Writing an Operating System/2 application

by R. L. Cook
F. L. Rawson III
J. A. Tunkel
R. L. Williams

This paper illustrates use of the key facilities of Operating System/2™ (OS/2™). It provides some guidance on how to use the interfaces and functions implemented by the system and then introduces the program development environment. Two examples demonstrate the use of some of the more interesting capabilities. The paper discusses many of the significant differences between the functions of OS/2 and those of the Disk Operating System (DOS).

A new operating system must provide a way for its users to convert existing applications and to develop new ones. There must be suitable interfaces to the functions provided by the system, in addition to the programming tools that are needed to create executable code.

Operating System/2™ (OS/2™) offers the programmer a comprehensive set of interfaces. In addition, programming languages and tools provide the means for creating OS/2 applications.

Since OS/2 is intended to be a general-purpose operating system for Intel 80286-based personal computers such as the IBM Personal System/2®, it has many functions, and rules and principles guide the application developer in the use of the various features of the system.

Using the new features of OS/2

Although much in OS/2 should be familiar to IBM Personal Computer Disk Operating System (DOS) programmers, many new features have no analog in DOS. The creation of reliable and efficient applications requires that these facilities be used in the manner intended by the OS/2 designers. The following discussion, though not all-inclusive, touches upon those areas that have proven to be the most important to designers of OS/2 applications.

For more information on the design of OS/2, including a more detailed presentation of its terminology, see the paper in this issue by Kogan and Rawson.[1]

Performing tasks. OS/2 multitasking uses the concepts of processes and threads.

A *process* represents the execution of a program. Processes have an identifier known as the Process ID (PID), which is often needed when requesting services from OS/2. A process owns resources—storage, files, pipes, queues, semaphores. A process also has at least one thread of execution. Processes are protected from other processes by OS/2 and the 80286 processor architecture, and coordinate their actions by means of the synchronization mechanisms described in the next subsection.

A *thread* is the unit of dispatchability within OS/2. Threads have an identifier known as the Thread ID which is only valid within a process. Whenever a Thread ID is used in a request for OS/2 services, it is assumed to refer to a thread within the current process. Threads, in general, do not own resources; threads use the resources of their process. Threads have full access to the resources of their process and thus are not protected from other threads within the same process. Threads can cooperate with other threads within the same process through the use of synchronization mechanisms.

Processes require allocation of resources when they are started. Hence, there is inherent overhead in starting a process. Threads can be created and destroyed rapidly with little overhead.

OS/2 provides various services that may be used to manage processes and threads. The following services are primarily used to manage processes:

DosExecPgm starts a program, executing it as a new process. The requesting process can either continue executing asynchronously, or it can wait for the new process to terminate.

DosExit ends the current thread within the current process or the current process and all threads under it, furnishing a Result Code to any thread which has issued a DosCwait for this Process ID. Even a process that has not created any additional threads should issue a DosExit for the entire process rather than for the thread, since it is possible that a subsystem or OS/2 itself has created additional threads on behalf of the process.

DosKillProcess requests the system to terminate the indicated process and, optionally, any child processes. If the indicated process has a SigTerm signal handler, which is set by the DosSetSigHandler function call, it may intercept the Kill Process request and decide whether to terminate at this time.

DosGetInfoSeg retrieves system-wide and local variables of the process under which it is invoked. This call provides pointers to two read-only areas which may be directly read by the process. The information available includes date and time, OS/2 version information, information on current and active sessions, process IDs, and process and thread priority.

The following services are primarily used to manage threads:

DosCreateThread creates another asynchronous thread of execution within the current process. The new thread is of the same priority as the current thread and has equal access to any process resources.

Time-critical threads execute before any other execution classes.

DosSetPrty sets the execution priority or priority class of a process or of a thread. There are three priority classes, and each class has 32 priority levels. In descending order, these classes are

1. Time-critical
2. Regular
3. Idle-time

Time-critical threads execute before any other execution classes. OS/2 never modifies the priority of a time-critical thread: Only an explicit request using DosSetPrty can change its priority. Regular threads have a base priority level, but OS/2 modifies this level around the base in order to most effectively allocate the processing resources of the system. Effective allocation ensures optimum user responsiveness. Idle-time threads are of the lowest-priority class and, like the time-critical class, have static priority levels. Most processes and threads should belong to the Regular priority class.

DosCwait waits until a child process has ended and then returns the Process ID and the Result Code of the ending process.

DosSuspendThread suspends the execution of another thread in the same process. The target thread is not suspended until it has released any system resources it may be using. The thread can be restarted by issuing the DosSuspendThread function call.

Handling critical sections and task synchronization. The problem of mutual exclusion is well-known in the computer science literature. Any system that provides multitasking capabilities must also provide

mechanisms for guaranteeing mutually exclusive access to resources by several processes or threads. An example is the manipulation of a linked data structure that is accessible to multiple threads.

OS/2 provides several such mechanisms. The simplest mechanism permits a thread to suspend the dispatching of all other threads in its process. A critical section of execution is defined as a set of instructions whose sequential execution cannot be interrupted by other threads in the same process. To accomplish this, a thread may issue a *DosEnterCritSec* function call, which disables thread switching for the current process. Thread switching can be enabled after the critical section of code has completed execution by issuing the *DosExitCritSec* function call. To prevent deadlocks, a thread should not make OS/2 system calls while in a critical section.

Differences between RAM and system semaphores are primarily in performance and reliability.

An advantage of this mechanism is that it does not require the cooperation of the other threads in the process. A disadvantage is the easy creation of deadlock situations if the critical section thread waits for a semaphore "owned" by another thread in the process. Since subsystem services are often serialized by means of semaphores, two threads using a subsystem (for example, the video subsystem) could deadlock if one of them makes a video call within a critical section.

A more general synchronization mechanism is provided through the use of semaphores. *Semaphores* are objects that can be owned, unowned, set, and cleared atomically. Semaphores require that the processes or threads cooperate in their use of the resource. This cooperation takes the form of explicit requests and releases of the semaphores. The state of a semaphore at any given time is unambiguous.

OS/2 provides two types of semaphores: random access memory (RAM) semaphores and system semaphores. A number called a *handle* represents a semaphore to a program. A RAM semaphore is a minimal-function, high-performance mechanism allowing two or more threads, or processes accessing shared memory, to synchronize through a double word in memory. A system semaphore is a full-function mechanism allowing synchronization among any processes or threads in the OS/2 system. The storage for the system semaphore data structure is allocated and managed by OS/2.

The differences between RAM and system semaphores are primarily in performance and reliability. RAM semaphores are intended to be used by threads within a process. Since the handle used to access a RAM semaphore is actually the semaphore address, access is very fast. However, RAM semaphore ownership is not tracked by OS/2, and should the owner of a RAM semaphore end abnormally, or end without releasing the semaphore, OS/2 does not recover the semaphore nor give it to the next thread requesting it. System semaphore ownership and allocation are tracked and managed in protected system storage. Should the owner end without releasing the semaphore, OS/2 has the ability to release the system semaphore on its behalf and notify the next thread.

