M. A. Auslander
D. C. Larkin
A. L. Scherr

The Evolution of the MVS Operating System

The mechanization of computer operations and the extension of hardware functions are seen as the basic purposes of an operating system. An operating system must fulfill those purposes while providing stability and continuity to its users. Starting with the data processing environment of twenty-five years ago, this paper describes the forces that led to the development of the OS/360 system design and then traces the evolution which led to today's MVS system.

Introduction

Computer operating systems first began to appear twenty-five years ago. Since then, an operating system discipline, complete with new terminology, new employment categories, large expenditures for research and development, and formal academic training, has evolved.

In this paper we review the evolution of operating systems in IBM, drawing primarily on our experience with the Multiple Virtual Storage (MVS) operating system [1] and its progenitors OS/360 and OS/370 [2]. [Another paper in this issue reviews the development of the Virtual Machine Facility/370 (VM/370 operating system) [3].] In the next section of this paper we recall the environment that prevailed throughout the past quarter century, showing some of the major changes in technology and applications. Then the major areas of operating system function are discussed in terms of the significant technological advances made in them. Finally, we consider current trends and likely future directions.

The fundamental purpose of operating systems is to facilitate the use of computer systems. Functions provided can be grouped into two categories: (1) automation of computer operations; and (2) extensions of hardware function.

The earliest systems automated operations by mechanizing inter-job transitions [4]. Jobs were executed sequentially, one at a time. In contemporary systems, separate jobs often exist simultaneously as interactive applications, with the attendant complexity of allocating resources. Modern systems provide automatic resource allocation and tuning to aid computer administrators in the scheduling of work and the balancing of resources for multiple applications.

The extension of hardware functions probably started with the symbolic assembler [5]. Other examples include higher level languages, such as FORTRAN, COBOL, BASIC, etc., and assembler macros to perform higher level arithmetic functions and input/output operations. Over the years, significant amounts of software have been produced to raise the level of the interface to hardware so that application programmers would not have to deal with such details as timing, hardware geometry, and error recovery, and could deal instead with macro- rather than micro-level operations. One aspect of this trend has been to mask the application programmer from the details of I/O devices and other hardware elements, so that hardware conversion could be done without requiring changes in application programs.

Another major source of function has been the movement of facilities common to many applications into the operating system. The earliest examples of this were the large card tubs of subroutines (*e.g.*, square root) that appeared in many machine rooms in the 1950s. These functions are first seen in new applications. Later, as they become more generalized and more popular, they find their way into the operating system. For instance, the

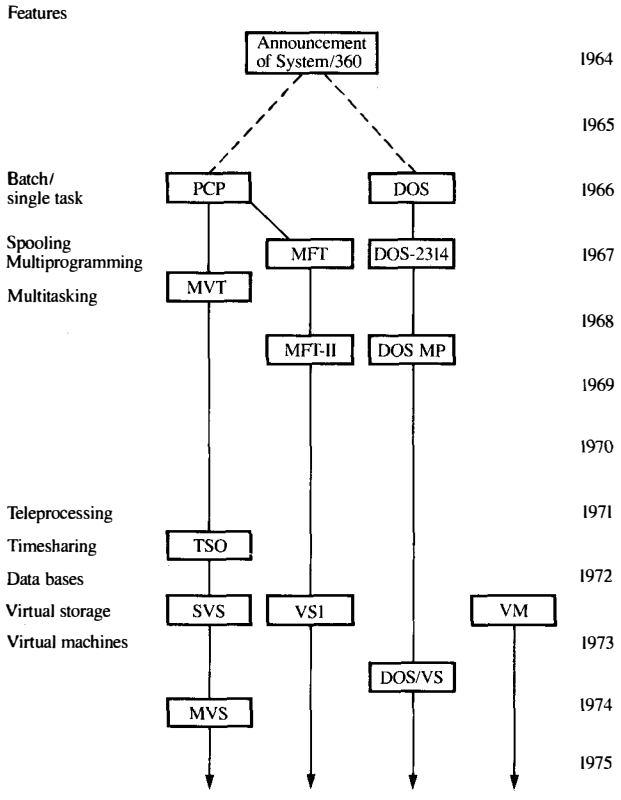

Figure 1 The evolution of OS/360, where PCP = primary control program, MFT = multiprogramming with a fixed number of tasks, MVT = multiprogramming with a variable number of tasks, TSO = time sharing option, SVS = single virtual storage, MVS = multiple virtual storage, DOS = disk operating system, and VM = virtual machine facility.

hierarchical data organization supported by IBM's IMS-DL/I program product was invented initially to support bill-of-materials processing in a manufacturing application of the late 1960s. As this support was generalized, it became clear that its usefulness ranged far beyond the manufacturing industry, and today DL/I is one of the most widely used methods for organizing data bases.

The environment: 1956–1981

To convey the environment in which designers of operating systems function, we first look at computing as it was done twenty-five years ago, contrasting it to that of today. In 1957 the total computer processing power installed in the U.S. was about ten million instructions per second. Today's installed capacity in the U.S. is three to four orders of magnitude greater. This tremendous growth of computing power led directly to the need for and definition of operating systems.

Another way to view the progress made since 1956 is to look at the size of a software system and how it has changed over the period. For instance, the FORTRAN Monitor System (FMS) and its programming libraries, widely used on the IBM 704/7090, fit on as little as one reel of tape. During application program execution, FMS required only the low 100 memory locations to contain the programming for job-to-job transition. The size of FMS itself was about 10 000 binary cards of programming, approximately one million bytes. This contrasts with today's MVS system, which arrives stored on 17 tapes that are the equivalent of 13 million cards, approximately 520 million bytes.

