The design of Operating System/2

by M. S. Kogan
F. L. Rawson III

The design of Operating System/2™ (OS/2™) is a result of matching the requirements of IBM and its customers for a new operating system for various models of the Personal System/2® with the need for continuity with a very large body of established DOS applications. The design of OS/2 represented a significant challenge both in meeting these requirements and in making efficient use of the hardware. In this paper, the design characteristics of OS/2 are discussed.

Operating System/2™ (OS/2™) is IBM's new operating system for its personal systems that incorporate 80286 and 80386 processors. Because new operating systems are expensive to develop and expensive for customers to install and to adapt to their own use, there has been a tendency in recent years to stretch existing systems rather than to implement new ones. OS/2 represents an effort to implement a new system that meets the needs of a growing marketplace while minimizing the development costs of both IBM and its customers.

The overriding goal behind the design of OS/2 is to provide a successor to the IBM Personal Computer Disk Operating System (DOS) that fully supports the hardware of the 80286 processor without giving up compatibility with DOS. OS/2 relieves a number of limitations of DOS while still permitting most DOS applications to run. Table 1 compares the key requirements placed on DOS and OS/2.

OS/2, like DOS, is a single-user system, designed to run well on personal systems. It is oriented toward providing a good interactive response while not occupying too much main memory or disk space. These requirements combined with the architectural constraints of the 80286 processor led to the development of the system structure that is characteristic of OS/2.

OS/2 product set. OS/2 executes on Models 50, 60, and 80 of the IBM Personal System/2®. It also executes on the IBM Personal Computer AT® and the IBM Personal Computer XT Model 286. Although IBM has announced two versions of the Standard Edition of OS/2 and two versions of the Extended Edition, this paper discusses the design characteristics common to all versions of OS/2.

System features. To meet its design goals, OS/2 had to provide a number of important system features (see Table 2).

Many of these features are common in current operating systems, and, where possible, OS/2 implements them in standard ways, using demonstrated techniques from earlier systems. However, two sets of design constraints forced the designers of OS/2 to use novel techniques in a number of areas to meet the requirements imposed on the system. The first set of design constraints is architectural in nature:

- OS/2 had to provide a way to run the vast body of existing DOS applications.

- OS/2 also had to use effectively the large physical memory that can be attached to the 80286 and 80386 processors.

Because of the nature of the architecture of the 80286 processor (described in the next section), these constraints are in conflict. Resolving this conflict in a working system is a significant design achievement of OS/2.

The other set of design constraints is one usually found in the design of small systems:

- OS/2 had to provide the features of a full-scale operating system such as might be found on larger microcomputers and on minicomputers.
- OS/2 also had to execute without consuming large amounts of system resources in order to fit on the machines it is intended to support.

As always with this set of constraints, the result is a compromise between complex and powerful ways of implementing features and simpler but less expensive means of providing them.

OS/2 architectural design constraints

In order to understand the conflict between the two sets of OS/2 architectural design constraints, it is essential to understand the hardware architecture of the Intel processor family and the environment assumed by DOS applications.

The architecture of the 80286 processor. Convenient use of the large-memory features of the Intel 80286 and 80386 processors necessitates use of their more advanced and incompatible architectural features. Since OS/2 is an 80286 operating system that treats the 80386 processor as if it were an 80286 processor, the term *80286* will be used to refer to the common features of both types of processor. For more information on the architecture of the Intel 80286 processor, see References 1 and 2.

The 80286 processor extends the addressing range of the 8086/8088 processor used in many personal computers by adding a new mode of operation called *protect mode*. The 80286 can also execute in 8086/8088 mode, which is called *real mode* on the 80286. The 80286 can switch beween real and protect modes under program control. The switch from real to protect mode is relatively simple and can be made fairly quickly. Although the exact mechanics vary from machine to machine, the transition from protect mode back to real mode on the 80286 (but not the 80386) always requires that the processor be reset. This operation is relatively slow and is done by the hardware on the machine external to the processor.

Like the 8086/8088, the 80286 processor divides memory into ranges of addresses called *segments*. Except on the 80386, segments are limited to at most 64K bytes in length because the processor can only generate a 16-bit offset into the segment. Although the 80386 can generate 32-bit offsets, OS/2 does not use this feature. Consequently, the 32-bit mode of the 80386 will not be discussed here.

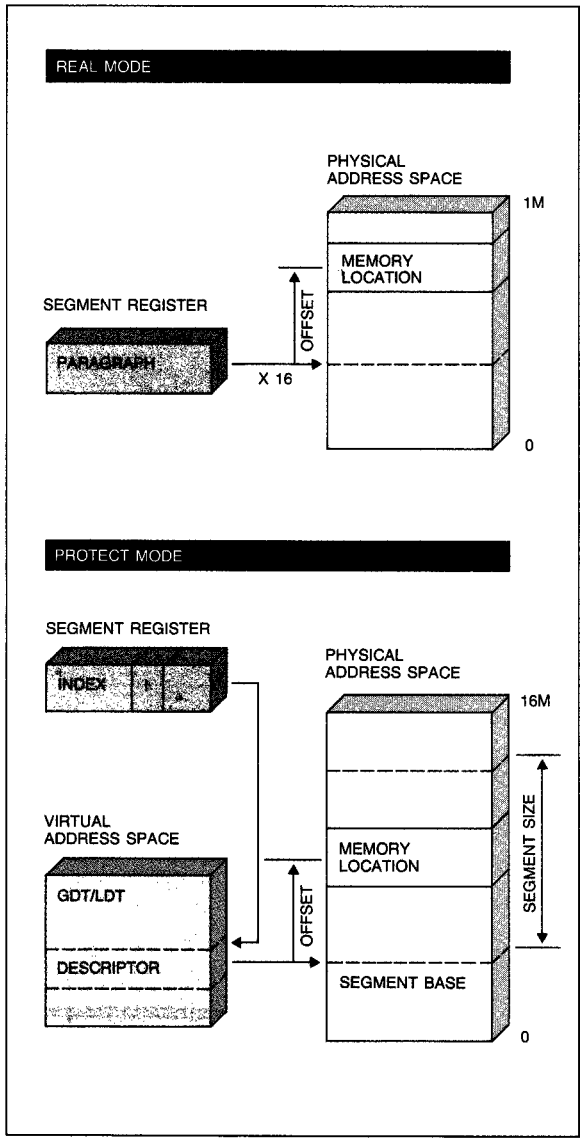
Table 1 DOS and OS/2 requirements

Area	DOS	OS/2
Processor architecture supported	8086	80286
Maximum memory size	640K bytes	16M bytes
Number of tasks	1	Multiple
Number of concurrent applications	1	Many
System interfaces	Fixed	User-extendible