RAM semaphores are created by allocating a double word of memory within the address space of the application and initializing it to zero. System semaphores are created by OS/2 upon the request of a process. *DosCreateSem* creates a new system semaphore. A system semaphore must be created before any process can use it. Once it is created, other processes must OPEN it before they can use it. The *Exclusive* option allows the creation of a semaphore that may only be cleared by the owner. *DosOpenSem* opens a system semaphore and returns a handle to be used by the current process when accessing the semaphore. *DosCloseSem* closes a system semaphore. Even though OS/2 closes system semaphores for a terminating process, the process should endeavor to close the semaphores itself as part of its termination.

Once a system semaphore has been created by OS/2, or a RAM semaphore has been allocated in memory, the same set of semaphore function calls is used to manipulate it. *DosSemRequest* requests a semaphore and, optionally, waits if it is already owned. If the semaphore is unowned, it is set "owned." Later the semaphore may be set "unowned" with the Dos-

SemClear function call. The double-word data structures for RAM semaphores should be initially set to 0, "unowned." If the required initial state is to be "owned," the application should issue DosSemSet. When system semaphores are allocated, they are also initialized to unowned. If the required initial state is to be owned, the application should issue DosSemSet immediately after obtaining the semaphore handle with DosCreateSem.

A process can specify a signal handler to intercept specific signals.

DosSemClear releases the semaphore. If any threads were waiting for the semaphore to become unowned, they are made dispatchable in the order in which they requested the semaphore. This call unconditionally clears the semaphore regardless of its current state. DosSemClear cannot be issued against an exclusive system semaphore that is owned by another thread.

DosSemSet sets a semaphore as owned.

DosSemSetWait sets a semaphore as owned and waits until another thread issues a DosSemClear for this semaphore.

DosSemWait waits for a semaphore to be cleared. The semaphore is not marked owned.

DosMuxSemWait waits until one of a list of semaphores is cleared. DosMuxSemWait does not set any of the semaphores owned. If any of the list of semaphores is clear, DosMuxSemWait returns immediately; otherwise, the thread is blocked until one of the semaphores is cleared.

All of the semaphore wait function calls, except DosMuxSemWait, return only if the subject semaphore has been cleared and remains cleared until the waiting thread is dispatched. It is possible for a subject semaphore to be cleared and set again before a waiting thread is dispatched; the thread continues to wait. DosMuxSemWait returns whenever any of the semaphores on its list is cleared, regardless of how long it may remain cleared. Therefore, it is possible for DosMuxSemWait to return from waiting and find that the triggering semaphore is set again.

Dealing with external events. OS/2 *signals* provide a way for the system to notify a process of an asynchronous, external event and for the process to handle the event if desired. Events that can appear as signals to a process are the pressing of either the Ctrl-Break keys or the Ctrl-C keys on the keyboard or calls to DosFlagProcess or DosKillProcess that point to the process as their target.

A process can specify a signal handler to intercept specific signals. Incoming signals are handled either by default action or by the signal-handler routine. The default action for some signals is to ignore the signal; for others it is to terminate the process. If a signal handler has been specified, thread 1 of the target process, the original thread which was created when the process was created, is diverted, in a manner similar to a hardware interrupt, to the proper signal-handler routine.

Incoming signals are presumed by the system to represent a time-critical event. If thread 1 is in the middle of a system call that cannot complete quickly, the system call is aborted, and control is passed to the proper signal handler. Typically, the calls that are aborted in this manner are device I/O calls, semaphore waits, and DosSleep. File system calls are not normally aborted.

An application that expects to make nonemergency use of signals, through DosFlagProcess for example, should reserve thread 1 and use another thread for program execution. Thread 1 is reserved by having it wait for a semaphore that is never cleared.

Processes may, by agreement, define external events which one process may signal to another. *DosFlagProcess* provides the mechanism for activating the signal. Three flags, or signals, may be sent with DosFlagProcess—Flag A, Flag B, and Flag C. Additionally, a one-word argument of undefined semantics may be sent along with the signal. If the target process has not set an appropriate signal handler, the default action, ignore, is processed. Otherwise, the signal is handled by the signal handler.

The Ctrl-Break or Ctrl-C signals may be sent to a process subtree, a collection of child and sibling

processes originating at a root process, by the function call *DosSendSignal*. The signal is given to the last-created child process in the subtree that has a corresponding signal handler. If, in the search for this process, the issuer of the DosSendSignal is checked, an error code is returned.

In many ways, processing of signals is analogous to the processing of interrupts. The requirement exists to disable and enable signal processing to protect critical sections of code. *DosHoldSignal* can be issued with a disable request to postpone the processing of all signals until it is again issued with an enable request.

DosSetSigHandler, the primary function call for signals, can install signal handlers for a process, install the default signal action, install an ignore action for a signal, or reset signal handling after the receipt of a signal. When activated by its associated signal, a signal handler must itself issue DosSetSigHandler with the appropriate action code to acknowledge the signal and re-enable recognition of that signal.

The signal to terminate may be sent to a process by invoking the system call DosKillProcess. If no SigTerm signal handler has been set by the target process, the effect is the same as if one of the threads of the process had issued a DosExit for the entire process.

Managing memory in an application program. Many aspects of application program memory management for OS/2 should be familiar to DOS programmers, many are new to DOS programmers but may be familiar to users of mainframe systems, and some originate with OS/2. The OS/2 memory model is based on the Intel 80286 architecture which defines storage as segments. Segments are contiguous regions of storage of variable length ranging in size from 1 byte to 64K bytes.

OS/2 provides four main memory management services to the application programmer.

Functions exist in OS/2 to allocate and deallocate segments of storage at the request of the application. A segment is identified by a 16-bit value called a selector. Segments are allocated in varying sizes up to 64K bytes, but the capability also exists to handle large storage needs by allowing the allocation of multiple 64K-byte segments. Programs may not treat multiple segment allocations as contiguous memory. Segments allocated by one process may be shared among several processes. Segments may be shared by name or by passing selectors.

Data segments marked as discardable may be created, and they can be discarded by OS/2 during periods of high storage use. Discardable data segments should only be used when the data that they contain can be quickly regenerated. When discardable segments are being used by an application, they should be locked, thus preventing them from being discarded until they are unlocked. If a program attempts to lock a segment that has been discarded, it receives an error code from OS/2, indicating that the segment has been discarded.

In general, OS/2 memory management handles entire segments. Quite often, however, an application needs to subdivide its segment allocation into smaller data-element-sized units. OS/2 provides a standard mechanism for achieving this downsizing without incurring the overhead of segment allocation. Suballocation provides a high-speed mechanism for managing storage within a segment.

Allocating memory. DosAllocSeg allocates a memory segment ranging from 1 byte to 64K bytes. The segment thus allocated can be flagged via *DosGiveSeg* or *DosGetSeg* as sharable and/or discardable. If the allocation is successful, a selector is returned pointing to the allocated segment.

DosFreeSeg can be used to free (unallocate) shared or unshared memory segments. When freeing segments, OS/2 decrements a usage count. When the count reaches zero, the segment is freed.

