IBM

Y20-0450-0

SALES and SYSTEMS GUIDE

TSS/360 TIME SHARING COMPENDIUM

This compendium, intended as a foundation for knowledge on TSS/360, consists of a brief introduction followed by a reprint of the following papers:

Lett, A.S., and Konigsford, W.L. _TSS/360: A Time Shared Operating System_
Martinson, J.R. _Utilization of Virtual Memory in Time Sharing System/360_
McKeehan, J.B. _An Analysis of the TSS/360 Command System II_
Johnson, O.W., and Martinson, J.R. _Virtual Memory in Time Sharing System/360_
Lett, A.S. _The Approach to Data Management in Time Sharing System/360_

RESTRICTED DISTRIBUTION

RESTRICTED DISTRIBUTION: This publication is intended for use by IBM personnel only and may not be made available to others without the approval of local IBM management.

INTRODUCTION

The time-shared operating system for the IBM System/360 Model 67, TSS/360 has a number of features that distinguish it from other systems, both conventional and interactive. When considering its suitability, you should have a basic understanding of time=sharing systems in general and of TSS/360 in particular. Such an understanding will help you develop a true appreciation of TSS/360.

To this end, you should be aware of the wide range of published material about this system. You should also be aware of the wide range of experience with TSS/360.

To provide a foundation of knowledge upon which you can build, this Compendium includes material designed to promote a basic understanding of the most important concepts in TSS/360. This basic material should be supplemented by reading the full range of publications devoted to the system, in particular:

- _Concepts and Facilities_ (C28-2003)
- _System Logic Summary_ (Y28-2009)
- _Command System Users Guide_ (C28-2001)
- _Introducing TSS/360: A Primer for FORTRAN Users_ (C28-2048)
- _Assembler Programmer's Guide_ (C28-2032)

A summary of much of the information in the latter four publications may be found in the TSS/360 _Quick Guide for Users_ (X28-6400).

In studying the accompanying material, particular attention should be paid to the following features of the system:

- Sharing
  - Code
    - Reentrant for efficient use of main storage
    - Saves loading time
  - Data
    - Saves external storage
    - Provides large shared data base capability
- Foreground-initiated background
  - EXECUTE/BACK commands
  - Converts conversational task to nonconversational
- Command system
  - User profiles
  - Permits user to define his own commands
  - Message handling
- Delta data sets
  - Installation flexibility
  - Addition of function
- Table-driven scheduler
  - Tailoring of system to meet installation needs
  - Flexible control of system resources
- Virtual memory
  - 24-bit addressing provides 16 million bytes
  - 32-bit addressing provides 4 billion bytes

Eliminates overlay problem
Eliminates main-storage fragmentation

- Monitoring and debugging aids
  - Time Sharing Support System (TSSS)
    - Online system programmer's maintenance tool
    - Dynamic debugging of system programs
  - Program Control System (PCS)
    - Dynamic object-time debugging tool
- Hardware serviceability aids
  - On-Line Test System (OLTS)
    - Hardware diagnostic tool
    - Does not interfere with other users
  - Virtual Memory Environment Recording Edit and Print (VMEREP)
    - Formatted recovery of error recordings to terminal
- Data management
  - Solves large-data-base problems
  - Dynamic storage allocation Dynamic storage allocation

GENERAL SYSTEM DESCRIPTION

Time sharing can be described as the concurrent use of the resources of a general-purpose computing system by a large number of users. It is a logical extension of the growth in sophistication and scope of the computing environment since its beginning nearly two decades ago. In particular, TSS/360 serves as a logical extension for the problem-solving needs that gave rise to the IBM System/360 hardware and to IBM System/360 Operating System.

In common with other third-generation multiprogramming systems, the accent in TSS/360 is on "resource sharing"; a general-purpose time-sharing system must be designed to share main storage, channel facilities, and direct access (disk and drum) file space among a large number of users. Strictly speaking, a single CPU does not share time; it operates on only one task at any moment.

Unlike the goal of a batch multiprogramming system, which is to maximize throughput, the goal of a time-sharing system is to make it easier to use a computer while maintaining a very high degree of utilization of the system resources.

To make TSS/360 easy to use, the following features are included:

Remote conversational terminals
An online command system
Conversational language processors
A conversational program control system
Dynamic execution-time program linking
A one-level store concept
Protection and sharing for data and programs
Large online storage for libraries of programs and data

To make efficient use of computer resources, TSS/360 employs:

System scheduled multiprogramming
Dynamic program relocation

Partition capabilities

For further information see:

Appendix A -- Lett, A. S., and Konigsford, W. L. _TSS/360: A Time-Shared Operating System_, pp. 15-16.

SHARING

Two types of sharing are supported in TSS/360: data and code. Data sets are shared through pointers in the system catalog (of which each user's catalog is a part). The creator, or "owner", of a data set may permit selective sharing by specified users or universal sharing by all users; furthermore, he may permit read-only or read/write access, again either selectively or universally, to his data set.

The sharing of code is accomplished through the segment and page tables that support the system's dynamic address translation facility. To take full advantage of code sharing, programs must be written in a fashion that separates read-only, address-free instructions and data from variable, address-dependent instructions and data.

For further information see:

Appendix A -- Lett, A. S. and Konigsford, W. L. _TSS/360: A Time-Shared Operating System_, pp. 26-28.

Appendix B -- Martinson, J. R. _Utilization of Virtual Memory in Time Sharing System/360_, pp. 4-7.

FOREGROUND-INITIATED BACKGROUND

After initiating a conversational task in TSS/360, three courses of action are available to the user: (1) to complete the task conversationally, (2) to initiate one or more independent, nonconversational tasks (as part of his original task), or (3) to complete his original task in nonconversational mode. The second alternative may be accomplished by using the EXECUTE command one or more times during a terminal session; each EXECUTE command will cause a prestored task to be initiated and handled as an independent nonconversational task. The third may be accomplished by issuing a BACK command during the conversational task; this will cause a prestored continuation (from the point at which the BACK command was issued) of the task to be handled as an independent nonconversational task.

For further information see:

IBM System/360 Time Sharing System: Command System Users Guide (C28-2001)

COMMAND SYSTEM

TSS/360 provides commands for managing tasks, managing data, using language processors, controlling program execution, and tailoring the command system to the installation's and/or user's needs. The system-supplied command names may be changed or abbreviated at the user's discretion, and the user may establish whatever default values for command parameters he desires.

The keynote of the TSS/360 Command System, therefore, is flexibility. The user has the option, and ability, of remaking the command system to fit his needs, even to the extent that his becomes a different system. An additional feature is a message-handling capability that permits the user to filter out system-supplied messages and/or to insert his own.

For further information see:

> Appendix C -- McKeehan, J. B. An Analysis of the TSS/360 Command System II.
>
> Appendix A -- Lett, A. S., and Konigsford, W. L. TSS/360: A Time-Shared Operating System, pp. 26-28.

DELTA DATA SETS

Changes to the system can be tested to ascertain their effect, without permanently updating the system, by means of delta data sets. These changes may be either IBM-supplied maintenance packages or user-initiated modifications. The changes, which remain in effect from startup until shutdown, are contained in delta data sets and must appear on one private volume (delta data set volume).

During the startup procedure, delta data sets are searched for initial virtual memory, resident supervisor, and resident support system control sections; then, system data sets on the IPL volume are searched. After the query "DELTA DATA SETS?", the order of search for these data sets is specified by the operator. There are three loadlists: one for the resident supervisor, one for initial virtual memory, and one for the resident support system. Only those control sections identified by name in a loadlist are loaded. Since these loadlists are control sections and are located by STARTUP in the same manner in which any other control section is located, an alternate loadlist from that contained in the system data sets can be included in a delta data set. Such a loadlist might be used to add control sections to initial virtual memory, to the resident supervisor, or to the resident support system. Tasks executed during the current session will run with all such modifications to initial virtual memory, the resident supervisor, and the resident support system.

For further information see:

> IBM System/360 Time Sharing System: System Generation and Maintenance (C28-2010).

TABLE-DRIVEN SCHEDULER

CPU time in TSS/360 is scheduled by means of a system table that permanently resides in main storage as a system control block. The supervisor refers to this table when scheduling tasks, both at task initiation and during execution of the task, to determine when next to schedule the task and for what amount of CPU time. The scheduling table may contain as many as 256 scheduling levels, called STE.

The STE given to a task initially reflects the user priority and task type (that is, conversational or batch). Once a task has been initiated, the supervisor may move a task to another level -- for example, when a task is switched from conversational to batch mode. Levels are also adjusted for tasks that have been determined to be I/O-bound or execute-bound.

A programmer may also affect the scheduling of a task. The PULSE macro-instruction changes the STE level of a task to another preset "pulse-level" that is associated with the current level. The CHANGE macro-instruction changes the task's level to a specified level. The PRESENT macro-instruction displays the current schedule level of a task.

For further information see:

Appendix A -- Lett, A. S., and Konigsford, W. L. <u>TSS/360 A Time-Shared Operating System</u>, pp. 19-20.

VIRTUAL MEMORY

The dynamic-address-translation feature of the Model 67 is utilized in TSS/360 to provide a virtual memory, or virtual address space, equal in size to the logical addressing capability of the system. When used with the 32-bit addressing structure, this provides each user of the system with 4 billion bytes of addressablee storage; when used with 24-bit addressing, 16 million bytes. To the user, this means freedom from overlay considerations and improved throughput because it provides a solution to the problem of main-storage fragmentation. The translation process, which utilizes a two-level page table structure and an associative register array, is completely automatic and the user need not know where his program is in main storage during execution.

For further information see:

Appendix D -- Johnson, O. W., and Martinson, J. R. <u>Virtual Memory in Time Sharing System/360</u>.

Appendix B -- Martinson, J. R. <u>Utilization of Virtual Memory in Time Sharing System/360</u>.

MONITORING AND DEBUGGING AIDS

From an external view, the two TSS/360 monitoring and debugging aids -- Time Sharing Support System (TSSS) and Program Control System (PCS) -- are very similar. Indeed, many of their commands have the same names and provide similar functions. However, TSSS is a system monitoring and debugging aid, while PCS is a user monitoring and debugging aid.

TSSS actually comprises a resident support system (RSS), which is used with the supervisor, and a virtual support system (VSS), which is used with a user's virtual memory (by a system programmer, not by the user). By implanting dynamic statements (with the AT command) at various points in the supervisor or a virtual memory and coupling them with conditional (IF) statements, patches can be tested, dumps or displays taken while execution is going on, and bugs corrected -- all while the system is operating.

PCS provides similar features to each user so that he can perform similar functions within his own program. While PCS has certain limited system applications, the power of TSSS makes the use of PCS unnecessary in this case.

For further information see:

IBM System/360 Time Sharing System: Time Sharing Support System (C28-2006).

IBM System/360 Time Sharing System: Command System Users Guide (C28-2001).

HARDWARE SERVICEABILITY AIDS

TSS/360 provides remote as well as on-site hardware serviceability facilities for the customer engineer. The remote capability is significant that a customer engineer may run the hardware serviceability aids from any physical location, provided he has terminal access to TSS/360. As a significant percentage of failures result in a "no-trouble-found" situation, this remote capability allows the customer engineer to verify that he can recreate the failure before going on-site to accomplish the repair. In addition, the on-site customer engineer who may not be trained on a particular I/O device can call for assistance from a remote location having terminal access to his installation.

The On-Line Test System (OLTS) is a set of programs provided for testing I/O devices in an operating TSS/360 environment. Since OLTS runs in a multiprogrammed, time-sliced, relocated environment, it does not require a physically partitioned subsystem. This greatly increases the availability of the total system by eliminating downtime caused by partitioning of unique components indispensable for the operation of the system.

OLTS is initiated by a customer engineer from a terminal after an appropriate log-on procedure. The customer engineer communicates through this terminal with the OLTS control program to specify the device to be tested, the test programs to be run, and options to be exercised. Through these means, he can obtain information about failure symptoms, perform device adjustments or preventive maintenance, and subsequently verify the successful repair of the device.

The core of OLTS is the control program with which the customer engineer interfaces. It allows him to obtain device allocation from an operational TSS/360, interprets and carries out his commands, sequences the test programs, and provides various other utility functions.

The Virtual Memory Environment Recording Edit and Print (VMEREP) program employs the preservation-recording facility of TSS/360. VMEREP retrieves the error-incident information recorded by TSS/360 and prints it in an English-language format useful to the customer engineer. After an appropriate log-on procedure, the customer engineer may request that the error-incident recording be printed at a terminal or scheduled for output at a high-speed printing device.

For further information see:

> _IBM System/360 Time Sharing System: On-Line Test Control System_ (Y28-2042).

> _IBM System/360 Time Sharing System: System Logic Summary_ (Y28-2009).

DATA MANAGEMENT

Data Management in TSS/360 is supported by three "virtual" access methods. They are termed "virtual" because, like virtual memory, they utilize only one physical block size -- the 4096-byte page. Each of the methods is physical-device independent; data set management is performed only in virtual memory.

The virtual sequential access method (VSAM) permits records to be retrieved in the order in which they were created. The virtual index sequential access method (VISAM) permits records to be retrieved by key. The virtual partitioned access method (VPAM) permits VSAM and VISAM data sets to be made members of a single data set in which each of the members can be processed independently of the others.

For further information see:

> Appendix E -- Lett, A. S. _The Approach to Data Management in Time Sharing System/360._

> Appendix A -- Lett, A. S., and Konigsford, W. L. _TSS/360: A Time-Shared Operating System_, pp. 22-24.

APPENDIX A

TSS/360: A time-shared operating system

by ALEXANDER S. LETT and WILLIAM L. KONIGSFORD

International Business Machines Corporation
Yorktown Heights, New York

INTRODUCTION

Experience with TSS/360 design, development, and application has been varied and interesting. For example, as we began putting the initial system together, significant performance problems were observed that had not been predicted by the earlier simulation efforts. These problems had not been anticipated because the paging characteristics assumed in the model development were significantly better than the actual system characteristics.

Measurements and analysis soon indicated that one significant problem was in the organization of the dynamic loader tables. This was overcome by splitting these tables into functional sub-tables, which greatly reduced paging during the loading process.

However, the paging problem was widespread. In most cases, the size of the actual code was two to three times the size expected by the model. In addition to the adverse effects on the available core space, this caused the paging input/output traffic to be significantly larger than expected, the level of possible multiprogramming to be smaller than expected, and the number of tasks wholly contained on the paging drum to be fewer than expected.

During the multi-terminal testing phase of TSS/360, another significant paging problem was discovered. Core-space control was based upon dynamically limiting the number of tasks active in core to the maximum that their estimated core usage would allow. A greater number of tasks would cause a high rate of unproductive paging within the system; a lesser number would not fully utilize the system facilities. The core-space control was not functioning properly due to bugs. A temporary solution was to restrict the number of active tasks in core to a fixed number. This reduced the performance problem but, when the same test cases were later run under the correctly operating dynamic algorithm for core-space control, the dynamic algorithm was found to be consistently more effective than the static algorithm had been. This generally valid conclusion has been demonstrated in all area of TSS/360.

Since the initial release of TSS/360 in October 1967, performance has improved significantly with each subsequent release. The initial emphasis was on building a stable system, followed by extensive measurement-and-analysis efforts to identify potential system modifications. Then, through comparatively small changes in coding and resource management algorithms, the system performance was significantly improved.

From the material presented in this paper, we feel that several conclusions—which are supported by our operating experience—can be drawn:

- Paging is a sound concept. As expected, it is a direct solution to the dynamic-core-storage-allocation problem.
- Paging also allows for a hierarchy of auxiliary storage which, in the case of TSS/360, involves high-speed drum and slower-speed but larger-capacity disk files. A larger number of users can be supported economically on a system by subdividing each user's space requirements between drum and disk storage. From experience, sound algorithms can be developed for management of this storage, which is critical to the overall performance of the system.
- A data set access method based on page-size fixed block images has the same simplicity of implementation and elegance of application as in the core-storage situation.
- In the improved command system, we have found that a highly adaptive, open-ended system is not only more valuable to terminal users, but simpler to implement.
- The best strategies for resource allocation are those that address the allocation of all system resources in an integrated way, rather than optimizing specific sub-portions. In general, the simple round-robin strategy is fairly good, but

This paper was presented at the 1968 Fall Joint Computer Conference and appears in Volume 32 of the Conference Proceedings.
© 1968 by the American Federation of Information Processing Societies, New York, New York, 10017.

the need to emphasize certain characteristics (such as response time to conversational requests) requires the separation of resource requests by priority.

It is the purpose of this paper to highlight the key elements of TSS/360—control system organization, user services, and task structure—in order to describe and explain the design of a time-shared operating system.

Control system organization

There are many possible resolutions to the questions concerning the division of functions within a control program, the interfaces between portions of the control program, and the decision as to which portions are to be resident in main storage and which nonresident.

In the design of TSS/360, the historical examples of the MIT Compatable Time Sharing System[1] and the IBM Time Sharing Monitor system[2] led towards the concept of a small, fully resident monitor whose primary function would be to create a multiprocessing, multiprogramming enviornment.

In TSS/360, this monitor is called the Resident Supervisor. The Resident Supervisor is interrupt driven, and is responsible for controlling the real resources of the system and for performing services in response to requests originating from tasks. A task represents the environment in which a user's processing is performed. There is one task for each conversational user of the system. A fundamental design decision was to provide within each task the facilities of a full operating system. Figure 1 depicts this overall system structure.

Task processing is always performed in relocation mode with the dynamic-address-translation feature activated. Tasks are therefore said to operate in virtual memory.

The Resident Supervisor, on the other hand, does not use dynamic address translation—that is, instructions within the Resident Supervisor have main storage addresses, not logical addresses, as operands. The decision to make the Resident Supervisor operate in the nonrelocation mode was based upon the efficiency resulting from eliminating dynamic-address-translation overhead and upon the increased protection resulting from the fact that no location within the Resident Supervisor can be addressed by a program operating in virtual memory. Resident Supervisor routines, however, are capable of addressing all of main storage and of executing all of the instructions in the System/360 instruction set.

Another basic TSS/360 design decision was to have tasks be interrupt driven like the Resident Supervisor. It was felt that this structure provided the maximum of flexibility in task development. Accordingly, task-control structure is in many ways analogous to the control structure of the Resident Supervisor.

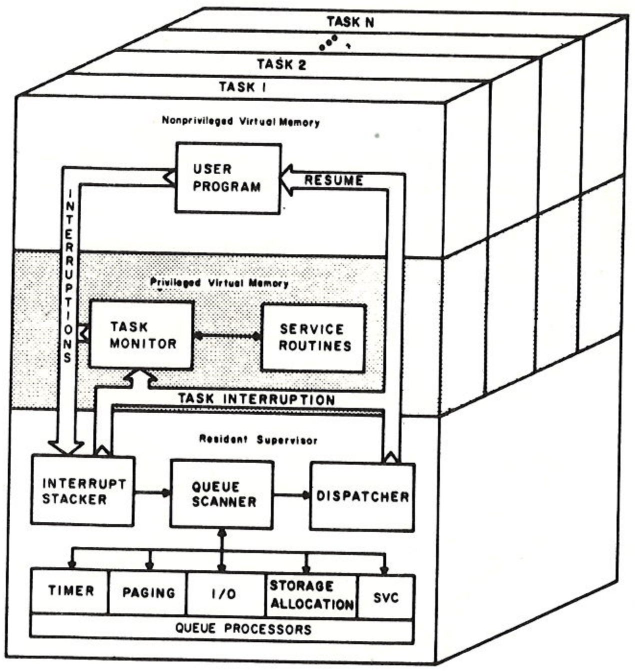


FIGURE 1—TSS/360 program structure

In order to provide a wide variety of control program services, while at the same time protecting user tasks from each other, task virtual memory routines are divided into two classes:

- privileged routines, which operate in the privileged state;
- nonprivileged routines, which operate in the nonprivileged, or user, state.

As the term implies, routines operating in the priviliged state are authorized access to many supervisor services denied to routines operating in the user state. In this way, most parameter validation and other protection checking can be eliminated from the Resident Supervisor. In addition to decreasing the overall size of the Resident Supervisor, this arrangement allows supervisor services to be more general and powerful.

The way in which the privileged state is implemented is as follows:

- In a task's virtual memory, pages that are allocated to privileged routines (and their associated tables and work areas) are assigned a storage protection key that differs from that assigned to user programs. This will cause a storage-protect interruption if the privileged part of a task's virtual memory is addressed by a user program. Privileged routines, on the other hand, can address all of the task's virtual memory.

- The dynamic loader service routine will not treat modules from a user's library as privileged routines. Thus, an ordinary user cannot cause his own version of a system routine to be loaded and executed as a privileged routine, a facility available to the systems programmer.
- A user program normally requests system services through instructions whose execution cause an interrupt to the Resident Supervisor. (In system/360 such interruptions are termed supervisor calls.) In response to the supervisor call, the Resident Supervisor, by manipulating CPU status, creates a task interruption to invoke a privileged system services routine. The privileged routine can then determine if the user's request is valid. If it is, the privileged routine may then invoke other TSS/360 supervisor calls while in the process of performing services. If the request is not valid, it will be rejected, thus preventing a nonprivileged routine from causing incorrect system operation. The reason for communicating between nonprivileged and privileged state via the Resident Supervisor is that only the Resident Supervisor can execute the instruction that alters a task's protection key and, therefore, its state.

The privileged system service routines constitute the bulk of the TSS/360 operating system. These routines are either shared by all tasks or are located in independent service tasks. Printers, for example, are serially shareable and thus are serviced through an independent task. On the other hand, the dynamic loader provides service to each task and is therefore shared in parallel.

System control elements

The Resident Supervisor is primarily composed of an interrupt stacker, a queue scanner, several processors, a number of error handling and service subroutines, a dispatcher, and the tables that form the system's data base.

Entry into the Resident Supervisor is via an interruption. Some interruptions are processed immediately either because of their urgency (e.g., interruptions denoting CPU malfunctions) or for efficiency (e.g., interruptions which require a change of task state).

For most interruptions, however, the interrupt stacker builds a record called a generalized queue entry (GQE), into which a description of the interruption is placed. This GQE is then placed upon an appropriate queue. A GQE is a standard control block used throughout the Resident Supervisor to contain a description of the work to be done by a device or facility that is controlled by the Resident Supervisor. Quite frequently, one control block may belong to several queues and contain forward and backward pointers to each of them. In processing these multi-threaded lists, the Resident Supervisor becomes, in effect, a list processor.

Interruptions are disabled during processing in the interrupt stacker. However, in contrast to many systems, the Resident Supervisor generally executes with interruptions enabled to facilitate processing of interruption queues, on a priority basis, without regard to sequence of arrival. When the interrupt stacker completes processing, it generally exits to the queue scanner.

Every system needs some facility for sequencing the work to be performed by the control program. In systems which operate with interruptions disabled, the hardware priority-interruption system provides this function for the interrupt-handling routines, and some other control-program routine provides a similar function for the system's resource-allocation routines. Within TSS/360, these two functions have been combined into one centralized queue scanner and a scan table. Each system queue is anchored in the scan table.

Because the queue scanner is a central facility within the Resident Supervisor, it must operate efficiently if the Resident Supervisor is to operate efficiently. To achieve this efficiency, the queue entries in the scan table are organized to minimize the number of entries that must be inspected when the scanner is searching for work. Moverover, the organization of scan-table entries reflects an awareness of the possible interactions among queues so that, for example, an exit is not made to a processor only to find that a needed facility (such as an I/O path) has been allocated to some other request.

When the queue scanner finds work that can be done, it passes control to the appropriate processor; when it determines that there is no currently available supervisor work, control is transferred to the scheduler and dispatcher.

TSS/360 was designed for a generalized multiprocessing environment in which multiple CPUs may be simultaneously executing the single copy of the Resident Supervisor. To facilitate multiprocessing, it was necessary to define a number of programmed interlock flags to prevent unwanted recursion and logical race conditions. In general, TSS/360 used the approach of defining a small number of interlocks, each covering a wide scope. These interlocks generally guard entrance to the queue processors and to the major system data bases.

The purpose in minimizing the number of interlocks is two fold:

- First, placing interlocks at the entrance to the queue processors tends to prevent a CPU from entering a path of logic only to soon be forced to await the resetting of an interlock. When the queue scanner finds an interlocked queue processor, it simply bypasses inspecting that queue and proceeds to the next entry in the scan table.

- Second, in a multiprocessing situation, it is desirable to permit one CPU to perform error-recovery procedures whenever another CPU encounters a processor or storage unit error. Because all processors use the single copy of the Resident Supervisor, it may be necessary for the recovery CPU to reset programmed interlocks initially set by the malfunctioning CPU. This means that the recovery CPU must be aware of the reason why the interlock was set. The fewer the system interlocks, the simpler the recovery procedures can be.

In general, a queue processor locks its associated queue upon entry and unlocks the queue as soon as the processor has dequeued a GQE for processing. In certain cases a queue processor may lock a queue until some specific future event or condition has occurred. Each scan table entry has indicators reserved for such use.

TSS/360 has adopted a policy of concentrating the physical locations of the interlock flags in an orderly fashion within a very few key system tables. This has proved to be a valuable aid to the development programmers, who can determine the status of the Resident Supervisor by inspecting or displaying these system tables.

The following is a brief description of the major processors within the Resident Supervisor:

- *Task Core Allocation:* Controls the overall core storage space; it processes requests for space allocation and responds with the location assignments
- *Auxiliary Storage Allocation:* Controls auxiliary-storage space; it processes requests for drum and disk page-space allocation.
- *Page Drum Request:* Processes input or output requests for the auxiliary paging drum. Because of the unique mechanical characteristics of the drum (several pages per track with instantaneous switching), the requests are sorted by angular position to maximize throughput.
- *Page Drum Interrupt:* Processes interruptions that are the result of paging drum input/output operations. This processor will attempt to keep the drum I/O channel busy by adding drum requests to an active drum I/O channel program. It calls the page-posting routine to process the results and releases core space when appropriate.
- *I/O Device Request:* Processes requests for I/O operations to devices *other* than the auxiliary storage drum; it first determines, by calling the path-finding subroutine, if a free path to the requested I/O device is available and, if possible, reserves a path.

  I/O device requests are either disk paging requests or other I/O requests. For disk paging requests, a subprocessor is called to convert the request into an I/O channel program. For other I/O requests, a request control block chained from the queue entry already contains the I/O channel program. This I/O program is normally created by the task requesting the I/O. The I/O operation is started by the request processor, which returns to the queue scanner.
- *Channel Interrupt:* Processes input/output interruptions that originate in other than the paging drum. It determines if the interruption is synchronous or asynchronous by verifying if a request on the corresponding device-request queue had initiated the operation. If the interrupt is synchronous, various processing is performed. If the interrupt is asynchronous, an interrupt entry is queued for the task currently associated with the device. If no task is currently associated with the device, and it is a terminal, the channel interrupt processor will call a routine to create a new task that will then be dispatched. The newly created task will begin execution of the appropriate task-initialization routines in response to its initial interrupt.
- *Timer Interrupt:* Processes timer interruptions; it determines if a task has reached the end of its time-slice or whether a task-specified time interval has elapsed. At time-slice end, various processing is performed. For task-specified intervals, a task-simulated timer-interrupt entry is queued for the task.

TSS/360 error recovery and retry procedures are designed to dynamically correct errors or to minimize the effect of errors on the system as a whole. Although the specific recovery procedures differ for each type of error, the general approach to recovery is the same. Failing operations are retried where possible, failing hardware devices (e.g., a CPU or I/O device) are checked and intermittent failures retried. Where an operation cannot be retried at all or is retried without success, a "hard" failure is recognized and fault localization, to the component level, is invoked. The failing element or device is removed from the system in an orderly manner, so that only the affected tasks are disrupted. An environment record is generated for later analysis by service personnel and the system continues operation. It is only as a last resort, when recovery is not possible and when removal of the failing component would render the system inoperative, that the system is shut down.

In addition to the queue scanner's scan table, the Resident Supervisor contains data bases to describe task status and to describe I/O path status.

Each task has associated with it a control table that is separated into portions. The first portion is needed for scheduling and control purposes, so it is kept continuously resident in main storage. The second portion contains the task's relocation tables that must be in main storage during a task's time-slice, but not necessarily between a task's time-slices.

To allow a user's program to be highly device-independent and to allow the Resident Supervisor to remain relatively insensitive to dynamic changes in system configuration, TSS/360 users normally employ device-class codes that describe a device as a member of a class of like devices. Furthermore, the TSS/360 access methods employ symbolic addresses to designate devices.

The Resident Supervisor uses a group of tables, called pathfinding tables, to translate a symbolic device address into a hardware address that specifies a path through a channel control unit, channel, and device control unit to the device. The supervisor-maintained pathfinding tables are used to determine if a device is busy instead of attempting to physically address the device. In a typical environment, it is expected that there will be multiple paths to most devices. In such a situation, the efficiency of I/O processing will be increased by reducing the number of "busy" or "unavailable" conditions encountered during an attempt to initiate an I/O operation. The use of common pathfinding tables also assists in synchronizing I/O processing in a multiprocessing environment, because an I/O interruption may be accepted by any available CPU, not just the CPU that initiated that operation.

In retrospect, the design of the Resident Supervisor has proved to be sound and remains, in outline, essentially as initially described in 1965.[3] Experience has shown that it is nearly impossible to predefine an optimal overall system. A significant amount of tuning of resource-control algorithms and processing procedures must be expected. We have found that the best method to do this tuning is by modification and measurement of the running system.

Task control elements

TSS/360 includes a scheduling algorithm for determining the sequence of allocation of CPU time to competing tasks. As implemented initially, the scheduling algorithm divided tasks into conversational and nonconversational groups.

