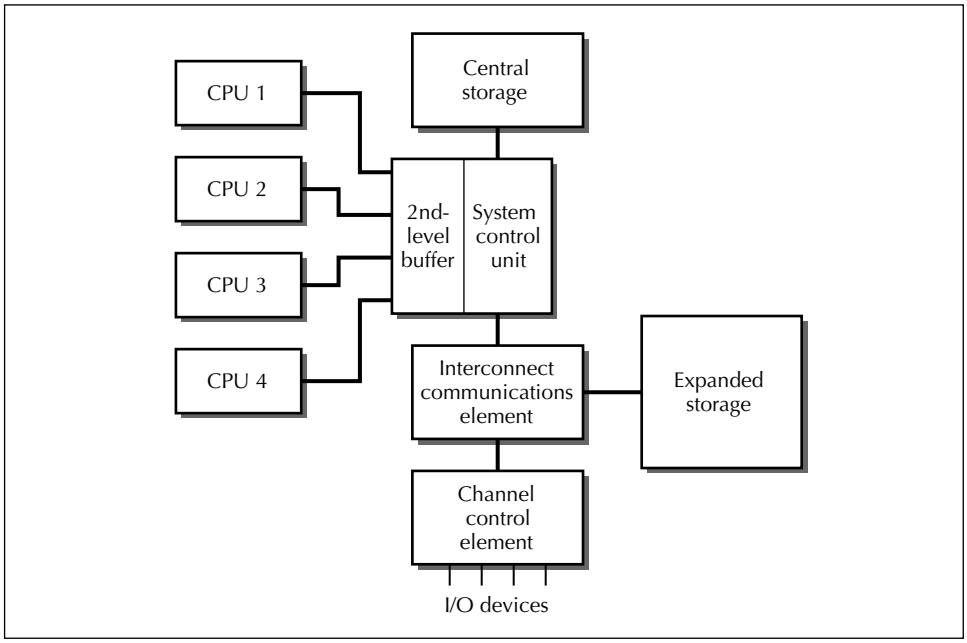



**FIGURE 2.12** Multiple CPUs.

or be physically partitioned into two independent operating systems that shared a single mechanical frame. The partitions (now called LPARS, logical partitions) might run different operating systems or different versions of the operating system, or hardware and software maintenance might be performed on one without affecting the other, as illustrated in Figure 2.13.

DASD was also changed. Large disk drives still abounded, but IBM also provided the RAMAC disk storage based on RAID technology. RAID (Redundant Array of Independent Disks) groups several smaller hard disks into a single unit. Data can then be stored sequentially in strips across the several individual disks rather than continuously on one disk to reduce access time. The hard disks are essentially those found on a PC. Because a RAID device has more disk units, there is more potential for hardware error. This is minimized by three levels of parity written onto a separate disk to allow all the data in an entire disk to be recovered, should it be lost. (The concept of parity is simple. If you stored the number 45 on one hard disk and the number 23 on another, the system could write the sum, 68, on the parity disk. Then if the number were lost on any one of the disks, the system could recover it from the other two.) All this enabled individual disks to be hot-plugged in a RAMAC unit without bringing the system down.

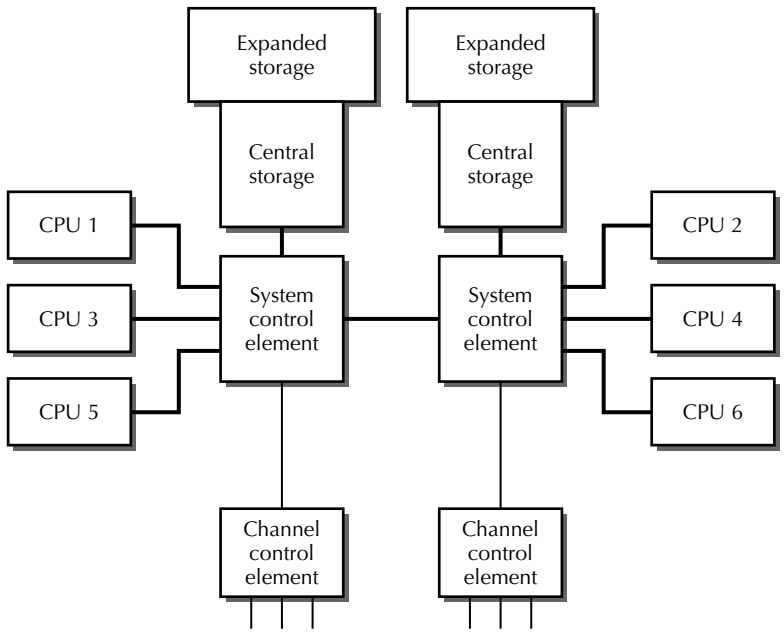



**FIGURE 2.13** N-way multiprocessor.