DOS as a Mach 3.0 Application

Gerald Malan, Richard Rashid, David Golub, and Robert Baron
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue Pittsburgh, Pennsylvania 15213

(412) 268-8744

Internet: grm@cs.cmu.edu, rfr@cs.cmu.edu, dbg@cs.cmu.edu, rvb@cs.cmu.edu

Abstract

We have implemented support for the DOS operating system on top of the Mach 3.0 kernel. This support included machine-dependent kernel modifications for the i386/i486 architecture to handle virtual 8086 mode, a multithreaded emulation of the IBM PC's VGA display and I/O devices, direct support for a number of common DOS functions and frequently loaded DOS drivers, and code to integrate DOS functionality with the existing 4.3BSD Unix Server. The resulting system allows multiple virtual DOS environments, supports DOS versions 3.1 to 5.0, and is capable of running common DOS software including performance sensitive PC entertainment software such as Wing Commander – a high-speed space combat simulation system.

Many lessons were learned during the course of this work. DOS stresses a number of Mach features infrequently used by Unix emulation. The timing and latency demands of DOS applications, especially those with animation and sound, are dramatically different from those of typical Unix applications. Because most DOS programs interact intimately with the underlying PC hardware, DOS emulation expanded our knowledge about the behavior of Mach 3.0 when asked to provide a virtual machine environment rather than pure client/server support. In particular, quirks of the PC architecture – such as support for the so-called "high memory area" above 0x100000 – had to be precisely emulated.

This paper describes our implementation, its capabilities and limitations, and the lessons learned about Mach in the course of our development effort.

1. Introduction

Approximately one year ago we undertook, as an experiment, the implementation of support for the i386/i486 family's "virtual 8086" mode and the DOS operating system. We were encouraged to begin this work by the success already achieved in the implementation on the Macintosh of support for the MacOS and Multifinder under Mach 2.5. We felt that DOS and the implementation of a virtual x86 machine would present a new set of challenges for Mach's abstractions and implementation.

Today Mach 3.0 DOS support consists of:

- machine-dependent kernel support for virtual 8086 mode

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035 and in part by the Open Software Foundation (OSF). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, or the U.S. government.

- a multithreaded emulator task which provides:
  - virtual Industry Standard Architecture (ISA) I/O devices;
  - BIOS emulation;
  - DOS ISA device drivers;
  - emulation of common DOS extensions such as Expanded Memory (LIM 3.2), Extended Memory, High Memory and Upper Memory Block management (XMS 2.0);
  - direct implementation of many common DOS functions; and
  - software to integrate DOS with Mach's existing Unix server.

The system supports DOS versions 3.1 to 5.0 and runs Windows 3.0 in real-mode. We have successfully run over 100 DOS applications ranging from business software such as Lotus 123 and Microsoft Word to entertainment software such as Wing Commander, Space Quest IV, King's Quest V, Populous, and the Microsoft Flight Simulator. Many of these applications are extremely demanding in terms of both graphics performance for VGA displays and sound generated by commercial monaural and stereo sound boards.

This paper describes our implementation, our experiences, the lessons we have learned and the potential impact of our work on future Mach development.

2. Overview of DOS

A limited knowledge of DOS and the Intel 80386 processor is needed to understand the Mach DOS support. There are three main areas of interest: the way DOS programs call the operating system, the organization of a DOS machine's physical memory, and the operation of the Intel 80386 processor's Virtual 8086 mode.

Two operating system modules are used by DOS programs for system support, the BIOS ROM and the DOS kernel. The BIOS is a set of ROM routines that support device specific system calls. The DOS kernel is located in RAM and supports a general operating system interface. The DOS kernel uses BIOS routines to perform many operations, but DOS programs frequently bypass the DOS kernel and directly call the BIOS.

DOS programs perform system calls by loading call parameters into machine registers and then generating a numbered exception. The registers used to pass parameters vary depending on the system call. Exceptions are generated by DOS programs with the `INT` machine instruction. The `INT` instruction takes the exception number as an argument which is used as an index into the interrupt vector table located in low memory. The interrupt vector selected by the `INT` instruction determines which interrupt service routine will handle the exception. Different exception numbers are used by different DOS and BIOS system calls. DOS provides a general operating system interface that only has one of these exception "entry points." The BIOS routines are grouped by device services, so there are many different BIOS exception numbers (eg. the BIOS video services are all entered via exception number 0x10, the disk routines all use exception 0x13).