The original algorithm followed a round-robin schedule for the active tasks (those not waiting for the completion of some event, such as terminal input). Conversational tasks were scheduled for dispatch in consecutive order to the end of the list. At this point, a test was made to determine if an installation-specified real-time interval had elapsed. If not, the system devoted the remainder of the interval to the round-robin execution of the nonconversational tasks. If the interval had been exceeded, the system went back to redispatch the first active conversational task.

As a result of system experience, this algorithm was modified. All active conversational tasks are now dispatched in round-robin fashion until no further active conversational tasks are available. Then the system begins to dispatch active nonconversational tasks, but with provision for pre-emption whenever a conversational task becomes active. Instead of round-robin execution of the nonconversational tasks, the system tends to run to completion as many nonconversational tasks as can be effectively multiprogrammed within the available core resource. This modification was incorporated because round-robin scheduling for the nonconversational tasks served no useful purpose and reduced system throughput by causing the system to do additional paging in switching resources.

The scheduling algorithm outlined above is not considered to be the optimum for general time-sharing operation in any specific customer's installation. Experience with scheduling algorithms and their effect upon the system dictated the need to provide a flexible facility for modifying the task-scheduling algorithm. TSS/360 is adding this facility, called the table-driven scheduler, in which table entries are made to define sequences of states and attributes that a task can assume. When created, each task is assigned an initial table entry in which specific parameters explicitly state:

- the relative priority of every task associated with that table entry
- whether such tasks may be interrupted by a higher priority task
- the time-slice quantum to be allocated to the task
- the maximum core space to be allocated to the task
- other parameters concerned with the action to be taken when execution of a task is suspended.

Execution of a task can be suspended for reasons such as time-slice end, terminal-wait condition, or excessive paging. Associated with each of these conditions is a value specifying the table entry to be assumed by the task on the occurrence of that condition.

The collection of schedule table entries, which can be prepared at each installation, specify the scheduling algorithm to be followed by the system. The table entries can range from extremely simple ones that simulate a round-robin queue, through exponentially related algorithms, to complex time-and-priority algorithms.

The allocation of the CPU resources to tasks, to best carry out the sequence selected by the scheduling algorithm, is controlled by the dispatcher. The dispatcher first determines if a new task can be placed into execution. This is determined by comparing an estimate of the core pages a task is expected to require during its next time-slice with the number of unreserved and available core pages. The estimate of a task's page requirements is based on its activity in the preceding time-slice. If enough core pages are available, the count of available core pages is reduced by the estimated number and the task is prepared for execution. This dynamic control of the number of tasks allowed to concurrently execute in core storage is vital to avoid overloading a paging system such as TSS/360.

A modification has been made to the dispatcher to dynamically detect CPU-bound tasks. When more than one task is ready for immediate execution, non-CPU-bound tasks are dispatched before CPU-bound tasks. Through this strategy, the system dynamically maximizes its probability of multiprogramming (overlapping I/O with computing).

When a task is selected for immediate CPU execution, a task-interrupt-control routine in entered. The need for a task-interruption mechanism arises because the Resident Supervisor processes requests for system services in a logically independent fashion, that is, the Resident Supervisor may be concurrently performing several services for a task. There is no way to forecast the order or time of completion of processing of each of these services.

Therefore, for a task to operate asynchronously with respect to the completion of system services, a task-interruption mechanism has been created that is analogous to the hardware-interruption mechanism that allows the Resident Supervisor to operate asynchronously with respect to the real computer system. Operation of task interruptions is similar to hardware interruptions. The major difference is that the hardware interruptions convey a change in the status of the entire system to the Resident Supervisor, while the task intertuptions represent a change in status of only that portion of the system currently allocated to the task being interrupted.

A task interruption is requested by a Resident Supervisor routine when it discovers an event, such as I/O completion, whose further processing is a task's responsibility. However, a task is not always prepared to receive an interruption; further, the task for which the interruption is destined may not be the next task to be dispatched. So there is a software queueing-and-masking facility that is analogous to the hardware facility.

Before control is given to a task, the dispatcher transfers control to the task-interrupt-control routine, which checks the task's interruption queues for unmasked pending interruptions. If none is found, control is given to the task at the location saved in its control table.

If pending interruption is found, the task-interruption-control routine changes the location pointer to point to an appropriate interruption processor of the Task Monitor. Now, control will go to the interruption processor. This action of influencing the dispatcher's transfer of control is called a task interruption.

The Task Monitor consists of a group of privileged service programs that receive and process task interruptions on a priority basis via queueing, scanning, and dispatching mechanisms analogous to those of the Resident Supervisor. The Task Monitor may thus be considered a task-interruption handler, whereas the Resident Supervisor is a hardware-interruption handler.

The Task Monitor performs these major functions:

- Provides an interface with the Resident Supervisor for receiving and analyzing task-oriented interruptions.
- Provides linkage to required service routines or user routines, either by immediate dispatching or by queueing the interruption for later dispatching in a priority sequence.
- Maintains the integrity of the task and service routines that are dispatched, primarily through save-area management.

The Task Monitor is designed to provide for flexible handling across a wide range of interruptions. Thus, it provides an ability to dynamically specify task-interruption-handling routines and to dynamically arm, disarm, and change the relative priority of these routines. As with the queue scanner of the Resident Supervisor, provision has been made to use this generalized process in an efficient manner.

User services

Because TSS/360 is a comprehensive operating system, it offers a wide variety of user services[4], such as:

- Command system
- Program control subsystem
- System programmer support system
- Catalog management
- Page-oriented data management
- Magnetic-tape and unit-record data management
- Dynamic program loading
- Virtual memory allocation
- External storage allocation

- Resource control and accounting
- Task-interruption control
- Language processors

From this list, we have chosen to describe in this section the command system, the page-oriented data-management services, and the dynamic-program-loading services. Not only does each of these represent a key aspect of TSS/360, but each has relevance to problems of general interest.

Command system

The command system is the principal interface between a time-sharing system and its users. Therefore, it has a position of special importance in TSS/360.

Initially, the TSS/360 project attempted to define a set of commands that would be satisfactory for all users. The result was a rigid set of commands that completely satisfied no one. This experience led to the conclusion that it is better to implement a command system than a command language.

As a result, TSS/360 now contains a flexibile command system that is delivered with a set of simple commands that can either be employed as is or be completely replaced and expanded in a straightforward fashion. This approach allows each installation and, more important, each user at an installation to customize the system-user interface to his own needs.

In TSS/360, the syntax of the command system has been separated from the semantic content of command statements. This regularization of syntax and structure has resulted in a simpler implementation utilizing a single, centralized command analyzer and execution facility.

The command-system syntax is simple and natural. Each command consists of an operation name, which is usually followed by one or more operands. As supplied with the system, the delimiting character for the operation name is a blank or tab; the delimiter between operands is a comma; the delimiter between commands is either a semicolon or the end of a line of input; and the line-continuation flag is a hyphen entered as the last nonblank character of a line.

When an individual enters his commands conversationally, he is told of the actions taken by the system in response to each command and, when necessary, he is prompted for additional non-defaultable information needed to complete an action, is informed of errors (if his command entry is either incomplete or incorrect), and is told of the options he may exercise in response to an error. Special care has been taken to make the types of options consistent for all commands. Nothing, for example, could be more frustrating to a user than to be required to resubmit an operand with delimiters in one situation and without delimiters in another.

Each user can establish his own spellings, abbreviations, or operation names for commands through a SYNONYM facility. Use of this facility sets up one or more equivalences for the original name but does not destroy it.

Any command operand may be entered either by position or by keyword. Keywords may appear in any order and have the general form KEYWORD = value, where KEYWORD is the name of the operand and "value" is the actual value of the operand. For each command operand, the user may select the form that is most convenient for him. A keyword has a global meaning since it is associated with the value to be passed, not with the particular command invoked. Therefore, the SYNONYM facility, available for command operation names, is also available for keywords. In contrast to many other systems, almost every command operand has a default value. Moreover, the user need not accept rigid default values for operands, for he can easily override those supplied with the system. For example, a standard default for the FORTRAN compiler might be to produce an object code listing. Any TSS/360 user can individually change this default so that, in his case, the language processor will not produce an object listing unless he specifically requests it.

TSS/360 maintains a special prototype data set that is copied into the user's library when he is initially joined to the system. This data set, called a user profile, contains three tables: the first specifies the initial default values for command operands; the second contains his character-translation list (to allow redefinition of printing characters and control characters); and the third contains command operation names and equivalences. The user can modify any of the entries in these three dictionaries, which, in conjunction with the command system, define his command language.

The command system includes as a fundamental feature a command procedure facility, which permits the user to create a stored procedure comprising commands and logical statements that control the flow of command execution. Invocation of a command procedure is identical to invocation of a system-supplied command. The command statement consists of the procedure name followed by a series of parameters, whose values are inserted by the command system at the proper points in the procedure. The resultant statements will be interpreted as though they had originated in the input stream. For maximum power, command procedures can be nested and/or recursive. When defining a procedure, a user can utilize the facilities of the TSS/360 text editor. Once defined, a procedure may be edited, shared, copied, etc., as with any other file.

Another interesting feature of the command system is the use of "null commands." For example, immediately after a user has signed on the system but before control is returned to the terminal, TSS/360 automatically invokes a command procedure called ZLOGON. As initially supplied with TSS/360, ZLOGON is a "null" command—it does nothing. However, the individual may redefine the ZLOGON command procedure to perform functions to augment the initialization of his task. Thus, "null" commands are conceptually similar to the "user exits" frequently associated with general-purpose programs.

The command system also provides a facility for defining new command primitives. Efficiency can be enhanced through use of this "Built-in" facility as the command system can directly bypass much of the interpretive processing required in the expansion of command procedures.

Still another feature of the command system available to the user is the ability to augment system-message handling:

- He can request explanation of system messages or of key words in such a message; word explanations may continue to a number of levels.
- He can dynamically specify the classification of messages he is to receive; this filtering, or masking, capability provides different message-severity levels and message lengths.
- He can construct a personal message file that will be issued in lieu of the corresponding system-supplied messages.

The command system also provides a flexible system for handling attention interruptions that is quite useful. For instance, suppose a user has forgotten to identify a library that contains a subroutine required by his mainline program. When he receives a system diagnostic message, he can use the attention button to re-enter the command mode, define the library, and then resume processing at the point where the message was issued.

The program control subsystem of the command system is a powerful facility that permits a user to inspect and modify programs during execution. These dynamic control facilities eliminate the need for user-written debugging and control instructions that must be preplanned, coded into the user's programs, and then later removed.

The output from the TSS/360 language processors may optionally include a dictionary containing the values and other attributes associated with the symbols or variables used in the source program. Through the use of this dictionary, the program control subsystem can properly interpret debugging statements utilizing source program symbols and can properly format its input and output.

Even during the initial shakedown of TSS/360, there were many users who insisted upon using the system only because of the power associated with a dynamic execution-control system. This has made clear that an essential element of any interactive system must be a dynamic symbolic debugging and control facility.

Page-oriented data management

The access methods that support page-oriented data management in TSS/360 are called virtual access methods. The name "virtual" was given to these access methods to reflect the fact that they utilize only one physical block size—that of a page. The virtual access methods were specifically designed for a time-sharing environment and present a clear division between data set management and physical device management. Each of the three virtual access methods provides access and processing capability for a specific type of data set organization:

- Virtual sequential access method (VSAM)
- Virtual index sequential access method (VISAM)
- Virtual partitioned access method (VPAM)

In all three of these access methods, only data set management is performed in virtual memory; the construction and execution of channel programs and error recovery (i.e., physical-device management is performed by the Resident Supervisor. The direct-access volumes, on which TSS/360 virtual organization data sets are stored, are entirely formatted into fixed-length, page-sized data blocks. No key field is required. The record-overflow feature is utilized to allow data blocks to span tracks as required.

The page-sized block for data storage was selected for a number of reasons. For example, rotational delay is a significant factor in direct-access throughput, since it cannot be overlapped as mechanical-seek time can. Any block size significantly smaller than a page would be extremely wasteful of total direct-access capacity unless elaborate strategies were utilized to avoid rotational delay.

The need for a large block size is also apparent when the simultaneous direct-access activities of multiple users are considered. Due to conflicts in demands for access arms, a mechanical seek may frequently be required before accessing a data block. A larger block size makes better use of the total access cycle while, at the same time, reducing the frequency of access requests by each user.

The direct-access volume-packing efficiency is also quite high for page-sized blocks. First, the data-recording space is utilized at better than 90% of the theoretical capacity that could be obtained by the use of cylinder-length blocks. Second, the smallest external-storage allocation unit is a single page; hence, a large number of small data sets can be kept on one volume. Furthermore, large data sets need not be allocated physically contiguous external storage space. This contributes to higher volume packing efficiency by reducing external-storage space fragmentation.

The physical representation of a typical virtual sequential organization is shown in Figure 2. The specification of any virtual data set is contained within the data set's external page map, which is stored on the direct-access volume together with the data pages. There is one entry in the external page map for each page-sized block occupied by the data set. The content of an entry specifies the location of a block in external storage. The position of the entry within the external page map signifies the relationship of the associated block relative to the other blocks in the data set.

For the three-page data set shown in Figure 2, the external page map shows that the first data block is between the other two pages of the data set. This example emphasizes that block relationships in the data set are determined by the contents of the external page map rather than by their physical position within the volume. This concept allows the virtual access methods true device independence across the range of direct-access devices. That is, it is perfectly feasible for a data set to have physical records recorded on, say, the IBM 2311 Disk Storage Drive and the IBM 2314 Direct Access Storage Facility in any mixture. Furthermore, because information is referenced relative to the beginning of the data set and not by its location with respect to an external-storage device, it is entirely practical to move data sets (or portions of data sets) among a hierarchy of devices.

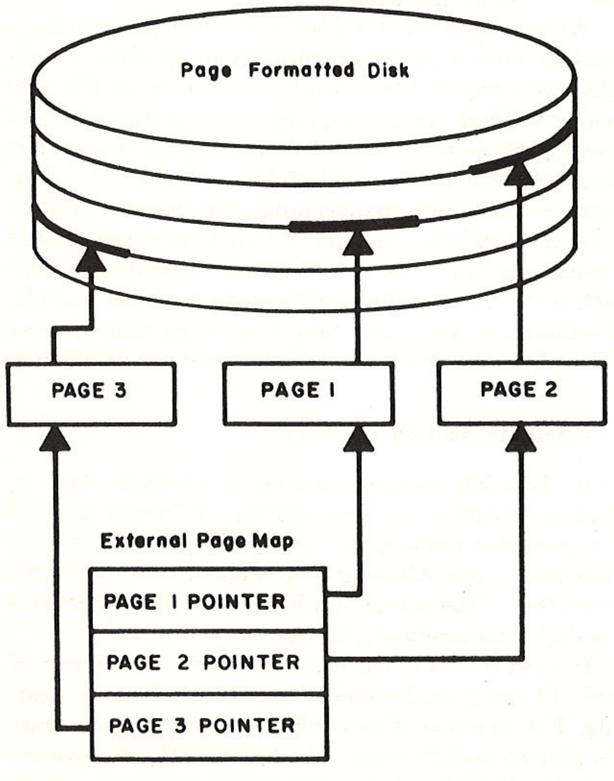


FIGURE 2—Typical virtual sequential organization

In a typical virtual index sequential organization, three classes of blocks can be specified within the external page map: directory pages, data pages, and overflow pages. One entry, corresponding to the lowest record key in each data page, is placed in the directory. Records are maintained in collating sequence within the the data set by key value. To find a given record, the directory is searched and then the data page containing the record is searched. Locator entries, corresponding to each record within a data page are stored in the back of the data page. Space in overflow pages will be assigned when record insertions exceed the capacity of a data page. The record locators in the primary data page will point to secondary locators within the overflow page. The placement of data and locators within the same block is a significant convenience associated with choosing a fixed block size, and is in contrast to many contemporary systems.

In a typical virtual partitioned organization, two classes of page blocks can be specified within the external page: directory pages and member pages. The partitioned organization directory contains an entry describing each member, which is specified as a contiguous group of entries within the member-data portions of the external page map. Members are subsidiary data groups that may have sequential or index sequential organizations (or any combination of the two). Members can be expanded or contracted by simply adding or deleting entries within the external page map. The partitioned organization allows a user to manipulate individual members or to conveniently treat a group of data sets as a single entity for purposes such as creating libraries or sharing data sets through the system catalog.

Two types of interlocks are provided to coordinate simulatenous access to shared data sets by more than one user:

- Read interlock: prevents another user from writing into the interlocked data space; other users may have read-only access at the same time.
- Write interlock: prevents another user from reading or writing the interlocked data space; can be set only when no other interlock is set.

Interlocks are established at various data space intervals, depending on the data set organization. Virtual sequential organizations are interlocked at the entire data set level. Virtual partitioned organizations are interlocked at the individual-member level. Virtual index sequential organizations, however, are interlocked only at the individual data-page (block) level; this allows a much finer level of sharing than is available in most other systems. The control mechanism for sharing has been simplified significantly by the choice of placing interlocks at the level of the physical block, rather than at the level of the individual record.

When a logical record is wanted (in a straightforward case), the flow of control is as follows. The appropriate external-storage address of the record's page is obtained from the external page map. This address and the virtual memory address of a buffer are passed to the Resident Supervisor in a request list.

The Resident Supervisor places the symbolic device address and relative block number in the relocation-table entry associated with the buffer's virtual address. However, the page itself is not yet read into main storage. It is only when a user addresses a record in his virtual memory buffer that a paging-relocation-exception interruption occurs, causing the Resident Supervisor paging processors to bring the page into main storage.

The virtual access methods write onto external storage only those pages of the buffer that have been modified. When it is necessary to write a buffer page onto external storage, the appropriate virtual access method routine obtains an external-storage address for the page from the external page map and passes the virtual memory address of the buffer, together with this external-storage address, to the Resident Supervisor. The appropriate Resident Supervisor routines then write the buffer page into the data set on external storage.

The external page table maps the external-storage locations of a given portion of the data set into a virtual memory buffer. The size of the buffer controls the extent of virtual memory allocated to the data set. This second level of mapping allows the user to process a page-oriented data set that can be as large as 65,000 pages, which is a great deal larger than the 4096 pages available in a 24-bit-addressed virtual memory.

TSS/360 brings into the buffer only those pages of the data set that are currently needed. The size of this buffer need not be limited to one page; it may be as large as a segment (256 pages), thereby allowing a user to address all or a portion of a data set in the same manner as main storage.

The TSS/360 user thus has a choice that allows him to treat a properly organized data set as a file or as one-level storage. There are several advantages, however, involved in the use of traditional data management macro instructions, such as GET and PUT. For example, while information within auxiliary storage is vulnerable to a system failure, information that is maintained through macro instructions is updated directly on external storage, and is thus preserved across system failures. In addition, macro instructions directly signal when buffer contents are no longer required and thus enhance efficient auxiliary space management.

As described, the virtual access methods perform a programmed search of data set indexes in virtual storage. Conceptually, this amounts to combining the benefits of paging large indexes with the benefits of substituting high-speed auxiliary drum storage for slower speed disk storage.

This concept of programmed searches can be extended by user programs to secondary indexes for data sets. For example, the TSS/360 Assembler macro library is maintained as a line data set for maintenance purposes. However, the library must frequently be accessed alphabetically on the basis of macro instruction name. A list of such names combined with the line numbers locating the macro instruction is maintained, alphabetically sorted, in a separate sequential data set. When it is desired to locate a particular macro instruction, the entire alphabetically arranged name list is brought into virtual memory and a programmed search is performed to locate the appropriate index (i.e., line number) to the macro library.

We have found that implementation of the virtual access methods required significantly fewer lines of code than were required for a corresponding set of TSS/360 access methods used to support physical-sequential devices, such as printers and magnetic-tape units. It is apparent that the removal of device-dependent operations (with complex channel programs), the standardization of block size, and the elimination of exceptional procedures (such as end-of-volume operations) simplified the actual coding for the virtual access methods. Furthermore, the separation of data set management from physical device management simplified debugging.

Program loading services

In TSS/360, program loading is dynamic; that is, during execution one program may reference another program that has not been previously processed by the dynamic loader. Although not unique to TSS/360, this is another of the means by which a user is given flexibility during his terminal sessions.

In most conventional systems, there are a number of difficult design trade offs associated with dynamic loading. For example, the available memory space must be apportioned in some way between the storage requirements of the link-loader and the option to leave the program to be loaded in main storage. As another example, the cost of performing basic linking and unlinking functions during program execution must be traded off against the potential inefficiencies of passing inter-module parameters by value.

In TSS/360, the loading process is performed in virtual memory. The large virtual memory environment of TSS/360 permits a disassociation of claims on address space from claims on main storage, and thus allows the allocation of storage to be optimized on a system-wide basis. Moreover, because of the large virtual store environment, it is seldom necessary to unlink program modules. This makes it unnecessary to place system restrictions upon the form of intermodule references.

A program module generated by a language processor resides in the system as a member of a partitioned data set before being loaded and, in this state, consists of at least two parts: text and module dictionary. A third part, an internal symbol dictionary, used by the program control subsystem, is optional.

The text of the program module is divided into control sections. This division is determined by source language statements for output generated by the Assembler, and automatically for output generated by the FORTRAN compiler.

From a system standpoint, the purpose of control sections is to allow a program to be divided into portions whose virtual memory locations can be adjusted (independently of other sections) by the dynamic loader without altering or impairing the operating logic of the program.

For the user, a control section is a segment of coding or data that can usually be replaced or modified without reassembling an entire program. A control section also represents a segment of coding or data to which attributes can be assigned independently.

At the time the user creates a control section, he may assign a variety of attributes to it, such as:

- fixed length
- variable length
- read-only
- privileged
- shared

The module dictionary consists mainly of a group of control-section dictionaries, one for each control section of the program module. A module dictionary describes the text: its length, attributes, external-symbol references and definitions, and information to be used in relocating address constants. Collecting all linkage data into one module dictionary allows the TSS/360 dynamic loader to calculate linkage addresses without bringing the larger text portion of the module into main storage.

In TSS/360, the dynamic loader resides in virtual memory. The basic functions of the loader are to load programs into virtual memory—not into main storage—and to relocate only those address constants that are in pages of text actually referenced during execution of the program.

The process of loading a program into virtual memory does not involve the movement of any text and is performed in the allocation phase of the dynamic loader. Loading a program into virtual memory consists, in largt part, of establishing the addressabilit*y* of the program within the virtual store.

When the dynamic loader's allocation phase is invoked, it utilizes the virtual access methods to locate the program library containing the requested object program module.

Utilizing information from the module dictionary, the loader requests the allocation of a virtual memory for the object module text. Virtual memory allocation involves the creation of relocation table entries for the text and the assignment of protection keys according to the attributes of each control section. The loader next places the external-storage addresses of the module's text pages into the relocation table entries just created. Locations within a program are addressed through base registers, index registers, and displacements. Base registers generally contain values obtained from address constants. For each text page that contains address constants, an "unprocessed by loader" flag will be set in the appropriate relocation-table entry.

Among other functions during this phase, the dynamic loader examines all external references of the module, and obtains and processes the module dictionaries for any additional object modules required to satisfy these external references. This process results in the dynamic loader recursively invoking itself as long as additional dictionaries must be obtained.

When the allocation phase is complete, the dynamic loader exits, supplying location values that correspond to entry points in the loaded program.

The second phase of the dynamic loader is invoked when a page containing address constants is referenced and consequently brought into main storage during program execution. Address constants on the page are adjusted to reflect the values calculated during the allocation phase of the loader.

A secondary function of the dynamic loader is to enforce the TSS/360 protection rules concerning the loading and referencing of program modules.

During the allocation phase of the dynamic loader, the content of each module dictionary is placed in a private task table. Called a task dictionary, this table contains the information needed to load (and unload) modules for particular task. A task dictionary consists of a header containing three hash tables, and a body containing one module dictionary for each module loaded for the task.

To link programs dynamically, the dynamic loader must be able to look up all external-symbol definitions in an efficient manner; hash tables, consisting of headers and a number of hash chains, are used for this purpose.

To reduce the number of pages referenced during the loading process and to prevent a nonprivileged user from accidentally linking to a system routine or a system routine from erroneously linking to a nonprivileged user routine, three symbol tables are defined: privileged-system, nonprivileged-system, and user.

The privileged-system table contains external symbols defined in control sections with the privileged attribute.

The nonprivileged-system table contains nonprivileged external symbols defined in control sections with the system attribute. A further convention has been adopted: the initial entry points of nonprivileged system routines directly invoked by a nonprivileged user (such as a language processor) may begin with certain reserved characters. This has the effect of making these routines "execute-only" to the user.

With the two system symbol tables, instead of just one, the dynamic loader does not need to search a hash chain containing a large number of privileged symbols when looking up nonprivileged symbols. As will be shown, the loader does not normally reference the privileged system symbol table during system operation.

The third symbol table, constructed for the user, is primarily protective. It provides close control over the interface between the user and system routines by separating the user's symbols from system symbols.

Although the loading and protection facilities just described are quite powerful, it has already become apparent that future computer systems might require extensions to these facilities. This is currently a subject of study within IBM and elsewhere.[5,6]

Task structure

Within TSS/360, tasks function in the environment of a large, segmented virtual store. Our knowledge of the proper way to utilize this environment evolved as the system was built and used.

Because of the large size of this address space, the need for specifically declared overlays is eliminated. This does not remove the need to plan program organization when efficient execution is desired; it merely makes it possible to minimize planning. In a time-sharing environment, where there is a premium placed upon solving a problem quickly, this added flexibility is significant and frequently desirable.

Initial virtual memory

During the initial stages of development, it was realized that certain system service routines must reside in each task's virtual store when the task is initiated (e.g., the dynamic loader). This virtual-store image would be created during system startup. As the system developed, it became apparent that efficiency could be enhanced by including a large number of other system routines in this initial virtual memory.

The TSS/360 routines that currently make up initial virtual memory include all privileged system service routines and many nonprivileged system programs, such as the FORTRAN compiler.

By tightly pre-loading most system programs at system startup, the overhead usually associated with library searches, binding, and unbinding is significantly decreased. The trade off here is time versus the auxiliary-storage space needed to hold the fully bound copy of those routines included in initial virtual memory.

Still another advantage is obtained by binding at system startup. Efficiency in a paging system is closely associated with the degree of locality of reference over a time-slice. In a highly modular system, it frequently occurs that there are groups of routines that follow a pattern such that all members of the group tend to be referenced within a short period of time whenever any one of them is referenced. Page-reference patterns associated with system programs can be significantly improved by ordering routines with an affinity for each other so that they are packed, as a group, into a minimum number of pages.

In TSS/360, this ordering is based upon a control section name list that can be altered easily to optimize the packing of system programs to minimize paging. This is especially significant in TSS/360 because many control sections are much less than a page in length.

Sharing

Virtual memory sharing in TSS/360 is utilized in three ways:

- When users share programs, they share the pure-procedure sections of the program. Each user receives a private copy of any modifiable data contained in the program.
- When users share data sets, they share a common external page map control table.
- All tasks share certain common control tables (such as the I/O device allocation table).

Program modules designed for simultaneous sharing by more than one task are called re-entrant. Such modules are characterized by their division into a shareable control section that does not change in any way during execution and a private control section (PSECT) that contains modifiable data and address constants.

While most system programs are re-enterable modules with PSECTs, it is not necessary to use a PSECT when composing a TSS/360 program. With greater effort for special cases, it is possible to write re-enterable programs where all parameters are held in CPU registers or where working space is dynamically acquired.

When a re-enterable program is composed, all modifiable data, work areas, and address constants may be placed within a PSECT. Allowing the composer of a program to create the PSECT relieves the caller of that program of the requirement to know precisely what address constants the called program requires.

The use of PSECTs has effects upon the structure of programs within TSS/360. Whenever a user loads a shared re-enterable module, a private copy of its PSECT is placed into the user's private virtual memory, while shared access is established to a single copy of the program's re-enterable control sections. Programs are shared in such a way that the PSECTs and the re-enterable portions of the called routines are separately mapped into the task's virtual memory. Moreover, because each user's virtual memory is allocated dynamically and independently, the single physical copy of a re-entrant control section may be mapped into different virtual memory locations for each concurrent user (see Figure 3). Therefore, to perform linkage to a re-entrant routine, two virtual memory addresses must be supplied

The first address specifies the location at which execution of the program module will begin when control is transferred. This is the conventional external reference value.

The second address can be used to specify where the PSECT of the linked module has been mapped within the task's virtual memory. If this pointer were not supplied, the re-entrant module would have no way of knowing, for instance, where the appropriate private modifiable data are located since the PSECT may be placed in different virtual memory locations in each concurrent task.

Putting all address constants and modifiable instruction sequences into one or more PSECTs does not guarantee that the resulting routine will be re-enterable under all conditions. While this provides for intertask re-enterability (i.e., sharing by a number of tasks), intratask re-enterability must be considered.