Over the last twenty-five years, IBM designers and programmers have created over twenty major, separate families of general purpose operating systems. The systems culminating in MVS are shown in Fig. 1 and their capabilities in Table 1. Generally these have been motivated by unique hardware, but in recent times IBM has developed different operating systems for the same hardware systems. For instance, on the latest IBM System/370 computers, there are eight IBM operating systems in use. These are: Disk Operating System/Virtual Storage Extended (DOS/VSE), Operating System/Virtual Storage 1 (OS/VS1), OS/VS2 (now called Multiple Virtual Storage or MVS), VM/370, Airlines Control Program (ACP), and Time Sharing System/370 (TSS/370), with continued substantial use of the System/360 Operating Systems, OS/360 and DOS [6].

Even within an operating system, diversity pervades. At least six different telecommunication access packages have been offered for OS/360: BTAM, QTAM, TCAM, RTAM, XTAM, VTAM [7]. In addition, several users have developed their own.

While we have created a large number of programming products over the last twenty-five years, only a relatively small number have faded from existence. An extreme example of the longevity of these products is that programming and hardware shipped as recently as 1980 still provides the capability to execute programs originally written to run on the IBM 650 and 1401 computers, first shipped in 1954 and 1960, respectively. A more meaningful example is that today's MVS systems must be capable of executing many of the programs that were written to run on the 1966 OS/360, Release 1. One of the most challenging aspects of providing software for IBM users is the huge investment in application programs for IBM computers, and one of the primary requirements placed on any operating system is to continue to provide the capability to successfully use these programs. Thus, designers of IBM operating systems have been faced with the need to provide for extensions of function while at the same time preserving the ability of existing interfaces to operate the same way they had in the past.

Table 1 Capabilities of MVS and its predecessors.

Generation	*Operations*	*Extensions of function*	
		Hardware functions	*Application functions*
Pre-operating system (early 1950s) with, *e.g.*, the 701	Manual (*e.g.*, each job step required manual intervention) No multiple-application environment support	Symbolic assembler Linking loader	Subroutine libraries in card tubs, manual retrieval
First generation (late 1950s and early 1960s) with, *e.g.*, FMS, IBSYS on the IBM 704, 709, and 7094	Automatic job batching Manual device allocation, setup, work load scheduling No multiple-application environment support Off-line peripheral operations	Higher level languages—FORTRAN, COBOL Primitive data access services with error recovery	Subroutine libraries on tape, automatic retrieval Primitive program overlay support
Second generation (late 1960s) with, *e.g.*, OS/360 on System/360	Multiprogramming Primitive work load management Primitive tuning (device, core allocation) Spooling, remote job entry Operator begins to be driven by the system Primitive application protection Initial multiprocessing (loosely and tightly coupled)	More higher level languages—PL/I, ALGOL, APL, BASIC Device independence in data access First random access data organizations Primitive software error recovery, full hardware ERP's Array of hardware function extensions Supervisor call routines	DASD subroutine libraries Full facilities for programmed overlays Interactive program development support Primitive automatic debugging aids First application subsystems Checkpoint/restart
Third generation with, *e.g.*, MVS OS/VS on System/370	Integrated multiprocessing (loosely and tightly coupled) Work load management extensions More self-tuning, integrated measurement facilities Less operator decision making, fewer manual operations Full interapplication protection, data and program authorization Primitive storage hierarchies for data	Virtual storage Device independence extended Hardware error recovery extended to CPU, channels Operating system functions begin to migrate to hardware	Growing libraries Overlay techniques obsoleted by virtual storage Symbolic debugging aids Primitive data independence Integration of application subsystems Software error recovery for system and applications

However, there are exceptions. For instance, in DPPX [8], one of the operating systems for the IBM 8100 Distributed Processing System, compatibility requirements could be relaxed because DPPX was aimed at new application environments. In spite of this, DPPX pays homage to the past in that it has COBOL and FORTRAN and must be able to interchange programs and data with other systems.

In addition to the need to provide continuity with the past, the other major constraints dictating design tradeoffs in operating systems are the capacities and speeds of the hardware for which the software is being designed. These two factors, compatibility with the past and hardware parameters, constitute the major reasons for maintaining separate operating system designs in contemporary systems. The occasional unique function in a particu-

Table 2 Numbers and varieties of devices that can be included in an MVS system.

Disk magnetic storage units	10 types
Drum magnetic storage units	1 "
Mass storage system	1 "
Diskette	1 "
Magnetic tape	8 "
Card readers/punches	5 "
Paper tape reader punch	1 "
Printers	10 "
Optical character recognition unit	5 "
Magnetic ink character recognition unit	1 "
Operator's console	7 "
Telecommunications control units	6 "
Display terminals	7 "
Keyboard/printer terminals	4 "
Remote high-speed terminals	6 "
Other terminals (including other processors)	30 "

This table does not depict all actual different model numbers; rather, it gives the number of different devices of each type.

lar system (*e.g.*, the virtual machine facility in VM/370) generally exists because of the relative cost of providing it in other systems compared to its benefits, rather than technical feasibility.

Generally, software has adapted to the rapidly changing circumstances of the data processing industry. This includes new application types, new hardware variations, and new styles of usage. On the other hand, change has been heavily dependent on the past because of the large investment all of us have in maintaining the ability to run existing programs. Over the years what has happened can be viewed as a process of natural selection, similar to other evolutionary processes.

Another dimension of the problem of designing operating systems is the diversity of hardware configurations that must be supported by a single operating system. MVS, for example, runs on eleven different processing units, some of which allow symmetric and some asymmetric two-way multiprocessing. The performance range supported is well over an order of magnitude. The minimum MVS production system has two megabytes of main storage and as few as four disk drives. Conversely, one large MVS configuration in the United States has seven interconnected processing units in a single room. This configuration has more than 450 direct access devices and services approximately 15 000 terminals in more than 50 locations. The total instruction execution capacity is over 40 million instructions per second.