The size of storage segments can be changed as desired by the application within the range of 1 byte to 64K bytes by using *DosReallocSeg*. However, if a segment is shared, it may only be increased in size. Issuing DosReallocSeg for a segment that is discardable causes it to be locked in storage.

DosMemAvail returns the size of the largest available block of free storage at the moment of execution. Since this value is likely to vary at any time as a function of system activity, it should be used as an approximation.

Allocating objects as discardable allows the memory manager to reclaim their space when the system is low on memory and they are not locked. Prior to using a discardable segment, an application should

lock it with *DosLockSeg*. A locked discardable segment may still be swapped to disk if necessary, but it is not discarded until unlocked with a *DosUnlockSeg* request. Locking a discarded segment results in an error code. The application must then reallocate the segment and rebuild the data, if necessary. A use count is kept of the Lock and Unlock requests issued for a discardable segment. If the use count reaches 255, the segment is considered to be permanently locked.

OS/2 can make large amounts of memory available to an application.

Sharing memory. Memory is shared on a segment basis. There are two sharing mechanisms, one based on names and the other on selectors.

Processes can share named segments by agreeing on names and then obtaining the memory with *DosAllocShrSeg* or *DosGetShrSeg*. The naming convention used for named segments is the same as the file-system naming convention, although no files are involved. Shared segments must all be named \SHAREMEM\nnnnnnnn.eee, where n and e conform to the file-system name rules.

Processes can share unnamed segments by allocating them as "sharable by GetSeg" or "sharable by GiveSeg." Additional interprocess communication is required to pass the sharable selectors among the cooperating processes.

If it is assumed that a segment is allocated as "sharable by GetSeg" and that the selector to that segment has been passed to the current process, access to that segment can be obtained by issuing DosGetSeg.

DosGiveSeg is analogous to DosGetSeg, except that the segment must be allocated as "sharable by GiveSeg," and the process ID of the sharing process must be available. Typically the sharing process is a child of the current process. The selector of the child process to the shared segment is passed to the process through some means of interprocess communications unrelated to the shared segment in question.

Allocating huge segments. OS/2 has the ability to make large amounts of memory available to an application. Huge allocations simplify use of memory objects greater than 64K bytes. A huge allocation is a group of 64K-byte segments that can be allocated, shared, and freed as a unit. The selectors that point to the segments of a huge allocation are algorithmically derived from the selector that points to the initial segment of the allocation.

DosAllocHuge allocates a number of 64K-byte segments. The segments may, as a group, be "sharable by GiveSeg," "sharable by GetSeg," or discardable. Whenever any selector of a huge allocation is shared, locked, or discarded, the operation is performed on all selectors of the allocation. The segments are allocated with noncontiguous selectors, and only the first selector is returned to the requestor. The selectors to the succeeding segments can be calculated from the first selector and a shift count by the following algorithm:

Obtain shift count by issuing DOSGETHUGESHIFT

Place the value 1 in a register and shift left by the shift count

Add this value to the previous selector to obtain the next selector

DosGetHugeShift returns the shift count to be used in the algorithm required to calculate the selectors needed to access a huge allocation.

DosReallocHuge expands or reduces the size of a huge allocation (up to the maximum specified in the original allocation). The algorithm used to calculate selectors remains the same, but more or fewer valid selectors may be available.

When freeing segments obtained with DosAllocHuge, supplying the selector to the first segment frees the entire set of selectors for the huge storage.

Segment suballocation. Subdividing memory within a segment is something that DOS programs have typically had to do for themselves. OS/2 provides a simple high-speed mechanism that has little system overhead. Segment suballocation cannot be used across a huge allocation.

DosSubSet initializes a segment for suballocation. This function is also used if the size of a segment is expanded.

DosSubAlloc and *DosSubFree* respectively allocate and free intrasegment memory. Memory is managed in units of four bytes.

Program access to hardware. It has been common practice on single-tasking systems such as DOS for application programs to query and control hardware devices directly. This activity is not desirable on OS/2 because of the potential for conflict among concurrently executing programs. OS/2 programs typically access hardware by making requests to OS/2 subsystems, for example, the video, mouse, or keyboard subsystems, or device drivers through the Device I/O Control (DosDevIOCtl) Interface. There are, however, situations in which direct hardware access is necessary, and OS/2 provides two mechanisms for doing so.

The primary mechanism for accessing and controlling hardware devices is the *device driver*. There are OS/2 device drivers to support the standard IBM hardware for the Personal Computer AT®, the Personal Computer XT/286, and the Personal System/2 Models 50, 60, and 80. For special-purpose adapters and devices, the programmer must write specific device drivers. See the paper in this issue by Mizell for a description of OS/2 device drivers.[2]

The Intel 80286 processor architecture defines four privilege levels for programs and data. These privilege levels are called *rings* and are numbered 0 through 3. Device drivers run at the most privileged level of the processor, Ring 0, and have complete access to the hardware and hardware interrupts that are generated by adapter cards. Device drivers are the only system component that can receive hardware interrupts. The I/O Control function call, DosDevIOCtl, is provided to permit application programs and subsystems to communicate with device drivers.

In some applications the program may need to access hardware directly, yet has no need to process device interrupts. Examples of such programs are full-screen graphics applications that directly manipulate the video hardware, including the video controller registers. To implement these applications on OS/2, one creates a separate code segment containing the code that does the I/O instructions and gives it I/O privilege, or *IOPL*. IOPL code segments run at the Ring 2 privilege level and can issue the 80286 I/O instructions IN and OUT. IOPL code segments are also used in building subsystems that work in conjunction with device drivers to provide device support. Although Clear Interrupt (CLI) and Set Interrupt (STI) instructions may be issued from IOPL code segments, these instructions do not guarantee atomicity of execution, since Not Present faults for swapped segments may occur when loading segment registers. It is a better coding practice to use semaphores to control access to critical sections of code.

OS/2 provides both dynamic linking and independently loadable segments.

The OS/2 development environment

OS/2 and its associated language and toolkit products offer a program developer a complete set of development tools for converting DOS applications and writing new applications. The OS/2 development environment should be familiar to programmers who have written software for DOS. Many of the concepts are similar, and the programming tools are enhanced versions of their DOS counterparts. In fact, it is possible to use the OS/2 programming tools to create DOS programs.

The OS/2 program developer may choose from a number of language processors that run on OS/2 and can generate object code for either OS/2 or DOS. The new OS/2 language processors are C/2, COBOL/2, FORTRAN/2, Macro Assembler/2, Pascal/2, and BASIC Compiler/2.

The OS/2 Linker is based on the DOS Linker, but it provides new features to support the new facilities of OS/2. Unlike DOS, OS/2 provides both dynamic linking and independently loadable segments. Dynamic linking is an OS/2 facility which allows *late binding* of program segments. The binding between segments can be delayed until load time, run time, or until

needed during program execution. This delay is an advantage because binding may never actually occur in some cases. Segments may be statically linked into the executable module as in DOS, dynamically linked and preloaded by the OS/2 program loader at program initiation, dynamically linked and loaded on call when an actual reference is made by the program, or explicitly loaded by the program using the facilities of OS/2.