A single task can re-enter the same program when it receives a task interruption while executing a system routine or when a routine is called recursively. In such a situation, the PSECT will not protect task integrity, since within a single task there is only one copy of the PSECT. This is why the Task Monitor provides either

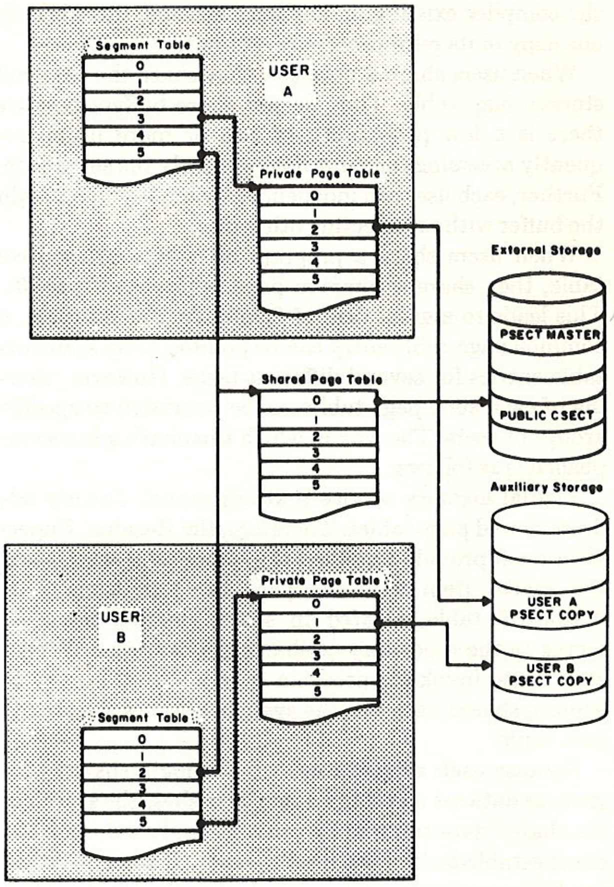


FIGURE 3—Sharing of programs in TSS/360

a push-down save-area or a means by which a routine can protect itself from unwanted intratask re-entrancy.

A PSECT is generally used to hold the register save-area for a re-entrant routine. Placing a save-area within a PSECT, rather than into a push-down stack, reduces overhead and facilitates tracing linkages during debugging.

The sharing of programs in virtual memory is based on many users actively using pure-procedure sections of the same program (such as the FORTRAN compiler) with resultant decreases in the paging overhead and utilization of main storage. Because of the amount of shared code in TSS/360, the probability that shared pages will be simultaneously used is high only for a few system routines. The primary value of shared code thus lies in its read-only attribute, which allows only one copy of a page of code to be on auxiliary storage. During the lifetime of the average task, there is a high probability that a number of users will invoke, say, the FORTRAN compiler. Thus, instead of many copies of the compiler existing on auxiliary storage, there is only one copy of its pure-procedure sections.

When users share a data set, they share the external storage map table. They do not share buffers because there is a low probability of two or more users frequently accessing the same data at nearly the same time. Further, each user can independently modify his copy in the buffer without affecting other users.

When users share a program or a data set control table, they share a common page table (see Figure 3). This leads to a great deal of flexibility. For example, a common page-table entry can be pointed to by segment-table entries for several different tasks. However, sharing of each such page table can be restricted to specific groups of users. The way in which this sharing is accomplished is as follows:

Virtual memory service routines cannot directly address shared page tables. Therefore, the Resident Supervisor must provide a method of symbolically associating the shared item with the page table that maps it. A control table, located in shared virtual memory, serves as the repository of sharing information. Whenever a user invokes a program from a shared library or opens a shared data set, the system searches the shared data table.

Because each system user can catalog a shared program or data set using any name he wishes, the search of the shared data table is, by convention, based upon the name established by the item's owner. If the entity has not been previously referenced during the session, then an entry for this name will be created in the table.

Next, shared virtual memory is obtained for the entity. The Resident Supervisor creates the required number of shared-page-table entries and sends back the symbolic identification number of the shared page table and the location of the requested allocation within the segment. This information is stored into the shared data table. Thus, there is now an association between the name of the entity to be shared and the page table that maps the entity.

When another user invokes the shared module or optns the shared data set, a search of the shared data will yield a match on the name. The symbolic page table number can then be used in a supervisor call to request that a segment-table entry for this user be made to point to the proper shared page table.

Virtual memory sharing requires the use of programmed interlocks to prevent destructive intertask interference. The use of interlocks for sharing, however, requires careful control. For instance, system operation can be severely affected if one task sets an interlock in a system table and then becomes inactive for a long time. Furthermore, substantial system overhead is incurred if tasks waiting for the interlock to be reset are continually being dispatched only to find that the interlock is still set. This type of problem is representative of the many subtle considerations involved in the control of extensive sharing among tasks in a time-sharing environment. We are still gaining experience and insight into this aspect of the TSS/360 time-shared operating system.

REFERENCES

1 *The compatible time-sharing system: a programmers guide*
P A Crisman ed MIT Computation Center Cambridge Mass MIT Press 2nd ed 1965

2 H A KINSLOW
The time-sharing monitor system
Proc AFIPS 1964 FJCC Vol 26 Washington DC Spartan Books 1964 pp 443–454

3 W T COMFORT
A computing system design for user service
Proc AFIPS 1965 FJCC Vol 27 Part I Washington DC Spartan Books 1965 pp 365–369

4 *IBM system/360 time sharing system: concepts and facilities*
Form C28-2003 IBM 1968

5 R M GRAHAM
Protection in an information processing utility
Comm ACM Vol 11 No 5 May 1968 pp 365–369

6 E B VAN HORN J B DENNIS
Programming semantics for multiprogrammed computations
Comm ACM Vol 9 No 3 March 1966 pp 143–155

APPENDIX B

OCTOBER 28, 1968 TR53.0001

UTILIZATION OF VIRTUAL MEMORY IN TIME SHARING SYSTEM/360

by J. R. Martinson

ABSTRACT

The concept of virtual memory provides the fundamental ability to separate the address space that is used by a program from the allocated real memory that supports program execution. Thus, with a virtual memory, program design does not need to consider the availability of real memory and, as with conventional systems, to plan for overlays. Real memory becomes a true system resource that can be dynamically allocated by the system to adapt to real-time requirements.

Operating experience with the IBM System/360 Time Sharing System (TSS/360) has indicated that the effective utilization of virtual memory requires a reevaluation of currently accepted programming techniques. The emphasis shifts from a program's total space requirements to a program's short-term demands on real memory. The objective becomes the optimization of useful information in regions of virtual memory, even if this should increase total program size.

This report explores the TSS/360 definition of virtual memory and the program structure imposed upon it. Guidelines are presented to describe how programs should and should not be constructed for effective utilization of virtual memory.

Virtual Memory
Time Sharing
TSS/360
Memories
Memory Allocation
Memory Relocation
Segmentation
Program Structure
Dynamic Address Translation
Programming
IBM System/360
07 Computers
21 Programming

IBM

Systems Development Division · Time Sharing Department · Mohansic Laboratory
Yorktown Heights · New York

INTRODUCTION

A virtual memory is a zero-origined* sequential address space that is dynamically and continuously mapped, as it is referenced, into the address space of real memory.[1] The size of virtual memory is not limited to the availability of real memory, and sequential virtual memory addresses need not map to sequential real memory addresses.

The IBM System/360 Time Sharing System (TSS/360) virtual memory, as seen by a task**, has an extent equal to the addressing capability of the hardware in the System/360 Model 67.[2] System/360 standard 24-bit addressing thus provides 16-million bytes (as a special feature, 32-bit addressing provides 4-billion bytes) of virtual memory.

To facilitate utilization of this address space, the TSS/360 virtual memory is organized as a sequence of contiguous segments, with a maximum of 16 segments in the 24-bit system and a maximum of 4096 segments in the 32-bit system. Each segment comprises a sequence of 256 pages, and each page comprises 4096 bytes. To be consistent with this page size, real memory is viewed as being divided into 4096-byte blocks.

A page is the smallest allocatable unit in virtual memory, for the system controls a task's use of virtual memory on a page basis. An allocated virtual memory page is, at any time, in one (or more) of four states:

- Its content is in, and therefore mapped into, a block of real memory.
- Its content is in a data set on external storage*** (IBM 2311 Disk Storage Drive or IBM 2314 Direct Access Storage Facility).
- Its content has, at some time, changed from its initial value, and is on auxiliary storage**** (IBM 2301 Drum Storage, IBM 2311 Disk Storage Drive, or IBM 2314 Direct Access Storage Facility). In this case, both the initial and latest versions of the virtual memory page must be retained.
- It has no initial content and has never been referenced. The system will map the virtual memory page, when it is referenced, into a real memory block containing all 0s.

Zero-origined: beginning at 0; the first address of a virtual memory has a value of 0.

***Task:* the basic unit permitted time-shared access to the available computing facility. To facilitate system control over a variable number of independent tasks, TSS/360 associates a unique virtual memory with each task.

****External storage:* in TSS/360, a collection of direct-access storage on which a user's cataloged public and private data sets are maintained.

*****Auxiliary Storage:* in TSS/360, a collection of direct-access storage used by the system to store the current copy of any virtual memory page that is altered as a result of task execution.

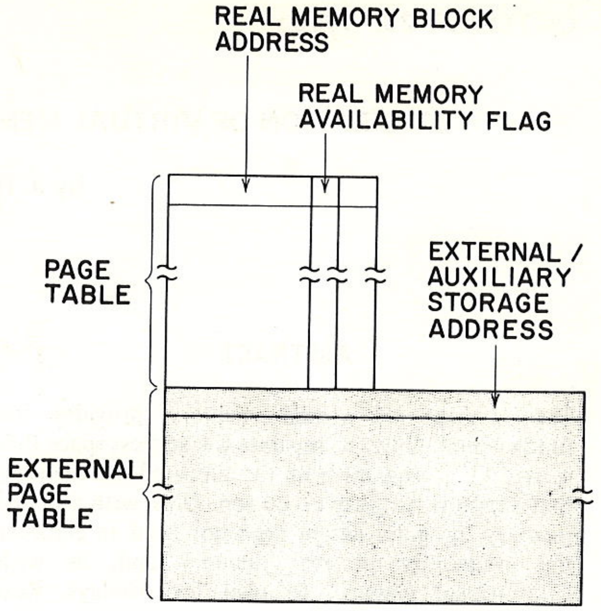

REAL MEMORY BLOCK ADDRESS
REAL MEMORY AVAILABILITY FLAG
PAGE TABLE
EXTERNAL / AUXILIARY STORAGE ADDRESS
EXTERNAL PAGE TABLE

Figure 1. The status of each allocated virtual memory page is maintained in a page table and an associated external page table. Each task has its own tables.

Paging

The system maintains the status of each allocated virtual memory page in a page table and an extension to it, the external page table (see Figure 1). The page-table entry indicates whether or not the virtual memory page has been mapped into real memory and, if it has, gives the associated real memory block address. The corresponding external-page-table entry provides the location of the virtual memory page image on external or auxiliary storage.

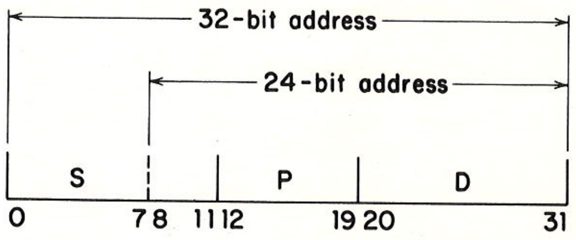


Figure 2. A virtual memory address comprises a segment number, S, a page number, P, and a page displacement, D. In the 24-bit addressing system, S occupies four bits; in the 32-bit system, it occupies eight bits.

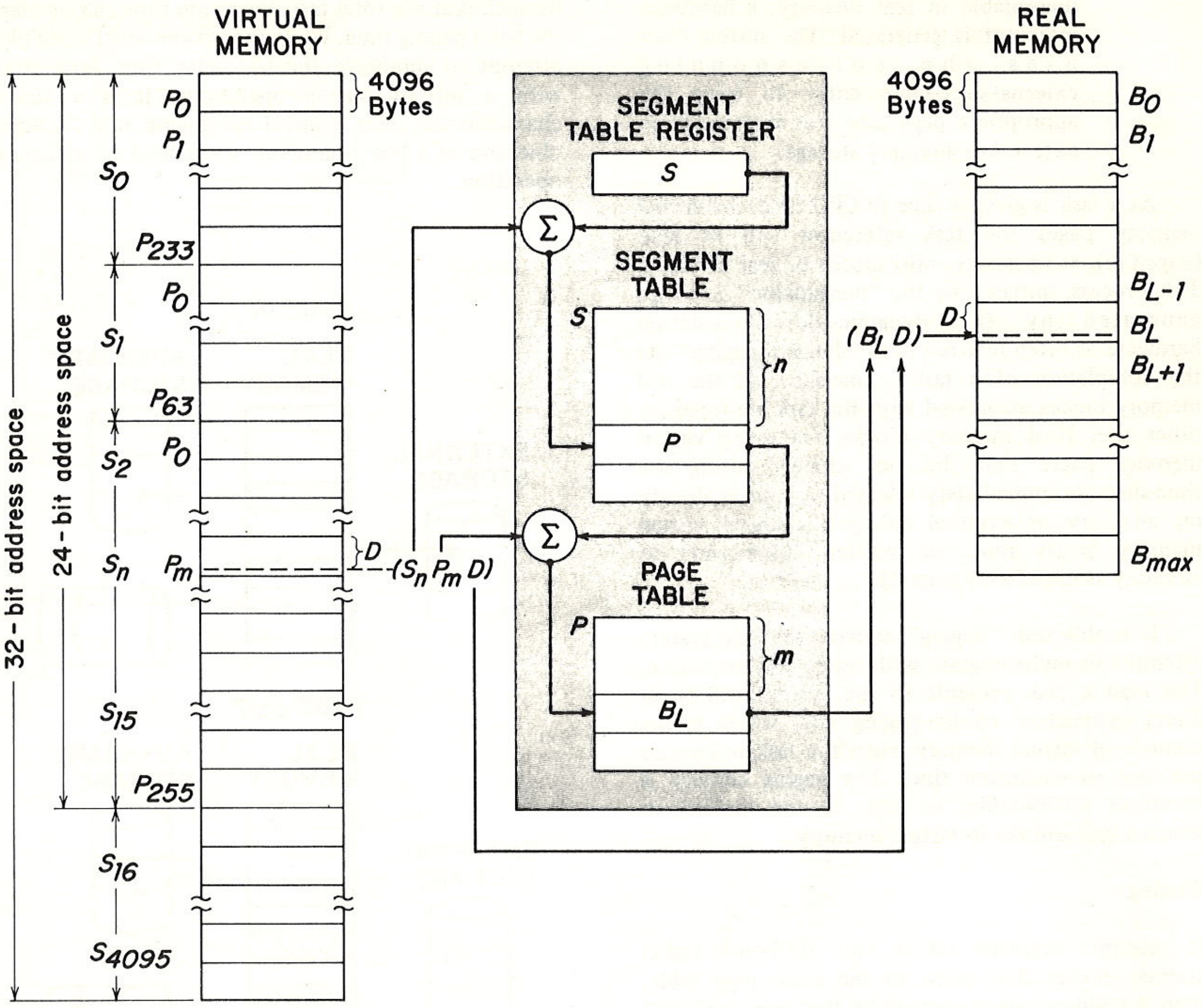


Figure 3. A simplified view of dynamic address translation. The virtual memory address has the form $S_n P_m D$, where S_n is the segment number, P_m is the page number, and D is the page displacement. Using the segment and page tables, this is translated into a real memory address. The real memory address has the form B_L D, where B_L is the real memory block address and D is the same displacement as in the virtual memory address.

Any virtual memory address can be viewed as a 4- (or, in 32-bit mode, 12-) bit segment number, S; an 8-bit page number, P; and a 12-bit page displacement, D (see Figure 2). The System/360 Model 67 dynamic-address-translation hardware translates such a virtual memory address into a real memory address. This is done by translating a virtual memory page address, which is uniquely identified by its segment number and page number, into a real memory block address, and subsequently applying the indicated displacement (see Figure 3):

- The association between a task and a specific virtual memory is accomplished by associating the address of the task's segment table with the CPU assigned to the task. A special hardware register, the segment table register, is reserved for this purpose

- The pointer to a segment table, which is contained in the segment table register, and the segment number locate a segment-table entry.

- The resultant pointer to a page table, which is contained in the segment table, and the page number locate a page-table entry.

- The page-table entry is used as follows:

  - When the content of the page-table entry is the real memory block address of the virtual memory page, it is used with the displacement to form the complete real memory address.

  - When the content of the page-table entry indicates that the virtual memory page is unavailable in real memory, a hardware interrupt is generated. The system then uses the corresponding external-page-table entry to bring the appropriate page into real memory from external or auxiliary storage.

As a task is given a slice of CPU time, the virtual memory pages the task references will be read (paged-in), as necessary, into blocks of real memory. This process, initiated by the "unavailable" interrupt generated by the dynamic-address-translation hardware, is often referred to as "demand paging". At the completion of a task's time-slice, all the real memory blocks associated with the task are freed for other use. Real memory blocks containing virtual memory pages that did not change during the time-slice are immediately released (a copy is already on auxiliary or external storage); changed virtual memory pages must be written (paged-out) to auxiliary storage (see Figure 4).

It is this task "paging" activity that the system attempts to multiprogram with other task execution. The load a task presents to the system will be in direct proportion to its paging rate, which is the number of virtual memory pages the task references per unit of execution time. The paging activity is therefore attributable, in part, to the manner in which a task utilizes its virtual memory.

Sharing

If multiple segment tables (for different tasks) contain entries that point to the same page table, then the address space spanned by that page table will be addressable in each of those virtual memories. This mechanism provides for the dynamic sharing of entire segments of a virtual memory among many tasks.

The system attempts to retain referenced shared virtual memory pages (as compared to private pages, which are released at the end of a time-slice) in real memory for an extended time. The intent is to keep such shared virtual memory pages in real memory for use by tasks other than the task making the initial reference. Thus, the shared virtual memory portion of the system, which forms a base for the virtual memory operation, will tend to remain in real memory and reduce the system paging rate.

EFFECTIVE TASK UTILIZATION OF VIRTUAL MEMORY

The paging rate a task presents to the system is the single most important factor contributing to the system's effectiveness in processing the task. If efficient multiprogramming is to be achieved during a given real-time interval, the cross section of paging rates for all tasks executing in that time interval must be such that the total task-execution time can overlap the total paging time. While the system must certainly attempt to minimize the I/O delay time associated with a set of paging operations, it is a task's characteristics that control its paging rate. A task objective of a low paging rate is essential for efficient operation.

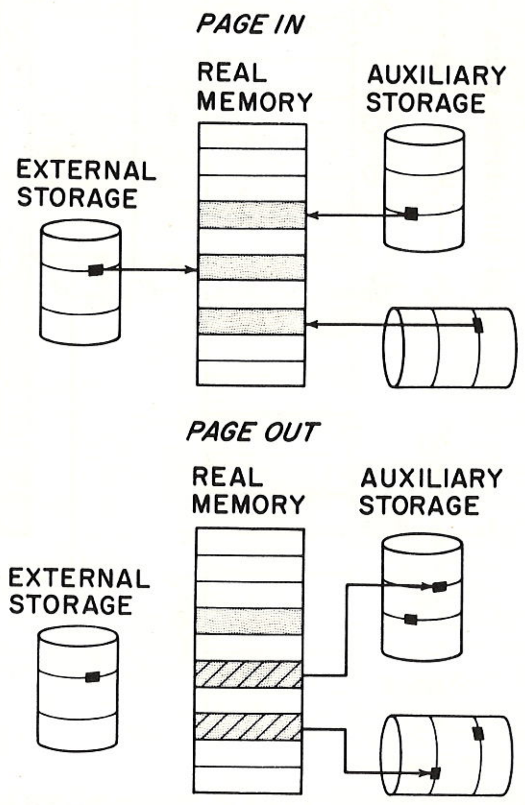


Figure 4. When a task references a virtual memory page that has not yet been mapped into real memory, that page is read (paged-in) from external or auxiliary storage into a block of real memory. When the task's time-slice ends, only those pages that have been modified are written (paged-out) onto auxiliary storage.

Another consideration is the effective use of auxiliary storage, which serves during task execution as an extension to real memory (having significantly slower access time). Any page of virtual memory that changes (is written into), however so slightly, will at some time reside in its entirety on auxiliary storage. Furthermore, the most recently changed copy of a virtual memory page will remain on auxiliary storage until the task explicitly deletes the page from its virtual memory. Once a virtual memory page is allocated to auxiliary storage, the subsequent read-access time to this page will differ, by as much as an order of magnitude, depending on whether the page is on drum or disk storage. The system maintains a given task's auxiliary storage on both drum and disk, giving preference to the drum.

A task's unjustifiable requirement for auxiliary storage space can adversely affect system operation.

- It can seriously compromise the system objective of maintaining a task's auxiliary storage solely on drum.
- It can limit, by decreasing the availability of auxiliary storage, both the extent to which a single task can use its virtual memory and the number of tasks that will be granted concurrent access to the system.

Virtual Memory Program Structure

A principal objective of the standard program-module structure in TSS/360 is to permit program modules to be shared selectively among active tasks. This objective has been achieved by creating a program structure that provides intertask program reentrancy without requiring a program to dynamically allocate its working storage.

A program is a collection of program modules. Each program module usually consists of two independently relocatable control sections (CSECTs): the first, usually referred to as "the CSECT," contains read-only, address-free instructions (pure procedure) and data; the second, usually referred to as "the PSECT" (prototype CSECT), contains all variable, address-dependent instructions and data. The CSECT is task-independent and the PSECT is task-dependent; hence, while the CSECT can be actively shared among tasks, each task must have its own private copy of the PSECT.

A CSECT contains:

- Executable read-only code
- Data constants
- Nonrelocatable (absolute) address constants
- Nonrelocatable literals
- Any other nonmodifiable, address-free information

A PSECT contains:

- Save-areas (used in the linkage process)
- Local temporary (working) storage
- Parameter lists
- Relocatable address constants
- Relocatable literals
- Any other modifiable or location-dependent information

Special attention should be given to the placement of address constants in the PSECT. First, address constants change as a result of relocation (the process of relocating modules to virtual memory) and binding; second the value of an address constant is dependent upon the location of the object in virtual memory (the location may differ from task to task).

The dynamic-loading process in TSS/360 allocates and relocates program modules to virtual memory; the allocation function thus implements the desired reentrancy when it allocates the CSECT to shared virtual memory and the PSECT to private virtual memory. Further, when executing shared code in a multiple-CPU TSS/360 environment, this program structure permits different tasks to be dispatched to multiple CPUs without conflict.

Program Organization

Program organization refers to the external packaging of program-module CSECTs to form the page images of a complete program. The TSS/360 linkage editor provides the facility for external packaging.

When program modules are dynamically loaded from program libraries into virtual memory, every CSECT is allocated at a page boundary (note that a PSECT is a CSECT with the prototype attribute). Therefore, as program modules interact, every distinct CSECT reference will result in a corresponding virtual memory page reference. When this interaction is known in advance, a significant reduction in the task's paging rate can be achieved by properly combining program-module CSECTs into more compact pages. Packaging can be achieved as a three-step process (see Figure 5).

First, the external referencing characteristics of each module are determined, indicating how the modules can interact (shown in Figure 5A).

Second, the dynamic module interaction is ascertained. The objective is to determine the high-frequency, mainline path through the set of modules; this is the path that should be optimized (shown in Figure 5B).

Third, the CSECTs of the dynamically interacting modules are combined into a single compact CSECT (shown in Figure 5C). Also, the PSECTs of the

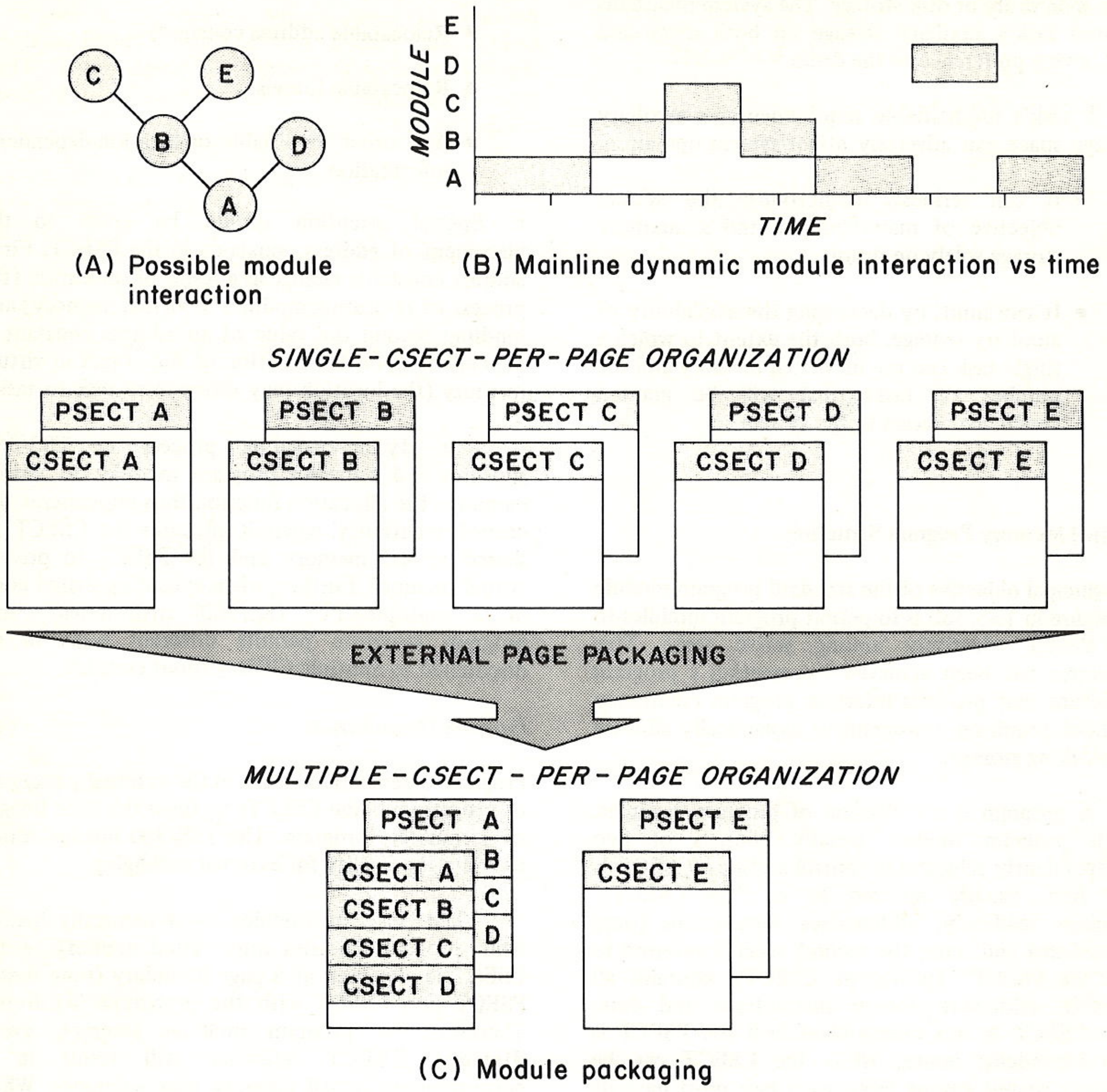


Figure 5. Program organization can be optimized by external packaging, which can be achieved as a three-step process. First, the external referencing characteristics of each module are determined *[A]*. Then, the dynamic module interactions are ascertained *[B]*. Finally, the dynamically interacting modules are combined in a single CSECT and PSECT *[C]*. In the case shown here, in which module E normally is not referenced, the eight mainline page references required in the single-CSECT-per-page organization are reduced to two in the optimized organization.

dynamically interacting modules are combined into a single compact PSECT. A fundamental assumption follows from the imposed program structure: Module interaction can be used as a basis for efficient CSECT and PSECT packaging.

Note, however, that a CSECT should not be packaged across a page boundary because, generally, this will double the number of pages referenced by the CSECT. (Of course, this assumes that the CSECT is less than a page in extent.) This leads to a rule-of-thumb: the extent of a CSECT or a group of packaged CSECTs should be less than or equal to a page.

Another important consideration when packaging to optimize the program paging rate are the resultant auxiliary-storage requirements. Observing the rule-of-thumb, above, CSECTs that are subject to change during program execution should be packaged in as few page images as possible to conserve the auxiliary storage required by the task.

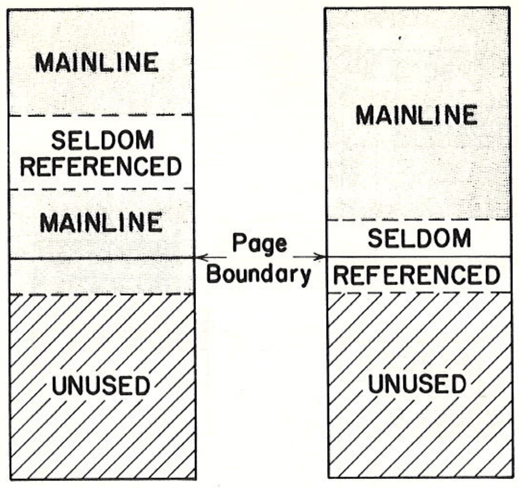


Figure 6. Program-module organization can be optimized by internal packaging. In the case shown here, prior to the organization of the control section *[left]* two pages normally would be referenced, since the mainline spans a page boundary. After reorganization so that all of the mainline is contained on one page *[right]*, only that page normally would be referenced.

Program-Module Organization

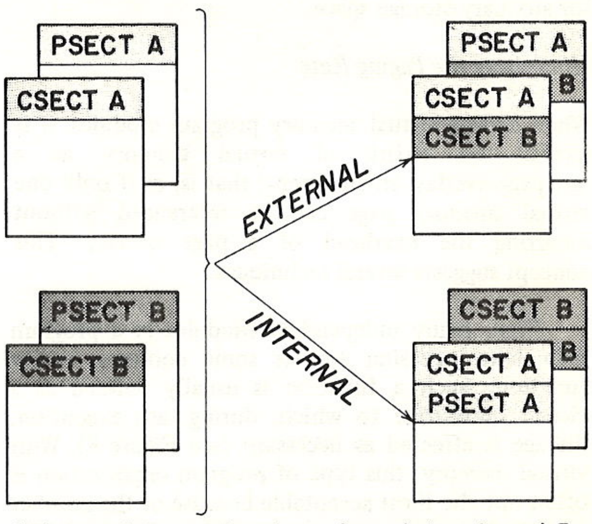
Program-module organization refers to the internal packaging of program modules. For modules that are self-contained (do not interact with other modules), internal packaging is most appropriate and is preferable to external packaging. As with external packaging, the TSS/360 linkage editor can be used to perform internal packaging.