The largest processor currently supported by MVS has a real main storage capacity of 32 megabytes and a maximum of 1023 devices connected to its I/O channels. Anywhere from two to 16 I/O channels are supported.

Table 2 shows the numbers of devices of various types that can be selected for inclusion in an MVS system. Providing for this large number of combinations influences the way operating systems are designed. The I/O configuration can change dramatically from installation to installation and, in fact, from month to month. Of course, it would be impractical to include all of the programming necessary to run all of these devices in every system. Moreover, as desirable as device interchangeability would seem to be, it ought to be possible to exploit new technology and new hardware capabilities. For these reasons, operating systems like MVS are designed to be modular, with support for particular device types selectable for a particular installation.

A final problem for IBM operating systems designers is the number of installations using each system. Changes and extensions made to our systems are, soon after release, experiencing extremes in work loads and varieties of applications. A change that is incorrect in the most obscure special case will soon generate a problem report or "APAR." A modification to an undocumented and unguaranteed internal characteristic of an interface is often exposed by applications which work only because of that characteristic. This necessarily contributes to a design approach that is conservative and evolutionary and that groups modifications so that extensive tests can be performed on modified systems before their shipment. Yet greater stability is one of the most asked for features by customers of our products.

In summary, our experience of developing operating systems is that of responding to requirements in the basic functional areas under the constraints of a wide range of hardware configurations, large numbers of installations, and compatibility. The following sections explore the approaches we have taken in response to these motivating requirements.

Operations

The earliest use of computers was characterized by totally manual operations; that is, the sequence of programs to be run was controlled manually, media containing needed data were mounted and dismounted by the operator, usually following written instructions. A typical compile-load-go sequence in the 1950s was accomplished by the operator placing the binary card deck for the compiler in a card reader, with a program to be compiled behind it, and pressing the Load Cards button on the console. The compiler assumed that certain "scratch" tapes were available and used these during the compilation. At the end of the compilation, output was printed, and cards representing the object program were punched. This I/O activity was not overlapped with any other

activity. The operator would then pull the object deck out of the card punch, walk over to the card tub, and select the required subroutines and an eight-card linking loader that was placed at the front of the resulting deck. These cards were then placed in the system card reader, and the operator once again pressed the Load Cards button. If any data were required by the program, the appropriate media would be prepared by the operator.

Generally, if operations were any more complex than described, written instructions were provided for the operator, or more likely the programmer would be personally present. "Programmer present" runs were so common that a space on the job card was provided to indicate that type of run. In these cases, the operator would telephone the programmer when his run was about to be made.

Accounting was done on a total system basis, often using a time clock to log a user on and off of the system. Users and operators took great pride in the speed with which they could mount tapes and operate the hardware to minimize the idle time between jobs.

By the late 1950s, this type of operation had been semi-automated, with the card tub of library subroutines placed on tape and the linking loader having the capability of searching this library tape for unresolved external references. Moreover, the ability to change from the compilation to the execution phase of the job without manual intervention had been provided. Still, communication with the operator was done for each program in pretty much an *ad hoc* way. The 704 had a series of lights on the console called Sense Lights, and often instructions to the operator had statements like "if Sense Light 4 lights, turn on Sense Switch 3."

Through the early 1960s, systems were characterized by manual device allocation, setup, and work load scheduling. Multiprogramming was not provided, and only a single application would run at a time. It had been recognized very early that the printing of output and the reading and punching of cards were operations that could not be performed efficiently on a high-speed central system. Thus, tape input and output operations were substituted, and the card-to-tape, tape-to-printer, and tape-to-card operations were done off line. At first this was accomplished with special purpose equipment designed specifically for these functions. Then these off-line operations were done with small computers, such as the IBM 1401. Later the IBM 7040 was linked via a high speed connection to the larger IBM 7094 to do these operations and related job scheduling functions. This "Direct Coupled System" is one of the earliest examples of automated job scheduling and setup. It significantly improved the utilization of the controlled systems by overlapping setup and peripheral input and output with system operations. This approach survived into the System/360 era as the Attached Support Processor (ASP) package and is now embodied in the JES3 subsystem of MVS.

It was not until the multiprogramming support provided in the mid-1960s that job scheduling and peripheral operations were done simultaneously with other work on the same processor [9]. It was at this time that the operator, rather than having control of the system, began to be provided with instructions by the system. Comprehensive job control languages were introduced to allow the application programmer to completely specify sequences of job steps, as well as device and data requirements for each, so that the system could allocate and sequence the appropriate programs and resources. The OS/360 Job Control Language (JCL) was widely regarded as complex and difficult to use. Yet its contribution was to separate application programs from such operational considerations as device types, addresses, sequences, conditions, and timing of execution. Thus, it was possible to create new applications by writing JCL programs to apply existing processing and utility programs in new ways.

Today, the most significant operational challenges remaining are the mechanization of the remaining manual functions and the improvement of existing algorithms for work load management so as to react effectively to changing work loads and to the lack of availability of particular pieces of hardware or other resources due to errors, failures, or contention. In the future these decisions will be increasingly pre-programmed into the system, and more of the manual media operations will be eliminated through the use of storage hierarchies and automated media handling techniques.