Table 2 OS/2 system features

Use of physical memory up to 16M bytes
Multiple concurrent applications
Protection between independent applications
Multitasking
Interprogram communication and synchronization
Dynamic linking to system and subsystem services
Demand loading of code and data
Call Application Programmer Interface (API)
DOS Environment for executing DOS applications
Methods and tools that permit the development of applications that can run on either DOS or OS/2
File and media formats identical to those of DOS
Separate device drivers for IBM and non-IBM devices
Character monitors to filter input and output streams
Subsystems for video, keyboard, and pointing devices
An extended DOS command processor
A set of utilities extending those of DOS
An installation program for the system and associated development tools
National-language and country-specific support

Figure 1 80286 real- and protect-mode address generation



In protect mode, segments are described to the processor by data structures known as *descriptors*. The descriptor for a segment gives the starting address of the segment in physical memory, its length, and its usage and protection attributes. Descriptors that the operating system can protect and manipulate as necessary to implement its memory management policies are gathered into tables. There is a single, system-wide *Global Descriptor Table* (GDT) as well as multiple *Local Descriptor Tables* (LDTs). At any one time the GDT and one LDT are accessible: the GDTR and LDTR are hardware registers that point to the current GDT and LDT, respectively. Addressing is done indirectly through either the GDT or the current LDT. The 16-bit values in the segmentation registers are no longer addresses, as they are in real mode, but rather are indexes called *selectors*. The selector indexes to a descriptor in the GDT or LDT. The processor calculates the address of the location being accessed by adding the segment base address to an offset value. By making the result of the address calculation 24 bits wide in protect mode, the 80286 provides for a maximum physical storage size of 16M bytes.

Figure 1 shows the address generation of the 80286 in real and protect modes.

The 80286 and 80386 processors provide four levels of protection, called *protection rings*. The rings are numbered, with the most privileged ring being 0 and the least privileged being 3. Privilege is associated with segments, and the privilege level of a segment is stored in its descriptor. The current privilege level of the machine is the privilege level of the currently executing code segment.

Certain instructions relating to the processor state are designated as privileged and can only be executed at Ring 0. Other instructions relating to I/O operations are classified as trusted and require I/O privilege access, known as IOPL. In order to use these instructions, the current privilege level must be at least as privileged as the minimum privilege set globally for these trusted instructions. See page 10-5 of Reference 1 for a list of these instructions.

The processor provides a disciplined way of going from one privilege level to a more privileged level, called a *gate*. When a program calls a gate using the FAR CALL instruction and the gate points to a more privileged code segment, the processor automatically switches privilege level. When switching privilege levels, the processor automatically copies the parameters specified on the call to the stack used at the new privilege level.

Built into the 80286 there is a hardware tasking model which uses a *Task State Segment* (TSS) to describe the state of a task and the stacks that it is using. In order for privilege transitions in protect mode to work correctly, there must be a TSS. A hardware register, the TR, holds the address of the current TSS.

An *Interrupt Descriptor Table* (IDT) is used to vector interrupts to the routines that service them. This table is laid out differently from the low-storage interrupt vector table used in real mode, containing descriptors for special types of gates. This table is pointed to by a special register, the IDTR, that allows it to be located anywhere in physical memory rather than being restricted to low storage as is the interrupt vector table of real mode.

The DOS programming environment. DOS applications are coded to a well-established DOS programming environment. This environment is very tightly tied to the details of the hardware provided with the original 8088-based IBM Personal Computers. It is a very open environment in which the application writer has almost total freedom to control what the machine does. Although this is a significant advantage in many cases, it makes it difficult to move to more advanced architectures and make multiple applications run together on the same system.

The most important architectural features of the DOS programming environment are summarized in Table 3.

Capturing this open environment in a system that provides traditional operating system resource management is a significant technical problem.

DOS and DOS programs execute only in real mode. Although there is a BIOS[3] interrupt that puts the machine in protect mode, its usage is very limited and specialized; for example, there are no DOS system services available in protect mode. BIOS itself also assumes real-mode operation.

DOS assembly language programs commonly use segmentation registers for base registers and perform arithmetic on the values contained in them. Doing this when running in protect mode generally leads to protection violations. The segmentation registers cannot be used as base registers in protect-mode programs, since the values in them are indexes rather than addresses.

The DOS programming environment assumes an addressing maximum of 640K bytes for a usable programming area. Although the processor can address in real mode up to 1M bytes, the address range between 640K bytes and 1M bytes is reserved for BIOS read-only memory (ROM) code and video buffers.

Table 3 DOS programming environment

Real-mode execution on the 8088/8086/80286/80386
Modification of segmentation register values
Addressing range limited to 1M bytes
Use of both DOS and BIOS interrupt interfaces
Applications execute with full privilege, using all machine facilities
Multiple application programs supported with special features such as terminate and stay resident
Application programs may alter each other or DOS

Many of the interfaces that are commonly used by DOS programs are not provided by DOS but rather by BIOS. DOS programs are written to the combination of the DOS and BIOS interfaces.

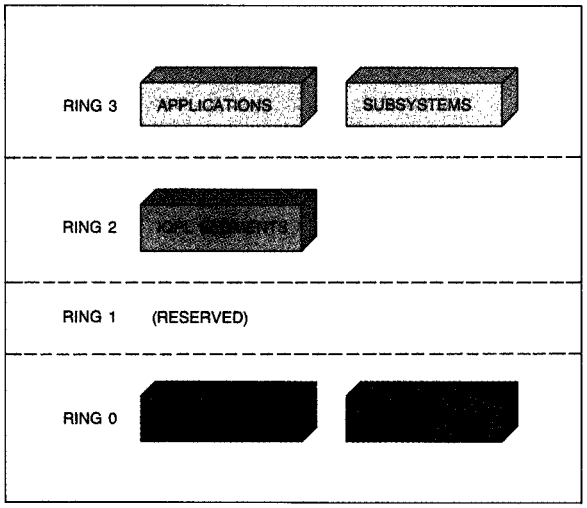
The DOS programming environment gives the programmer full access to the facilities of the hardware. The program may do low-level I/O instructions, disable and enable interrupts, and perform its own interrupt processing. The memory buffer that represents the display screen is directly accessible, so that programs may update the screen by storing information in the appropriate memory locations.