Dynamic Link Libraries are repositories of code and data segments created by the Linker from the object modules produced by the language translators. Application programs may explicitly import code and data segments at load time or run time. OS/2 provides a number of its facilities using dynamic link libraries, and programmers may create their own if desired.

Family Applications. The developer may wish to create applications that can execute on either DOS or OS/2. OS/2 applications that restrict themselves to a defined subset of the OS/2 interfaces called the *Family Application Programming Interface*, or *Family API*, may be implemented in such a way that either DOS or OS/2 can load and execute them.

The OS/2 Toolkit provides the mechanism for building such Family Applications. The functions of the Family API are OS/2 functions and are implemented directly by OS/2. On DOS a layer of mapping code maps each function to a corresponding DOS or Basic Input Output System (BIOS) programming interface. As a result, the program executes correctly on either DOS or OS/2.

The Family API mapping is created by an OS/2 Toolkit utility called the Family API Binder (BIND). The BIND utility is run after the compile and link steps. Its input is the OS/2 application (.EXE) plus the Family API mapping library (API.LIB), and it produces a Family Application (.EXE) version of the same program. The OS/2 Loader will load only the OS/2 segments of the .EXE file, whereas the DOS Loader will bring into memory all the code necessary for the applications to run in a DOS system.

Message files. Modern programming practice dictates that text strings, such as user messages, should not be inextricably bound into a program. Rather, they ought to be gathered together into data objects, separate from program logic, where they can easily be modified without danger of impacting any of the program. Following this practice also allows simple language translation of application programs: Only the data objects that contain the messages need to be modified during the translation process.

The OS/2 Toolkit provides two utilities for creating such objects in the form of message files and message segments, and the OS/2 system provides function calls to retrieve, modify, and display these messages. OS/2 uses this mechanism for all of its own messages and is distributed in versions with 11 different message files to support 11 national languages.

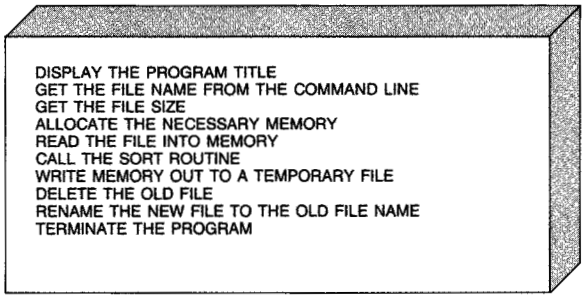
The utility Make Message File (MKMSGF) converts an ASCII text message file created by the programmer or the message translator into an indexed file suitable for rapid retrieval by the DosGetMessage function call. The utility Message Bind (MSGBIND) places messages from the message file into a message data segment that is bound to an executable module. This is useful for those instances when very rapid display of messages is required or when, for diskette-resident programs, the disk containing the message file is not available. Programs that use bound messages are guaranteed that the messages are in storage when the program is executing. Since the DosGetMessage function call checks for the message in the message segment before accessing the message file, no programming changes are required to use the MSGBIND utility.

Implementing some simple programs

This section presents two OS/2 programs. The first is a simple application which demonstrates some memory-allocation techniques. The second is a more complex program using some of the multitasking and synchronization facilities of OS/2.

Since real applications are too large to present in this paper, the programs are shown in simplified form. Although these programs are written in IBM C/2 and have been run on OS/2 Standard Edition Release 1.0, not everything necessary to compile, link, and run them is shown here. In particular, the link and bind steps are not described in any detail. Also, only those data declarations necessary to explain the OS/2 programming techniques are included in the listings. The other declarations are contained within the INCLUDE files but are not completely shown. The programs are presented in a manner that highlights the OS/2 function calls, rather than the program logic. For this reason, although all OS/2 system calls return error codes which can and should be checked, no error checking is shown. Finally, some concurrent programming practices have been simplified, thereby allowing certain race conditions to exist.

Figure 1 High-level logic for memory management example



A memory management example. Figure 1 shows the outline of a simple in-storage sort program that uses the memory management features of OS/2—the sorting routine itself has been omitted. The C source code is given in Appendix A.

Parameter passing. Command line parameters in OS/2, unlike their DOS counterparts, must be obtained from the environment segment. On entry to the program, the AX register contains the selector to the environment segment and the BX register contains the offset to the command string. However, when programming is being done in C, standard conventions are used for passing command line parameters.

File system. Using the OS/2 file system is similar to using the handle-based I/O system of DOS. However, the system calls are more powerful; for example, the OPEN function call, DosOpen, attempts to open a file and, if it does not exist, can create it. DOS requires that the program use separate function calls for OPEN and CREATE. Information about a particular file is obtained with a query to the file system, DosQFileInfo, and the required information is in the data structure addressed by the returned pointer. The read and write function calls, DosRead and DosWrite, move data directly between the user's buffer segment and the file system without the need to specify a Disk Transfer Area. The MOVE function call, DosMove, moves a file between subdirectory paths on a given disk or diskette or, as in this case, renames a file.

Memory management. The sample program uses DosAllocHuge to allocate enough main memory to hold the entire file. Since the size of the file may be larger than 64K bytes, the program uses a huge allocation. When the program moves from one segment of the huge allocation to the next, it generates the new selector from the current selector by using the shift count returned by DosGetHugeShift.

DosAllocHuge allocates enough segments to hold all of the data in the file. If there is not enough physical memory available to keep all of the data in main memory at the same time, OS/2 invokes swapping to keep available the data currently being used. Thus, the operating system rather than the application manages the physical memory resource. The application simply allocates the memory needed.

Message file. Keeping messages in an external message file makes the program easier to maintain and easier to translate for use in another country. The messages may be retrieved by using the Get Message function call, DosGetMessage, and appropriate fields may be inserted into the message by using the Insert Message, DosInsMessage, function call.

After the message file is written, the Make Message File (MKMSGF) Utility converts the ASCII text into a file indexed by message number that can be accessed rapidly using the DosGetMessage call.

The message file for this program is shown in Figure 2. In this message file, the first two lines, beginning with semicolons, contain comments. The next line contains the message file component ID, which in the example is SRT. The last two lines are the actual messages numbered 0001 and 0002. The I is an indication that this is an information message versus an error (E), prompt (P), or warning (W) message. The %1 in the second message is a place holder for variable-message text.

External calls. All of the calls made by the example program to OS/2 functions are dynamic link calls to external routines because the OS/2 programming interface is implemented as a set of dynamic link libraries.

Program termination. OS/2 programs terminate with the DosExit function call. The call permits the program to exit with a return code and provides the option of terminating the entire process, as in this case, or just the currently executing thread.