- *Maintenance and service*

Large numbers and sizes of programs are typically involved in the modern data processing installation. Moreover, the problems of fixing errors, installing new versions of software packages, updating critical data, etc., must all be done in the face of the need to operate more continuously. In the early 1960s, it became apparent that *ad hoc* techniques would no longer suffice. By the 1970s several data base oriented systems had been created in IBM to keep track of modifications, error fixes, and versions of programs. Because code modifications made to fix errors sometimes themselves contain errors, such a system must allow swift recovery if such errors occur. Thus, if a modification contains an error, it must be possible to restore the system to its original state very

rapidly. With the large installations of today, and especially with the distributed systems that are beginning to appear, it is essential to provide the ability to manage the servicing of large numbers of systems with a single centralized package. Thus, record keeping and distribution facilities must be expanded to allow for tracking up to hundreds of systems.

Since the time to diagnose problems is important, and since often problems encountered in a system at one installation have been found and fixed earlier at others, it is useful for an installation to have access to a central data base of problem symptoms and fixes. During the 1960s this was accomplished by distributing this information on listings. Recently, remote interactive access has been provided to this data, and, in the future, symptom descriptions to be used to search these data bases will be generated automatically.

- *Resource allocation*

OS/360 was the first general purpose IBM operating system to provide for complete sharing of hardware resources. The major resources considered for shared usage in the original design were the processor, its memory, direct-access storage devices (disks, drums), I/O devices (tapes, card equipment, printers, etc.), and the space on individual disks and drums.

OS/360 provided for processor sharing by introducing the notion of a "task" [9]. An OS/360 task is an implementation of what computer scientists eventually came to call a process. An additional characteristic of a task in OS/360 is that resources, including other tasks, are allocated to a task. Further, tasks are designed to allow full multiprogramming of their execution. The OS/360 designers recognized that operating system functions should usually contend for resources by being themselves treated as tasks.

The task structure was originally intended to support not only long running batch jobs but also the short running, response-oriented processing associated with transaction and telecommunications environments [9]. Unfortunately, the generality of the task concept was too great to allow for efficiency. The fact that the task is interruptable by other higher priority tasks and is the anchor for resource accounting, allocation, and recovery required significant processing just to create a task. Thus, the overhead was impractical for very short-lived pieces of work. A widely used technique to overcome this problem was to have a task waiting for a signal to restart, rather than creating a new one each time. This technique created long queues of mostly dormant tasks that had to be searched, and it added overhead in other areas. Also, because the task did not fully support all of its characteristics in its first implementations, early application subsystems like QTAM, CICS, and IMS [10] provided their own substitutes for the task. In MVS, a new construct called the System Request Block (SRB) was created to be purely a unit of work for the processor. Interruption ability was not provided, and no other resources were anchored to the SRB. Thus, an SRB in MVS can be created and given control of the processor using a trivial number of instructions, compared to the thousands required to create a task.

The original intent in the OS/360 design was to allocate memory to each task on demand [9, 11]. Memory would then be held by the task until it was either explicitly released by the task or the task terminated. The existence of this interface, and the crucial and new requirement that programs ask the operating system for memory space and accept that space at whatever location it was provided, made for some early difficulties. However, this interface once and for all solved the problem of how the operating system and the application program can share the processor memory, even though they are independently designed. For this reason alone, similar storage allocation mechanisms persist in all operating systems today.

From the beginning, the designers of OS/360 were concerned with the possibility of a resource allocation deadlock [12]. They chose the philosophy of complete deadlock prevention by predetermining the order of allocation: I/O devices, data sets, memory, then the processor. It was not recognized until late in the design that the earlier assumption of complete dynamic allocation of main memory from a common space would lead to deadlocks. To overcome this problem, the application programmer had to specify the aggregate storage requirement of his job in advance. This "region" could then be allocated, and the possibility of a deadlock during execution was avoided.

By the late 1960s, OS/360 was being severely strained by two emerging requirements. The under-utilization of memory caused by region allocation was becoming ever more troublesome. At the same time, it was recognized that application development could be eased if programs did not have to be fitted into the smallest possible address space. Dynamic address translation hardware and demand paging techniques were applied to these problems throughout the 1960s in such IBM systems as the 7044 and System/360 models 40 and 67 [3, 11]. In 1970, demand paging was added to OS/360. In MVS, a large number of regions can be created, with the size of each region the same and usually not a factor to be considered by the programmer. The technique of demand paging is

used to share real memory among the existing regions. Because demand paging allows dynamic reallocation of memory, deadlocks cannot arise.

The second emerging requirement was for time sharing [13]. This technique can improve the capabilities of program developers and problem solvers by allowing continuous communication with the computer. Time sharing was first provided by a number of special purpose systems in the early 1960s. By 1967 there were over 30 different special purpose time sharing packages running or planned for System/360 hardware. The Time Sharing Option (TSO) was intended to integrate time sharing functions into the OS/360 base. However, the batch allocation of devices and data space present in the base were inappropriate. In practice, time sharing is workable when the storage devices in use are permanently mounted and when space on them is dynamically allocated. To accomplish time sharing in OS/370, dynamic disk space management interfaces were added to the existing batch mechanisms. It is noteworthy that TSO uses essentially the same interfaces and mechanisms of OS/360 for purposes for which they were not initially intended.

Time sharing was implemented in TSO without address translation hardware. Instead it used a memory swapping technique pioneered at MIT on the IBM 7094 [14] in the early 1960s. The fundamental resource management problems of time sharing, for which these special systems had previously been developed, can be dealt with by the multiprogramming and paging mechanisms of MVS. In creating MVS from the OS/360 and TSO componentry, many of MVS's major structures came out of generalizations of techniques introduced in TSO.