To minimize the paging rate, the program-module organization selected should maximize program information content on a module basis. The design of the program module should be such that if part of a program module is required during task execution, the entire module is required. For example, the portion of a program performing exception-condition processing should be defined in a program module that is separate from the module performing a mainline function of the program.

In some cases, this objective can be attained through proper organization of code or data within a single module (see Figure 6). This situation arises when the CSECT or PSECT of a module is in excess of a page *and* contains information seldom referenced in the mainline processing of the module. The placement of the less frequently used information at the "end" of the CSECT or PSECT will probably reduce the number of pages referenced during task execution.

Still another means of minimizing the paging rate is the combination of a PSECT with its CSECT. Based on the TSS/360 program structure, a program module usually consists of a read-only CSECT and a variable, task-oriented PSECT, each on its own page. Thus, whenever the module is executed, a minimum of two virtual memory pages will be referenced. It may be desirable to package the content of a module's CSECT and PSECT into a single control section, thereby reducing the number of pages to one. This type of packaging should be attempted only if these conditions exist:

- The CSECT must not be allocated to shared virtual memory. (Such packaging will generally result in a variable, task-oriented CSECT).
- The combination of the CSECT and PSECT should not cause the new component to span a page boundary.
- Appropriate external packaging of this module with others is not possible (see Figure 7). The concern here is that improper packaging of this type will increase the number of variable virtual memory pages associated with the task and, consequently, the task's auxiliary-storage requirements.



(A) Possible packaging of modules A and B

	PAGES REFERENCED TO ACCESS		
	A	B	A & B
EXTERNAL PACKAGING	2	2	2
INTERNAL PACKAGING	1	1	2

(B) Resultant pages referenced

Figure 7. To determine whether external or internal packaging is more appropriate for given modules, the possible packaging combinations and the resultant number of pages referenced must be ascertained.

Auxiliary-storage requirements should also be considered in the internal packaging of modules. If any part of a virtual memory page changes as a result of task execution, the entire page will be written to auxiliary storage. Planning that avoids the mixture, on one page, of read-only data with variable data will contribute to the efficient use of auxiliary storage. For example, consider a large read-only table prefixed with a header that is modified during an initialization process: If both the header and the table are allocated to the same control section, then both will reside on auxiliary storage at the completion of the initialization process. A better organization would locate the table in a read-only CSECT and the header in a variable PSECT, as suggested by the TSS/360 program structure.

Coding Techniques for Virtual Memory Program Modules

Effective utilization of virtual memory requires that the characteristics of virtual memory programs differ from those generally accepted for programs operating in real memory. The prime considerations are the minimization of the paging rate and the requirements for auxiliary-storage space.

Minimizing the Paging Rate

When coding virtual memory program modules, it is useful to think of virtual memory as a one-page-overlay environment; that is, as if only one virtual memory page can be referenced without incurring the overhead of a page overlay. This concept suggests several techniques.

First, many independent modules in a program may have occasion to use some common utility function. Such a function is usually defined as a closed subroutine to which, during task execution, linkage is effected as necessary (see Figure 8). With virtual memory, this type of program organization is often not the most acceptable because of the number of virtual memory pages that must be referenced. The short-run number of pages referenced may be less if the subroutine is actually duplicated in-line at some, or all, of the points of use.

Generally, this in-line duplication is feasible whenever the extent of the resultant CSECT or PSECT of the referencing module is not increased to span a virtual memory page boundary. This duplication can be achieved by using in-line macro-expansion techniques.

Second, a common data area should be utilized to contain the parameters, work areas, constants, or other data that are frequently referenced by an interactive set of program modules. This will tend to increase the information content in the virtual memory pages referenced by the modules.

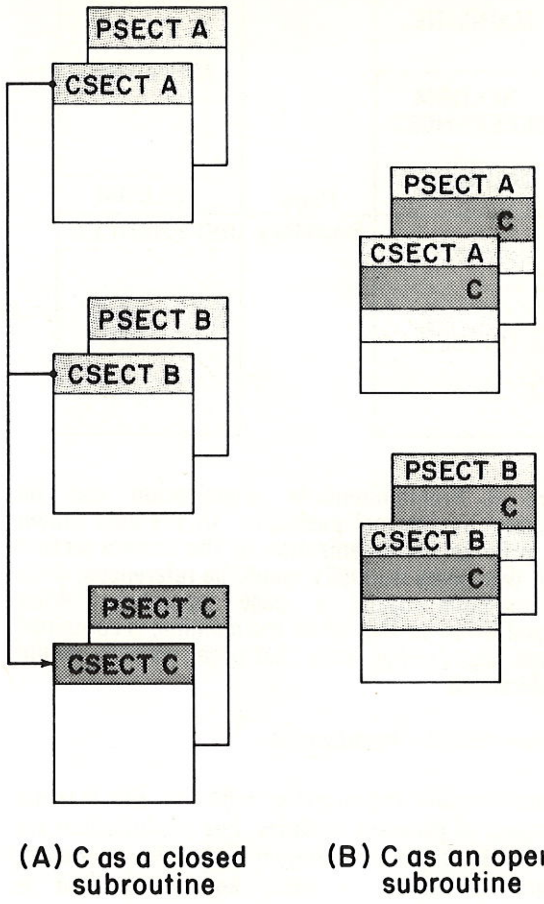


Figure 8. For virtual memory programs, coding a common utility as a closed (out-of-line) subroutine may result in a greater number of pages being referenced than by coding it as an open (in-line) subroutine.

Third, multiple-page list structures must be used carefully (see Figure 9). Of particular concern is a table organization that employs a list structure to provide fast access to a given element in the table. A linear search in a single virtual memory page often makes more efficient use of the system than does a linked search in multiple virtual memory pages, even if the instruction execution time of the former is greater. (The true virtual memory task-execution time is the sum of the central processor time *and* the required paging time.)

Fourth, data sets accessed by using the TSS/360 virtual access methods[3] can be processed in a particularly efficient manner because of the size of virtual memory. An entire data set can consist of one logical record (up to 256 pages) so that, when the data set is first accessed, its entire content is immediately mapped into virtual memory. Subsequent access to the data set content is then

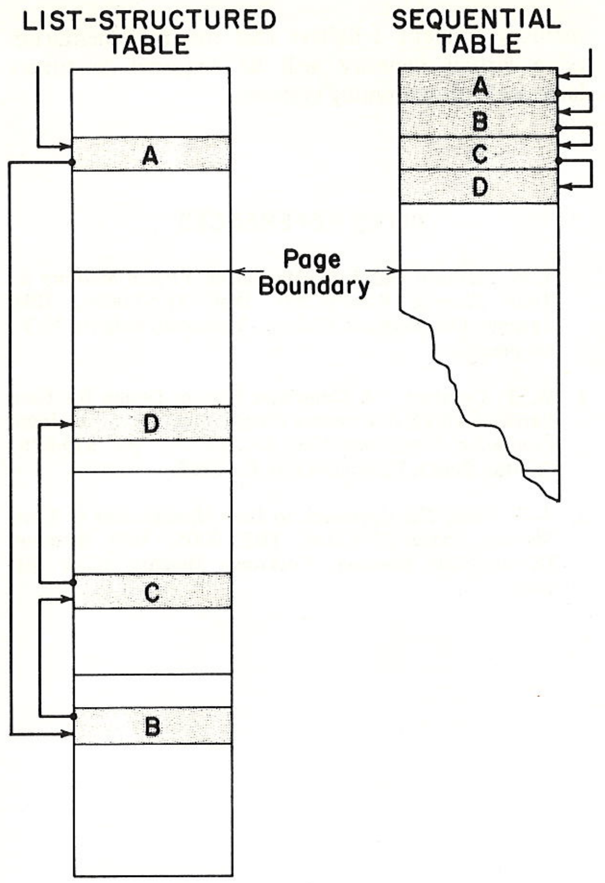


Figure 9. **When coding for virtual memory applications, a linear search in a single page may be more efficient than a linked search in multiple pages. In the case shown here, four page references are needed to locate D in the list-structured table, but only one page reference is needed to locate D in the sequential table.**

possible through direct virtual memory references; the access method overhead (including paging) normally associated with data set access is thereby eliminated.

Fifth, the techniques employed in initializing large areas of virtual memory requires special consideration. If such areas are completely initialized at the same time, a significant burst of virtual memory paging activity will be encountered as each of the respective virtual memory pages is referenced. The initialization of a virtual memory page should be delayed until that page is otherwise used by the program for the first time. In this way, the paging activity associated with initialization is included with the program's required paging activity. It is noted that the system will automatically initialize to 0 all referenced virtual memory pages not having a prespecified content.

Sixth, the design of a set of interacting program modules should avoid or minimize the repetitive invocation of one module by another during task execution. In such a situation, the total number of virtual memory pages referenced during the time either module is active would probably include virtual memory pages that contain both modules. Rather, data should be blocked into or out of a program module in a single invocation. For example, the TSS/360 data management facility provides the ability to open multiple data sets at the same time.

Auxiliary Storage Requirements

Coding techniques are also important in determining how much auxiliary-storage space is required during task execution. To achieve the objective of keeping the use of auxiliary storage at a minimum, several techniques are available.

First, virtual memory push-down stacks should not be allowed to reach depths of more than 20 or 30 pages; exceeding this will contribute heavily to a task's auxiliary-storage requirement. Every virtual memory page that contains data and that at some time was in a stack will be on auxiliary storage, even after the stack is "popped up". The task must explicitly free the associated virtual memory to release the auxiliary-storage pages.

Second, virtual memory should not be used as long-term storage for variable data without consideration for the associated auxiliary-storage requirement. In some cases, such data can be more effectively processed by the data-management facilities that will, given suitable directions, control the amount of data in virtual memory and, therefore, the amount of auxiliary storage.

Third, the use of the TSS/360 dynamic virtual memory allocation facility for variable tables that are in excess of two pages is important in the proper use of virtual memory. If such tables are dynamically allocated, it is possible to dynamically free the associated virtual memory when the table content is no longer required, thus releasing the corresponding auxiliary storage. The alternative is to locate such tables in a program module. However, the auxiliary-storage space used for the tables then cannot be released until the program module is deleted.

Fourth, TSS/360 provides for the dynamic allocation of program modules to virtual memory; such modules may also be deleted from virtual memory. It is good practice to delete a program module from virtual memory when it is no longer required. This will release any auxiliary-storage space allocated to virtual memory pages belonging to a module that may have changed (the module's PSECT) during program execution.

CONCLUSIONS

Operating experience with TSS/360 has clearly demonstrated that, as is true with real memory, efficient utilization of virtual memory depends upon the programmer. As I have already indicated, the characteristics of virtual memory programs differ from those generally accepted for real memory programs.

The techniques for coding in a virtual memory environment that are given here are derived from experience in the TSS/360 project; certainly, as the project continues and our experience grows, it can be expected that these techniques will be refined and others developed to further optimize utilization of virtual memory.

Nevertheless, operating experience with TSS/360 has already demonstrated the value of the virtual memory concept. I believe that the implementation of a virtual memory will be required in future large-scale programming systems.

CITED REFERENCES

1. O. W. Johnson and J. R. Martinson, *Virtual Memory in Time Sharing System/360,* IBM TR53.0004, IBM Systems Development Division, Yorktown Heights, N. Y. (in press).

2. W. T. Comfort, "A Computer System Design for User Service,"*AFIPS Conference Proceedings, 1965 Fall Joint Computer Conference,*Vol. 27, Part 1, pp. 619-626, Spartan Books, Washington, D. C., 1965.

3. A. S. Lett, *The Approach to Data Management in Time Sharing System/360,*IBM TR53.0005, IBM Systems Development Division, Yorktown Heights, N. Y. (in press).

APPENDIX C

OCTOBER 28, 1968

TR53.0002

AN ANALYSIS OF THE TSS/360 COMMAND SYSTEM II *

by J. B. McKeehan

ABSTRACT

The design of a second-generation command system for the IBM System/360 Time Sharing System (TSS/360) has provided new insight into the relationship of man and the machine-system. By emphasizing the human aspects of a command system, an extremely flexible structure was produced.

Such a structure is easily achieved. Certain basic structural elements are clearly defined: these include a set of syntax rules, a central dictionary, a structured event list, and a message processor. The proper use of these elements produces a system that is responsive to user needs and can easily be tailored to customer requirements. Examples of this structure are taken from the TSS/360 Command System II.

Certain by-products of this design have proved to be extremely powerful. In particular, the dynamic quality of the system allows the user to create his own form of the language. This attractive feature is balanced by the fact that user-made changes to the command system can be removed easily to allow a basic system to re-emerge for troubleshooting purposes.

* This report, under the same title, was presented at the IBM Systems Development Division Programming Symposium, Atlantic City, N.J., Sept. 24-27, 1968.

Command System
Job Control
Languages
TSS/360
Time Sharing Systems
IBM System/360
21 Programming
07 Computers

IBM

Systems Development Division · Time Sharing Department · Mohansic Laboratory

Yorktown Heights · New York

PROLOGUE

With the advent of Command System II in the IBM System/360 Time Sharing System (TSS/360),[1] a significant step forward in the logic and structure of command systems was achieved. The idea of Command System II began with the recognition that our design approach to the initial command system had been faulty. We had failed to produce a system responsive to user needs because of a lack of central purpose in the design.

In designing the new command system, the first step was to agree on the purposes for which it had to exist.[2] Following that set of purposes inevitably led to a basic structure, and with that to build on the command syntax and semantics came into view. Finally, the coding structure was laid on to develop the structure. Let us now follow this analytical process in some detail.

PURPOSES

The principle function users of any system wish to accomplish is the processing of data. However, data comes in many forms. Some of the more evident are the source statements input to language processors, the substantive information input to various processes designed by the user, and the displays, listings, and dumps output from the system to the user. It is important to note that the classification of data is meaningful only in context. What may be meant for compilation at one time may be meant to be interpreted on another occasion.

Thus, there is no rigid distinction between "object code" and other data. The same rules apply to all of the forms data may assume. In order to achieve data handling, several primary functions are required. In the first place, a naming scheme must be devised, and in the second, a way of controlling the residence and motion of various pieces of data must be available to the user.

One of the things most users wish to do is to manipulate data, and we have found that they wish to do this in a great number of ways.[3,4] They want to erase large pieces of data; to create large pieces of data; to generate data within their programs; to correct data; and to reorder, restructure, and rearrange data. Such functions are a prime consideration of a command system.

Most importantly, users want to exchange and share data. The ability to carefully define the access other users may have to this data is a prime prerogative. It is not sufficient to have only two categories of sharing–yes and no. There must be options to share at the read/write levels in addition. Further, there must be an open set of sharing rules to allow the eventual inclusion of such things as "execute only" and "write but not read."

A second, and only slightly less important, function to be performed by a command system is to provide a comfortable, efficient, and reasonable interface with various subsystems. For instance, in TSS/360 we consider as subsystems the Assembler, FORTRAN Compiler, and certain functions used to construct new commands. We also provide as part of this interface the ability to insert new subsystems into the structure. Among these, we allow the general categories of user applications, that is, systems which users may devise for whatever purposes. Of course, most users will wish to insert additional languages or language processors into the system. Many have the requirement to insert a desk calculator mode of operation.[5,6,7,8] Others want to put in text-editing facilities similar to those in IBM's ATS system, or Lincoln Laboratory's edit system.[3] If we did not provide a comfortable interface to such subsystems, one which would allow the users to conveniently hook their applications into our command system, then we would have failed to provide a satisfactory interface.

A most important function the command system must exhibit is that we must provide users of the system with control of their environment; that is, the degree of interaction between the system and the users must be controllable by the users and not by the system. The users must be able to specify the kinds of devices and the residences of data, and they must be able to sequence their operations in almost any order they choose. Clearly, we must allow the users to provide their own set of names and defaults for various items, functions, and ideas in the system.

Last, but not least, is our acknowledgment that the command system, at any stage of its development, is incomplete and must therefore be extendable in directions we cannot now predict.[9] To this end, a maximum effort was expended to make this system open ended: We must be able to add commands, sub-systems, new data handling methods, and even interfaces to other kinds of devices–for instance, graphics, remote processors of various kinds, and perhaps even someday a remote-job-entry structure.

BASIC STRUCTURE

In order to achieve the above purposes, we devised a structure that provides the flexibility customers desire without severely constricting them. We foresee that, over a long period of time, this flexibility in the structure of Command System II will essentially allow the user to create his own time-sharing system structured basically on what we provide. Indeed, one of the things which we feel to be essential to the structure of any command system is the ability of the user to get what he wants out of the system. There are those who feel that this be-all and end-all approach is obviously too expensive, too loose, too flexible, and too costly. We hope to show that providing this function is really no harder, and in fact in many cases easier, than providing the rigid, limited structures we have seen in the past.

Syntax

One of the difficulties in preceding systems, and indeed in some of our current co-systems, is that a language has been specified without allowing the differentiation between syntax and vocabulary: the assumption is made that vocabulary is a part of syntax. We know this not to be the case. In Command System II, the language is defined by a set of syntax rules and a dictionary which provides the vocabulary of the language itself.

Vocabulary

One of the principle elements of the structure is a centralized dictionary, which represents the vocabulary of the system. This dictionary, which contains the names and to a certain extent the control characters—syntax characters, if you like—to be used by the system, can be substituted for at any time or modified. Thus, in TSS/360, synonyms, default values, and new commands may be defined and put in the dictionary. It is this dictionary which, in combination with the rules of syntax, provides the essential command-system structure.

Command Stream

Another important function of the command system is to allow the creation of what we will call a symbolic command stream; that is, we do not expect that the system must go step-by-step with the user through a series of commands. It is perfectly reasonable for the user to point to a whole collection of commands he wishes to utilize as his next functional entity, and to refer to this series of commands by a pseudo command name.

In order to accomplish such a function, it is necessary to have a residence inside the system of remembered commands that are yet to be executed. This permits the user to enter commands at any speed he desires and to allow the system to catch up with, but never get ahead of, him. This concept of an internal command stream is such that we can allow eventually an asynchronous input mechanism. This structure exists even though the asynchronous part of the input stream has not yet been accomplished. The insertion of groups of commands into the stream has already been well established and is available in Command System II.

User Profile

An important part of the new command system is the user profile. This is a dossier of the user's needs, requirements, fetishes, etc., and presents a picture of the user to the system. It is with this profile that the user may establish the environment of the system he wishes to have. The impression that the user wishes to make will be read out of his profile when he logs on and made a part of his central dictionary. This profile includes all of the synonyms and defaults the user wishes to establish which are different from those the system normally provides. It also includes those special symbols he wishes to create for subsequent reference, such as individual counters or a little mail box for communication purposes. In addition, it includes certain translate tables he is allowed to specify that will allow him to convert the characters of his input according to his desires. The user profile also contains certain syntactical characters, such as continuations and promptings, which will allow the user to describe the way the system looks at his input.

We find the concept of the profile of great value to users. Yet, experience has shown that users are a little afraid to establish their own environment. For this reason, we provide what we call a system prototype profile. Thus, if the user is of the timid variety and does not wish to establish his own environment for control, the system will provide a reasonable form of basic system on which he may operate without any exercise of his own personality.

Command Vocabulary

Another feature of the structure of Command System II is the open-ended form of its command vocabulary. Rather than making command tables a preassembled part of a centralized processor, we have allowed commands to be added and subtracted from the system at will.[9] This can be done at two levels: one, the user may add and subtract commands for his own environment at any time he desires and, two, he may write commands which override system commands with the same names. For example, if he doesn't like the way the system's CATALOG command works, he can write a CATALOG command of his own with the same name (or, if he prefers, a different name) and it will do what he wishes CATALOG to do. Of course, he does this at his own risk, for he may lose as well as add functions by such manipulation.

Perhaps more important is the fact that it is perfectly possible for the user to add new functions as new commands to the system and to give them names he can use conveniently. We provide a mechanism which allows him to add his code as a command or to link combinations of existing commands and his code together into new functional entities that, to all intents and purposes, look like commands.

Message File

In the same vein, the user is allowed to write his own message file. We use exactly the same mechanism for message file that we use for commands. The system has one, a system prototype message file, which may be overlaid by the user. If he does not care for the phraseology of some system message and substitutes another, the system will choose his version in lieu of its own.

SYNTAX

The basic syntax of Command System II is independent of the vocabulary of the system. A set of syntax rules were devised that concern themselves with the line, the command name, and the command operands.

The Line

A logical line and a physical line are usually thought of as being synonymous, and frequently they may be. As long as each line of input contains a single complete command, the logical and physical lines are identical. The separation begins to occur when a command is continued onto a second physical line; here the logical line occupies some part of two physical lines and a "continuation" character is required to indicate this to the system. A third distinction also may be made for those cases in which more than one command is entered on a single physical line; separation between commands then must be made by specifying an "end-of-command" character.

If each command entered is to be executed at once, these distinctions are fairly inconsequential. The need for more exactitude arises when the series of commands is stored and invoked indirectly at a later time. Let us examine a few sample lines:

```
DDEF DDNAME=X,DSNAME=Y,-    (1)
DCB=(RKP=4,LRECL=132,KEYL-
EN=7,RECFM=V),UNIT=(DA,231-
4),VOLUME=(,MYVOL),DSORG=-
VI
```

This is an example of one logical (or, at least, technically logical) command occupying more than one physical line. (Note that the Command System II default character for "continuation" is the hyphen.)

```
DDEF X,,Y ; FTN A ; A             (2)
```

This shows several commands occupying one physical line. (Note that the default character for "end-of-command" is a semicolon.) In this case, all three commands will be executed before a return to the user to prompt for a new command. Sophisticated, well-organized users find this a convenient way to reduce the frequency of interaction with the system and at the same time to achieve efficiency inside the system.

A note here is appropriate. If the amount of useless output can be reduced, the slowest part of any terminal system is the human reaction. Thus, when a user reacts to a system prompt, he should be able to say as much as he wants to.[2] In this way, the user—not the system—controls the scope of his input logic. Many users are frequently annoyed by knowing what they want to do for the next several commands but being constrained to enter each command only when the system feels it is appropriate.

There is no reason to suppose that a user's thought processes are somehow magically related to the physical size of the input device.

```
DDEF X,,SOURCE.MYPROG,UNI-   (3)
T=(DA,2311),VOLUME=(,USERS-
1),DISP=OLD ; FTN MYPROG,Y ;
MYPROG 3,1,7D14
```

Here the user's thought process clearly extends over the physical line limit.

The Command Name

No system need be so presemptuous that the user is constrained to use words that he doesn't like for commands. Perhaps the greatest confusion generator in the past has been to change a command name. Nor has there ever been a set of command names which satisfied more than the designer.

In the non-human environment of the remote job on the distant machine, users suffered strange names gladly in order to get the system power that they represented. Now, in the more nearly human timesharing environment, the system must be prepared to meet the user on his ground, that is, on his terminal. Here the user will be more affected by jargon names because he will use them more often. Attempts to make names more meaningful or palatable in the past have only resulted in a new jargon set to replace the old for most users. Hence, the solution lies in a different direction: Let the user make up his own names.

In Command System II, a synonym capability allows the user to establish the equivalence between character strings of his choosing and the system vocabulary. One restriction imposed is that the new string of symbols cannot be longer than eight characters or contain certain delimiters, such as a blank. For instance,

```
SYNONYM ?@%!=DDEF                    (4)
```

is a perfectly valid synonym, and renames the DDEF command to ?@%! which may be more meaningful to some users.

Of course, this facility does not destroy the original name. Hence, it is possible to create several synonyms for the same command name.

```
SYNONYM DD=DDEF                      (5)
```

will supplement (4) and allow any of the three strings

```
DD                                   (6)
?@%!
DDEF
```

to reference the same command.

The inclusion of this facility greatly eased the implementation of the command system. It was necessary to recognize only a single name for each command, not both a name and its abbreviated form. Nor did the single name have to be esoterically satisfying. The only requirement was that each name be unique and conform to the rules of syntax.

The user, then, is expected to rename the commands in the system to his taste. If these renames look to him like "abbreviations," then that is what they are. It is now his prerogative.

The Operands

The problem of communicating variable data to systems has long been an awkward and sore point. Considerable effort is being expended in the direction of coherent programming to address this problem at the coding level. At the human level, there are as many schemes proposed as there are systems.

One of the basic tenets of the TSS/360 command system is that input should be concise. This is a normal, conservative approach. The design of Command System II is such that the system can be asked to make the maximum number of inferences about the user's needs and desires. A monstrous example of the contravention of this concept is that of (1), above. However, the flexibility of the system allows considerable reductions in this input. The DEFAULT command establishes values to be utilized (inferred) when the user fails to enter them explicitly:

```
DEFAULT DCB=(RKP=4,LRECL-            (7)
=132,KEYLEN=7,RECFM=V)
DEFAULT DSORG=VI
```

Now (1) can be rewritten as

```
DD DDNAME=X,DSNAME=Y,UN-             (8)
IT=(DA,2311),VOLUME=(,MYVO-
L)
```

which also utilizes (5) to shorten the input stream. Suppose the user must enter this command many times. He may eliminate more of this input by recognizing that a positional form of this command is equally valid. Thus, (8) may be rewritten

```
DD X,,Y,,(DA,2311),,(,MYVOL)         (9)
```

But by now a new irritation is creeping in—the intervening separator comma. Although the user has saved key strokes, he has also introduced some uncertainty, not on the system's part but on his own. How many commas does he truly require between UNIT and VOLUME?

There is another way out. The synonym facility used so conveniently for command names can also be used on operand names.

```
SYNONYM B=UNIT,C=VOLUME              (10)
```

allows the user to say

```
DD X,,Y,B=(DA,2311),C=(,MYVOL)(11)
```

which, although slightly longer than (9), is a lot more certain. Note that, throughout the above, the interchangeable use of keyword and positional forms has been acceptable. This greatly enhances the options available to the user. There is no need to restrict the use of positional parameters by requiring that they be entered before keyword parameters, as is the case in the assembler macro language. For instance, (11) could be rewritten as

```
DD DDNAME=X,B=(DA,2311-              (12)
),Y,C=(,MYVOL)
```

It will be noted that Y is still in the third position (it follows the second comma not enclosed in quotes or parentheses).

Frequently the user will want to turn off synonyms and defaults. It is as easy to disconnect them as to create them.

```
SYNONYM B=,C=, DD= ,?@%!=            (13)
DEFAULT DCB=,DSORG=
```

will restore the original state. An equally effective way would have been to LOGOFF and then LOGON again. Since the user had not made these a permanent part of his environment, the equivalences would have been lost.

In order to make synonyms and defaults part of the user's permanent environment, they must be placed in the user profile. This data set is associated with each user as soon as he issues a PROFILE command for the first time. At that point, all the synonyms and defaults the user has established are written into the user profile. From then on, each time the

PROFILE command is issued, a new copy of the user profile replaces the existing one.

It is interesting to note that, in certain procedural syntaxes proposed for command systems, a different approach is to use reserved words rather than keywords.[10] The use of these reserved words does not permit the restructuring of the language as conveniently as the scheme described above. The principle reason is that the reserved word is not associated with a value to be passed but with the command invoked. Thus default and synonym functions are without handles in the operands of the command.

COMBINED DICTIONARY

Having established all these facilities for the user, it is mandatory that a convenient and rapid access to these synonyms and defaults exist. As may well be imagined, a centralized directory is a solution. Originally, the concept was taken to include all the items that the user or the system might want to refer to. This meant commands, stored procedures, data set names, program names, program symbol tables, defaults, etc. It was discovered, however, that the size of this table militated against system performance. The final design, therefore, included only those elements—synonyms, defaults, command variables, textual procedures, and built-in procedures—newly added to the system. This directory is called the combined dictionary.

The concepts of synonyms and defaults and some of their usage have already been discussed. Therefore, only command variables, textual procedures, and built-in procedures will now be covered.

Command Variables

Command variables are a set of values whose names reside in the combined dictionary. Characteristically, users would like to be able to set some storage aside for elements, such as recursion counts, limits, and communication cells, that are not tied to a given program. In Command System II, it is possible to create a command variable by means of the SET command.

```
SET ABLE='NO' , BAKER=10          (14)
```

will establish these two symbols in the combined dictionary, provided there are no external symbols with the same names (ABLE or BAKER) and that these names do not appear in an internal symbol dictionary that is currently loaded. In other words, if the named item does not appear in any of the extant dictionaries, then an entry will be made for it. Once these command variables or symbols have been established, they may be referred to in other commands quite freely; for example,

```
IF ABLE¬='YES'; SET BAKER=B-  (15)
AKER + 1
DISPLAY BAKER
```

If a command references these, they may be reset, displayed, and tested by suitable commands. Of course, each time a reference is made to a command variable, the entire set of pertinent dictionaries is again searched to ascertain that no symbol with the same name has been introduced in a user's program.

It is possible to retire these command symbols in several ways:

```
SET ABLE=,BAKER=                (16)
```

will erase these symbols from the combined dictionary. Another way is for the user to LOGOFF and then LOGON again. Since he had not made these symbols a permanent part of his environment, they will not be reestablished when he starts a new task.

If the user so wants, he may save his command variables in the user profile in a fashion similar to that for saving synonyms and defaults, which was discussed earlier. He accomplishes this by issuing the PROFILE command with the first operand equal to Y:

```
PROFILE CSW=Y                   (17)
```

This writes all existing command variables into the user profile in addition to synonyms and defaults.