Interrupt vectors that are set up by DOS or by BIOS may be altered by application programs. This practice is usually called *hooking* an interrupt. The alteration is done by simply storing a pointer to the routine that the application wishes to get control of into the correct interrupt vector table location in low storage.

There is no protection between programs under DOS, which assumes that one program is running at a time. Although there are many schemes for actually executing multiple programs together, such as terminate-and-stay resident routines, no mechanism is provided by the system or by the hardware when running in real mode that enforces any type of separation between programs. There are also no system mechanisms for managing the concurrent execution of multiple programs. Although this may be an advantage when structuring a set of programs that are tightly coupled and cooperating, it is a significant disadvantage when programs from different sources are being collected together.

A number of alternatives to protect-mode operation have been proposed and implemented in the past. These alternatives certainly increase the utility of DOS, but all of these solutions are limited by two

Figure 2 OS/2 protection ring usage



important factors. First, they are external solutions, different from the processor architecture and the natural trend of development for systems based on the Intel processor family. They require special hardware, usually on the bus or on memory cards, which increases the total cost of the machine. Second, they are all mapping schemes. Although they increase the amount of physical memory that can be attached to the machine, they do not increase the instantaneously addressable memory that a program has. The programmer must divide the code and data into sections that do not have to be addressable concurrently and then must manage the hardware correctly to ensure that the correct section is mapped at any particular time. This procedure is complex, especially when using higher-level languages. For this reason, OS/2 uses protect mode to extend the addressing range of the application program.

System structure

Since OS/2 runs real-mode applications in real mode and protect-mode applications in protect mode, and since in protect mode the architecture of the 80286 requires the use of multiple protection rings in order to enforce privilege constraints, the structure of the OS/2 system is tailored to support these features.

Mode usage. Many portions of the OS/2 system are designed to run either in real mode or protect; these include

- Device management
- Interrupt management
- Mode switching
- Context switching

Code that can run in either mode is called *bimodal* code. The device drivers supplied with OS/2 are also implemented as bimodal code. The balance of the OS/2 system, including virtual memory management and the file system, is implemented as protect-mode-only code. Low-level system services are provided that permit system code to determine the current mode of the machine and to ensure that it is executing in the correct mode.

An attempt is made in OS/2 to minimize the amount of mode switching that is done on performance-critical paths such as device and interrupt handling, since mode switching is relatively slow, especially when going from protect to real mode. This led to the decision to use bimodal code for the device drivers. In comparison, services such as segment allocation that are usable only in protect mode are written as protect-mode-only code. Although it is used by real-mode programs, the file system is implemented as protect-mode-only code to reduce the amount of storage below 640K bytes required by the OS/2 system. When the file system is used by a real-mode program, there are two mode switches, one from real mode to protect mode when calling it and one from protect mode to real mode on return.

Protection ring layout. OS/2 uses three of the four protection rings of the 80286. Figure 2 shows use of the protection ring by the system.

The Ring 0 operating system code consists of two parts. The basic system services are linked together to form a program called the *kernel.* The programs that are used to run devices are separately loaded modules called *device drivers.* Both the kernel and the device drivers execute at Ring 0 because they require the highest level of system privilege in order to be able to handle interrupts.

Subsystems are system services and extensions that do not require hardware privilege. Both applications and subsystems execute at Ring 3. Ring 2 segments are used by subsystems and applications for code that executes instructions requiring I/O privilege. This allows these programs to do I/O operations that do not require servicing interrupts.

The kernel is implemented as a monolithic monitor with one entry and one exit. The OS/2 kernel is

nonpreemptible; when a program is executing in the kernel, it will not be preempted to run another program. However, a program may be interruped while in the kernel, so that interrupt routines can run and perform functions that have to be done within a relatively brief period of time, such as clearing I/O buffers. This kernel structure permits simplicity of implementation and provides acceptable performance for a single-user, interactive, but nonreal-time system such as OS/2. With this design it is essential that the length of time spent by any system call in the kernel be held to a minimum.

Implementing large memory

By using protect mode, OS/2 is able to use the full addressing range of the 80286. Memory is a key system resource that is laid out and managed carefully by the system. Figure 3 shows the memory layout of OS/2.

OS/2 programs execute as a *process*. In particular, each process has its own addressing environment. The tasking aspects of processes are covered in the section on multiprogramming and multitasking.

The kernel consists of one code and one data segment in low physical memory below 640K bytes and several code and data segments in high memory above 1M bytes. The low-memory segments are *tiled*, which means that these segments can be addressed with the same value in a segment register in real and protect modes. Tiling is accomplished by taking the real-mode virtual address of these low-memory segments, converting this address to a physical address, and placing this value into the base address field of the descriptor whose selector matches the segment value from the original real-mode virtual address. Tiling lets bimodal code execute correctly regardless of the current mode of the machine. Figure 4 illustrates the concept of tiling. The device drivers and the portion of the kernel that is in low memory are tiled.

Real-mode programs are run in an area below 640K bytes, so that they have the same addressing environment as they do in DOS. The split of the kernel into low- and high-memory segments is done to maximize the amount of space available to a DOS program running in real mode.

Virtual memory management. Although any individual segment is still limited to 64K bytes in size, a protect-mode program can use a large number of segments; thus, it may have an address space that is much larger than the 640K-byte limit imposed by DOS. In addition, OS/2 provides ways of allocating and using collections of segments as single units to permit the implementation of large data structures.