Today, three resource allocation issues concern operating system developers. The first is the impact of multiple processors. It might seem that a multiprogramming operating system could be converted to multiprocessing with minimum difficulty. In practice, however, an operating system contains many internal information resources which are frequently locked and unlocked. As the number of processors increases, so does the contention for these internal resources with the attendant loss of performance when conflicts arise. To control such conflicts, MVS uses carefully designed lock structures to minimize the cost of locking an available resource and the probability of needing an unavailable resource. The MVS approach is based upon the experience gained during the 1960s with the OS/360 and TSS support for multiprocessing. The OS/360 structure had a single lock and produced serious degradation in many important environments. The TSS approach used myriad locks but in an uncontrolled sequence that created deadlock and recovery difficulties.

The other two resource allocation problems deal with resources that were not shared when OS/360 was first designed: communication facilities and data stored at the logical record level [15]. The former is the subject of the Systems Network Architecture (SNA) [16] and has required substantial innovation. The second is primarily a problem of data base subsystems [17].

- *Security and integrity*

Over the years, operating systems have played an increasingly important role in providing tools to allow an installation to protect itself against unauthorized access to data or other computer facilities [18]. In the earliest systems, little or no provision was made to protect against such use. In the early 1950s, before the advent of multiprogramming, such protection was generally not needed, since a job's data were loaded onto the system with the job and taken down at its completion. When jobs were first batched on an input tape, there was the ever-present danger of one job forward or back spacing the input tape, causing other jobs to be skipped or repeated. At some universities, this was an annual occurrence as new programming students "tested the system." It was not until late in the 1950s that IBM computers began to provide protection. There was a multiprogramming special feature for the 7090 that provided for privileged instructions—only programs in certain states could execute instructions that performed I/O operations and other management tasks.

In the System/360, the separation of privileged instructions from those intended for use by application programs was made quite clear; it was theoretically possible for software to prevent unauthorized use of any function. The original direction taken in OS/360, however, was that deliberate penetration attempts would not be prevented; that is, the system was designed only to prevent casual or accidental penetration or unauthorized use. It was not until the late 1960s that it became apparent that this philosophy was inadequate. Operating systems had to prevent a program from gaining access to data, services, or other facilities that it was not authorized to use. MVS was the first IBM system for which a systematic attempt had been made to eliminate all exposures to unauthorized use and for which a commitment had been made to correct any such exposures found. MVS had a number of authorization schemes by which particular users or programs could be allowed or denied the use of certain data and functions.

Since the introduction of MVS in 1974, additional loopholes have been identified and closed. Also, more elaborate authorization capabilities have been provided, as well as detection capabilities, to facilitate the identifi-

cation of a perpetrator, and hardware assistance, such as encryption devices [19].

Extensions of hardware function

The following subsections deal with extensions to hardware function to enable application programmers to more easily exploit computer systems. The first subsection deals with extensions that raise the level of function of the processor's instruction set. The second deals with similar extensions in the I/O subsystem area. The final two subsections deal with error recovery, which could be viewed as the creation of ideal computer elements that apparently never fail.

- *High level languages*

Early operating systems, such as the FORTRAN Monitor System, were based on an intimate connection between their language processors and the system which managed compilation, loading, and execution of programs written in those languages. An important advance of OS/360 was to separate the compiler from the operating system. A compiler is, from the operating system view, just another user program which takes input and produces output. Whether or not the user then decides to have the program loader load and execute that output is unimportant. The only tie between the compiler and the system is the conventions for representing machine language programs as operating system files.

This separation has a number of advantages. Compilers can be offered and maintained independently of the operating system. New languages or new compilers can be developed easily. Various organizations inside and outside IBM can offer languages and compilers which do not depend, for their success, on operating system modifications. This has led to a healthy growth in both languages and compilers. Finally, this decoupling encourages general purpose operating system interfaces. However, the danger is that significant functions may not be available to higher level language programmers.

Another aspect of the interaction between high level languages and operating systems is the implementation language of the system. Until the late 1960s, IBM software was written in macro assembly language. It was believed that compilers could not produce code efficient enough for operating systems. While this assertion was true in 1970, the reduction in programming errors and the increase in program extensibility resulting from the use of high level languages offset the space and execution time penalties. By 1980, most programmers could not have consistently exceeded the efficiency of compiler optimized code. Now essentially all new operating system code is written in a high level language.

- *Access methods*

As operating systems evolved in IBM, they were driven by a number of forces towards the development of generalizations of their hardware input/output devices. The primary forces operating were:

Ease of programming Programs must often deal with the mechanical complexities of the device, variations in the storage medium, complicated procedures, and error detection and recovery strategies. These often change with each new hardware innovation.

Device independence For purposes of data storage, the important distinctions from the point of view of the application programmer are record size, retrieval order, selection techniques, etc., rather than media dependence. A program written to process a sequential stream of 80-character records should be the same no matter where the data are stored.

Data integrity The shared use of direct access storage device (DASD) space and the shared (often read-only) use of data require some trusted program between the application and actual manipulation of the media.

Concurrent operation The involvement of the operating system in every input/output operation makes possible the incorporation in its implementation of strategies for overlapping computation and input/output activity without requiring that individual programs include complex code (such as double buffering) to produce local concurrency. The concurrency arises rather because one job is computing while another is doing input/output operations.

Early operating systems often dealt with the simpler aspects of some of these needs. Beyond that, the libraries for some high level languages contained input/output routines which eased the programming burden for users of that language. With the introduction of OS/360, the notions of an access method and of a data set organization were introduced in a uniform way [7].

Data set organizations are abstractions that expose to the application program those distinctions which it must deal with and shield it from other device distinctions. The original data set organizations of OS/360 were sequential, partitioned, indexed sequential, and direct access.