The importance of getting this data into the user profile has not yet been made clear. Each time a task is started and the user logs on, his user profile is used to construct a part of the combined dictionary. Thus, the only items with any persistance between tasks are contained in the user profile.

One of the most frequent uses for command variables is as user flags, or latches, to control the action of commands. This use is not yet clear but will become so in the following discussion of procedures.

Procedures

In order to provide as uniform an interface as possible, all the system commands are defined in exactly the same way that users are expected to define their own commands. There are two mechanisms for command creation. The first allows a command to be built up out of existing commands and modules of code. The second allows a new function to be introduced as a command and linked to directly.[5,11,12,13] It has been suggested that the facilities for command creation must be highly guarded in order to keep the system from being overrun with superfluous commands. This view is quite foreign to the basic design philosophy of the command system. The entire thrust of the system is to allow the users to create their own systems from this basic structure.

Textual Procedures

The first and simplest form of command is the textual procedure. This is a series of commands put together to achieve some function. A name may be assigned to this collation and the result will be indistinguishable from one of the system's original set. Let us take a most practical example. As you will remember, we spent some time in the preceding section worrying about the most convenient form of the DDEF command. The conclusion was that there was no really good, simple way to provide all the operands. In the case in question, the definition was of a line data set that the language processors could accept as input and that the text editor could manipulate. The general form of this command is as follows:

```
DDEF DDNAME=?[1],DSORG=V- (18)
I,DSNAME=?[2],DCB=(RKP=4,L-
RECL=132,RECFM=V,KEYLEN=-
7)
```

for a data set residing on public storage. If the data set resides on a private volume, the following fields must be added:

```
UNIT=(DA,?[3]),VOLUME=(?[4]) (19)
```

where the ?[] form indicates the variable information. In the first instance, there are only two values to be inserted, the DDNAME and the DSNAME of the data set. If we assume that the data set should be of a form suitable for the language processors, then the form of the DSNAME is even more rigidly controlled and that operand should appear as:

```
DSNAME=SOURCE.?[2]             (20)
```

Let us create a command with only one variable operand by making the DDNAME the same as the module name, that is, by making the values to be substituted for ?[1] and ?[2] identical. We will call this command EDITPUB and proceed to define it as follows:

```
PROCDEF EDITPUB                (21)
PARAM EDITNAME
DDEF EDITNAME,VI,SOURCE.E-
DITNAME,(RKP=4,LRECL=132,-
RECFM=V,KEYLEN=7)
_END
```

As soon as the command PROCDEF was given, an entry was made in the combined dictionary adding the word EDITPUB to the vocabulary available to the user. By the time the END command had been executed, a copy of the text of this command had been stored in the procedure library, which is a data set much like a macro instruction library except that it has no index. The index to it is maintained in the combined dictionary.

The effect of this command is to allow the user to say

```
EDITPUB MYPROG                 (22)
```

and get the following DDEF command executed:

```
DDEF MYPROG,VI,SOURCE.MY- (23)
PROG,(RKP=4,LRECL=132,REC-
FM=V,KEYLEN=7)
```

But this only solves the problem for the case in which the data set resides in public storage. Of course, it is possible to define a second command for the private-volume case, but there would be a lot of duplication in the second command compared to the first. It should be possible to create a single command that will take care of both cases at once. The significant difference is that in one instance there is no volume information and in the second there is. The unit data is pretty much standard in any given installation, but we should allow it to be changed easily. Let us examine this next sequence in some detail:

```
PROCDEF EDITDDEF               (24)
PARAM EDITNAME,EDITVOL,U-
NIT=XXX
IF'EDITVOL'='';DDEF EDITNAM-
E,VI,SOURCE.EDITNAME,(RKP=-
4,LRECL=132,RECFM=V,KEYLE-
N=7)
IF 'EDITVOL'¬='' ; DDEF EDITN-
AME,VI,SOURCE.EDITNAME,(R-
KP=4,LRECL=132,RECFM=V,KE-
YLEN=7),UNIT=XXX,VOLUME=-
EDITVOL
_END
```

Now this definition may seem rather esoteric, but in fact as we shall see it is quite simple. In the first place, we have defined three parameters; EDITNAME, as before; EDITVOL, to specify the volume of residence of the data set; and UNIT, to specify the residence device when necessary. Note that the third operand is specified in a strange way. The establishment of a dummy operand for internal substitution allows us to use UNIT as a keyword of the new command and yet be able to specify UNIT as a keyword of the resultant DDEF command. We have established by this mechanism that the string XXX is the object of the internal substitution and that the keyword UNIT only appears in calls to EDITDDEF.

The prime consideration is that the absence of volume data means that the data set is to reside on public storage. Thus, the first IF command tests the existence of a real value for EDITVOL. If EDITVOL is a null string (i.e., not explicitly given or defaulted in the combined dictionary), the public form of DDEF will be issued. If the first test is untrue (there is a value for EDITVOL), the first form of DDEF will be skipped. The second IF command must necessarily be true if the first was not and so the longer form of DDEF will be given. It is expected that the user who creates this command would set up a default for UNIT as follows:

```
        DEFAULT UNIT=(DA,2314)               (25)
```

This will allow him to omit the UNIT operand unless he wishes to change the residence device. Now the command given as

```
        EDITDDEF MYPROG                      (26)
```

will achieve exactly the same result as EDITPUB did before, while

```
        EDITDDEF   MYPROG,(,MYVOL)           (27)
```

will result in the following DDEF being issued:

```
        DDEF  MYPROG,VI,SOURCE.MY-           (28)
        PROG,(RKP=4,LRECL=132,REC-
        FM=V,KEYLEN=7),UNIT=(DA,23-
        14),VOLUME=(,MYVOL)
```

which is just what we wanted.

This simple example has only scratched the surface of the possibilities of this textual procedure scheme. Many commands in the system have far more elaborate procedures. The user's imagination is the only limit.

Now it is possible to demonstrate the use of command variables in a textual procedure. Consider the following circumstances: The user wishes to print more than one copy of a particular data set. He would like to be able to specify the number of times explicitly when he names the data set, which he can accomplish with the following sequence:

```
        PROCDEF MULTPR                       (29)
        PARAM NAME,TIMES
        SET DUMMY = TIMES
        PRMULT NAME
        _END
        PROCDEF PRMULT
        PARAM NAME
        IF DUMMY>0; SET DUMMY = D-
        UMMY-1; PRINT NAME
        IF DUMMY¬=0; PRMULT
        _END
```

In this instance, the first procedure is merely the necessary housekeeping required to initialize the command variable DUMMY. The textual procedure PRMULT now uses DUMMY as a basis for decisions. If DUMMY indicates that there are still more copies to be made, it will call PRMULT again to get the next PRINT issued. Thus the command

```
        MULTPR MYDS, 2                       (30)
```

requests two copies of MYDS. First, the value of DUMMY is established as 2 and PRMULT is called with MYDS as the name of the data set. Now the sequence is as follows:

```
IF DUMMY>0                   true
   SET DUMMY = DUMMY-1       DUMMY now = 1
   PRINT MYDS                   First Time
IF DUMMY¬=0                  true
   PRMULT
IF DUMMY>0                   true
   SET DUMMY=DUMMY-1         DUMMY now = 0
   PRINT MYDS                   Second Time
IF DUMMY¬=0                  false
```

and no more recursion takes place. However, it should be obvious this scheme will blow up if the command is given as

```
        MULTPR MYDS, -1                      (31)
```

Although this possibility seems remote, there is no need to permit infinite loops. A better way to do this is as follows:

```
        PROCDEF PRMULT                       (32)
        PARAM NAME
        IF  DUMMY>1  ; SET DUMMY=D-
        UMMY-1 ; PRMULT
        PRINT MYDS
        _END
```

In this case, the data set will always get printed at least once, but more than once only if DUMMY is greater than 1. Here the PRINT commands are collected until as many as are required are present; then, as the system works its way out of the recursive stack, the PRINT commands are executed.

Built-in Procedures

In order to be as efficient as possible but still preserve the centralized syntactical scheme, it was necessary to evolve a mechanism that would provide the operand analysis to resolve synonyms, defaults, and explicit values, and to avoid the necessity of referring to a data set for the operand list.[12] The most desirable circumstance would be to have each hard-coded command specify the keywords and the order in which they were to be treated positionally. This implies that the necessary information would be a part of the coded module itself. From that idea the rest followed. We define an entry point in a section of the object code into which data may be written. Enough space is set aside to allow each operand to be represented by a pointer, and with each such set-aside position a keyword equivalence is given.

Thus evolved the concept of a built-in operand descriptor and linkage pointer. Since the system would insist on a rather easily followed but fairly rigid linkage, a macro instruction called BPKD (built-in procedure key definition) is provided. All that remains is to give the macro instruction the internal pointers to the keyword character strings to be used in the linkage and the macro instruction generates the rest. The final touch is to allow the specification of the entry point to be used when the linkage is filled in. The idea is that it would be possible to have several "built-in" linkages in a single module without confusion.

So far this has taken care of the code. But how is the system to know that for a given name a special piece of code exists? For this part of the linkage, a command called BUILTIN is provided. It creates an entry in the combined dictionary and puts a remembered version into the procedure library. The dictionary entry consists merely of the name associated with the command and the entry point containing the BPKD description. Thus, the dictionary entry is sufficient to find the right piece of code and to retrieve the operand list from that code. Once the operand list is obtained, the resolution proceeds just as though the keywords were on a PARAM line of a textual procedure. The only difference occurs when all the operands are resolved; then the system passes control directly to the object code involved at the entry point specified without further ado.

Perhaps this sounds too elaborate to be quite efficient. The alternative is to allow each piece of object code to set up its own analysis of the input operands and their defaults. However, the loss to the system is immense. No longer will a single set of rules obtain. The user will have to remember that this command needs one form and that one another form. Further, the centralized mechanism will be lost and unavailable to users who need to construct new commands. Last, and perhaps least important, is that the analysis code will be repeated endlessly–in every command module that needs to resolve operands.

Dictionary Handlers

This, then, is the sum of all the types of entries that reside in the combined dictionary. In order to manage this dictionary, a set of generalized dictionary handlers have been developed. Although the only dictionary they will work on now is the combined dictionary, they are not locked to it and can be used on any dictionary the user desires. All that needs to change is the pointer to the beginning of the dictionary.

LANGUAGE = SYNTAX + DICTIONARY

A language definition has always depended upon two elements. First, the language must have form. In general, this means that it must have an agreed upon set of syntax rules. Of course, punctuation and juxtaposition are the cornerstones, and the concept of operators and operands fill most of modern grammar.

Second, and perhaps more imperfectly understood, a language must be capable of conveying meaning, or intelligence. The only way in which this may be achieved is to have semantic values associated with the symbols of the language. Then the rules of syntax become useful.

It is just such a combination of grammar and semantics that define most languages. Recently, the world of programming has shown a tendency to treat these concepts lightly. The imprecise definitions associated with programming languages have hindered the development and exchangeability of most of the "languages" currently in vogue. In the case of Command System II, a conscious effort has been made to achieve a clean structure. The rules of syntax dictate the "grammar" of Command System II, and the combined dictionary entries give semantic value to the terms used.

These two elements taken together represent the language definition of Command System II. However, it is not enough to have a language. Those command systems which only define a language are woefully incomplete. Most languages are considered dead unless they are in use, and a command system must be considered in this light. Without an underlying system that can comprehend and react well with the language, no purpose is served. As will be seen, some considerable thought has been devoted to the creation of a meaningful foundation to this system. No great amount of detail will be expended on those parts of the underlying system which are essentially unchanged from previous systems. Rather, we shall now consider those new or vastly improved areas.

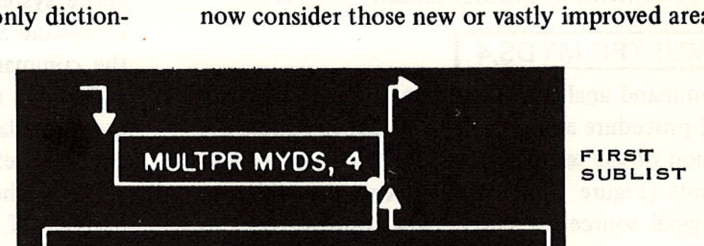


Figure 1. Initial sublist expansion of the MULTPR textual procedure.

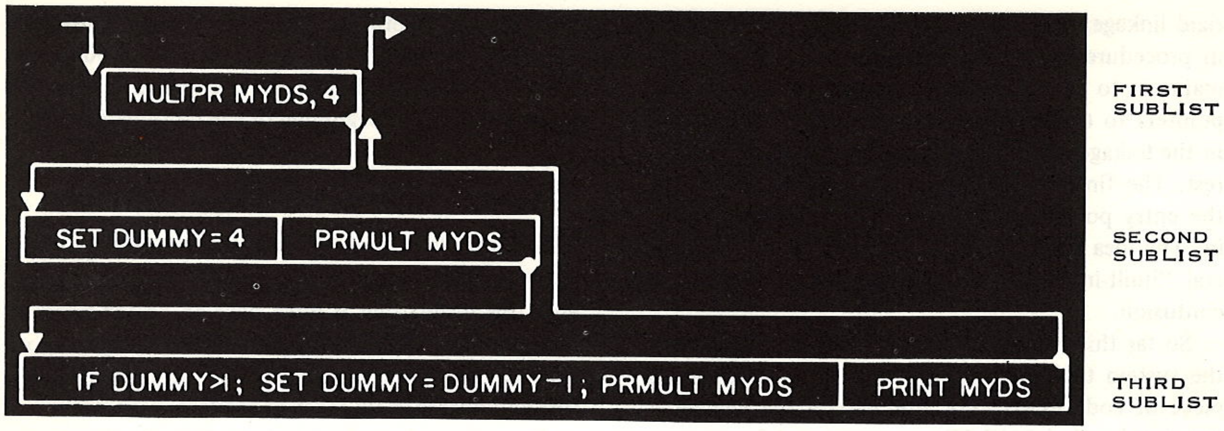


Figure 2. Continued expansion of the MULTPR textual procedure, including PRMULT, into sublists.

COMMAND STREAM

Throughout the preceding sections, much has been said about how commands are interpreted and executed. In particular the expansions of textual procedures make a number of "sub-commands" available at once. You will remember that in the multiple print procedure–examples (29) through (32)–the PRINT commands themselves were "saved up" until all of them had been accumulated and only then were they executed. In a system that allows the nesting of commands or procedures, the problems of switching the attention of the system in and out of these convoluted schemes is at best difficult. In an effort to make this as easy as possible for users to comprehend, the system provides a set of "source list" handlers. These complement the dictionary handlers in that they deal with lists rather than tables.

In order to make this clear, let us examine the "recursive" procedure used to exhibit multiple print requests. If we wish to follow this expansion, it is necessary to invoke the MULTPR procedure–example (29) as modified by example (32)–with a set of explicit parameters:

MULTPR MYDS,4 (33)

When this is taken into the system, it occupies a position in the source list. If it came from the terminal and was the only command on the logical line, it is the entire contents of the first sublist.

$\boxed{\text{MULTPR MYDS,4}}$

The command analyzer recognizes that MULTPR is a textual procedure and prepares a sublist to handle the expansion of the set of commands that the procedure represents (Figure 1); it then connects the sublist to the original source list entry. Now control passes to the SET command and after that to the PRMULT procedure. Again the command analyzer prepares a sublist for the expansion of PRMULT (Figure 2).

This process continues until each command on some sublist can be executed without resorting to another sublist. At that point, the sublist at the bottom is exhausted and control reverts to the next higher sublist. The entire sequence will look something like Figure 3. It is clear that the four PRINT commands are executed last as control passes back through the six sublists. In order to emphasize the sequential nature of this event, the various values of DUMMY as the source list is executed are shown in the circles adjacent to the occurrences of DUMMY in each sublist.

The source list handlers are generalized to manipulate any list-structured string. All that is required to make them work on strings other than command sequences is a pointer to a new string origin. Since we allow procedures to invoke procedures, the source list must be structured to permit "levels" of commands to exist. Thus, each call on a textual procedure is taken to require a new level. The completion of the execution of commands at any level is a signal to revert to the previous level. This push-down/pop-up scheme is not new. What is new is to make the handling of commands nestable. Many systems in the past have recognized single-thread (level) command sequences. A few, notably the TX-2 RECKONER, have allowed the invocation of a new level of the entire system. The essential features that are new in Command System II are that the push-down level of the command sequences is independent of all other activity in the system, that all levels have access to the same data, and that the degree of involution does not affect execution.

Once the mechanism is established, there are a number of interesting side effects that bear noting. For example, it is now possible to call the command system recursively. That is, the command system can be invoked from user's object modules to execute commands. At the completion of an exercise such as this, the system will revert to the object code at the next level. This produces the effect of making any command or command procedure, either user or system provided, available in a macro instruction form. The technique here is to create a new sublist upon the occurrence of the OBEY macro instruction call. This sublist is isolated from the rest of the command source list in that its predecessor list is object code and not another list. Thus, the command invoked via OBEY may in itself create as many sublists as necessary and then work its way back out again. When it is finished and the created list is complete, the return is not to the next higher list but rather to the object code from which the OBEY call occurred.

Another benefit possible is that the occurrence of an attention interrupt allows the system to create a new sublist. In this case, the commands the user enters are kept on this sublist level until the user elevates the system to the previous list by use of the GO command. In this way, the user can intervene in a command sequence and start a new sequence of commands. This sequence will continue until the user explicitly asks the system to revert to the higher level. There is no requirement that the user ever revert to the original list, however. Our experience to date has been that over a period of some time the user gets deeper and deeper into sublists. The only cost of this process to the user is that somewhere in his current virtual memory reside the unfinished ends of these suspended strings. If no reference is made to these, they will not intrude and they will be eliminated at LOGOFF.

When a sublist is completed, the space it occupied is freed to be reused by the source list handlers. In fact, the source list handlers are quite conservative of space. Even parts of lists that have been executed are available for reuse. The only time that the system will require very much space for the source list is in the case in which a great number of commands are stacked up at once for eventual execution or in the case in which the user has made extensive use of the attention button to leave uncompleted sublists around.

The concept of the source list will eventually allow the addition of soft copy devices, such as graphic terminals. All that is needed is the added feature of saving the lists before executing them. Thus, the remember function, or audit trial, can be internally collected without any great fuss.

The way the system operates is that the command analyzer tries to find work in the source list. If none remains, then the appeal is to the SYSIN to get the next command. Of course, the user has the option of

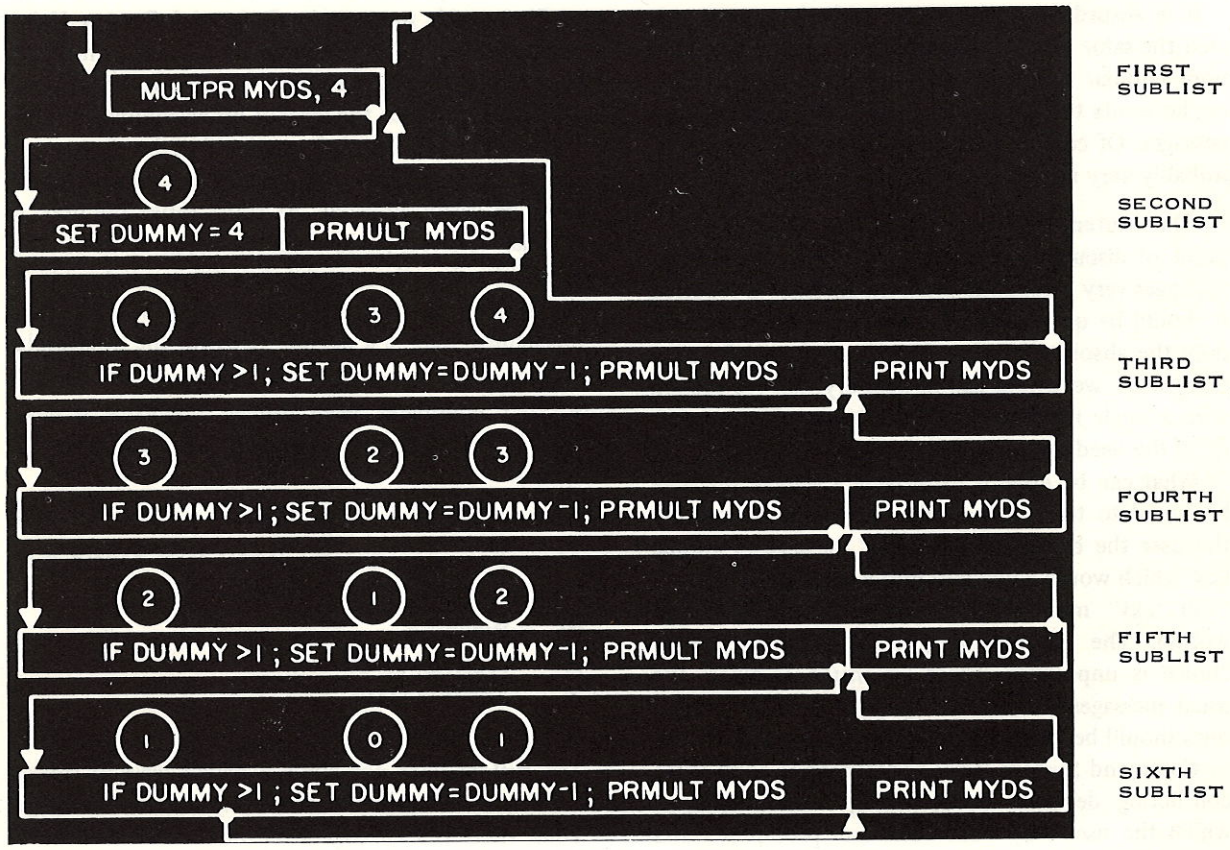

MULTPR MYDS, 4
FIRST SUBLIST
4
SET DUMMY = 4
PRMULT MYDS
SECOND SUBLIST
4
3
4
IF DUMMY >1; SET DUMMY=DUMMY-1; PRMULT MYDS
PRINT MYDS
THIRD SUBLIST
3
2
3
IF DUMMY>1; SET DUMMY=DUMMY-1; PRMULT MYDS
PRINT MYDS
FOURTH SUBLIST
2
1
2
IF DUMMY >1 ; SET DUMMY=DUMMY-1; PRMULT MYDS
PRINT MYDS
FIFTH SUBLIST
1
0
1
IF DUMMY >1 ; SET DUMMY=DUMMY-1 ; PRMULT MYDS
PRINT MYDS
SIXTH SUBLIST

Figure 3. Sublist structure for the MULTPR textual procedure when DUMMY is initially set to 4. The value of DUMMY as the source list is executed is shown in the circle above that symbol in each sublist.

fetching his input from the source list or the SYSIN at will. If his data is to be generated by a procedure, it may well be that the data will reside in the source list. In other systems, the user is constrained to access data from outside the system, either from the equivalent of a SYSIN or from an independent data set. Seldom is the user free to generate data as though they were commands and then access them internally. The advantage of this scheme is that the distinction between commands and data disappears. The user, then, is perfectly free to construct a subsystem using large parts of the existing structure. The input to the subsystem may appear just as a command string to the user and all the facilities of textual procedures are available to his subsystem.

MESSAGE HANDLING

The perennial problem of when to publish a message to the user and how much to say had plagued interactive systems from the beginning. There are several schools of thought, and each disagrees violently with the others.[2,14] One of the problems is to describe the different environments surrounding users. These range from the neophyte who has never engaged in a conversation with a system to the ultrasophisticated user who has had a wealth of experience over several years.

It is absurd to suppose that both these extremes need the same level of messages. As a general rule, the more familiar a user is with interactive systems the less he wants the system to intrude with unnecessary messages. Of course, what he considers unnecessary is probably very necessary to the new user.

The content of the messages themselves is also a point of discussion. There are those who don't want messages very often, but when the message is required it should be quite explicit. There are those who want only the absolutely most abbreviated form of a message, what we would call at best cryptic. No system with a single message handling scheme will ever satisfy all the needs expressed by users.

What can be done is to provide a vehicle that can be tuned to the user's needs. Older systems offered the user the choice of a "coded" message (a unique key which would guide him into a message book) or a "full text" message (a paragraph of richly chosen words). The all-or-nothing flavor of this type of choice is unpopular. There are instances where the usual messages should be very short but the unusual ones should be long enough to be "meaningful."

Command System II has endeavored to solve these conflicting demands by providing a skeleton upon which the user may build. The user prompter starts with the assumption that all messages can be categorized into severity classes:

- *Informational:* a message which may tell the user something about system activity but will have little or no effect if left out.
- *Warning:* usually a diagnostic message that might reveal some symptom the user should be aware of; certainly not so serious that the system should reject the current action.
- *Normal error:* a message issued concurrently with termination of the current command but not indicative of system problems, such as when the user misspells the name of his program or tries to do something entirely illegal; the large majority of messages in any system seem to fall into this category, usually because the message fabricators tend to overestimate the severity of the conditions that gave rise to the message.
- *Serious error:* a message indicating the system has detected a symptom that places the user's task in grave danger of being aborted; in general, such messages should engender a vigorous response from the user.
- *Terminal error:* this is the bitter-end message; it is the one which sadly informs the user that his task is dead. There are probably two levels even here—the one which involves only one user and the major disaster which crashes the entire system.

Since each message in Command System II has been classified in this fashion, the user may choose to suppress all messages below a suitable level of importance. As he raises the level of his acceptance, he gets correspondingly less information but he does speed up the action. Naturally, the choice of threshhold must be easily adjusted. At some times the user, in familiar territory, will want to suppress most messages; at other times, using less frequented parts of the system, he will want more data to go on. The mechanism here is a default value for the threshhold. This implicit operand is set and reset by the DEFAULT command, even though the value is never used in any command. To get all the messages in the system

```
DEFAULT LIMEN=I                    (34)
```

where I specifies that the lowest level desired is informational. The other codes are:

W	Warning
N	Normal error
X	Serious error
T	Terminal error

The system does not allow a higher class of threshhold, one which could block the terminal message. Thus, there is a message level that can be counted on to get through to the user in any circumstance.

This scheme, however, still does not address the problem of how much to say when the message is finally issued. To exert some control, two mechanisms are employed. The simplest is a variation on the coded/full dichotomy. Now we recognize three categories of length. In the shortest, we issue only the message identification; this is a unique string of characters and will always be associated with some text. Next we recognize a normal message; that is, a short sentence which provides a synopsis of the problem or event being reported. The third category is the extended message; here the emphasis is on completeness and clearness. The user may choose any one of these lengths and can shift back and forth with another default:

```
DEFAULT BREVITY=S                    (35)
```

where S stands for standard, or normal, message. The other codes are

E	Extended messages
M	Message identification only

But most users are not satisfied to set this threshhold and live with it. Most want a short message of some sort with the possibility of obtaining a longer version if the short message is unclear. To provide this function, we have incorporated an explanation scheme in the message handler. Thus, it is possible to place explanation messages—which will never appear except on user demand—into the message file. Then, when the meaning of a message is unclear to the user and he wants elaboration, he says:

```
EXPLAIN                              (36)
```

and the system will retrieve the explanation message with the same message identification as the last-issued message. If there is no explanation message, the system says so.

Once we had gone this far, it was easy to add a word-explanation system. Thus, in a message, certain words may be flagged as "explainable." For example, the user can say

```
EXPLAIN DSNAME                       (37)
```

and, if the word is explainable, the system will retrieve the word explanation associated with DSNAME. Of course, this explanation is keyed to the last-issued message again. In this vein, the user is allowed to indicate the scope of his explainable word. He does this by assigning it a message identification which is all or some part of the original message. Since message identification is unique, the assignment of message identifications may be regularized and assigned in some logical fashion. In this sense, the scope is indicated much in the same fashion as a Dewey decimal classification is assigned: the more places in the identification, the finer the scope.

Lastly, the user may not want to refer to the last-issued message. for this he must say

```
EXPLAIN TEXT CZASA037                (38)
```

where CZASA037 is the identification of the message he wants the explanation for.

By such schemes, the wealth and power of a message file is materially enhanced. The user has been given some control of his environment again.

In all the foregoing it was assumed that the texts of messages were fixed in the system. This is quite natural, since no system in the past has been otherwise endowed. Now we add the last bit of the message-freedom bill. The user may create his own message file and expect the message-handling facilities to operate on it just as well as they do on the system message file.

Even better, the user may not like the language or phrasing of some of the system messages. He may then overlay these by creating in his own message file a new message with the same identification as the system message. The user prompter will select his message in preference to the system version. The message will still be issued under the same conditions, but the text is now what the user wants to see and not what the system presumed was meaningful.

The system does not incur the extra cost in time if there is no user message file. It examines the user's environment at LOGON and, if there is then no user message file, it will not try to search it. This means that the user will have to LOGOFF and then LOGON after he has created his message file before it becomes available to the system.

EPILOGUE

The entire thrust of the foregoing scemes has been to provide a well-defined structure for the user while allowing him to control his environment. There have been no assumptions made about the kinds of useful work the user may wish to accomplish. Such assumptions are dangerous and narrowing. There is every indication that, as in all preceding systems, the pattern of usage will grow and change as users begin to really investigate the interactive system. Since this is the case, the system must be prepared to adapt to the new applications.

Yet the system must not be so loosely defined that simple things cannot be done simply. It is imperative that the system be able to make inferences about the user's wishes. The system must never be at a loss as to what to do next. There should be no action that is barred to the user or that leads the system into omphaloskepsis.