Converting to a Family Application. To make this program executable on either OS/2 or DOS 3.3, it must be run through the BIND utility. The bind command used to bind this program is

```
bind sort.exe api.lib doscalls.lib
```

Figure 2 OS/2 message file for sample program

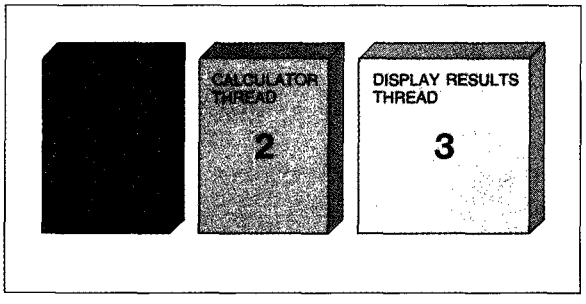
```
;  Sort Example Program Message File
;
SRT
SRT0001I: Family API and Memory Mgt Example - Sort Program
SRT0002I: File %1 has been sorted.
```

The Bind utility adds logic to the SORT.EXE file which maps the OS/2 system calls to their equivalent DOS and BIOS software interrupts. Following the Bind process, SORT.EXE is an executable module that can run on either OS/2 or DOS 3.3. If the programmer chooses to use calls that are not in the Family API, there is no equivalent mapping routine. However, it is possible to issue DosGetMachineMode to test the machine state and branch around OS/2 specific logic when executing on DOS 3.3.

A multitasking example. Figure 3 depicts the use of OS/2 multitasking to implement a postfix calculator using three threads. Figure 4 shows the structure of the program in pseudocode. The user of the program enters operands on a stack. When an operator is entered, the program pops the top two operands off the stack, performs the calculation, and pushes the result back onto the stack. Meanwhile, the stack data structure is constantly displayed on the screen. Figure 5, Figure 6, and Figure 7 show the logic of the three threads. The C code itself is given in Appendix B.

The main procedure is responsible for initializing the display, creating the two additional threads, and

Figure 3 Tasking structure of calculator program



then terminating all of the threads in the process when the program is finished. Three primary procedures (C functions) are run during normal execution of the program. In addition, there are auxiliary functions for manipulating the stack and the display.

IPC. This example uses several interprocess communications mechanisms, as shown in Figure 8.

There are two RAM semaphores, StartCalc and ResultAvail, that coordinate the execution of the

Figure 4 Structure of the program

```
Variables declared in include file

Main Procedure
   Initialize the display
   Create threads 2 and 3
   Call the user input function with thread 1
   Terminate all threads of the programs

User Input Procedure executed by thread 1

Calculator Procedure executed by thread 2

Display Results Procedure executed by thread 3

Additional functions used throughout the program
   Push an operand/result on the stack
   Pop an operand off the stack
   Initialize the display
   Clear the user input field
```

Figure 5 Keyboard input thread

```
User Input Procedure
dowhile the quit key is not pressed
   Read a string from the keyboard
   capture the screen resource with ScreenOwnership semaphore
   if an operator was entered
      then (
         place operator in shared operator variable
         pop top value off stack into shared operator 1 variable
         pop next value off stack into shared operator 2 variable
         clear StartCalc semaphore to signal thread 2
         clear the user input field on the display
      )
      else if the quit key was pressed
         do nothing since we are going to quit
      else (
         push the new number on the operand stack
         clear the user input field on the display
      )
   release the ScreenOwnership semaphore
return
```

Figure 6 Calculation thread

```
Calculator Procedure
do forever
   wait on StartCalc semaphore to clear
   convert operands to integers
   perform the calculation
   convert the integer result to a string
   clear the ResultAvail semaphore to signal thread 3
```

Figure 7 Screen display thread

```
Display Calculation Result
do forever
   wait on ResultAvail semaphore to clear
   capture the screen resource with ScreenOwnership semaphore
   push the result on the stack
   release the ScreenOwnership semaphore
```

Figure 8 Interprocess communications mechanisms

```
Thread Signaling

   RAM Semaphore - StartCalculation, thread 1 signals thread 2

   RAM Semaphore - ResultAvailable, thread 2 signals thread 3

Resource Ownership

   System Semaphore - ScreenOwnership, thread 1 and thread 3
                      serialize their access to the display
                      and the stack.
```

three threads. Since the threads all share the same address space, RAM semaphores can be used rather than system semaphores. Since StartCalc and ResultAvail are RAM semaphores, they do not have to be explicitly created by the program: They are double words in shared storage referenced by their addresses.

StartCalc and ResultAvail are used as signaling semaphores. They are initialized as set, and the threads waiting for them wait for another thread to clear them. Once a clear is done, the waiting thread is awakened. The DosSemRequest used to wait for the clearing of the semaphore also sets it again. When the function loops back to the DosSemRequest, the thread waits until the signaling thread does another DosSemClear.

One exclusive system semaphore, ScreenOwnership, is used to serialize access by the threads to the display image and the data structure representing the stack of the calculator. As ScreenOwnership is a system semaphore, it must be explicitly created by an OS/2 system call. OS/2 tracks which thread, if any, currently owns an exclusive system semaphore, and only the current owner can clear it. This feature allows the semaphore to control resource ownership. When a thread is ready to update the data structures guarded by ScreenOwnership, it does a DosSemRequest to indicate that it wishes ownership. If no other thread currently owns ScreenOwnership, the requesting thread is granted the semaphore. When it is done with the data structures, it must do a DosSemClear on ScreenOwnership to permit other threads to acquire the resource.

Shared data structures and code. Since threads of the same process share the same memory address space, it is very easy to define data structure that can be shared by all of the threads. The shared data structures used by the threads of this program are shown in Figure 9. However, since the data structures are shared and the threads of the program execute concurrently, it is important to serialize access to any shared data that are updated by one or more of the threads. Static, updated data must be managed carefully in a multithreaded environment.

Threads may also share common functions very simply. The functions can be invoked in the usual

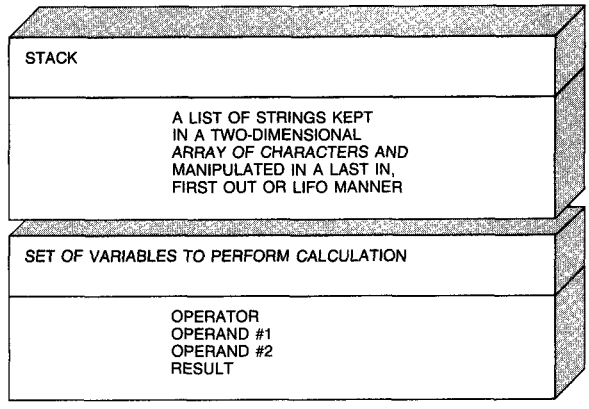
way in the C language. Functions that use only parameters and automatic variables do not require special precautions because these functions are passed or allocated on the stack, and each thread has its own stack. References to static variables must be explicitly serialized, as indicated above.

Keyboard handling. Thread 1, which manages the keyboard, uses the KbdStringIn function to read lines from the keyboard. This simplifies processing by permitting the application to receive all of the characters of an operand or operator with one call in a single buffer. Although an application can handle keystrokes directly, it is usually simpler to request lines unless the application must handle individual keys immediately. KbdStringIn implicitly waits for a sequence of characters terminated by a return to be entered at the keyboard. No polling of the keyboard is required, and, in general, polling techniques are not recommended for OS/2 programs.