Access methods are collections of operations that can be applied to read and write data in these organizations. The original access methods were Basic Sequential (BSAM), Queued Sequential (QSAM), Basic Partitioned (BPAM), Indexed Sequential (ISAM), and Basic Direct (BDAM). The distinction between basic and queued had to do with whether or not the access method supported

implicit blocking and deblocking internally, as well as whether or not the access method was capable of a read-ahead input/output strategy. In hindsight, one could argue that this distinction was short sighted. However, in 1963, it was difficult to decide to isolate the programmer from the device at all, let alone to impose a blocked, buffered interface as the only one available. Real concerns with performance and a confusion of the technique of buffering with the function of sequential access both contributed to this viewpoint.

These small sets of organizations and access methods served for access to all devices and data during the early history of OS/360 and OS/370. Changes in this area have been remarkably rare, considering that these interfaces were simply invented and implemented all at once by the system architects. The major changes to occur in the intervening years reflect a new requirement and the recognition of a new technology.

The new requirement is the desire to attain uniformity and device independence for terminals. BTAM, TCAM, and VTAM represent three attempts at communications access methods. A number of reasons exist for this relative lack of success in standardizing communications access methods.

First, the communication devices did not become important until after the introduction of OS/360. Without a body of experience, and with dramatic and fundamental changes in hardware capability (from teletypewriters to CRT display terminals and beyond), it was difficult to foresee all future needs in designing an interface. Beyond that, more than in any other input/output area, the application program must deal with the details of the display device. Adding more bytes to a disk track can easily be hidden from the application program using that disk to store information, but adding more rows or columns to a display necessarily affects the application design decisions relating to the human interface that the display represents. Providing for terminal device independence in applications is one of the remaining challenges that will get much future attention.

The other important change was a response to a new technology. The indexed sequential access method was designed to support data organizations in which contents retrieval (keyed records) was used to maintain the appearance of a sorted file while allowing efficient random update and retrieval. When it was designed, it was commonly believed that the best way to do such retrieval was to exploit the key search capability of the DASD hardware. After the introduction of OS/360, the use of tree structured, balanced indexes (B-trees) as a technique for managing content addressing was developed, and it became clear that this organization was superior to the ISAM approach. To offer this new technology, IBM introduced VSAM, an access method which in some sense provides all the organization and retrieval functions of the other access methods. Again, history and the need for continuity tie us to using and supporting the new and the old. While the use of ISAM is waning, the use of the sequential, partitioned, and direct access methods continues and will continue for the forseeable future.

The OS/360 access methods went a long way towards achieving ease of programming, device independence, data integrity, and concurrent operation. As much as any other feature of OS/360, they have made it possible for programs written fifteen years ago to continue to operate on today's totally new and often quite different hardware and operating systems.

In OS/360, and subsequently in MVS, the format of recorded data and the available operations differ by access method and organization. Thus, device dependence was, to some extent, replaced by data set organization dependence. For example, sequentially organized data cannot be accessed by record number without a format conversion. It is possible, as is demonstrated by the DPPX operating system and by CMS, to achieve uniformity and provide better usability as a result.

- *Hardware error recovery*

IBM's first computers were shipped with little more programming than that used for hardware error diagnosis. Typically, engineers develop such programs for the early models of a system. As computer programs became more sophisticated, these diagnostics became less effective. By the end of the 1950s, it was commonplace for a nontransient error to be undetected by the manufacturer-supplied diagnostics but to show up readily when trying to execute application software. It was during this period that the gathering of hardware error status and statistics was first introduced into IBM software. During the early 1960s, the software was designed to stop upon the detection of any hardware error, under the assumption that to continue would produce unpredictable results. This philosophy was reinforced by the fact that the hardware often signaled a status that was undefined. Nevertheless, elaborate algorithms were created to re-try operations if errors might be transient or, in the case of input/output functions, to attempt alternate paths to devices. For instance, if a control unit for disks had paths to two I/O channels, one of the strategies to recover from a channel failure would be to attempt access through the second channel.

It was not until the early multiprocessing systems that serious attempts to recover from central processor errors

were made. The IBM 9020 System [20], created to assist in air traffic control, was a fully redundant multiprocessing system that could continue despite the failure of any single element. Much of the effectiveness of the software in accomplishing this goal was due to the nature of the application. Since the major function of the system was to process incoming radar data, little stored data had a useful life of more than a few seconds. Thus, it was possible to reconstruct the state of the application a relatively short time after a failure.

The general purpose systems did not fare quite as well. It was not uncommon when a single element failed, even though spare hardware was available, for the software to be unable to untangle itself well enough to continue. By the late 1960s it was apparent that a major undertaking was required to address the effectiveness of software in recovering from hardware errors and, indeed, hardware failures. If this were successful, a two-processor system could reliably continue with one processor remaining. One aspect of this work involved a joint effort by hardware architects and software designers to ensure that adequate status information was presented so the software could continue from a known state. In MVS, this type of recovery was implemented on a large scale within the operating system. Thus, in the event of failure of a single processor in a two-processor system, the software can move the work being done on the failed processor over to the remaining one and simulate two processors until a pre-defined state has been reached. Subsequently, the system can operate as if it were a normal single-processor system.

- *Software error recovery*

One of the lessons learned during the original implementation of MVS was that software errors manifested themselves in more complex and unpredictable ways than hardware errors. Because the relative frequencies of hardware and software errors were comparable, it was decided to attempt to handle them both with a single set of facilities that would signal the program in control at the time of the error. The signal took the form of an interrupt to a pre-specified exit program. If no such exit program had been specified, then control would be given to an exit specified by the next higher level program (*i.e.*, the program that called the program in control). Since the operating system itself is the ultimate caller of every program, ultimately a recovery exit would be reached.