Many systems in the past have exhibited a high degree of overprotection. The user was forbidden to do things on the basis that "he had no need to do that" or that he might damage his environment. This kindergarten approach must be a part of the past, and users must be presumed to be willing to accept the consequences of their own actions. Command System II will treat the users as adults capable of creative thoughts. In this way, we may learn something about the future directions of interactive systems.

CITED REFERENCES

1. *IBM System/360 Time Sharing System: Command System Users Guide,* C28-2001-2, IBM Data Processing Division, White Plains, N.Y.
2. J.I. Schwartz, *Observation on Time-Shared Systems,* SP-2046, System Development Corporation, Santa Monica, Calif., Sept. 15, 1965.
3. M. ben-Aaron, et al, *Design Specification for 360/67 Editor System (EDS),* TR 1412-TR-1, Auerbach Corporation, Philadelphia, Pa., Feb. 1, 1967.
4. C. Weisman, *Context: A Proposed Edit Program for IBM S/360 (TSS),* TM-3134/000/00, System Development Corporation, Santa Monica, Calif., Sept. 13, 1966.
5. M. Greenberger, et al, *The OPS-3 System for On-Line Computation and Simulation,* M.I.T. Press, Cambridge, Mass., 1965.
6. B. Arden, "Some Notes on the Time-Sharing Requirements of the University of Michigan," University of Michigan, Ann Arbor, Mich. (unpublished).
7. "GMR Time-Sharing System Capability Requirements," General Motors Research Corporation, Warren, Mich., Sept. 6, 1966 (unpublished).
8. J.F. Lubin, "Some Observations on SDS-940 'Time Sharing' System," University of Pennsylvania, Philadelphia, Pa., July 15, 1966 (unpublished).
9. T.A. Dolotta, and A. Irvine, "Proposal for Time-Sharing Command Structure," SHARE SSD, Aug. 30, 1966 (unpublished).
10. N. Rochester and P. Woon, "A PL/I-Based Terminal Language for the Allen-Babcock Conversational Programming System," IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. (unpublished).
11. B.M. Leavenworth, *Syntax Macros and Extended Translation,* IBM TR17-196, IBM Advanced Systems Development Division, Yorktown Heights, N.Y., May 1966.
12. A. Stowe, D. Yntema, et al, "The Lincoln Reckoner," MIT Lincoln Laboratory, Lexington, Mass., July 1966 (unpublished).
13. J.B. McKeehan, "A Proposed Desk Calculator Language," IBM Systems Development Division, Yorktown Heights, N.Y., June 16, 1966 (unpublished).
14. C. Weissman, *Communication Technique for an International Time-Sharing Users Group,* SP-2029, System Development Corporation, Santa Monica, Calif., Apr. 8, 1965.

APPENDIX D

NOVEMBER 25, 1968

TR53.0004

VIRTUAL MEMORY IN TIME SHARING SYSTEM/360*

by

O. W. Johnson and J. R. Martinson

ABSTRACT

This report describes the evolution of the virtual memory concept through the point of implementation in the IBM System/360 Time Sharing System (TSS/360). The characteristics of virtual memory as implemented in TSS/360 are discussed. Segmentation as a programming concept, the loading and binding of programs, and the allocation of memory are contrasted to conventional systems. The advantages of working with a virtual memory are presented together with some examples of its use.

**This report, under the same title, was presented at the IBM Systems Development Division Programming Symposium, Atlantic City, N.J., Sept. 24-27, 1968.*

Descriptors: IBM Technical Information Retrieval System

Virtual Memory
Time Sharing
TSS/360
Memories
Memory Allocation
Memory Relocation
Space Allocation
Segmentation
Address Translation
Dynamic Address Translation
IBM System/360
Programming
07 Computers
21 Programming

IBM®

Systems Development Division · Time Sharing Department · Mohansic Laboratory

Yorktown Heights · New York

INTRODUCTION

Since the advent of computer operating systems, their development has been in two distinct directions: first, to efficiently use machine resources; second, to provide a better computing facility for the users. It has become apparent that multiprogramming can be used to improve computer-resource utilization. At the same time, it has been learned that the solution of many user problems is hampered by the limited addressing capability of the computer, which requires planned overlays and data organizations that can effectively use them. On the other hand, multiprogramming efficiency is critically dependent upon any program using a minimum amount of main storage, or real memory, so that as many programs as possible can be coresident and, therefore, multiprogrammed. Virtual memory offers a solution to both the efficiency-minded computer-center manager and the problem-oriented user, for it tends to maximize resource utilization while it minimizes real memory requirements.

A virtual memory is a zero-origined (beginning at 0) sequential address space that is dynamically and continuously mapped, as it is referenced, into the address space of real memory. The size of a virtual memory is not limited to the availability of real memory and sequential virtual memory addresses need not map to sequential real memory addresses.

Such a virtual memory has been implemented in the IBM System/360 Time Sharing System (TSS/360). The utility of this virtual memory is being explored by IBM and customer alike.

ADVANTAGES OF VIRTUAL MEMORY

Virtual memory purports to solve two problems. One is that it separates real memory management from program address space needs. The other is that it increases the program address space available.

In non-virtual memory systems, the programmer is the manager of real memory. He must estimate how big his program and data are and make arrangements

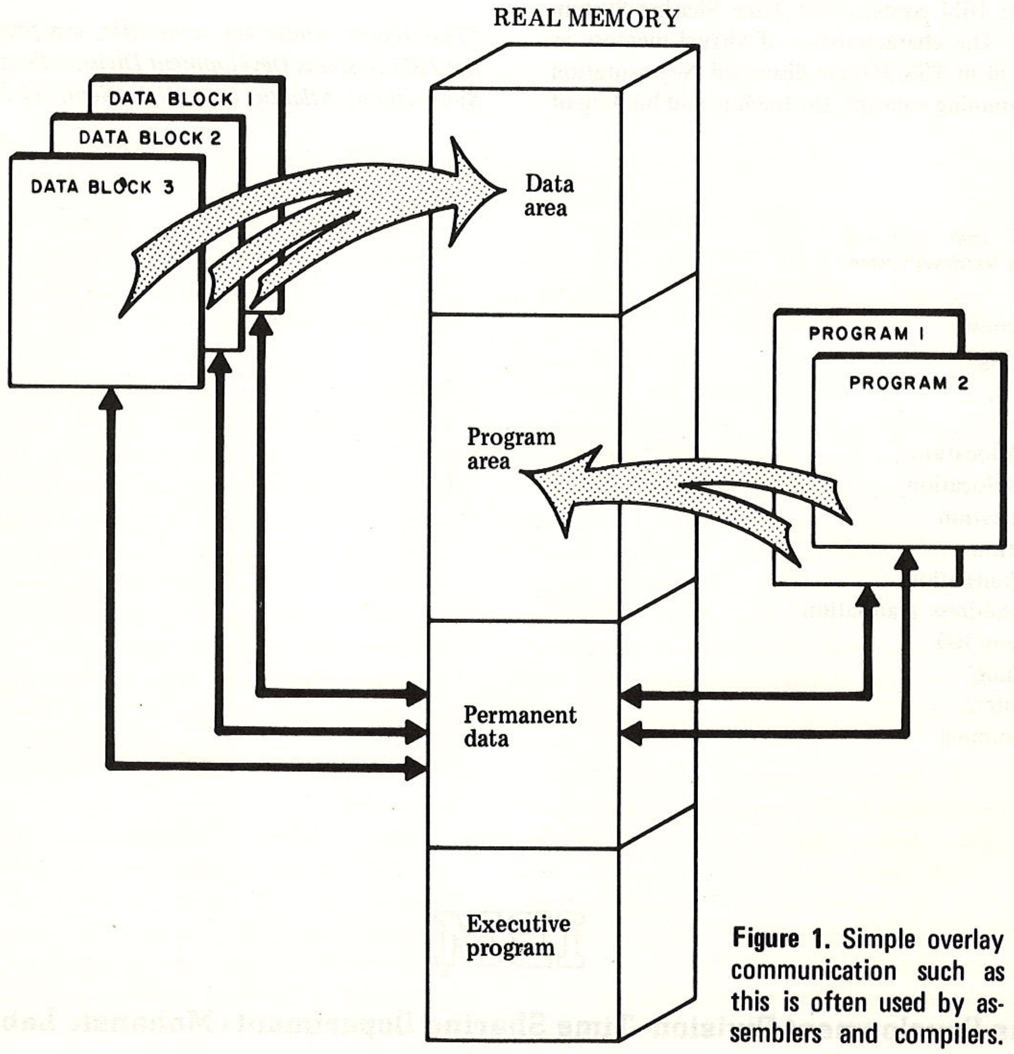


Figure 1. Simple overlay communication such as this is often used by assemblers and compilers.

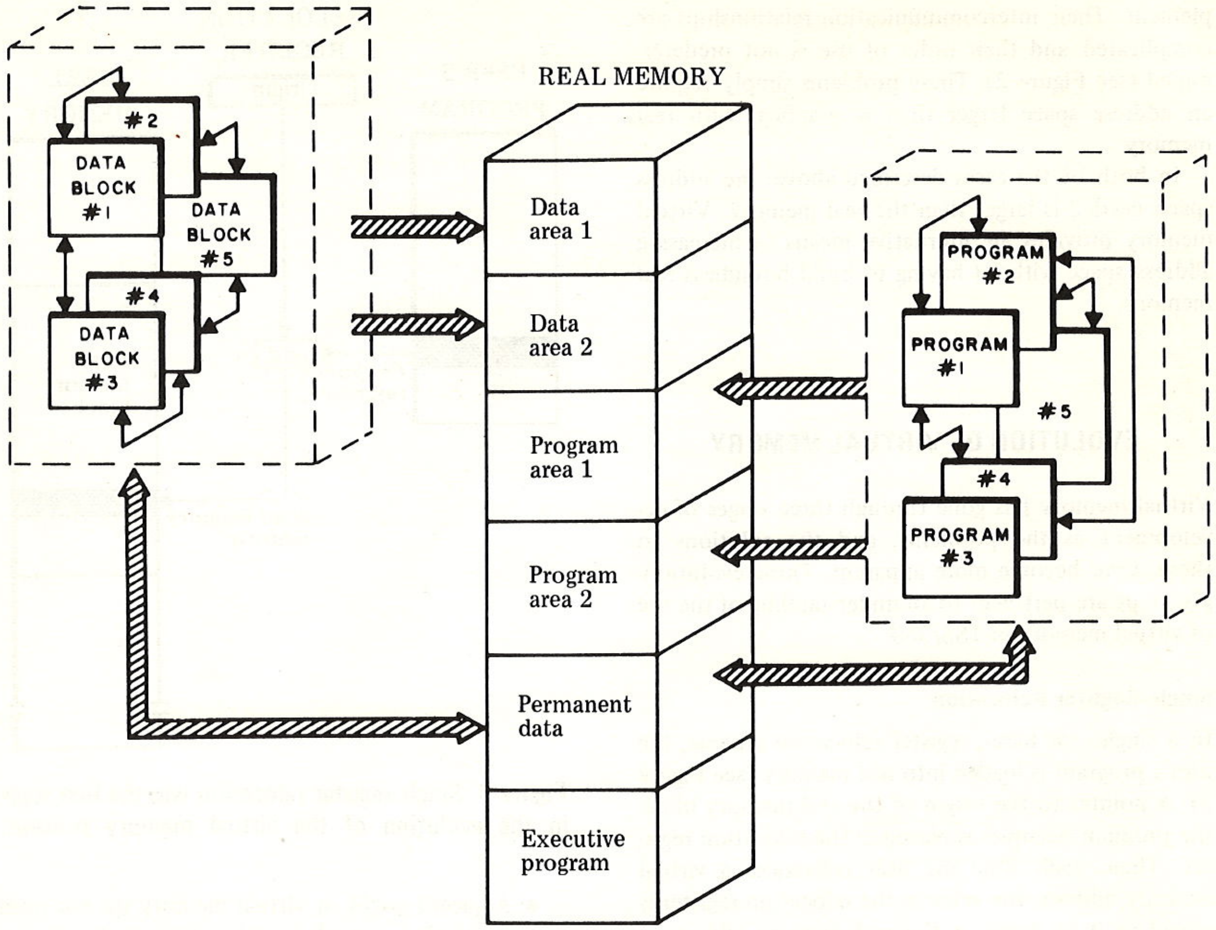


Figure 2. Complex overlay communication is typical of contemporary problem programs. Since data can require coresidence [*left*], as shown by the solid arrows, the data blocks cannot be overlayed in a simple sequence; since the programs call each other in a complex sequence [*right*], as shown by the solid arrows, the programs cannot be overlayed in a simple sequence.

to have the space (real memory) allocated. However, the estimates are difficult to make, especially if the program space is data dependent. Furthermore, he is usually penalized for making bad judgments. First, if the program requires more space than he defined, it is thrown off the machine. Second, if he requests more space than is needed, his job is apt to get a low priority and a long turn-about time. In addition, the system may run poorly as a result of his over-specification of space needs or even because he needed the full amount of space for only a portion of the computation time.

From the system's point of view, it has the problem of accomplishing multiprogramming. It must take the information it demanded of the user and manage the real memory space as part of its job-scheduling function. The system has no real knowledge of the actual address space being used, but only knows what the user has told it.

In virtual memory systems, the address space is always of the size needed to solve the problem. The mechanism used to map virtual memory into real memory can be as dynamic as necessary to suit the system's needs. With virtual memory, the user can penalize neither his program's operations nor the system's efficiency because his use of address space is not a binding decision to allocate that much contiguous memory. Even a dynamic increase in virtual memory address space imposes no problem to the system's real memory management facility.

Often, an algorithm requires more address space than is available with real memory. The commonly used technique for overcoming this problem is to plan a set of overlays, which come in two forms, instructions and data.

Some algorithms are easily broken down into overlays. For instance, assemblers and compilers are readily broken into phases and often break data into overlay blocks. These have the general intercommunication relationship illustrated in Figure 1.

However, a class of problem programs has emerged that makes preplanned overlays very difficult to implement. Their intercommunication relationships are complicated and their order of use is not predetermined (see Figure 2). These problems simply require an address space larger than is practical with real memory.

In both of the cases described above, the address space needed is larger than the real memory. Virtual memory provides an alternative means of increasing address space without having to build boundless real memory.

EVOLUTION OF VIRTUAL MEMORY

Virtual memory has gone through three stages of development as the problems, and the solutions to them, have become more apparent. These evolutionary steps are pertinent to an understanding of the use of virtual memory in TSS/360.

Single-Register Relocation

In a single-, or base-, register relocation scheme, the user's program is loaded into real memory (see Figure 3). A pointer to the origin of the real memory block the program occupies is placed in the relocation register. Then, each time the user references a virtual memory address, the value in the relocation register is added to it to arrive at the real memory address at which the reference is located.

Although it can be debated whether single-register relocation is a form of virtual memory, it is safe to say that this technique separates the address space defined by the user from the actual memory space to be used.[1] However, in this scheme, virtual memory was always confined to an area smaller than the system's available real memory and therefore did not make any headway in solving the user's overlay problem.

One-Level Page Table

The basic step in the creation of a virtual memory is the definition of a table. The index to the table is a virtual memory page number; the value in the entry is the real memory origin of the corresponding virtual memory page. As part of the table definition, each entry must contain an indicator of whether or not that virtual memory page is actually in real memory (see Figure 4).

By establishing such a system, we have created an address space which is logically contiguous, although not necessarily physically contiguous. It is divided into pages, each of which is allocated to real memory independently. This paged virtual memory has the following properties:

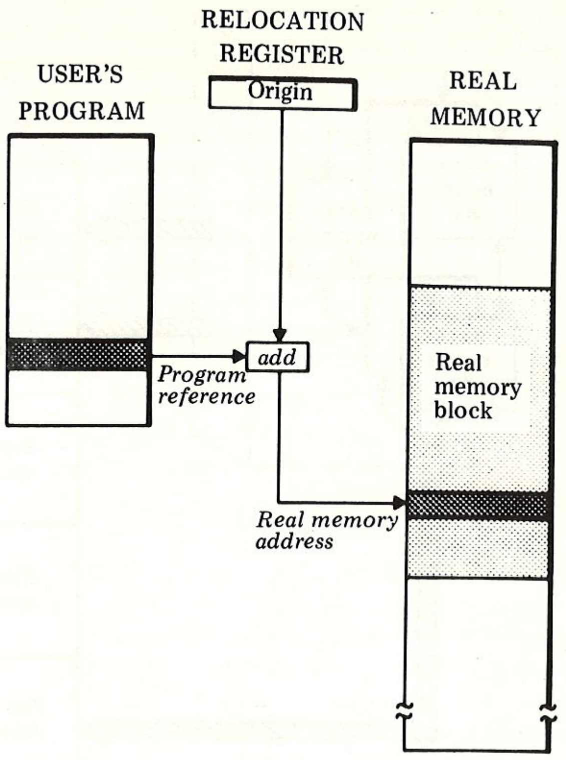


Figure 3. Single-register relocation was the first stage in the evolution of the virtual memory concept.

- Adjacent pages in virtual memory do not need to be allocated adjacent blocks in real memory. Real memory management can limit itself to handling only page-size blocks and thus solve the fragmentation problem.

- Not all of a virtual memory is needed in real memory simultaneously.[2] Therefore, the system incorporates a mechanism that interrupts execution when the indicator for a page being referenced shows that the page is not in real memory. The system can then use such an interrupt as a signal to bring the referenced page into real memory. This means that the system can allocate real memory as it is needed by the virtual memory program execution rather than as address space is requested by the program.

- The available address space is expanded to the smallest of (a) page-table size limitations, (b) CPU instruction-addressing capability, or (c) auxiliary space (drums, disks) that must hold a copy of that portion of virtual memory not now in real memory.

- The system has the complete responsibility to detect the real memory needs of the problem program and manage real memory accordingly.

The one-level-page-table implementation of virtual memory, although it provides advantages over previous systems, has the following limitations:

- Simultaneous access to the same data by different virtual memories is not accomplished.
- Some algorithms require variable size data spaces depending upon the data to be processed. (For example, the size of a FORTRAN compiler symbol table depends upon the program being compiled.) Leaving a large enough space to accommodate the maximum symbol table would leave holes in the page table for most users.
- The page tables could themselves require large contiguous blocks of real memory.

Two-Level Page Tables

A two-level page table (see Figure 5) is designed to overcome these problems.[3,4] It defines a segment table with each of its entries pointing to a page table. Each page table is constructed as in the one-level page table case. The virtual memory is thus divided into segments, each of which is divided into pages.

Simultaneous access to a segment by different virtual memories (sharing) is provided by allowing different segment tables to point to the same page table. In this way, the segment is simultaneously accessible to all those virtual memories whose segment tables contain a pointer to the shared page table. This facility allows selected users to share portions of the same virtual memory.

Variable-length data spaces are allocated in such a manner as to begin on segment boundaries. Thus, a page table can be made to grow as space is needed. However, page tables are limited in size (to the size of the segment) and can never require large, contiguous blocks of real memory.

CHARACTERISTICS OF VIRTUAL MEMORY IN TSS/360

The TSS/360 virtual memory is created using the two-level-page-table structure inherent in the design of the IBM System/360 Model 67. The characteristics of the two-level structure, combined with the attributes of TSS/360 software, provide the following characteristics of TSS/360 virtual memory.

Segmentation

A virtual memory broken into several pieces, which in some ways can be treated as different virtual mem-

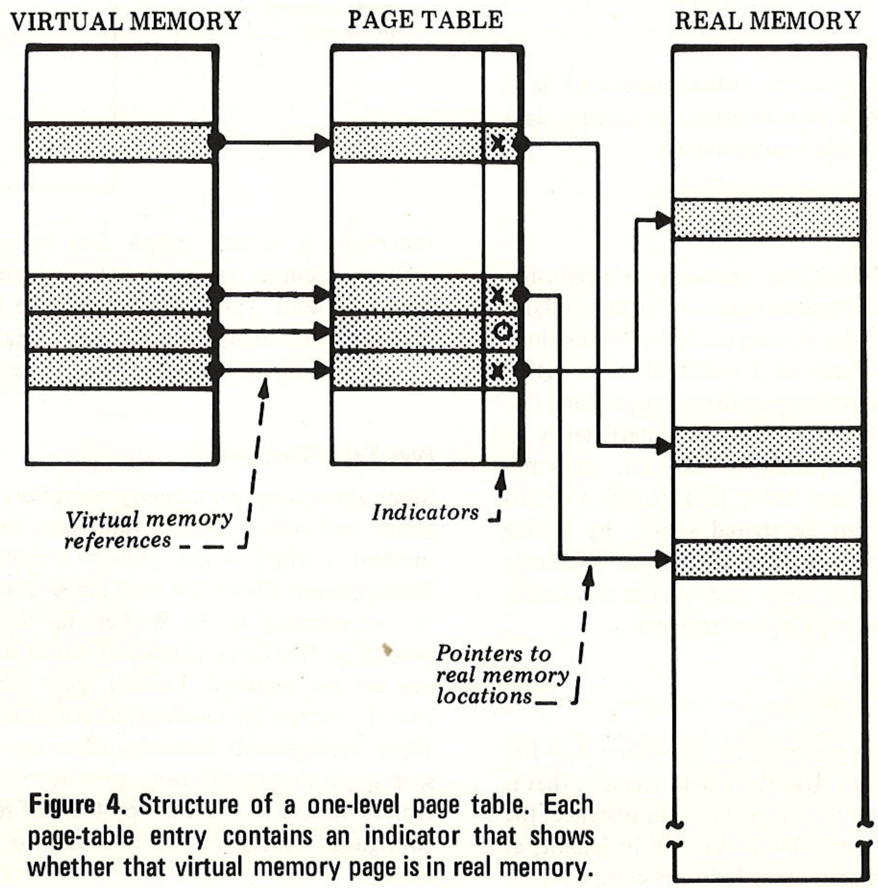


Figure 4. Structure of a one-level page table. Each page-table entry contains an indicator that shows whether that virtual memory page is in real memory.

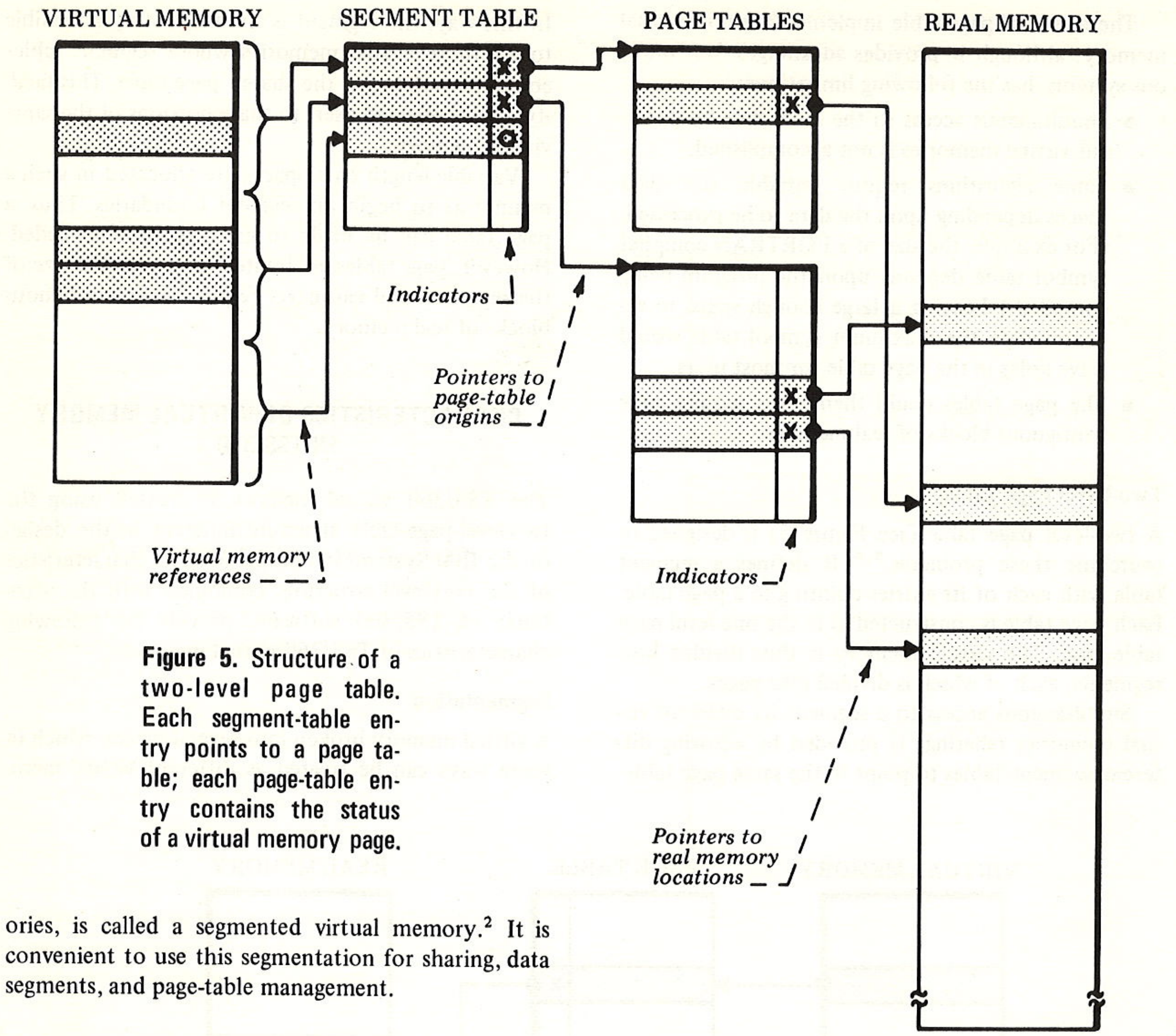

VIRTUAL MEMORY
SEGMENT TABLE
PAGE TABLES
REAL MEMORY
Indicators
Pointers to page-table origins
Virtual memory references
Indicators
Pointers to real memory locations

Figure 5. Structure of a two-level page table. Each segment-table entry points to a page table; each page-table entry contains the status of a virtual memory page.

ories, is called a segmented virtual memory.[2] It is convenient to use this segmentation for sharing, data segments, and page-table management.

Sharing

In designing TSS/360, it was necessary to establish a mechanism for simultaneous access to both programs and data. Although this sharing could have been done on a page-by-page basis in a one-level-table virtual memory, all known techniques for accomplishing this would have introduced considerable inefficiency in the supervisor by requiring a separate, software-implemented shared-page table. In TSS/360, a virtual memory segment can be shared simply by having segment-table entries for separate virtual memories point to the same page table. Each user in this case is allowed to share any page in that segment.

Data Segments

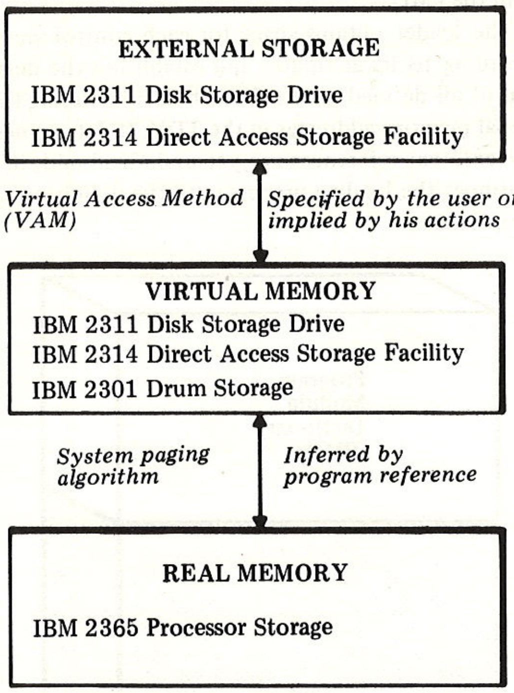
In the course of programming an algorithm, it is frequently convenient to describe a data segment that is variable in length (up to maximum). For instance, the symbol table of a compiler is variable in length, at least until the first-pass compilation is complete. By describing a variable-length data segment, the real memory address space is not necessarily allocated until it is used. The ability to describe such data segments considerably simplifies the table-allocation problems inherent in conventional systems.

Page-Table Management

Since a user's virtual memory comprises both his programs and certain system functions, such as access-method routines, it can grow to a considerable size. Segmentation allows the page tables describing such a virtual memory to be broken up into convenient groups, so that large chunks of contiguous real memory are not required. Further, page tables need not contain entries for unallocated portions of segments. These management functions allow the supervisor to save space that would ordinarily be required for page tables and also to avoid the problem of requiring large contiguous blocks of real memory to contain page tables.

Data Sets

It is often necessary to assign an entire data set to a portion of virtual memory in order to conveniently address it. The TSS/360 virtual access method allows the user to specify that a data set be assigned to a segment for convenient processing.[5]



EXTERNAL STORAGE
IBM 2311 Disk Storage Drive
IBM 2314 Direct Access Storage Facility
Virtual Access Method (VAM)
Specified by the user or implied by his actions
VIRTUAL MEMORY
IBM 2311 Disk Storage Drive
IBM 2314 Direct Access Storage Facility
IBM 2301 Drum Storage
System paging algorithm
Inferred by program reference
REAL MEMORY
IBM 2365 Processor Storage

Figure 6. The TSS/360 storage hierarchy.

In order to implement such a virtual access method (VAM), note was taken of the storage hierarchy in TSS/360. We had already defined virtual memory to be contained on devices such as the IBM 2311 Disk Storage Drive and IBM 2314 Direct Access Storage Facility (auxiliary storage) or in real memory (see Figure 6). Both were page formatted. The job to be done by VAM was to map external storage into virtual memory and further to provide record and data set services. To simplify the problem, VAM external storage was defined as a page format. The capability for a virtual memory program to change page-table entries was also provided. This allowed VAM to simply map the data set pages of external storage into virtual memory. The supervisor already had developed the paging mechanism to accomplish any needed record transfer. VAM then was left with the simple problem of managing data sets and their records for the user, a clean though complex task. Error recovery and similar functions are a supervisor responsibility and should not be confused with record management, variable record lengths, etc.