Since, as is standard in multitasking systems, the memory resources of individual processes have to be protected from interference by other processes and

Figure 3 OS/2 memory layout

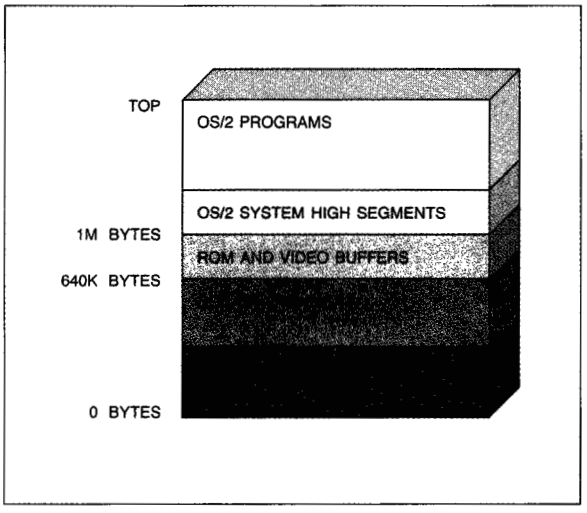
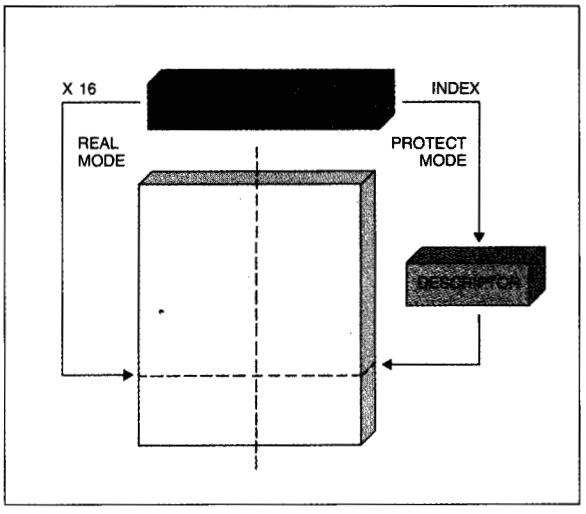


Figure 4 Tiled addressing



since those of the system itself have to be protected from interference by user programming, the descriptor tables are accessible only at Ring 0 privilege. Thus, the kernel must manage segments, determining their base addresses in physical memory and their lengths. This function is performed by the Virtual Memory Manager, which uses the services of the Physical Memory Manager (described in the subsection "Physical Memory Management") to allocate and deallocate physical memory as needed and to guarantee that memory can be overcommitted by the Virtual Memory Manager.

Although segments are allocated and deallocated by processes, the Virtual Memory Manager maintains a single, system-wide table, called the Handle Table, that describes all memory objects currently allocated in the system. Objects are mapped to segments and selectors by the Handle Table.

Since the smallest unit of storage managed by the kernel is a segment, OS/2 provides a subsystem to manage storage within a single segment.

Memory sharing. OS/2 supports memory sharing of code and data segments. The system provides both named and unnamed shared segments. The names of named shared data segments have the same form as the names of files. Processes gain access to named shared segments by knowing the name and using the shared-segment system calls. OS/2 permits sharing of application and subsystem code segments as well as global subsystem data segments. In fact, the entire OS/2 subsystem concept is built around the notion of memory sharing: Processes are almost always sharing segments with other processes. This is in contrast to systems like the UNIX® operating system which typically share only reentrant code between processes. See Bach[4] for a description of the UNIX process model and the recent enhancements made to it to support memory sharing between processes.

Segments in the Global Descriptor Table (GDT) are, of course, automatically shared by all processes as a result of the architecture of the 80286 processor. However, except for the kernel, most code and data segment descriptors are located in the Local Descriptor Table (LDT). In order to support segment sharing in a convenient way and in order to make sure that addresses can be passed correctly between processes for shared code and data, the Virtual Memory Manager ensures that if a slot is allocated for a shared segment in the LDT of one process, that slot is reserved in every other LDT in the system. This guarantees that the slot will be available should any other process request access to the segment. Each selector in the LDT is classified as either public or private. Public selectors are managed as sharable resources across the LDTs of all processes. Sharable items, such as segments that are allocated as being shared, and subsystem code and data are mapped into public selectors. The private selectors are used for segments allocated by the process owning the LDT. Reference counts and owners are tracked by the Virtual Memory Manager in the Handle Table.

Although this arrangement has the obvious cost of requiring that all LDTs be scanned on every allocation or deallocation of a shared segment, and although it increases the size of the LDTs in the system, it ensures that addresses can be shared validly across processes. As long as the size of the LDTs and the number of processes are both small, the cost is reasonable.

OS/2 descriptor table management. The GDT is statically allocated when the OS/2 kernel is built. It contains descriptors for global system segments, as shown in the memory layout of Figure 3. Resident and installable device drivers' code and data segments are also mapped into the GDT. In addition, the call gates for the callable interfaces implemented by the kernel reside in the GDT.

Aliases to processor-specific data structures are also in the GDT. The alias to the GDT itself is present, along with one LDT descriptor for the current LDT. There is also an alias to the LDT so that it can be referenced as data. Other aliases include an IDT alias, a descriptor for the current Per Task Data Area (PTDA), a TSS descriptor for the TSS of the system, a descriptor for the tiled selector to the ROM data area at 40:0, and a descriptor for an interrupt stack. The PTDA, which is the control block used by OS/2 to describe a process, will be described more fully in the next section. The rest of the GDT entries can be dynamically allocated by portions of the kernel and the device drivers.

There is one LDT for each process in the system. The fact that there is only one LDT descriptor and one PTDA descriptor in the GDT implies that these descriptors are remapped on a context switch by OS/2. PTDAs and LDTs are allocated when a program is started, and the LDT can be dynamically grown.

Physical memory management. OS/2 manages physical memory in such a way that more memory can be allocated than the machine actually has. This

feature, known as *memory overcommit*, allows the user more flexibility by permitting the system to execute programs that are larger than the amount of physical memory installed on the machine. Segments that are not actively being used may be swapped out to a fixed disk. The system brings them back into memory when they are needed. Since all of the memory used by a segment must be contiguous, OS/2 moves segments around in main memory in order to maximize the amount of free space available in physical memory. This placement is called *compaction* or *segment motion*. Code segments are discarded in overcommit situations rather than being swapped, since they can be reloaded from their original disk images when needed again. OS/2 uses segment swapping rather than paging to support memory overcommit because the 80286 processor does not have paging hardware but does provide the hardware support necessary to implement demand segmentation.

The areas of low physical memory used to run DOS programs and OS/2 itself are not subject to being moved or swapped. In addition, the system or an application may temporarily lock a segment in storage, so that, for example, an I/O buffer is guaranteed to remain fixed in physical memory until the I/O operation is completed.