The general ground rule in the design of MVS was that every part of the operating system should be designed to include such recovery facilities. The job of each recovery exit was to assess any damage and either repair it or remove ongoing work from the system. In some cases it was deemed acceptable to lose resources for the duration of the program or job that was running or until the next time the system was initialized. For instance, if an error destroyed the records of free storage and if those records had to be reconstructed but some were lost, then perhaps several thousand bytes of storage might be unavailable for a period as a consequence of the recovery action. This is preferable to losing the entire system and having to reinitialize it.

The MVS programs associated with error exits are called functional recovery routines (FRRs). As would be expected, these were more effective in dealing with hardware errors than software errors. Because they were executed only when an error was detected, their effectiveness could really be assessed only over time and usage in real situations. Over the years this experience has led to increasing the effectiveness of these routines, and MVS is a more robust system as a consequence of having them. A remaining challenge is to substantially improve the effectiveness of detecting software errors.

The difficulties in creating an effective recovery program increase with the generality of the program for which recovery is being attempted. For example, if a failure occurs during a basic supervisor service, such as establishing a connection to a file (*e.g.*, OPEN), and if in fact that data set cannot be made available to the application program, the specific actions to be then taken are really best left to the discretion of the application program. Depending on which file cannot be accessed and its importance to the application, the program may or may not be able to continue. Thus, the burden could be shifted entirely to the application programmer.

In the interests of application programmer productivity, a great deal of effort has gone into the error recovery facilities of the data base and data communications subsystems of operating systems. For instance, the designers of the IMS system [10, 15] have gone to extraordinary lengths to protect the integrity of data under its control from errors introduced by both hardware and software failures. The concept of a transaction has been incorporated into IMS, and the system is designed so that the data base reflects changes made only by successfully completed transactions. Thus, if a program performing the work of a transaction fails, no updates to the data base are ever actually reflected. To implement this concept, IMS maintains journals of changes and undoes any changes made by a failing transaction program. In addition, IMS can maintain journals of data base changes so that if a disk file is lost due to an error, recovery can be accomplished by loading a check-pointed version of the data onto the disk and by processing the journal of

changes against it, bringing it up to date. Transaction-oriented recovery has been implemented on a variety of systems to some degree, including CICS and DPPX/DTMS. Over the past ten years, the concept of transaction-oriented data base recovery has been generalized and extended. It is now accepted as a standard way to deal with this type of data processing and error recovery.

A future trend in this area will certainly be the combining of all the techniques so as to provide more effective and rapid recovery in the event of a hardware or software failure. As systems get larger, restart times are increased, making faster recovery more critical. Also, the significance of failures is aggravated because systems are being directly used more by human beings. Thus, the effective use of standby equipment, allowing switch-over in seconds, will become more important.

- *Application subsystems*

The earliest application subsystems were implemented to provide batch remote job entry (RJE), improve batch job scheduling, and provide interactive facilities for time sharing and other problem solving applications. These were implemented on the earliest versions of OS/360 in the middle 1960s. HASP and ASP are two examples of the RJE and batch job scheduling subsystems. APL [21] on OS/360 is an example of the latter type. As the techniques pioneered in these application subsystems became better understood, they were integrated into the operating systems, and use of the separate subsystems gradually faded.

The early 1970s also saw the popularity of the data base/data communications (DB/DC) systems. These systems used different forms of resource allocation to achieve goals similar to those of the time sharing systems. The chief difference between the DB/DC systems and the time sharing systems was that the former generally did not need to maintain program context between input messages. That is, an incoming message could be processed and the data bases updated without the need to maintain program variables until the arrival of the next input. On the other hand, interactions with users in a time sharing environment were more conversational, and systems generally had to provide the ability to relate successive inputs and programs needed to maintain continuity from one input to the next. This difference led to a number of DB/DC resource allocation strategies which provided for higher efficiency than was available in time sharing systems. This improved efficiency was significant enough to warrant special purpose implementations. Here again, usage during the 1970s gravitated toward two or three packages for the System/360, and in the late 1970s the movement of functions from the subsystems into the operating system base had begun to occur.

The overall trend in this area is that, when operating systems do not schedule resources or provide similar functions in a way which is adequate for new functions, these new functions are provided in application subsystems. As these functions are identified it is expected that they will be integrated into subsystems and then the subsystems themselves into operating systems. Thus, a natural selection process identifies the soundest techniques and the most generally acceptable solutions.

Conclusions

The most unusual aspect of the IBM programming environment is undoubtedly the combination of forces brought to bear by the large number of installations and their diversity of applications. This, coupled with the ongoing stream of new hardware devices, has created a situation in which small changes are amplified substantially. In other words, the law of large numbers is operative, and it is easy to see the results of any change, both positive and negative. The rate of change has been immense and progress has been made in virtually every dimension of the computer programming craft.

Over the years, there has been substantial movement of function out of the manual realm, as well as the realm of the applications programmer, into the operating system itself. The operating system has taken over functions relating to increasing the usability of the hardware, to protecting application programs from changes in hardware, and to saving them from the need for conversion in the event of such shifts. Operational considerations have changed from the stand-alone single batch systems of the 1950s and early 1960s to the multi-system, geographically dispersed, but centrally managed, complexes of today, with the attendant operational and control requirements of such systems being placed in the various operating systems. The evolution of increasingly complex and capable data organizations, including the relational facilities that are now being used, represents another thrust of increasing function. Other trends include more facilities to ensure automatic error recovery and data integrity in the event of failures.

In all cases the effect has been to reduce the need for application programs to provide various functions. The net effect is that, compared to twenty-five years ago, the average application program as written by the programmer represents a much smaller fraction of the cycles executed by the hardware itself. In 1956, it was indeed rare that IBM written software took more than a few cycles of the system. Today it is not at all uncommon for IBM provided software to use 90 percent or more of the capacity of the system, with the user written application programs using the remainder.