Loading and Binding

Before discussing loading and binding, it is necessary to define some terms as clearly and explicitly as possible.

Loading: the act of placing program modules and/or data into the address space to be used in execution. In TSS/360, this means the assignment of program modules and/or data to virtual memory. Note that real memory is not involved in loading a virtual memory.

Dynamic loading: the act of loading a module at the time it is determined to be required. This facility allows a program to be constructed so that only the required modules will be loaded when they are determined to be needed.

Relocation: when a module or control section is loaded, it usually has some address constants that must be incremented by the difference between the origin of the address space and the address within that space assigned for the origin of that module or control section. The incrementing of these address constants is the process of relocation, and it is nominally done at the time loading is accomplished.

Binding: modules or control sections often contain external (to themselves) references. These references are to external symbols which are to be defined by other modules. The filling in of the intermodule address constants "binds" the modules together into a program and is called binding. In the MULTICS system, program modules are called program segments.[6] An original intent of program segments was to eliminate the need for binding program modules. Both MULTICS and TSS/360 have linkage sections that require the eventual substitution of virtual memory addresses for external symbols.

Address translation: a prerequisite to execution of a program is that the address constants must already have been resolved, i.e., relocation and binding must already have taken place. In a virtual memory system, these address constants point to virtual memory; the CPU in effect executes a virtual memory program. To relate the virtual memory addresses to real memory addresses requires a translation unit. In the System/360 Model 67, this is called the dynamic-address-translation (DAT) unit. Its purpose is to translate the virtual memory address called for by the CPU into the real memory address at which the appropriate virtual memory reference can be found. This address translation is accomplished entirely by the Model 67 hardware so long as the virtual memory being referenced is in fact in real memory.

Mapping: on occasion, the address-translation process will discover that a required page of virtual memory is not contained in real memory. An interrupt is given to the supervisor, whose responsibility it is to page the required virtual memory into real memory and establish the correspondence between them. The establishment of this correspondence is called mapping, and the page is said to be mapped into physical memory.

Although they are often confused with loading, binding, and/or relocation, *address translation and mapping do not change any address constants* of the virtual memory and can be ignored from a loading-and-binding point of view. It is the province and responsibility of paging to service the address-translation hardware by accomplishing the appropriate mapping as needed. Paging is ignorant of address constants in virtual memory.

Module Formats

TSS/360 object-program-module formats are specifically designed for on-line storage and a paged virtual memory. This avoids some of the problems seen in other systems. In addition, it provides functional advantages. The most notable change from previous systems is the elimination of the card-formatted object module. The language processors produce modules in page format; these can be conveniently loaded without the intermediate step of linkage editing. Another significant change is the provision for an internal symbol dictionary (ISD), optionally produced by TSS/360 language processors. The ISD, which is contained in a separate page (or pages), is used during debugging operations.

The format of the TSS/360 object module is shown in Figure 7. The program module dictionary (PMD) contains relocation information, external symbol definitions, and reference information such as is contained in the RLD and ESD of the IBM System/360 Operating System (OS/360) object-module format. It also contains the attribute definitions for each control section. The text of a module is laid out as it will be required in virtual memory. Each control section is origined on a page boundary and is defined as an integral number of pages. The length of the control section (in bytes) is contained in the PMD.

The Loading Process

Loading in TSS/360 is accomplished dynamically. The loader has available to it an initial virtual memory, including a task dictionary (TDY) of defined external symbols. It has one library defined by the system for the user and perhaps others explicitly defined by the user.

When the user (by command or program macro instruction call) asks that a module, control section, or entry point be loaded, the loader checks the TDY for the symbol. If the symbol is not found there, the loader proceeds to search the libraries (partitioned data sets) until the module to be loaded is found. It then adds the appropriate information to the TDY from the PMD.

The loader obtains space for each control section according to its attributes and establishes the definition of all defined external symbols by entering their virtual memory addresses in the TDY. It determines if any external references are yet undefined and, if so, continues the loading process until the definitions are completed.

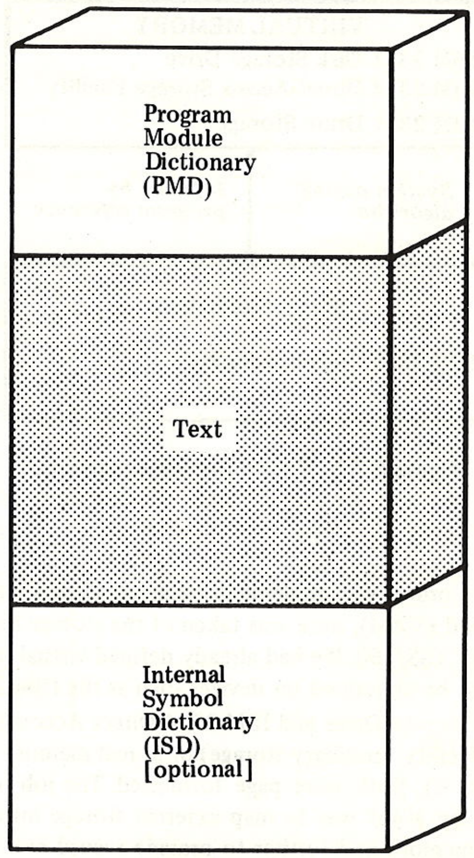


Figure 7. The TSS/360 object-module format.

Note that although all external symbols have been defined, no address constants have been relocated since the text has not been referenced. However, the loader did establish the correspondence between a virtual memory page and the location of the text on external storage by asking the supervisor to make appropriate page-table entries. It also identified those pages that contain relocatable or external address constants. The first time they are referenced, the loader will be called to insert the values as defined in the TDY.

Allocation

Allocation of virtual memory is necessary to provide system control. By providing for management of the allocation (and freeing up) of virtual memory, these serious problems are avoided:

- Once used, a virtual memory page must be kept for potential later use unless explicitly freed. This could impose much larger auxiliary-storage requirements upon the system than are actually needed.
- Virtual memory programs may have bugs. Wild references should be caught and treated rather than allowed to go undetected.
- Since certain system programs, as well as the user's programs, may be incorporated into his virtual memory, a means of keeping track of the assigned (and unassigned) space is essential to avoid conflicting virtual memory assignments.

The TSS/360 virtual memory requires that the user define his allocation requirements. A virtual memory allocation facility is common to system functions and user programs.

Protection

In TSS/360 virtual memory provides the primary means of protecting against unauthorized access to programs and data. Three kinds of protection are implemented.

Protecting the Supervisor from User Programs.

This protection is accomplished by not allowing the supervisor to be contained within the virtual memory of the user. The supervisor is executed with the dynamic-address-translation (DAT) unit turned off. The user program is executed with the DAT unit turned on.

Protecting One User's Virtual Memory from Another's

Since the virtual memories of different users are disjoint unless specific sharing (of segments) is permitted, protection against unauthorized reference is accomplished.

Protecting the System's Virtual Memory Programs from the User

TSS/360 uses storage keys to separate the levels of access within a virtual memory (see Figure 8). Since storage keys are used, protection is accomplished on a page-by-page basis.

When a user program is operating the PSW key is 1. This provides read/write access to information stored under key No. 1 and read-only access to information stored under key No. 2 without fetch protection. The program has no access to system information that is fetch protected. All calls to the system go through the task monitor, which assures that proper protection rules as well as proper linkages are used. The task monitor serves to queue and handle interrupts for virtual memory, which parallels the real memory interrupt-handling facilities of System/360 and its associated programming.

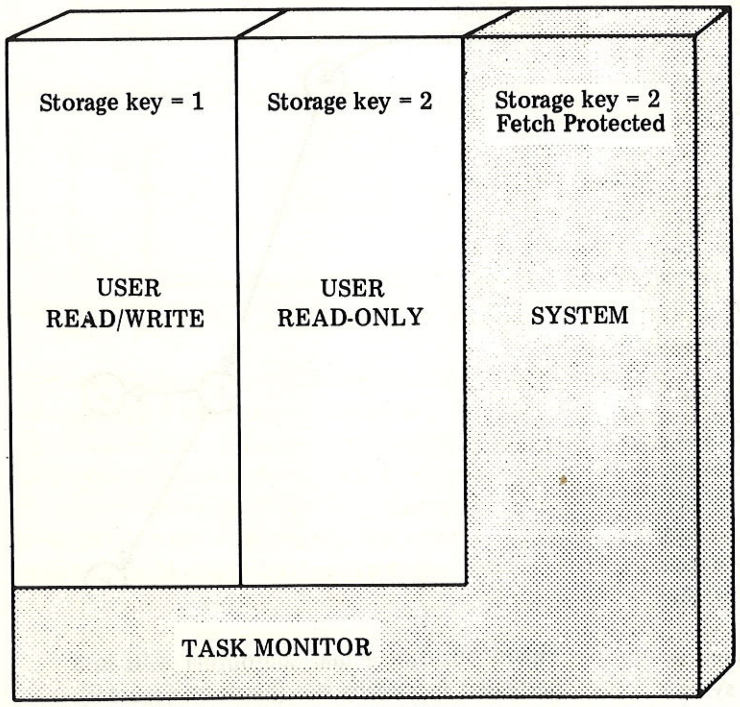


Figure 8. The use of storage keys in real memory. When a user program is operating, the PSW key is 1; when a system program is operating, the PSW key is 0, which provides unlimited access.

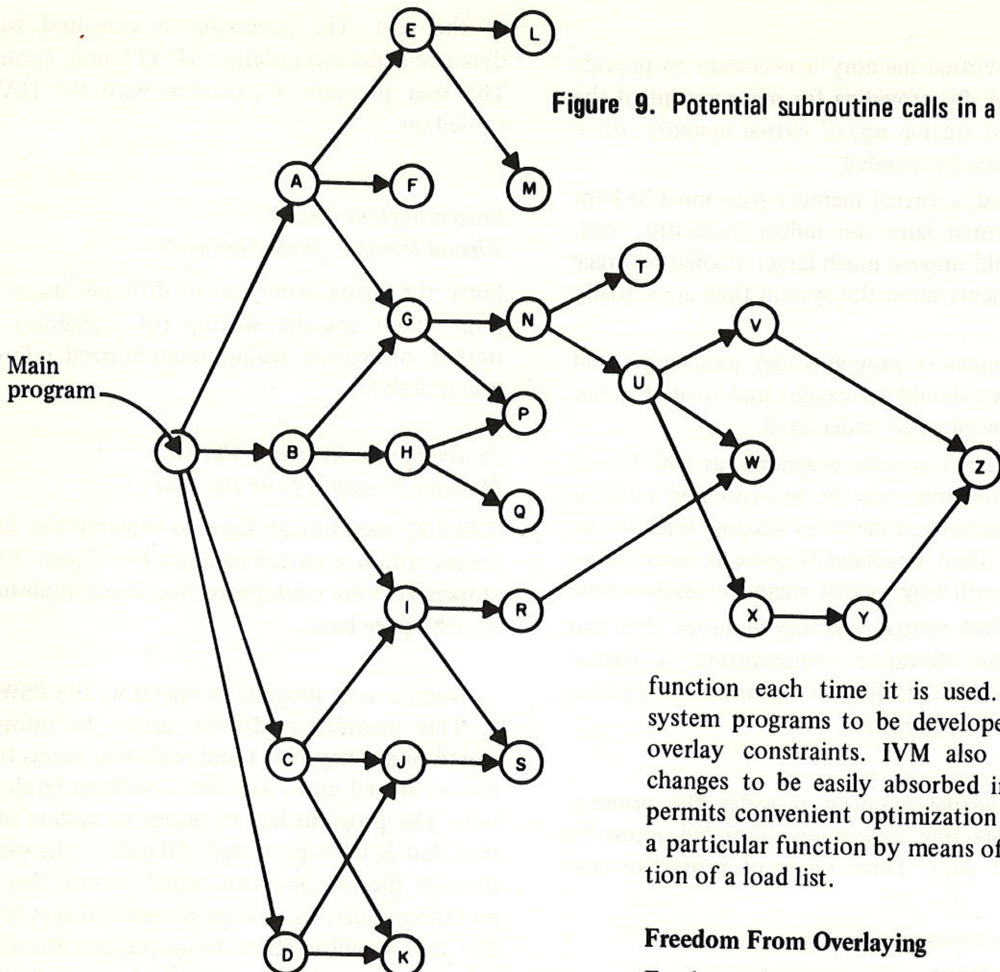


Figure 9. Potential subroutine calls in a complex program.

USE OF VIRTUAL MEMORY

Virtual memory, as implemented in TSS/360,[7] has a number of applications that are being explored. These applications involve one or more of the following:

Initial Virtual Memory

Initial Virtual Memory (IVM) is a set of program modules that are prerelocated and bound to each other during system startup (IPL). Initial virtual memory is defined by an IVM load list, which describes the program components to be made a part of IVM.

Resident on one or more paging devices, IVM is separated into private and shared virtual memory segments. Shared segments always have only one copy per system. Private segments have a pristine copy that each virtual memory is given to start with. Any change to one of those pages will cause a private page to be generated that will last the duration of that virtual memory.

The IVM concept provides an efficient system by not requiring relocation of a frequently used system function each time it is used. Further, it permits system programs to be developed without the usual overlay constraints. IVM also allows development changes to be easily absorbed into the system, and permits convenient optimization of pages spanned by a particular function by means of a simple reorganization of a load list.

Freedom From Overlaying

For large algorithms, such as a compiler or assembler, planning overlays is a substantial problem. Often the overlays do not efficiently utilize real memory be-

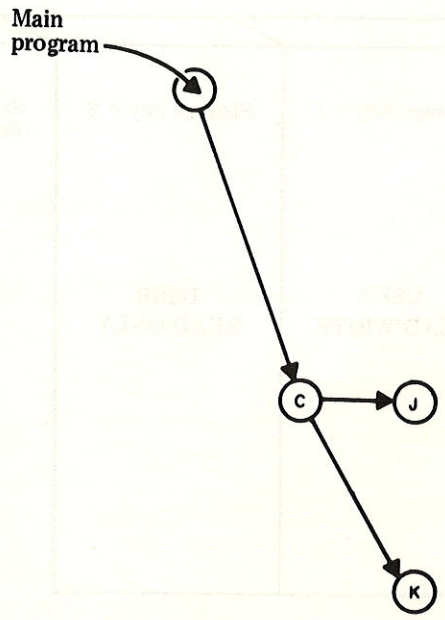


Figure 10. The actual subroutine calls in a particular processing run (based on Figure 9).

cause the program (or data) needed within the area is small compared to the overlay area. Further, overlays must be designed to fit a specific real memory address space. When it becomes necessary to provide (1) a small language processor and (2) a fast-running processor which is allowed to get larger, two compilers are required if planned overlays are used.

It is feasible with paged virtual memory to design a single processor that is operable in a small real memory environment and efficient in a large real memory environment. Not only is the duplicate cost eliminated, but also incompatibilities are eliminated between the two environments by the virtual memory approach.

For problem programs, not only is a lot of effort put into breaking the problem down into logically complete divisions, but also just the debugging of the overlay strategy takes substantial effort. In a paged virtual memory, it is beneficial (perhaps even mandatory) to minimize the number of pages needed to execute a particular function, especially if it is frequently used. However, exception handling need not adhere to these criteria. Exception-handling routines can cross page boundaries, provided they are logically complete, with a minimum degradation in performance. This permits a substantive saving in programming effort not possible in non-virtual memory systems.

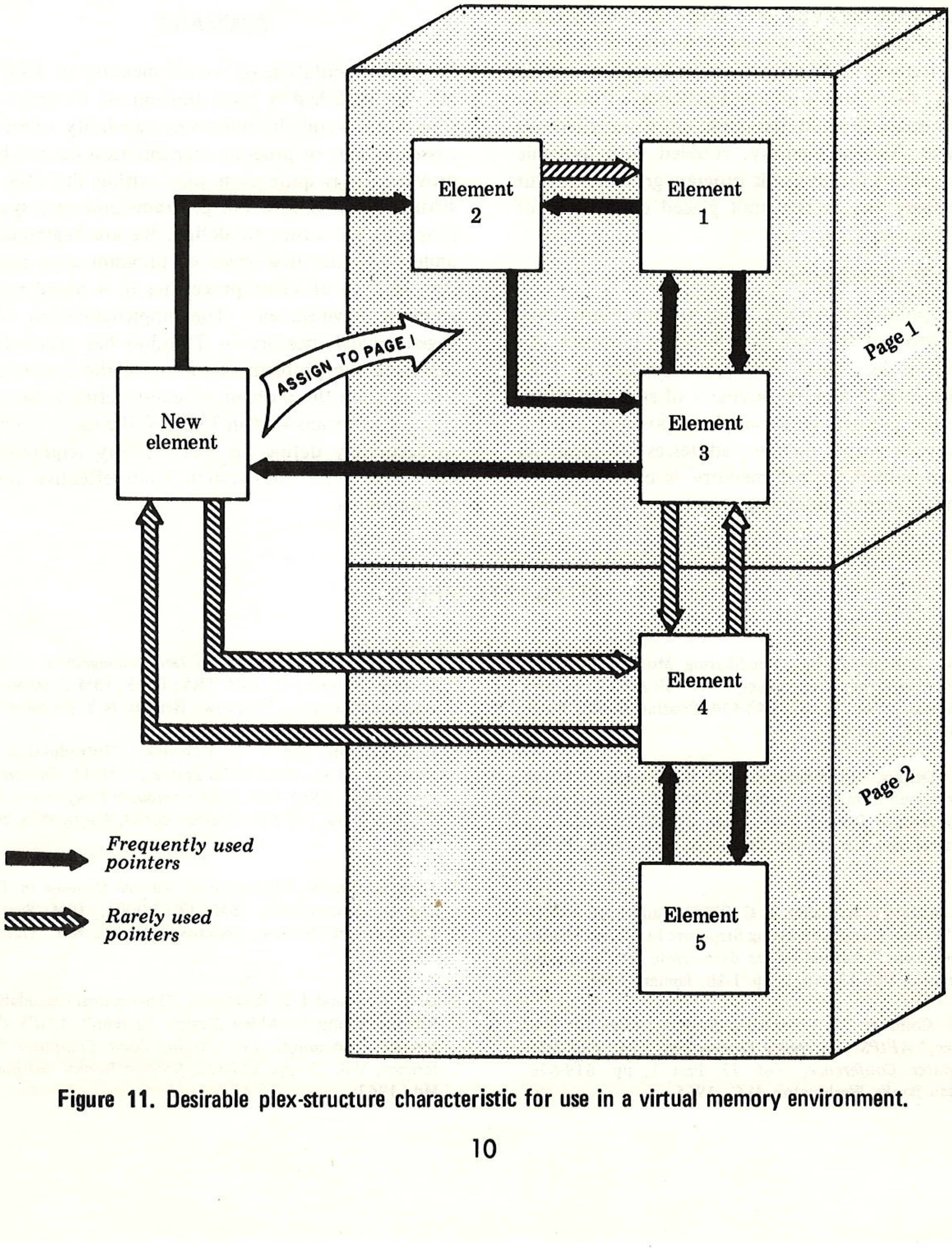


Figure 11. Desirable plex-structure characteristic for use in a virtual memory environment.

Dynamic Program Growth

Assume that a program has been developed by a number of people to process some highly variable data stream. The possible interconnections (subroutine calls) are shown in Figure 9. Assume also that each person makes changes from time to time to improve his particular subroutines. The latest checked-out version of each subroutine is desired.

Assume further that the subroutines to be called are a function of the data provided, and that the data processed in one operation will require references to only 10% of the subroutines. For a particular processing run, the subroutines to be used are shown in Figure 10.

The best way to accomplish the processing, and in some cases the only way, is to load a subroutine when it is determined to be needed. This kind of problem was extremely difficult to solve without virtual memory, for at any time the likelihood of making a program-space demand that could not be satisfied was too great. Virtual memory, coupled with dynamic loading, allows this dynamic program growth to occur as necessary (up to the limit placed on the size of virtual memory).

Flexible Data Structures

Several pointer-dependent data organizations have been developed for various processing purposes. An example is the "plex" structure described by Ross and Rodriquez.[8] One requirement of such structures is that the pointers be as simple as possible, and for this purpose virtual memory addresses are ideal. As has been shown, virtual memory is convenient for defining a simple addressing structure. This, in turn, greatly simplifies the programmed use of the structures.

When such structures are adapted to virtual memory, the convenience of addressing solves a substantial problem. However, it leaves the user wide open to another problem: he must organize his data space so that the probability of reference to a new page is minimized. For example, if a new element of data is being defined (see Figure 11), it should be put in the same page (or group of pages) as those elements with which it is most closely associated. If the user does a reasonable job of preventing random references to virtual memory, he has preserved efficiency as well.

SUMMARY

The implementation of virtual memory in TSS/360 has accomplished a large portion of its goals. Although the symbolic-addressing capability often discussed as part of program segmentation has not been provided, it is quite clear that, within the TSS/360 virtual memory, problem programs and even system programs are easier to define. We are beginning to understand the new rules of program organization that lead to efficient processing in a paged virtual memory environment. The implementation of a paged virtual memory in TSS/360 has successfully separated the problem of managing the system's resources from the problem of constructing a user program. This means that in TSS/360 the user's inability to accurately define his real memory requirements does not deter the system from effective multiprogramming.

CITED REFERENCES

1. H. A. Kinslow, "The Time-Sharing Monitor System," *AFIPS Conference Proceedings, 1964 Fall Joint Computer Conference*, Vol. 26, pp. 443-454, Spartan Books, Washington, D.C., 1964.

2. J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems," *1965 IEEE International Convention Record*, Part 3, pp. 214-225, Institute of Electrical and Electronics Engineers, New York, N. Y., 1965.

3. B. W. Arden, B. A. Galler, T. C. O'Brien, and F. H. Westervelt. "Program and Addressing Structure in a Time-Sharing Environment," *Journal of the Association for Computing Machinery*, Vol. 13, No. 1, pp. 1-16, January 1966.

4. W. T. Comfort, "A Computing System Design for User Service," *AFIPS Conference Proceedings, 1965 Fall Joint Computer Conference*, Vol. 27, Part 1, pp. 619-626, Spartan Books, Washington, D.C., 1965.

5. A. S. Lett, *The Approach to Data Management in Time Sharing System/360*, IBM TR53.0005, IBM Systems Development Division, Yorktown Heights, N.Y. (in press).

6. F. J. Corbatô and V. A. Vyssotsky, "Introduction and Overview of the MULTICS System," *AFIPS Conference Proceedings, 1965 Fall Joint Computer Conference*, Vol. 27, Part 1, pp. 185-196, Spartan Books, Washington, D.C., 1965.

7. J. R. Martinson, *Utilization of Virtual Memory in Time Sharing System/360*, IBM TR53.0001, IBM Systems Development Division, Yorktown Heights, N.Y., October 28, 1968.

8. D. T. Ross and J. E. Rodriguez, "Theoretical Foundations for the Computer-Aided Design System," *AFIPS Conference Proceedings, 1963 Spring Joint Computer Conference*, Vol. 23, pp. 305-322, Spartan Books, Baltimore, Md., 1963.

APPENDIX E

DECEMBER 2, 1968 TR53.0005

THE APPROACH TO DATA MANAGEMENT IN TIME SHARING SYSTEM/360*

by Alexander S. Lett

ABSTRACT

New direct-access organizations were developed for the dynamic-paging and time-sharing environment of the IBM System/360 Time Sharing System (TSS/360). These organizations are evaluated with respect to performance, recovery, and programming.

A device-independent scheme, based on page-size blocks and utilizing preformatted volumes, is the foundation of the organizations. A design objective was to provide a variety of data set organizations, with emphasis on logical rather than physical records. In TSS/360, the organizations are sequential, index sequential, and partitioned. Users are provided with full dynamic sharing of data sets that are within a pool of public data storage.

The key control table for each data set is the external page map, since all operations are performed relative to the data set and the data page. A single routine translates data set references to external-storage references by using the external page map. Input/output operations are performed by the resident supervisor. Data are recorded so that block boundaries are of no concern to the user.

**This report, under the same title, was presented at the IBM Systems Development Division Programming Symposium, Swampscott, Mass., June 11-14, 1967.*

Descriptors: IBM Technical Information Retrieval System

Time Sharing
IBM System/360
TSS/360
Software
Data Management
Programming
07 Computers
21 Programming

IBM®

Systems Development Division · Time Sharing Department · Mohansic Laboratory

Yorktown Heights · New York

INTRODUCTION

The IBM System/360 Time Sharing System (TSS/360) is the software support provided for the IBM System/360 Model 67. The Model 67 is a large, high-speed processor, derived from the Model 65 through the addition of dynamic-address-translation hardware.[1] The TSS/360 operating environment involves multitasking, dynamic paging, and time slicing.[2] Major components of the system are:

- Resident supervisor
- Data management
- Command system
- Support programs

This report presents the novel aspects of TSS/360 data management in terms of the physical organization of data storage, the user interface, and the method of implementation. Also, in the final section, the approach to data management selected for TSS/360 is evaluated.

Among its many benefits, the selected approach had significant implementation advantages that should be considered by other groups developing data management components.

PHYSICAL ORGANIZATION

A portion of the TSS/360 resident supervisor provides for the automatic input and output of page-size blocks of data to and from main storage, as required by each user's program.* This paging feature facilitates simultaneous operation of many independent user programs (tasks).

A block size of 4096 bytes for data storage was selected since it was equal to the smallest unit of main-storage allocation and because it could use the page-oriented input/output facility provided in the resident supervisor. TSS/360 data organization is called *virtual* to reflect the page-size orientation of its data blocks.

The data management component of TSS/360 must support the data set requirements of a large number of simultaneous users. Since direct-access storage devices permit this shared operation, they are the primary medium for data set storage in TSS/360.

The direct-access volumes on which TSS/360 data sets are stored have fixed-length page-size data blocks. No key field is required. The record-overflow feature is utilized to allow data blocks to span tracks as required. Eight page-size blocks are stored in each IBM 2311 Disk Storage Drive cylinder and 32 page-size blocks are stored in each IBM 2314 Direct Access Storage Facility cylinder. The entire volume, with the exception of the cylinders used for labels and for identification, is formatted into page-size blocks.

* Division of the user's program into *pages* of 4096 bytes is an essential feature of TSS/360. The program space that is subject to paging is called *virtual memory.*[3]

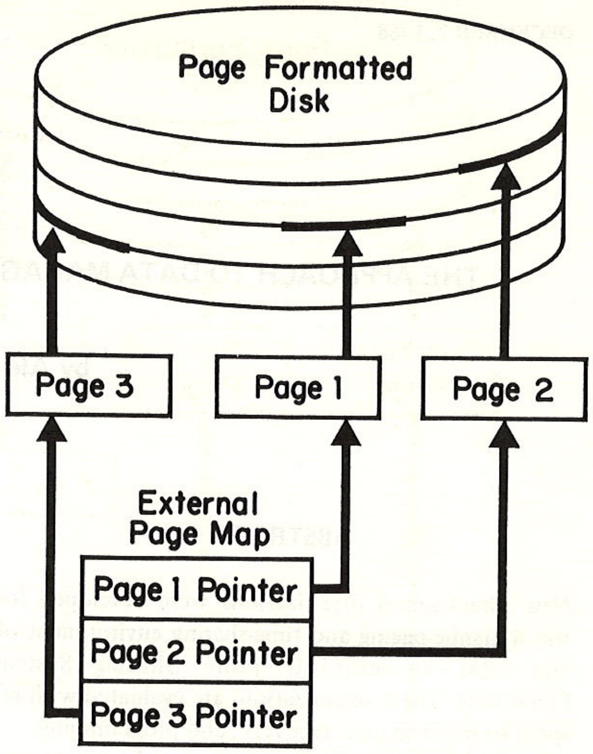


Figure 1. Typical virtual-sequential organization.

Most disk storage devices, such as the IBM 2311, have removable packs. In TSS/360, the number of concurrent users is much greater than the number of separate volumes available. Consequently, most of the available direct-access drives are designated as *public* storage devices; their packs are not removed during normal system operation and they are shared by all users of an individual system. The use of private volumes, with associated mounting and demounting, plays a minor role in TSS/360 operations.

Three data set organizations are provided by TSS/360 data management:

- *Virtual sequential:* data records are retrieved in the sequence in which they were created.
- *Virtual index sequential:* data records are retrieved according to the value of a unique key within each record.
- *Virtual partitioned:* a single data set is divided into named members, each of which can be processed independently of the others.