Calculator. The calculator thread is in an infinite loop, waiting for input and performing calculations when enough input is available. It waits for the keyboard handler to clear StartCalc. When that happens, it does the necessary string-to-integer conversions, computes the result of the operation, and converts the result to a string. It then signals thread 3, the screen handler.

Screen handler. Thread 3, the screen handler, waits for thread 2 to clear ResultAvail, indicating that there is some output to display. Since the screen is shared with thread 1, thread 3 must capture it by using the ScreenOwnership semaphore. Once it owns the screen, the screen handler updates the screen to show the result of the calculation and releases the screen resource again.

Figure 9 Shared data structures



Summary

OS/2 gives the programmer a set of interfaces with which to use its large memory and multitasking features. It also provides file-system and other services in a manner that is an extension of the functions and features of DOS and BIOS. With the OS/2 Toolkit and language products, the programmer can convert existing DOS applications to OS/2 and write new applications that utilize the new features of OS/2.

Appendix A: Source listing for memory management example

```
#include <doscall.h>
#include <string.h>

#include <sort.h>

void main(argc, argv, envp)
int argc;
char *argv[ ];
char *envp[ ];

{
                    /* DISPLAY PROGRAM TITLE */

/*    DosGetMessage obtains Message Number 1 from the message file named ''srt.msg'' and      */
/*                  places it in MsgDataArea                                                  */
```

```
DOSGETMESSAGE((char far * far *)lvTable,
                    lvCount,
                    (char far *)MsgDataArea,
                    MsgDataLength,
                    MsgNumber = 1,
                    MsgFileName = ''srt.msg'',
                    (unsigned far *)&MsgLength);

/*    DosPutMessage writes the message retrieved above as output to the console                */

DOSPUTMESSAGE(OutputHandle = CONSOLE,
                    MsgLength,
                    MsgDataArea);

        /* OPEN THE FILE NAMED IN argv[1] */

/*    DosOpen opens the file (named in argv[1]) with sharing permitted.                        */

DOSOPEN(FileName = argv[1],
          (unsigned far *)&FileHandle,
          (unsigned far *)&ActionTaken,
          FileSize = 0,
          FileAttribute = 0,
          OpenFlag = OPEN_FILE_IF_EXISTS,
          OpenMode = DENY_NONE_READ_WRITE_ACCESS,
          ReservedDWord = 0);

/*    DosQFileInfo is used to obtain the size of the file. This information will be used to determine   */
/*                the size of the required memory allocation needed.                                    */

DOSQFILEINFO(FileHandle,
                FileInfoLevel = 1,
                FileStatusStructPointer,
                FileInfoBufSize = 22);

                /* ALLOCATING MEMORY TO CONTAIN THE FILE */

/*    DosAllocHuge is used to obtain several 64K Byte segments plus a shorter segment to contain   */
/*                the file. The allocation is not expandable or sharable.                           */

DOSALLOCHUGE(NumSeg = (FileStatusStructure.file_size / 65535),
                Size = (FileStatusStructure.file_size - (NumSeg*65535)),
                (unsigned far *)&Selector,              /* Selector pointing to initial SEGMENT  */
                MaxNumSeg = 0,
                AllocFlags = 0);

/*    DosGetHugeShift provides a shiftcount which is used to calculate the selector increment.  */

DOSGETHUGESHIFT((unsigned far *)&ShiftCount);
SelectorIncrement = 1 << ShiftCount;

/*    The file is completely read into the Huge Memory Allocation                               */

LoopIndex = FileStatusStructure.file_size;
```

```
CurrentSelector = Selector;
while (LoopIndex > 65535){
                                                        /* Calculate next selector */
    PointerBuilder = CurrentSelector;
    SegmentPointer = PointerBuilder << 16;

    DOSREAD(FileHandle,                                 /* Read 64K Bytes into a segment */
            (SegmentPointer,
             ReadBufferLength = 65535,
             (unsigned far *)&BytesRead);

    CurrentSelector = CurrentSelector + SelectorIncrement;
    LoopIndex = LoopIndex - 65535;
}
                                                        /* Read in the rest of the file */
PointerBuilder = CurrentSelector;
SegmentPointer = PointerBuilder << 16;

DOSREAD(FileHandle,
        SegmentPointer,
        ReadBufferLength = Size,
        (unsigned far *)&BytesRead);

DOSCLOSE(FileHandle);

/*************************************************************************************************************/
/*                                          CALL SORT at this point                                          */
/*************************************************************************************************************/

                        /* OPEN OUTPUT FILE FOR SORTED DATA */


/*    DosOpen opens a new file (named ''Examp1.tmp'') with sharing permitted.                                */

DOSOPEN(NewFileName = ''Exampl.tmp'',
        (unsigned far *)&NewFileHandle,
        (unsigned far *)&ActionTaken,
        FileSize = FileStatusStructure.file_size,
        FileAttribute = 0,
        OpenFlag = CREATE_FILE_REPLACE_IF_EXISTS,
        OpenMode = DENY_NONE_READ_WRITE_ACCESS,
        ReservedDWord = 0);

/* Write Sorted Data out to new file                                                                         */

LoopIndex = FileStatusStructure.file_size;
CurrentSelector = Selector;
while (LoopIndex > 65535){

    PointerBuilder = CurrentSelector;
    SegmentPointer = PointerBuilder << 16;

    DOSWRITE(NewFileHandle,
             SegmentPointer,
             WriteBufferLength = 65535,
             (unsigned far *)&BytesWritten);
```

```
    CurrentSelector = CurrentSelector + Selector Increment;
    LoopIndex = LoopIndex - 65535;
}

PointerBuilder = CurrentSelector;
SegmentPointer = PointerBuilder << 16;

DOSWRITE(NewFileHandle,
         SegmentPointer,
         WriteBufferLength = Size,
         (unsigned far *)&BytesWritten);

DOSCLOSE(NewFileHandle);

/* DosDelete removes the named file                                    */

DOSDELETE(FileName,
          ReservedDWord);

            /* RENAME THE NEW FILE */

DOSMOVE(OldPathName = ''Examp1.tmp'',
        NewPathName = FileName,
        ReservedDWord = 0);

DOSGETMESSAGE((char far * far *)lvTable = &FileName,
              lvCount = 1,
              (char far *)MsgDataArea,
              MsgDataLength,
              MsgNumber = 2,
              MsgFileName = ''srt.msg'',
              (unsigned far *)&MsgLength);

DOSPUTMESSAGE(OutputHandle = CONSOLE,
              MsgLength,
              MsgDataArea);

/*          Terminate the program                                   */


/* DosExit terminates the entire process with a Result Code = 0 (Success)  */

DOSEXIT(ActionCode = 1,
        ResultCode = 0);
}
```