When there is not enough contiguous physical memory to satisfy a request, the Physical Memory Manager attempts to compact memory to create a large enough space. If the compaction fails to produce such a space, the Physical Memory Manager performs a background operation to swap enough segments out of physical storage to permit the original request to complete. The swapper uses the file system to swap data to and from physical storage.

Since swapping and compaction impose some overhead on the system as a whole, the user may configure the system so that these functions are not performed.

The Physical Memory Manager tracks the usage of physical memory. Each block of physical memory, called an *arena*, has a header that describes it. All the arena headers are chained together in a doubly linked list. The Physical Memory Manager maintains two additional doubly linked lists, one for the free arenas and one for the allocated ones. The DOS Environment has a single arena that is not on the free list of the Physical Memory Manager. Memory management in the DOS environment is done by the DOS services running there.

Multiprogramming and multitasking

A *multiprogramming* system allows the concurrent execution of multiple applications. A *multitasking* system distributes processor time among multiple programs by giving each one short periods of time on the processor. OS/2 implements both multiprogramming and multitasking.

Sessions. OS/2 provides multiprogramming by supporting up to 16 concurrent *sessions*. Since the system uses four of these sessions for (a) the DOS environment, (b) the user shell, (c) the hidden session for detached processes, and (d) the hard-error handler, a user may start up to 12 concurrent sessions.

A thread is the unit of dispatching in the system.

OS/2 manages the video buffer so that the display can be used by multiple applications concurrently. Each session has a single logical display, keyboard, and mouse.

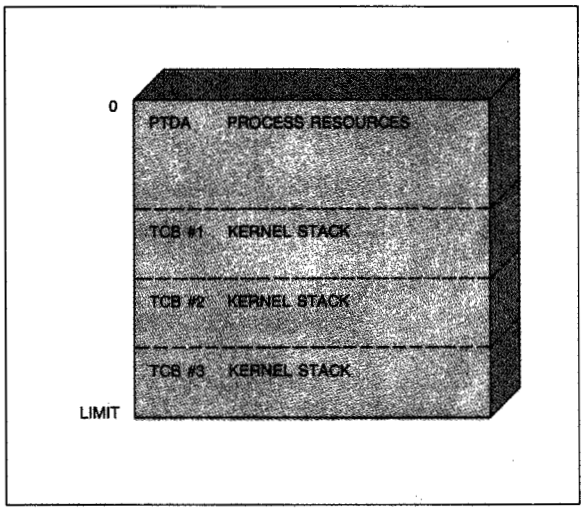
Processes and threads. The word *task* always refers to the hardware-defined task state. Although OS/2 executes multiple tasks concurrently, it does not use the concept of a task as it is defined by the 80286 architecture. Instead, OS/2 differentiates between processes and threads.

A *process* owns resources such as threads, file handles, semaphores, queues, and a memory map described by its own LDT.

The OS/2 system maintains a Per Task Data Area (PTDA) for each process in the system. The PTDA of each process is placed in a separate segment that is addressable via the GDT.

When a process is created, one thread is always created to run the code specified in the process creation system call. Thus, a process always has at least one thread. A *thread* is the unit of dispatching in the system. It is conceptually a stack and register

Figure 5 OS/2 PTDA segment



set that embodies a program. A process may have multiple threads sharing its resources. The system maintains a Thread Control Block (TCB) for each thread in the system. The TCB contains the register set and the kernel stack of the thread, and information for thread scheduling and I/O activity. Figure 5 shows the layout of the PTDA segment of a multithreaded process.

When a process is created, the system allocates a segment for the PTDA, including space for several TCBs. When subsequent requests to create threads exhaust the segment, the segment is resized to add space for more TCBs.

Dispatching mechanism. The dispatching mechanism in OS/2 is a software mechanism rather than the hardware mechanism defined by the 80286 architecture. Since the unit of dispatch is a thread, the system switches PTDAs when it switches to a thread that is not in the current process. Also, the stack fields of the TSS must be changed to reflect the correct stacks for each thread in the system. On a process switch operation, the physical address field in the LDT descriptor in the GDT is changed to point at the new target LDT, and the LDTR is reloaded to effect the LDT switch. The process switch operation is mostly completed when the hardware stack pointer (SS:SP) is reloaded to the new kernel stack and PTDA. A thread switch is the same as a process switch, except that the PTDA and LDT remain constant while the stack fields in the TSS are updated.

The decision to use a software dispatching mechanism is based on two major considerations. First, since the system sometimes runs in real mode, the hardware dispatching mechanism is not always available. Even when it is running in real mode, the system must track the currently executing program. Second, the dispatch mechanism is finely tuned, saving and restoring only those things that are absolutely essential to the correct preemption and resumption of execution. As a result, fewer clock cycles are expended in performing the OS/2 software context switch than would be required for an 80286 TSS switch.

Scheduling policy. OS/2 implements a preemptive time-slicing scheduler. A thread executes for some relatively short period of time before the scheduler gets control. When it does, it may determine that there is some other thread that ought to run. If so, it preempts the currently running thread by saving its state and dispatching the other thread. The OS/2 scheduler implements a multilevel priority scheme with dynamic priority variation and round-robin dispatching within priority level. Dynamic priority variation changes the priority of threads on the basis of their current activity. This is done to improve overall system performance and to make sure that the system responds rapidly to user interactions. Round-robin dispatching within a priority level ensures that if there is more than one thread at the same priority level, all of the threads at that level have an equal chance to execute.

For a single-user, personal-system operating system, OS/2 implements a sophisticated scheduling policy. This implementation permits the user to control the interactions of the programs being run on the system to make the whole system as responsive as possible to individual needs. Defaults built into the system permit it to run standard application mixes well, and built-in programming interfaces permit application programmers to control the behavior of their applications. This mechanism exists to ensure that the system provides the interactive response expected by the user.

Process synchronization and communication

Since it provides multiple processes and threads, OS/2 must also provide ways to coordinate their operation and to communicate among them, so that programmers can implement applications using more than one thread or process.

Semaphores. Since OS/2 applications may be multithreaded, it is important that they protect their resources. Common data areas shared by multiple threads in a process must be accessed in a serialized fashion. OS/2 allows an application to do this by using semaphores. A *semaphore* is a data structure which is owned by one thread at a time. If two different threads of a program need to access a data structure, each must request the semaphore first. OS/2 grants the semaphore to one of the requestors and blocks the other until the first relinquishes control.