Virtual-Sequential Organization

The physical representation of a typical virtual-sequential organization is shown in Figure 1. The specification of any virtual sequential data set is con-

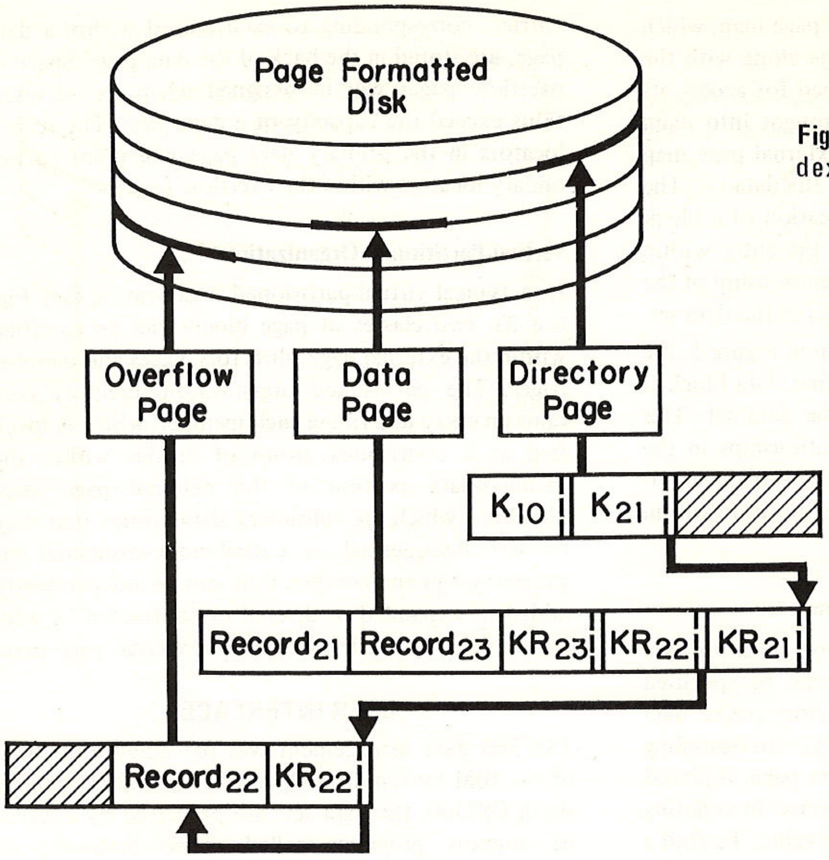


Figure 2. Typical virtual-index-sequential organization.

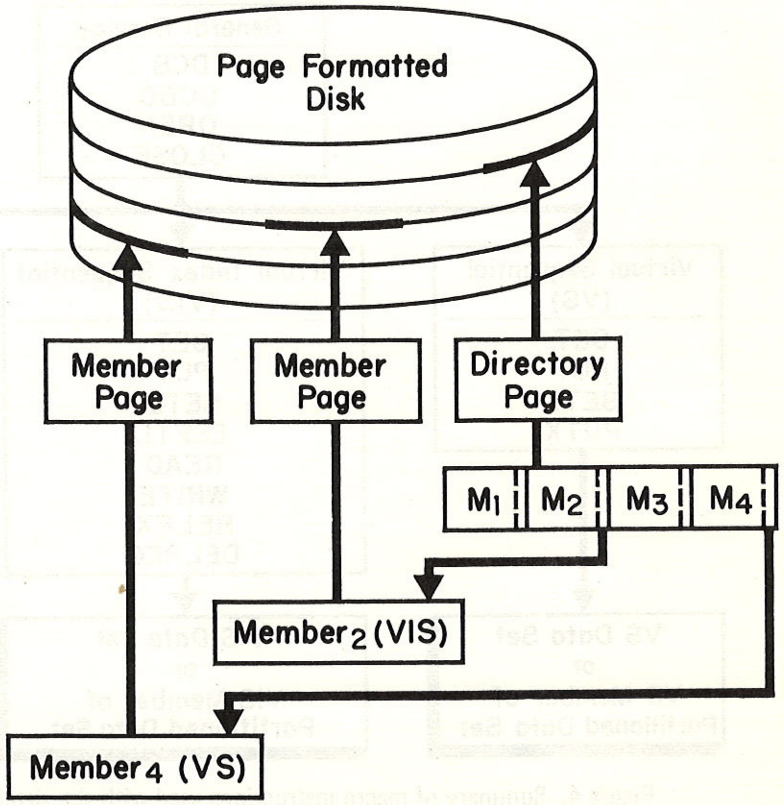


Figure 3. Typical virtual-partitioned organization.

tained within the data set's external page map, which is stored on the direct-access volume along with the data pages. When a data set is opened for access, its associated external page map is brought into main storage. There is one entry in the external page map for each page-size block occupied by the data set. The content of an entry specifies the location of a block in external storage. The position of the entry within the external page map signifies the relationship of the associated block and the other blocks in the data set.

For the three-page data set shown in Figure 1, the external page map shows that the first data block is between the other two pages of the data set. This example emphasizes that block relationships in the data set are determined by the contents of the external page map rather than by physical position within the volume.

Virtual-Index-Sequential Organization

In a typical virtual-index-sequential organization (see Figure 2), three classes of blocks can be specified within the external page map: directory pages, data pages, and overflow pages. One entry, corresponding to the lowest record key in each data page, is placed in the directory. Records are maintained in collating sequence within the data set by key value. To find a given record, the directory is searched and then the data page containing the record is searched. Locator entries, corresponding to each record within a data page, are stored in the back of the data page. Space in overflow pages will be assigned when record insertions exceed the capacity of a data page. The record locators in the primary data page will point to secondary locators within the overflow page.

Virtual-Partitioned Organization

In a typical virtual-partitioned organization (see Figure 3), two classes of page blocks can be specified within the external page: directory pages and member pages. The partitioned organization directory contains an entry describing each member, which is specified as a contiguous group of entries within the member-data portion of the external page map. Members, which are subsidiary data groups that may be virtual-sequential or virtual-index-sequential organizations in any combination, can be independently added or expanded or deleted or contracted by adding or deleting entries within the external page map.

USER INTERFACE

TSS/360 data management was patterned after that of the IBM System/360 Operating System (OS/360). As in OS/360, the data sets are processed by a series of support programs called access methods; in TSS/360, these are called virtual access methods.[4] The assembler-language/user interface to these access

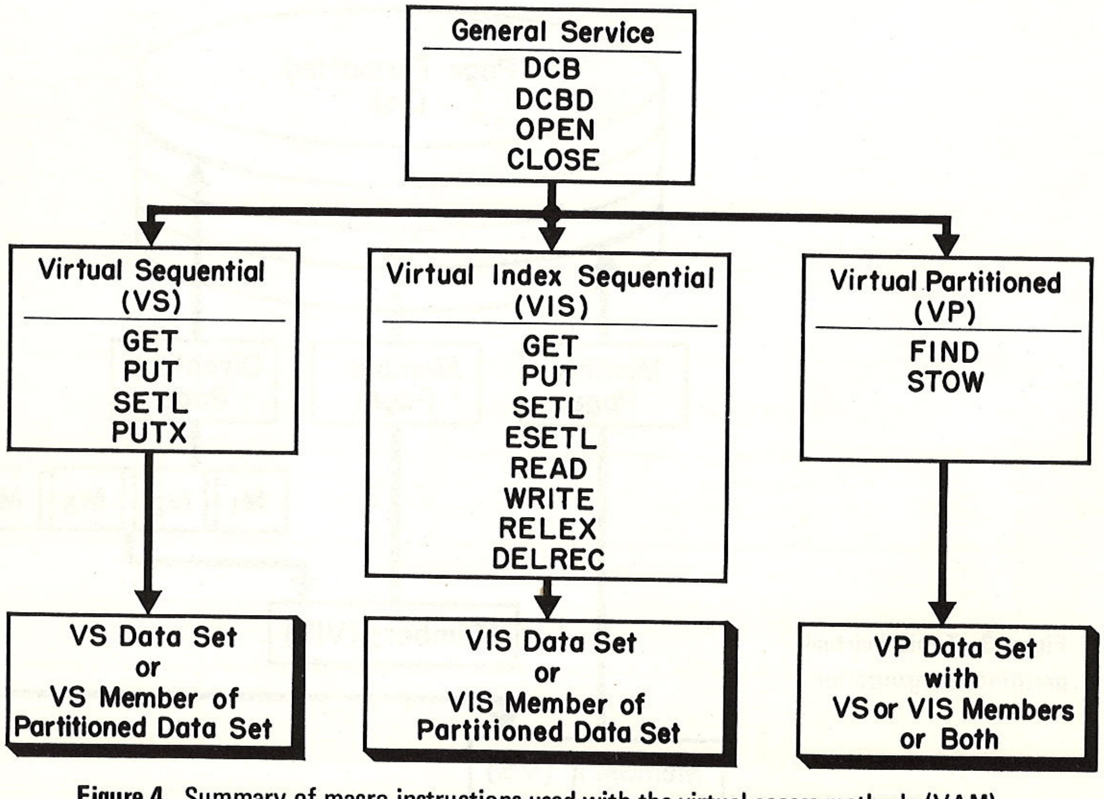

General Service
DCB
DCBD
OPEN
CLOSE
Virtual Sequential (VS)
GET
PUT
SETL
PUTX
Virtual Index Sequential (VIS)
GET
PUT
SETL
ESETL
READ
WRITE
RELEX
DELREC
Virtual Partitioned (VP)
FIND
STOW
VS Data Set or VS Member of Partitioned Data Set
VIS Data Set or VIS Member of Partitioned Data Set
VP Data Set with VS or VIS Members or Both

Figure 4. Summary of macro instructions used with the virtual access methods (VAM).

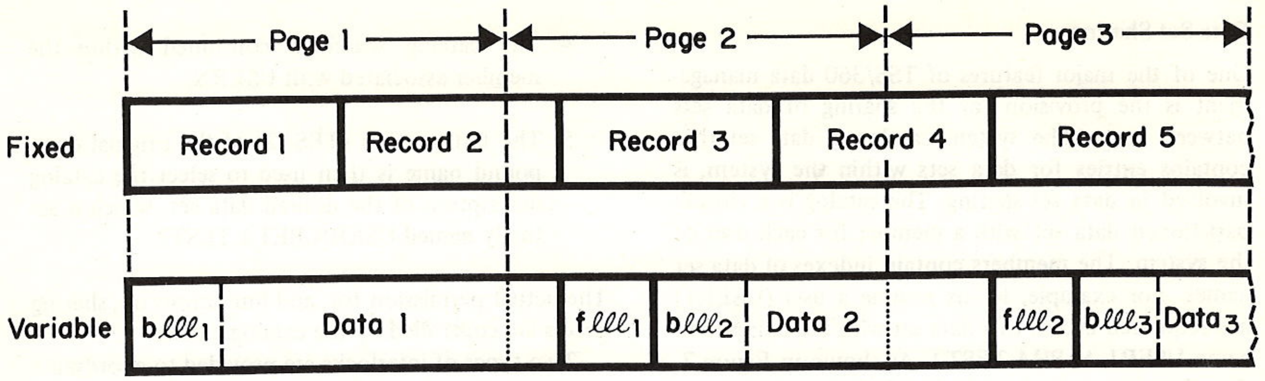


Figure 5. Fixed and variable formats for virtual-sequential records.

methods is via macro instructions.[5] The macro instructions available to ordinary users are tabulated in Figure 4.

The general-service macro instructions that specify a data control block (DCB), label the various DCB fields (DCBD), and open (OPEN) and close (CLOSE) a data set are provided for all data set organizations.

For virtual-sequential organizations, macro instructions are provided to retrieve logical records (GET), create logical records (PUT), set access location (SETL), and update logical records (PUTX). Access location can be specified as the beginning or end of the data set, as a particular logical record, or as the previous logical record (back space).

For virtual-index-sequential organizations, GET, PUT, and SETL macro instructions are provided. Since data-set sharing is allowed, the sharing interlocks can be released by the end of sequential access (ESETL) and the release exclusive control (RELEX) macro instructions. The capabilities to read (READ) and write (WRITE) individual records by key and to delete specific records (DELREC) are also included.

For virtual-partitioned organizations, only two macro instructions are needed: one (FIND) prepares for access to a specific member; the other (STOW) stores the member characteristics into the partitioned organization directory. A program may independently and simultaneously access several members of the same virtual-partitioned data set. New members can be added, and existing members extended or deleted, without restriction.

Record Formats

The formats of logical records for virtual-sequential and virtual-index-sequential organizations are shown in Figures 5 and 6. Both fixed-length and variable-length records are allowed. In variable-length records, an extra control field, which specifies the length of each logical record, is required at the front of each record. Virtual-index-sequential logical records must contain a key field, which may be either the first bytes of each logical record or embedded within the data portion of each logical record. Record lengths of up to 1,048,576 bytes for virtual-sequential organizations and of up to 4,000 bytes for virtual-index-sequential organizations are allowed. The apparent restriction on the length of virtual-index-sequential records can be easily circumvented if the user extends the length of the key field to utilize multiple records as a group.

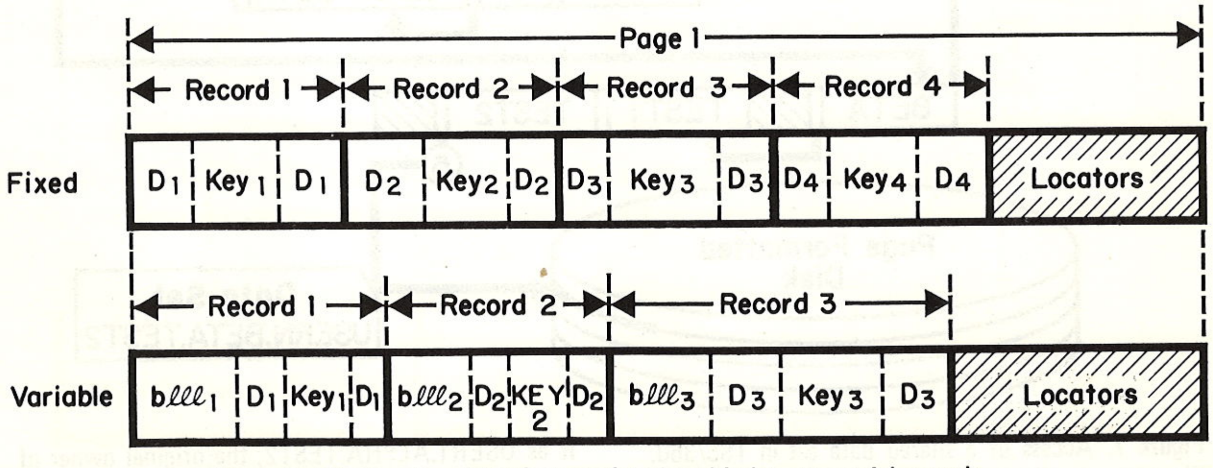


Figure 6. Fixed and variable formats for virtual-index-sequential records.

Data Set Sharing

One of the major features of TSS/360 data management is the provision for the sharing of data sets between users. The system catalog, a data set that contains entries for data sets within the system, is involved in data set sharing. The catalog is a virtual-partitioned data set with a member for each user of the system. The members contain indexes of data set names. For example, let us assume a user (USER1) has requested access to a data set with the compound name USER1.ALPHA.TEST2. As shown in Figure 7, the system catalog is searched as follows:

1. The master index (directory) of the catalog is searched to find the private catalog (member) for USER1.
2. The search of member entries shows that ALPHA is associated with a shared data set pointer (USERN.BETA); USERN is the original owner of the shared data set involved.
3. The master index is searched again, this time to find the private catalog for USERN to find BETA.
4. The catalog search is continued within the member associated with USERN.
5. The last portion (TEST2) of the original compound name is then used to select the catalog description of the desired data set, which is actually named USERN.BETA.TEST2.

The actual permission for, and limitations on, sharing access are controlled by the catalog entries.

Two types of interlocks are provided to coordinate simultaneous access to shared data sets by more than one user:

- *Read interlock:* prevents another user from writing into the interlocked data space: other users may have read-only access at the same time.
- *Write interlock:* prevents another user from reading or writing the interlocked data space; can be set only when no other interlock is set.

Interlocks are established at various data space intervals, depending on the data set organization. Virtual-sequential organizations are interlocked at the

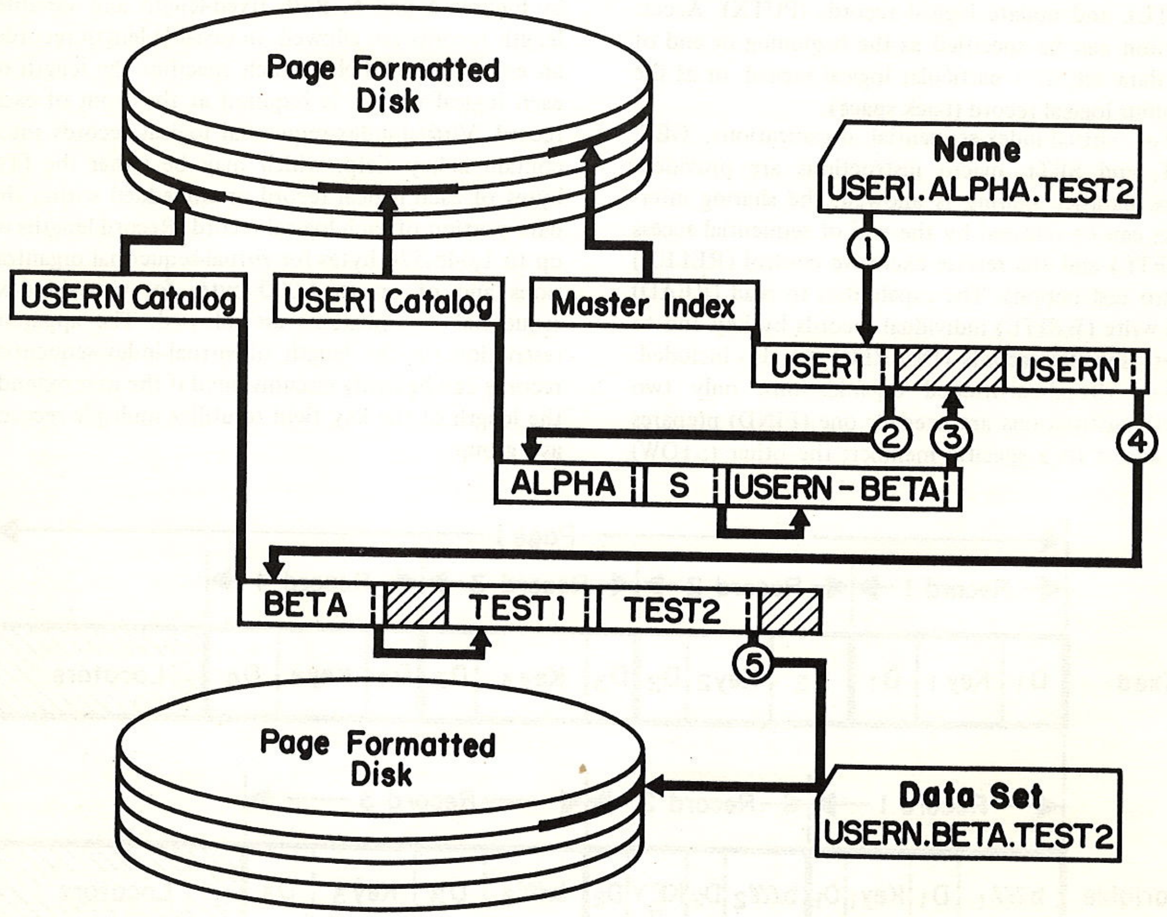


Figure 7. Access of a shared data set in TSS/360. The user permitted to share the data set references it as USER1.ALPHA.TEST2; the original owner of the data set has named it USERN.BETA.TEST2.

entire data set level. Virtual-partitioned organizations are interlocked at the individual member level. Virtual-index-sequential organizations are interlocked at the individual data page (block) level.

IMPLEMENTATION

The cornerstone for implementing the virtual access methods of TSS/360 is the external page map. There is one entry in the map for each page-size block assigned to the data set. The map itself, along with other descriptive information about the data set, is stored in a special portion of each volume, called the volume table of contents. When the data set is opened for access, the data set description, including the external page map, is brought into main-storage for use by the access methods.

An entry within the external page map is one word (32 bits) long. The high-order 16 bits specify the direct-access device on which the page-size block is stored; the remaining 16 bits specify the exact block within the device. The volumes are preformatted so they can be addressed as a string of pages, without regard for track or cylinder boundaries. Page-size blocks are addressed in a relative manner, with the initial page on the volume defined as relative page 0.

Pages pointed to by the external page map may be assigned as directory, data, or overflow pages, as required for the data set. Since there is no physical restriction concerning adjacency of data set pages, it is a simple operation to insert or delete pages by moving entries in the external page map. The virtual access methods treat the data set as a string of pages that are addressed relative to the origin of the data set. The position of an entry in the external page map corresponds to its position relative to the data set. When more than one class of page (directory, data, or overflow) is present, the external page map is subdivided into contiguous entry groups with separate pointers to each group. In this way, each class of data set pages may be expanded or contracted independently of other classes.

Whenever input/output reference is needed, a common routine is invoked to convert the data set references generated by the access methods into the external-storage references required by the resident supervisor routines. The conversion process utilizes the contents of the external page map to prepare the request list that is to be passed to the resident supervisor. The request list specifies the main storage address, external-device address, and the page block (within the device) involved in the operation. Since all items are of fixed length (4096 bytes) and the direct-access volumes are preformatted in a known pattern, this list completely specifies the major parameters involved.

The request list is processed by a portion of the resident supervisor concerned with the input/output of page-size blocks. A channel command word (CCW) list is developed. Because all direct-access volumes are preformatted, the basic CCW list is simple:

- SEEK: to appropriate cylinder and track
- SEARCHIDEQ: look for block on track
- TIC: transfer back to SEARCH if not equal
- READ (or WRITE): read (or write) a 4096-byte data block

The SEEK and SEARCHIDEQ values are calculated by using a single device-dependent table and processing routine to convert a relative page within a volume into the cylinder, track, and block number on the track. The READ (or WRITE) main-storage address was supplied in the request list. If the optional read-after-write verification is requested, an additional SEEK, SEARCHIDEQ, TIC, and READ sequence is added to the basic list. This basic CCW list is used for all page operations. After the operation is completed, control is returned to the access method.

The main portion of the access method for each data organization is concerned with operations within data pages. For virtual-sequential organizations, a main-storage buffer is provided; it is large enough to hold the maximum-length logical record, plus 4096 bytes, rounded out to a multiple of the page length. The required number of data set pages is brought in to fill the buffer. Thereafter, the location of each logical record within the buffer is determined by the access method. At some point, a logical record, which is not complete with the current buffer contents, will be encountered. In this case, the data set page containing the start of that record is shifted to the front of the buffer and succeeding data set pages are brought in to fill the buffer again. A similar procedure is followed for output. In this way, the user program always deals with complete logical records, without regard for page-size block boundaries.

In virtual-index-sequential organizations, the access method utilizes three one-page buffers. The first buffer is assigned to the last referenced directory page; the second is assigned to the current data page; and the third is assigned to the last-referenced overflow page. The access method utilizes the locator fields in each data and overflow page to find the logical records as needed.

In virtual-partitioned organizations, the access method brings the directory into a buffer for processing. When a member is accessed, pointers are set up in such a way that the virtual-sequential or virtual-index-sequential access methods can be used to process the member data.

One interesting logical operation is to backspace over a logical record. In virtual-sequential organizations of fixed-length records, this backspace involves moving the current-position pointer back a distance equal to the length of one record. For variable-length sequential records, the access method adds a control field, between the logical records, to specify the length of the preceding logical record. This control field supplies the information that moves the current-position pointer the correct backward distance. In virtual-index-sequential organizations, stepping back through the locator records accomplishes the back-space operation in the proper manner.

Associated with the external page map in main storage are control fields used by the access methods to process the data set. For nonshared (private) data sets, this information is placed in the main storage reserved for an individual user. For shared data sets, the control information (including the external page map) is placed in a portion of main storage accessible to all user tasks. It is in this area that the sharing interlocks are maintained by the access methods. Through use of the test-and-set (TS) instruction, logical conflicts are avoided.

For shared data sets, any directories are also placed in shared main storage when the data set is accessed. However, the data pages are not placed in shared main storage. Each concurrent user gets his own private copy of the data pages within his buffer. Only as a result of an explicit output request (PUT, PUTX, or WRITE) will an external data page be modified. When dealing with shared data sets, the control information (including the directories) is dynamically shared to maintain orderly access to the external data pages.

EVALUATION

The outstanding value of the approach to virtual organization that was selected for data management in TSS/360 is in the large reduction of implementation effort that resulted. Actual savings are difficult to estimate. Fortunately, within TSS/360, a meaningful comparison can be established.

In addition to the virtual access methods, TSS/360 data management includes physical-sequential access methods that are functionally similar to OS/360's basic sequential access method (BSAM) and queued sequential access method (QSAM).

The virtual access methods that support partitioned and index-sequential organizations as well as sequential organization required approximately 40% less code than did the TSS/360 physical-sequential access methods. It is apparent that the removal of device-dependent operations (with complex CCW lists), standardization of block size, and elimination of exceptional procedures (such as end-of-volume operations) for the virtual access methods simplified the actual coding. Also, checkout was simpler due to the separation of input/output from the access methods themselves.

Block Size

In terms of TSS/360 data management, the page size of 4096 bytes seems optimal. It is as small as the smallest unit of TSS/360 main-storage allocation. It is large enough so that direct-access throughput is high. Rotational delay is a significant factor in direct access throughput, since it cannot be overlapped as mechanical seek-time can. On the IBM 2311 Disk Storage Drive, the rotational delay is approximately 30% of the total data access time (exclusive of seek-time). For the IBM 2314 Direct Access the Storage Facility, the access time rises to 50% due to the faster byte rate. Any block size significantly shorter than 4096 bytes would be extremely wasteful of total direct-access capacity unless elaborate strategies were utilized to avoid rotational delay.

The need for large block size is also apparent when the simultaneous direct-access activities of multiple users are considered. Due to conflicts in demands for access arms, a mechanical seek may frequently be required before accessing a data block. The larger block size makes better use of the total access cycle while, at the same time, reducing the frequency of access requests by each user. In on-line printing operations, the block size of 4096 bytes will result in one access during approximately every two seconds, thereby holding access overhead to a minimal value.

The direct-access volume-packing efficiency also is quite high for page-size blocks. First, the data-recording space is utilized at better than 90% of its theoretical capacity (if cylinder-length blocks were written). The smallest allocation unit is a page-size block, so a large number of small data sets can be kept on one volume. Furthermore, the freedom from requirements for physically contiguous storage space will lead to higher volume packing.

Use of Direct-Access Devices

The virtual access methods minimize use of the data-searching facilities of the direct-access control units. The operation of these searching facilities would lengthen the data-access cycle and thereby reduce direct-access throughput. In the time-sharing environment, the throughput of direct-access devices will be critical to overall performance. It is better to conduct a programmed search in main storage than to extend the direct-access cycle.

A weakness of the virtual organizations is in their vulnerability to certain external-storage failures. If the volume copy of the external page map cannot be read into main storage, there is no way to access the data set. This particular weakness is not present in magnetic-tape storage (which has no need for a storage map) but is characteristic of direct-access storage organizations.

Another weakness is a result of the block independence of logical records in the virtual-sequential organization. If any data page cannot be read into main storage when processing variable-length records, there is no simple way to correctly process the remainder of the virtual-sequential data set. This is because the access method would have no sure method to determine the location of the first complete record that starts in the following page.

Both of these problems lead to a dependence on some form of back-up facility.

Extensions

Experience to date has turned up a performance problem with respect to the organization of direct-access volumes: In TSS/360, there is a high activity of opening and closing many small data sets. This involves processing the volume table of contents (VTOC) area of the volume, which is formatted in short (44-byte key and 96-byte data length) blocks like other volumes created by IBM operating systems (such as OS/360). Direct-access operations on the VTOC are predicted to be as high as two full seconds, just to allocate space for a new data set.

Accordingly, this procedure is being revamped to eliminate most of this direct-access overhead. First, the system catalog data set entries are being modified to point directly to data set control information in the volume. This will eliminate a direct-access search, which is required in the present procedure. Next, the storage-allocation records will be replaced by a byte map in a page-size block that records the status of each page block in the volume. This will eliminate the complicated chaining procedures required currently. Finally, the VTOC area will be eliminated with the data set control information packed into page-size blocks for ease of access.

The new storage-allocation map for each volume, in conjunction with the approach of using an external page map for each data set, provides the basis for the future development of an efficient checkpoint/restart facility. One of the basic requirements of such a facility is the ability to back up and reprocess data sets. This means that update-in-place operations on data sets cannot be allowed if restart may occur. The external page map provides the means to assign a new data page to hold the updated image, rather than reusing the original page. After the update, two versions of the data set (each with its own external page map) will exist. Their entries will differ only for the updated pages and will be identical for any unmodified pages. The total page storage space required for both data sets would be equal to the sum of the original data set pages plus the number of modified pages in the new data set.

If a restart were required, only the original version of the data set would need to be reopened for reprocessing. A potential problem with the sharing, between versions, of unmodified data pages is in the release of such pages when they are no longer required. In the new storage-allocation map, a count field is provided for each page in the volume. Whenever a page is included in a new version of data set, the count is incremented. Whenever a version is released, the count for each data page of that data set is decremented. A count of 0 indicates a page available for reassignment.

CONCLUSION

Fixed-block-length data organizations should be considered for future data management efforts. Experience with such organizations in TSS/360 has shown that they provide significant logical flexibility, with reduced implementation effort.

CITED REFERENCES

1. W. T. Comfort, "A Computing System Design for User Service," *AFIPS Conference Proceedings, 1965 Fall Joint Computer Conference,* Vol. 27, Part 1, pp. 619-626, Spartan Books, Washington, D.C., 1965.

2. C. T. Gibson, "Time Sharing in the IBM System/360: Model 67," *AFIPS Conference Proceedings, 1966 Spring Joint Computer Conference,* Vol. 28, pp. 61-78, Spartan Books, Washington, D.C., 1966.

3. J. R. Martinson, *Utilization of Virtual Memory in Time Sharing System/360,* IBM TR53.0001, IBM Systems Development Division, Yorktown Heights, N. Y., Oct. 28, 1968.

4. *IBM System/360 Time Sharing System: Concepts and Facilities,* C28-2003, IBM Data Processing Division, White Plains, N. Y.

5. *IBM System/360 Time Sharing System: Assembler User Macro Instructions,* C28-2004, IBM Data Processing Division, White Plains, N. Y.

Y20-0450-0

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N. Y. 10601
(USA Only)

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