Other strides during the period include decreasing dramatically the rate at which errors are introduced into programming. In the last ten years there is evidence to suggest an order of magnitude improvement in the number of errors per line of code introduced in IBM operating systems. Moreover, major efforts have been made to reduce the impact of errors by providing various recovery strategies in the software itself. Also, provisions are being made to allow for continuous operation, whereby changes and updates can be made to a system without stopping it. Finally, ease of use from the installation of the system to its actual use has lately received significant attention.

The overall increases in the capability of the operating system are really the result of what is perhaps the major economic trend of the last twenty-five years in the data processing industry—the steady decrease in the cost of data processing hardware in the face of steady increases in the cost of human labor. Thus, every year it becomes easier to justify automating manual functions through the use of computers and of augmenting those functions already automated to make them more efficient in their use of human time. We have lately seen the second and third implementations of applications that were originally created in the 1950s as batch programs. For instance, in 1956 many computer users were beginning to automate inventory control through the use of transactions written on sheets of paper, transcribed to punch cards, sorted, and processed in a batch mode against a master inventory file. Today such applications are implemented as interactive DB/DC programs that update inventory records on a real-time basis. Obviously, there is an incremental value to this type of operation, since inventories are maintained on an up-to-the-minute basis, and other aspects of operations can also be automated.

At the moment there seems to be no end in sight for this trend. It is anticipated that more and more emphasis will be placed on the traditional operating system areas of further expediting operations, extending hardware functions, and providing common application functions as part of operating systems. As was the case twenty-five years ago, it is apparent that the requirements of the next twenty-five years are not all known today, and the ability of our programs to be adaptable to unforeseen requirements will remain a very important characteristic. Implementing functions in a generalized way is the best preparation for unforeseen requirements.

Acknowledgment

The authors would like to acknowledge Mr. Bob O. Evans, IBM Vice President—Engineering, Programming and Technology, for stimulating the creation of this paper.

References

1. A. L. Scherr, "Functional Structure of IBM Virtual Storage Operating Systems, Part III: OS/VS2-2 Concepts and Philosophies," *IBM Syst. J.* **12,** 382–400 (1973).
2. G. H. Mealy, "The Functional Structure of OS/360, Part I: Introductory Survey," *IBM Syst. J.* **5,** 3–11 (1966).
3. R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM J. Res. Develop.* **25,** 483–490 (1981, this issue).
4. A. S. Noble, Jr., "Design of an Integrated Programming and Operating System, Part I: System Considerations and the Monitor," *IBM Syst. J.* **2,** 153–161 (1963).
5. Nathaniel Rochester, "Symbolic Programming," *IRE Trans. Electron. Computers* **EC-2,** 10–15 (1953).
6. G. Bender, D. N. Freeman, and J. D. Smith, "Function and Design of DOS/360 and TOS/360," *IBM Syst. J.* **6,** 2–21 (1967).
7. W. A. Clark, "The Functional Structure of OS/360, Part III: Data Management," *IBM Syst. J.* **5,** 30–51 (1966).
8. S. C. Kiely, "An Operating System for Distributed Processing—DPPX," *IBM Syst. J.* **18,** 507–525 (1979).
9. B. I. Witt, "The Functional Structure of OS/360, Part II: Job and Task Management," *IBM Syst. J.* **5,** 12–29 (1966).
10. W. C. McGee, "The Information Management System IMS/VS—Part 1: General Structure and Operation," *IBM Syst. J.* **16,** 84–95 (1977).
11. L. A. Belady, R. P. Parmelee, and C. A. Scalzi, "The IBM History of Memory Management Technology," *IBM J. Res. Develop.* **25,** 491–503 (1981, this issue).
12. J. W. Havender, "Avoiding Deadlock in Multitasking Systems," *IBM Syst. J.* **7,** 74–84 (1968).
13. Allan Scherr, "An Analysis of Time-Shared Computer Systems," Ph.D. Thesis, MIT Press, Massachusetts Institute of Technology, Cambridge, MA, 1967.
14. F. J. Corbató *et al.*, *The Compatible Time-Sharing System, A Programmer's Guide*, MIT Press, Cambridge, MA, 1963.
15. W. C. McGee, "Data Base Technology," *IBM J. Res. Develop.* **25,** 505–519 (1981, this issue).
16. J. H. McFadyen, "Systems Network Architecture: An Overview," *IBM Syst. J.* **15,** 4–23 (1976).
17. M. W. Blasgen *et al.*, "System R: An Architectural Overview," *IBM Syst. J.* **20,** 41–62 (1981).
18. C. R. Attanasio, P. W. Markstein, and R. J. Phillips, "Penetrating an Operating System: A Study of VM/370 Integrity," *IBM Syst. J.* **15,** 102–116 (1976).
19. W. F. Ehrsam, S. M. Matyas, C. H. Meyer, and W. L. Tuchman, "A Cryptographic Key Management Scheme for Implementing the Data Encryption Standard," *IBM Syst. J.* **17,** 106–125 (1978).
20. G. R. Blakeney, L. F. Cudney, and C. R. Eickhorn, "An Application-Oriented Multiprocessing System: II—Design Characteristics of the 9020 System," *IBM Syst. J.* **6,** 80–94 (1967).
21. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York, 1962.

Received July 6, 1981; revised July 27, 1981

M. A. Auslander is located at IBM Corporate Headquarters, Old Orchard Road, Armonk, New York 10504; D. C. Larkin and A. L. Scherr are located at the IBM System Communications Division laboratory, Neighborhood Road, Kingston, New York 12401.