A semaphore is a data structure owned by one thread at a time.

OS/2 has two types of semaphores: system semaphores and random-access memory (RAM) semaphores. The ownership of system semaphores is tracked by the system, and system semaphores may be used to serialize between threads in different processes. RAM semaphores are a very simple and very efficient form of semaphore that can be utilized to serialize among the different threads of a single process. When the thread owning a system semaphore terminates, the system notifies any pending requestors of the semaphore. However, it is up to the application using RAM semaphores to ensure that there are no deadlocks among threads requesting access.

Signals. *Signals* are used to notify OS/2 processes that some external event has occurred. A process can define signal-handler routines that are invoked when the various signals are received by the process. Among the signals defined for OS/2 processes are ones indicating that the user has keyed in the control-break function and that the process is to terminate itself. If there is no signal-handler defined when a signal is received, the system takes the default action, usually either ignoring the signal or terminating the process. If there is a signal-handler defined for the signal, it is invoked under the initial thread of the process. The system provides a function that permits a process to disable and to enable signaling.

Pipes. OS/2 provides both pipes and queues for interprocess communication. A *pipe* permits two processes to communicate by using the file I/O calls of the system. The first process writes data into the pipe, and the second reads data from the pipe. However, the data are never actually written to an external file but are kept in a segment in main memory.

Queues. The queuing system calls are implemented by a Ring 3 subsystem that uses shared memory, suballocation, and semaphores for serialization. These calls implement the mailbox model in which only the owner can read from the queue, but any thread can write to it. The owner can look at the elements on the queue, remove elements from the queue, purge the queue, and delete the queue.

Dynamic linking to services

OS/2 programs use *far call* instructions to invoke services from the system. The references generated by these calls are resolved when the program is loaded or when the segments of the program are loaded. This postponed resolution of references is called *dynamic linking*. The OS/2 load module format is a superset of the DOS load module format. It has been extended to support a demand segment-swapping environment with dynamic linking.

Dynamic linking reduces the storage requirements of OS/2 programs while making it possible to replace subsystems and service routines without relinking applications.

The dynamic linking mechanism. In protect-mode, whether a code segment selector or a call gate selector is used is transparent to the caller. When the request is for a system service that requires a privilege transition, a call gate is used. When the linkage is to a service provided by a subsystem executing at the privilege level of the application, the call is directly to a code segment selector.

Application structure. An OS/2 EXE module contains EXPORT and IMPORT records. When a program IMPORTs a far call reference, the linker does not attempt to resolve the far call to the function being imported and creates instead an IMPORT record in the EXE header. EXPORTs are usually found in subsystems; if a subsystem has EXPORTed a public far routine, the linker creates an EXPORT record in the EXE header to indicate that this EXE module contains candidates for target dynamic links.

Subsystem structure. Subsystems are also called *dynalink packages*, or dynamic link libraries, and reside in EXE files with the extension ".DLL." Subsystems are loaded into memory when they are referenced by an application at load time or run time. A subsystem does not have a stack segment but may contain Ring 3 and Ring 2 code segments. Subsystem modules also may contain data segments which are classified as instance data or shared global data. An instance data segment is allocated one per requestor of the package, whereas a shared data segment implies that the data segment is allocated once when the package is loaded. Since subsystems are mapped into the public selectors in the LDT, these slots are preserved in all LDTs, so that the subsystem appears in the same place in the address space of the process no matter which process is using the subsystem. Global data for subsystems are implemented by the loader allocating a public selector in the LDT for the segment, and then mapping this selector to a single copy of the data segment supplied by the subsystem module. This shared segment appears in the same place in all LDTs. Conversely, instance data are provided by the loader allocating one segment per LDT per process that has referenced the subsystem, and mapping the same public selector in each LDT to a unique copy of the data segment of the subsystem when the subsystem is first referenced by a thread in a process.

When a subsystem is loaded, an optional initialization routine supplied in the package will be run so that the subsystem is ready for clients. The initialization routine may be identified as instance or global depending on whether the subsystem writer wishes the initialization routine to be called once (global), or on a per-process-reference (instance) basis.

Load time dynamic linking. The OS/2 loader implements dynamic linking using the Module Table data structure which consists of Module Table Entries (MTEs). Each MTE is referenced by a module handle that is a pointer to an MTE. All the MTEs in the system are kept in a linked list of module handles. Each MTE contains the resident EXE header information extracted from the load module, including the exported entry points. Module tables are allocated as movable segments.

When the OS/2 loader attempts to load an EXE file, it builds an MTE if one does not already exist for the module. The MTE handle is stored in the PTDA of the requesting process. The MTE is then scanned for IMPORT records. Each IMPORT record tells the loader what module contains the code being referenced. IMPORTs can be by name or by ordinal number.

IMPORTs by name are references to subsystem code. If it is a Ring 3 code segment, the loader fixes up the program to call the Ring 3 code segment selector. If it is a Ring 2 code segment, the loader fixes up the program to call a Ring 3 call gate that points to the Ring 2 code segment. If necessary, the loader loads any subsystems referenced by the program. Subsystem loading is similar to application loading.

For IMPORTs by ordinal number the references are to the kernel, and the far call reference is fixed up to a Ring 3 call gate that contains a Ring 0 code segment selector and offset of the target. Ordinal numbers allow faster dynamic linking of kernel calls, since no search of the MTEs for the kernel is required.

When the system resolves dynamic links from Ring 3 to Ring 2, it allocates a Ring 2 stack. When there is movement from Ring 3 to Ring 0 through a DOS Call, the Ring 0 stack is already set up for the requesting thread in its TCB in the kernel. The TSS points to this as the Ring 0 stack.

Run time dynamic linking. OS/2 applications may explicitly load subsystems by using the DosLoadModule system call to load the subsystem that it wishes to use. The application then uses the DosGetProcAddr call to get the selector and offset of a given routine in the target subsystem. This permits routers to redirect requests to different subsystems at run time.

The program loader adds the explicitly loaded modules to the module table chain of the system and keeps track of them with a record structure containing the requestor's PTDA, the MTE handle for the runtime dynalinked module, and a reference count.

Demand loading. When the loader scans the MTE and finds segments with the PRELOAD attribute off, these are LOADONCALL segments that are demand-loaded. The loader postpones the actual loading of LOADONCALL segments; LDT selectors are allocated for them, but the descriptors are marked not present. On reference to the not-present segment, a not-present fault occurs, and the segment is demand-loaded. This condition may require the loader to do dynamic linking as a result of handling a segment-not-present fault.