Appendix B: Source listing for calculator example

```
#include <doscall.h>                    /* OS/2 API declarations          */
#include <stdio.h>                      /* C standard I/O run time        */
#include <string.h>                     /* C string library               */

#include <calc.h>
```

```
/* Semaphore Variables */

  unsigned long          RamSem1 = 0;
  unsigned long far     *StartCalc = &RamSem1;

  unsigned long          RamSem2 = 0;
  unsigned long far     *ResultAvail = &RamSem2;

  unsigned long          ScreenOwnership;

void main( )
{

/* DosCreateThread/Variables */
  unsigned              ThreadIDWord;                                   /* Thread ID */
  unsigned char         Thread2Stack[2000];  /* Stack for #2 */
  unsigned char         Thread3Stack[2000];  /* Stack for #3 */

/*****************************************************************************************************************/
/* Begin execution . . .                                                                                         */
/*                                                                                                               */
/*          Procedure that executes as Thread 1                                                                  */
/*                                                                                                               */
/*****************************************************************************************************************/

  InitializeDisplay( );

  DOSCREATESEM(NoExclusive = 0,                                  /* Exclusive system semaphore */
               (unsigned long far *)&ScreenOwnership,                    /* for the display */
               SemName = ''\\SEM\\MYSCREEN'');


  DOSSEMSET(StartCalc);                                                  /* Set RAM semaphore */
  DOSCREATETHREAD(CalculatorProcedure,                                /* Create Calculator Thread */
                  (unsigned far *)&ThreadIDWord,
                  (unsigned char far *)&Thread2Stack[1999]);
  DOSSEMSET(ResultAvail);                                                /* Set RAM semaphore */
  DOSCREATETHREAD(DisplayResultProcedure,                             /* Create Display Thread */
                  (unsigned far *)&ThreadIDWord,
                  (unsigned char far *)&Thread3Stack[1999]);

  /*  The initial program thread (Thread 1) is the input procedure                                               */
  UserInputProcedure( );                                          /* Get input from the keyboard */


  DOSCLOSESEM(ScreenOwnership);                                          /* Close the semaphore */

  DOSEXIT(ActionCode = 1,                                             /* Terminate entire process */
          ResultCode = 0);

}    /* MAIN Procedure Ends */
void UserInputProcedure( )
{
```

```
/* Loop is terminated when user enters ''q'' which is defined as QUIT
   do{                                                                    */

      KbdLength.Length = 32;
      KBDSTRINGIN((char far *)CharBuffer,                    /* Read string from keybd */
                  (struct KbdStringInLength far *)&KbdLength,
                  IOWait = 0,                                /* Wait for carriage return */
                  KbdHandle = 0);
      /* Substitute NULL for carriage return */
      CharBuffer[strlen(CharBuffer) - 1] = NUL;

      DOSSEMREQUEST(ScreenOwnership,                        /* Request Screen Semaphore */
                    Timeout = -1);                          /* No timeout - indefinite wait */

      /* Test to see if it is an operator.                                */
      if(CharBuffer[0]==PLUS || CharBuffer[0]==MINUS ||
        CharBuffer[0]==DIV   || CharBuffer[0]==MULT){

          Operator = CharBuffer;
          Pop(Operand2);
          Pop(Operand1);

          DOSSEMCLEAR(StartCalc);                           /* Signal calculation thread */

          ClearInputField( );
       }
      /* Test to see if it is the quit character ''q'', then do nothing   */
       else if(CharBuffer[0]==QUIT)
       ;

      /* Now assume it must be a number so push it on the stack.          */
       else {
          Push(CharBuffer);

          ClearInputField( );
      }
      DOSSEMCLEAR(ScreenOwnership);                         /* Release Screen Semaphore */


   }while(CharBuffer[0] != QUIT);
}
/*****************************************************************************************************************/
/*                                                                                                               */
/*            Procedure that executes as Thread 2                                                                */
/*                                                                                                               */
/*****************************************************************************************************************/
void far CalculatorProcedure( )
{
   int IntOp1,IntOp2,IntResult;
   for(;;) {


     DOSSEMREQUEST(StartCalc,                                 /* Wait for signal */
                   Timeout = -1);                             /* Indefinite wait */

     IntOp1 = atoi(Operand1);
```

```
      IntOp2 = atoi(Operand2);

      switch(*Operator){
         case '+':
            IntResult = IntOp1 + IntOp2;                    /*****************************************/
            break;                                          /*                                       */
         case '-':                                          /*    Perform                            */
            IntResult = IntOp1 - IntOp2;                    /*                                       */
            break;                                          /*      Appropriate                      */
         case '*':                                          /*                                       */
            IntResult = IntOp1 * IntOp2;                    /*        Calculation                    */
            break;                                          /*                                       */
         case '/':                                          /*****************************************/
            IntResult = IntOp1 / IntOp2;
            break;
      }
      Result = itoa(IntResult,ResultBuffer,10);                          /* Convert to ASCII */
      DOSSEMCLEAR(ResultAvail);                          /* Signal that Result is Available */
   }
}
/*****************************************************************************************************************/
/*                                                                                                               */
/*        Procedure that executes as Thread 3                                                                    */
/*                                                                                                               */
/*****************************************************************************************************************/
void far DisplayResultProcedure( )
{

   for(;;){
      DOSSEMREQUEST(ResultAvail,                              /* Wait for signal from Calculator */
                    Timeout = -1);                                /* Indefinite Wait           */


      DOSSEMREQUEST(ScreenOwnership,          /* Request screen resource                       */
                    Timeout = -1);            /* Indefinite Wait                               */
      Push(Result);

      DOSSEMCLEAR(ScreenOwnership);           /* Release Screen Resource                       */
   }
}
/*****************************************************************************************************************/
/*            Subroutines                                                                                        */
/*****************************************************************************************************************/
void Push(Operand)
char *Operand;
{
   /* Display stack change.                                                                                      */
   VIOSCROLLDN(TopRow=7,
               LeftCol=55,
               BotRow=19,
               RightCol=58,
               NumLines=1,
               (char far *)RedBlankChar,
               VioHandle = 0);
```

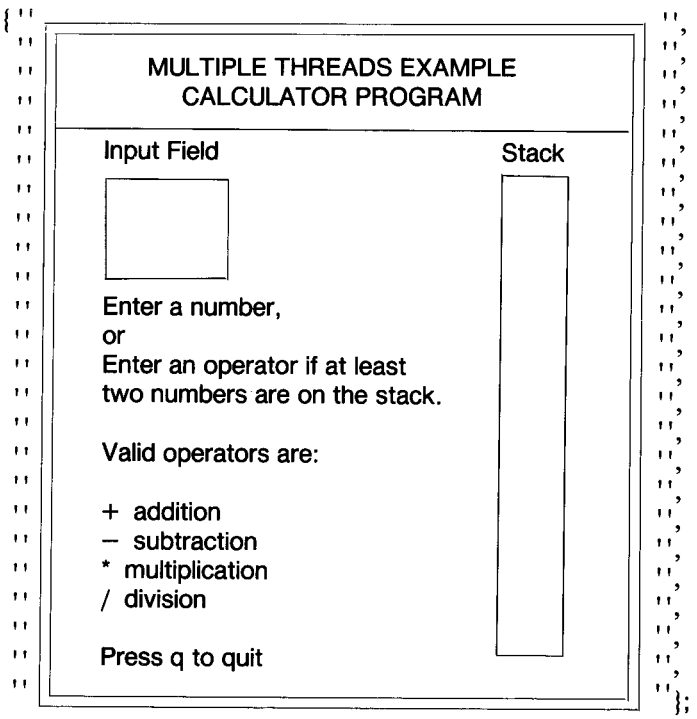
```
    VIOWRTCHARSTR((char far *)Operand,
                  VioLength = strlen(Operand),
                  Row = 7,
                  Column = 58 - VioLength + 1,
                  VioHandle = 0);

    /* Update the stack data structure.                                    */
    for( StackColumn=0;
         StackColumn<=strlen(Operand);
         StackColumn++)
          OperandStack[StackIndex][StackColumn] = Operand[StackColumn];
    StackIndex = StackIndex + 1;
}

void Pop(Operand)
char *Operand;
{


    VIOSCROLLUP(TopRow=7,                                  /********************************* /
                LeftCol=55,                                /*        Display stack          */
                BotRow=19,                                 /*          change               */
                RightCol=58,                               /********************************* /
                NumLines=1,
                (char far *)BlueBlankChar,
                VioHandle = 0);

    StackIndex = StackIndex - 1;                        /* Update the stack data structure */

    for(StackColumn=0;
        StackColumn<=strlen(&OperandStack[StackIndex][0]);
        StackColumn++)

        Operand[StackColumn] = OperandStack[StackIndex][StackColumn];

}
void InitializeDisplay( )
{


  VIOSCROLLUP(TopRow=0,                                 /  ********************************  /
              LeftCol=0,                                /*          Blank                 */
              BotRow=-1,                                /*            the                 */
              RightCol=-1,                              /*              screen            */
              NumLines=-1,                              /  ********************************  /
              (char far *)FillChar,
              VioHandle = 0);

  for( RowCounter = 0;
       RowCounter < WindowLength;
       RowCounter++)


  VIOWRTCHARSTRATT(IOWindow[RowCounter],
                   VioLength = strlen(IOWindow[RowCounter]),
                   Row = 1 + RowCounter,
```

```
                        Column = 20,
                        Attribute = WHITEONBLUE,
                        VioHandle = 0);

  VIOWRTCHARSTRATT(CharStr = '  ',
                        VioLength = 4,
                        Row = 7,
                        Column = 23,
                        Attribute = WHITEONGREEN,
                        VioHandle = 0);

  VIOSETCURPOS(Row = 7,
                    Column = 23,
                    VioHandle = 0);

}
void ClearInputField( )
{

  VIOWRTCHARSTRATT(CharStr = '  ',
                        VioLength = 4,
                        Row = 7,
                        Column = 23,
                        Attribute = WHITEONGREEN,
                        VioHandle = 0);

  VIOSETCURPOS(Row = 7,
                    Column = 23,
                    VioHandle = 0);
}
```

Appendix C: INCLUDE File "calc.h" for Calculator Example

```
/*****************************************************************************************************************/
/*          Multiple Threads Example — Calculator Program — Variables                                            */
/*****************************************************************************************************************/

                    .
                    .
                    .

/* Calculator Variables  ******************************************************************************************/

  char                  *Operator;
  char                   Op1Buf[8];
  char                  *Operand1 = Op1Buf ;
  char                   Op2Buf[8];
  char                  *Operand2 = Op2Buf ;
  char                  *Result;
  char                   ResultBuffer[8];

  char                   OperandStack[13][8];
  int                    StackIndex = 0;
  int                    StackColumn;
```

```
/* Semaphore Function Variables ********************************************************************************/

  unsigned long          RamSemaphore;
  unsigned long far     *SemHandle = &RamSemaphore;
  long                   Timeout = -1;

  unsigned               NoExclusive;
  unsigned long          SysSemHandle;
  char far              *SemName;

                .
                .
                .
/* Fill Characters and Attributes ******************************************************************************/

  char                   FillChar[2] = {0x20, 0x0F};
  char                   RedBlankChar[2] = {0x20, 0x4F};
  char                   BlueBlankChar[2] = {0x20, 0x1F};

/* Display window character definitions ************************************************************************/
                .
                .
                .
  char far *IOWindow[22] =
```




```
int         RowCounter = 0;
int         WindowLength = 22;
                .
                .
                .
```

Cited references

1. M. S. Kogan and F. L. Rawson III, "The design of Operating System/2," *IBM Systems Journal* **27**, No. 2, 90–104 (1988, this issue).
2. A. M. Mizell, "Understanding device drivers in Operating System/2," *IBM Systems Journal* **27**, No. 2, 170–184 (1988, this issue).

General references

E. Iacobucci, *OS/2 Programmer's Guide*, Osborne McGraw-Hill, Berkeley, CA (1988).

iAPX 286 Programmer's Reference Manuals, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051 (1985).

J. I. Krantz, A. M. Mizell, and R. L. Williams, *OS/2 Features, Functions, and Applications*, John Wiley & Sons, Inc., New York (1988).

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

Operating System/2 Programmer's Guide, 84X1448, IBM Corporation (October 1987); available through IBM branch offices.

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

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

Ross L. Cook *IBM Entry Systems Division, P.O. Box 1328, Boca Raton, Florida 33432.*
 Mr. Cook received a B.S. and an M.S. degree in computer science from the Florida Institute of Technology in 1969 and 1973, respectively. He began his career with IBM in 1965, working for the Federal Systems Division on the Apollo moon mission. His IBM programming experience includes hardware diagnostic programs for both the central processor unit and various peripherals, automated test controller for the IBM 2305 and 3330 disk facilities and file control units, development of a FORTRAN compiler and macro assembler for the System/7, and, since 1974, operating system design and programming for small to medium-sized systems. Mr. Cook was involved in early design work that led to Operating System/2; he joined the OS/2 design department in 1985. He has been instrumental in the detailed design of several components of OS/2 and has received an IBM Outstanding Technical Achievement Award for this work. Currently he is a senior programmer in the OS/2 design group.

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 OS/2 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 Personal System/2.

Jay A. Tunkel *IBM Entry Systems Division, P.O. Box 1328, Boca Raton, Florida 33432.*
 Mr. Tunkel is currently an advisory programmer in the OS/2 Systems Architecture Department, where he is involved in the design and enhancement of Operating System/2. He joined IBM in Endicott, New York, in 1968 and has worked on many projects relating to mid-size and personal computers, including the IBM 4300 Series, IBM PC XT/370, and IBM PC AT/370. Mr. Tunkel holds B.S. and M.B.A. degrees from the University of Connecticut and an M.S. in computer science from the State University of New York at Binghamton.

Robert L. Williams *IBM Entry Systems Division, P.O. Box 1328, Boca Raton, Florida 33432.*
 Mr. Williams is currently a member of the OS/2 Systems Architecture Department, where he is involved with the design of OS/2. He joined the Entry Systems Division in 1983 and has worked on several personal computer office system software projects in areas including telephony, LAN communications, and document retrieval. Mr. Williams is coauthor of a book titled *OS/2 Features, Functions and Applications* (John Wiley & Sons, Inc.). He holds Bachelor's and Master's degrees in computer science and engineering from the University of South Florida.

Reprint Order No. G321-5315.