Privilege transition on system call. When an application makes a system call to the kernel, the system

makes a privilege-level transition to Ring 0 by doing a far call to a Ring 3 call gate. Since the target code segment in the call gate is a Ring 0 code segment, the processor switches from the Ring 3 stack, which is allocated by the application during assembly or link time, to the Ring 0 segment specified in the TSS. Any parameters are copied from the Ring 3 stack to the Ring 0 stack by the 80286, and execution continues on the kernel stack of the thread which is allocated in the TCB. The dispatch mechanism ensures that when threads are switched, the Ring 0 stack fields of the TSS are changed so that the system always uses the correct kernel stack for the requesting thread.

On return, the system comes back to the caller, making the transition back to Ring 3. To be in kernel mode is not the same as to be executing at Ring 0. Once a system call has moved through the call gate,

OS/2 programs execute as a process.

the thread is executing at Ring 0, but it is not yet in kernel mode. Each system call entry handler invokes the System Call Interpreter, or SCI, which connects the Ring 0 entry point to its respective worker routine. A *worker routine* is an internal routine that executes the system call. SCI validates the stack-based parameters that are passed to the kernel and moves them to registers for use by the workers. SCI also ensures proper serialization for routines in the kernel. On return from the worker, it sets up the return status and returns through the call gate to the original requestor.

Application programming interface. OS/2 uses calls to implement its application programming interface (API), rather than the software interrupts (INTs) used by DOS and BIOS. The software-interrupt mechanism for invoking system services has a number of disadvantages, as follows:

- The number of INTs is very limited, restricting future growth.
- It is impossible to add new parameters to existing INTs in a compatible fashion, making it difficult to extend current interfaces.
- The software interrupt interfaces are easy to code in assembler, but they do not translate very well into higher-level languages.

Rather than using software interrupts, the OS/2 API is based on the dynamic linking mechanism of the system described above.

Extendability. The OS/2 API can easily be extended by adding new function names to the system library. Subsystems can also be added by introducing new dynamic link libraries. The application developer is provided with a single, consistent mechanism for accessing base system functions and extensions. The dynamic link mechanism enables the system to change APIs in future releases while maintaining compatibility for the existing calls.

Family API. There is a subset of the OS/2 API called the *Family API* which is supported by both OS/2 and DOS 3.3. Programs written to the Family API can be linked and bound to produce EXE files that are executable on OS/2 in protect mode, on OS/2 in the DOS environment, or on DOS. On OS/2 in protect mode, the Family API calls are handled in the usual manner by the system, whereas on DOS or in the DOS environment of OS/2, a special piece of code is bound to the application that is used to translate the Family API calls to software interrupts or to emulate the OS/2 system call in real mode. This code also loads and transfers control to the program, so that it seems to be executing in the OS/2 environment.

File system and I/O. Although there are many changes in the details of the implementation, the functions and overall structure of the I/O-related portions of OS/2 are very similar to those of the I/O support sections of DOS and BIOS.

File system. OS/2 uses the same file-system formats as DOS 3.3, so that media written by one system can be read by the other. Like DOS 3.3, OS/2 permits the user to organize the files on a disk or diskette into tree-structured directories. The file-naming and drive-letter conventions of OS/2 are the same as those of DOS 3.3. The OS/2 file system provides the same file-sharing features as DOS 3.3, so that a file can be shared among processes that are running concurrently. File access can be serialized among threads of the same process using semaphores.

Unlike DOS, OS/2 provides asynchronous file I/O operations as well as synchronous file I/O operations. This means that a program may issue a read or a write and then proceed with other processing while the I/O operation is being done by the system. When the program needs the data being read or needs to reuse the buffer containing the data being written, it can wait for a semaphore that is set when the I/O operation is completed by the system.

Also, since the file system runs under the thread of the caller or, in the case of asynchronous I/O operations, under the thread created to run the request, file system operations are multithreaded. This means that OS/2 does not wait for disk I/O activity to complete. Instead, it can dispatch other threads and do other work while the disk is returning the data requested.

For devices larger than 32 megabytes, OS/2 provides a method of partitioning the disk into logical drives, each with its own drive letter. This partitioning scheme is identical to that in DOS 3.3, so that media compatibility between the two systems is maintained.

OS/2 provides a single file system for all protect-mode programs and the DOS environment, and it enforces the file-sharing protocols between real- and protect-mode applications

Video, keyboard, and mouse. Although they are generally regarded as devices, OS/2 implements most of the device support code for video, for the keyboard, and for a mouse using subsystem code. There are device drivers for each, but they contain only a small portion of the code required to operate the devices. This arrangement permits the user to replace the OS/2-provided device support with customized support on a per-session basis.

The system manages sessions so that when a session is in the foreground, a process performing video output sends it directly to the hardware video buffer. When the session is moved to the background, the hardware video buffer is saved to a logical video buffer for the session. While the session is in the background, the process sends its video output to this logical video buffer. When the user returns the session to the foreground, the system copies its logical video buffer to the hardware video buffer. Keystrokes and mouse inputs are directed to one of the processes in the foreground session. There are programming interfaces to control which process receives them.

Device drivers. Like DOS, OS/2 has two types of device drivers—character device drivers and block device drivers. As with DOS, OS/2 device drivers may be broken down into strategy and interrupt routines. However, because OS/2 device drivers must operate in a multitasking environment, they must be written to relinquish control whenever they are forced to wait for I/O operations to complete. Device drivers are bimodal, tiled, Ring 0 code. Since they are bimodal, an asynchronous I/O operation may be started by a device driver in one mode, but the device driver may receive the interrupt for it in the other mode. Since the addressing structure is different in real mode and protect mode, device drivers translate virtual addresses to physical addresses and store the physical addresses for use at interrupt time. The physical addresses are guaranteed to be constant, independent of the mode of the processor.

OS/2 executes real-mode programs one at a time in low storage.

To assist in implementing such device drivers, OS/2 has a common interrupt manager that handles all hardware interrupts and routes them to the correct device driver. Also, the system provides a number of services called device driver helper, or DevHlp, routines that a device driver may call. These DevHlp calls provide access to kernel services, including some that are specifically tailored to assist in the implementation of device drivers. Among the functions provided are

- Converting virtual addresses to physical addresses
- Converting physical addresses to virtual addresses
- Segment locking
- Semaphore handling
- Request queue management

The kernel provides a DevHlp router that converts DevHlp calls to an interface to the kernel worker routines similar to the one produced by the System Call Interpreter for dynamic link calls. More information on OS/2 device drivers can be found in the paper by Mizell.[5]

Compatibility with DOS

OS/2 executes real-mode programs one at a time in low storage. Real-mode programs run only when the DOS environment is selected as the foreground session. Protect-mode applications may continue to execute in the background. When a real-mode program is put into the background by the user, it is frozen and does not run again until the user returns it to the foreground. This means that real-mode data communication functions cannot be reliably supported in the DOS environment when the user switches to a protect-mode OS/2 application.

The software interrupts issued by real-mode programs are serviced by OS/2. Included are both the DOS software interrupt function calls and the BIOS software interrupts. This arrangement permits the real-mode application and the protect-mode applications to share common resources and services such as the file system under the control of the operating system. Some BIOS functions which are particularly difficult to emulate are given to the ROM BIOS itself to execute. In that case, the bimodal device driver uses system services to serialize the use of ROM BIOS, so that the BIOS code operates correctly.

Because of the requirements of OS/2 for storage below 640K bytes, there is less application space available in the DOS environment than there is in a system running DOS 3.3.

A DOS program may hook an interrupt vector. When the DOS environment is dispatched, the interrupt vector table is set up to match the table expected by the DOS program. When OS/2 regains control, it compares the interrupt vector table with its previous image and alters the protect-mode IDT if necessary. If the system detects that the DOS program is hooking an interrupt in order to use a hardware resource, it ensures that this usage does not conflict with the usage being made by a protect-mode device driver or program. This feature is called *interrupt table shadowing*.

Although OS/2 is bimodal, the DOS environment does not get scheduled like protect-mode threads, since it cannot run in the background. There is a DOS PTDA in low, fixed memory.

Commands and utilities

The command processor for OS/2 is an extension of the COMMAND.COM command processor of DOS. With minor exceptions, the familiar command set of DOS is supported, and the syntax of the commands and the meaning of the parameters to the commands are the same.

There are some new commands that are unique to OS/2. In particular, OS/2 provides a START command that permits the user to start a program in another session and let it execute asynchronously to the command processor.

OS/2 supports a compatible extension of the DOS batch file language. This extension permits old batch files to be brought over from DOS and used on OS/2. Unlike DOS, OS/2 commands routinely set the error level, so it is easy to determine whether a command has succeeded.

The STARTUP.CMD command file, if present, is executed during system start-up and may be used to contain initialization commands. In addition, there may be an OS2INIT.CMD file that is executed at the beginning of each CMD.EXE session. This file is the functional equivalent of the traditional AUTOEXEC.BAT file of DOS. The AUTOEXEC.BAT file, if present, is executed when the DOS environment is started.

The utilities are implemented using the Family API, so the same programs can be used as real-mode programs in the DOS environment or as protect-mode programs. However, since the programs are able to determine the mode in which they are executing, some of them provide functions that are specific to one mode or the other. The Family API was used to implement the utility programs to save disk space, since only one program file is required per utility rather than two, as would have been necessary had there been both real- and protect-mode versions of the utility.

The DOS environment uses the COMMAND.COM command processor of DOS to process commands entered at the keyboard.

Summary

This paper has focused on the design of OS/2 Standard Edition Version 1.0, emphasizing the architectural constraints on the design of the system together with the requirements for providing a system that is both relatively small and fully functional.

OS/2 is a logical extension of IBM DOS 3.3 that provides

- Large real memory beyond 640K bytes
- Multiprogramming
- Multitasking

Yet OS/2 retains the ability to run most DOS applications. Thus, it is able to provide a bridge from the heritage of single-tasking, memory-limited personal systems of the past to the more complex but more powerful multitasking, large-memory systems of the future.

Cited references and note

1. *iAPX 286 Programmer's Reference Manual*, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051 (1985).
2. *iAPX 286 Operating Systems Writer's Guide*, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051 (1983).
3. BIOS stands for Basic Input Output System. BIOS is the low-level device support code that is packaged with the machine.
4. M. J. Bach, *The Design of the UNIX® Operating System*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1986).
5. A. M. Mizell, "Understanding device drivers in OS/2," *IBM Systems Journal* **27**, No. 2, 170–184 (1988, this issue).

General references

Disk Operating System Version 3.00 Technical Reference, 6138536, IBM Corporation (October 1987); available through IBM branch offices and authorized dealers.

iAPX 86/88, 186/188 User's Manual, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051 (1983).

Operating System/2, IBM Personal System/2 Seminar Proceedings, G360-2758, IBM Corporation; available through IBM branch offices.

Operating System/2 Technical Reference Volume I, 84X1434, IBM Corporation (October 1987); available through IBM branch offices and authorized dealers.

Operating System/2 Technical Reference Volume II, 84X1440, IBM Corporation (October 1987); available through IBM branch offices and authorized dealers.

Michael S. Kogan *IBM Entry Systems Division, P.O. Box 1328, Boca Raton, Florida 33432.* Mr. Kogan is currently a Staff Programmer in the OS/2 Design Department, where he is involved in the architecture and design of Operating System/2. In 1984 he joined IBM in Boca Raton, where he developed and tested several products in the IBM Engineering/Scientific Series. He then participated in the development and testing of IBM Personal Computer XENIX. Mr. Kogan holds a B.S. degree in computer science and mathematics from Emory University and an M.S. degree in computer science from Nova University; he is currently working toward an Sc.D. in computer science at Nova University.

Freeman L. Rawson III *IBM Entry Systems Division, P.O. Box 1328, Boca Raton, Florida 33432.* Mr. Rawson is currently a Senior Programmer in the Advanced Systems Architecture Department, where he is involved in the design and enhancement of Operating System/2. When he joined IBM in 1973 in San Jose, California, his first assignment was in the development of data management utilities for System/370 operating systems. Mr. Rawson transferred to Boca Raton in 1976 to work on the development of the Realtime Programming System for the Series/1 computer. In 1986, he joined the OS/2 design organization to work on its support for the IBM Personal System/2.

Reprint Order No. G321-5312.