The address space of a typical DOS machine is shown in figure 1. Key areas are the interrupt vector table, the BIOS scratch pad RAM, and the memory above 0xA0000. The interrupt vector table contains the addresses for the exception handling routines, referred to below as Interrupt Service Routines or ISRs. DOS programs often manipulate these ISR addresses to redirect different interrupts into their own ISRs. The BIOS scratch pad RAM is used to keep device state information for several important devices. It is in this scratch pad RAM that the BIOS keyboard routines keep the queue of incoming keyboard characters, the video driver keeps the scan line state, and the disk driver saves motor status and small input buffers. The area above 0xA0000 contains the video RAM, Extended Memory Manager swap buffers, BIOS ROM routines and High Memory area.

Central to the Mach DOS support is the Virtual 8086 (V86) operating mode of the Intel 80386 processor. When the processor is in this V86 mode, it interprets memory and executes instructions as if the machine were

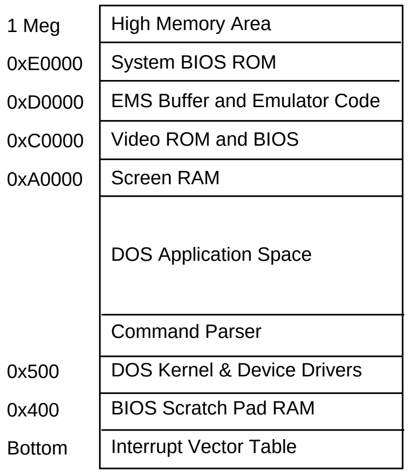

Figure 1: DOS Application Memory Organization.

a native DOS 8086 machine. A set of *privileged* instructions encountered by this V86 machine generates faults that are passed to a protected mode monitor. In the Mach DOS system, a thread operating in the V86 mode generates General Protection (GP) faults that are passed to the Mach Kernel when a privileged instruction is executed. The instructions that are privileged under V86 mode are those that: manipulate the interrupt state of the machine, generate or return from interrupts, and access privileged I/O ports.

3. The Organization of DOS as a Mach Application

The implementation of DOS as a Mach application consists of two pieces, support within the Mach kernel, and a Mach task used as a DOS emulator. The Mach kernel manages threads running in V86 mode and the faults that they generate. The emulator task provides DOS and BIOS system call support. It is important to note, that the DOS Emulator runs the native DOS kernel in a V86 mode thread and does not attempt to reimplement all of the DOS and BIOS system calls.

3.1. Microkernel support for DOS

The Mach kernel provides management for V86 mode threads, and the interface between a V86 thread and its associated Emulator task. Specifically the kernel provides:

- V86 thread creation and maintenance through existing Mach system calls
- V86 thread interrupt flag management using a “virtual” interrupt flag
- Handling of simple V86 General Protection faults
- Simulation of external interrupts within the V86 thread.

3.1.1. Manipulation of the Virtual Machine

Support for the creation and maintenance of V86 mode threads is implemented in the Mach kernel in the form of extensions to the thread get and set state system calls. Different types of thread *flavors* determine which kinds of operations the thread state calls will perform. The existing *mode* thread flavor was modified and two new flavors were introduced, the *port map* and *V86 assist* flavors.

- The *mode* flavor is used to take a thread into and out of the V86 operating mode. To turn on the V86 operating mode, the VM bit in the thread's flag register is set. To turn off the V86 mode, the VM bit is cleared.
- The *port map* flavor enables or disables the threads access to a given ISA I/O port. In this way, direct access to the VGA registers can be given to the V86 thread to speed up video intensive DOS programs.
- The *v86 assist* flavor accesses the kernel's V86 support data structures that contain a pointer to the Emulator task's interrupt queue, and the value for the V86 thread's "virtual" interrupt flag (see below).

3.1.2. Managing the Interrupt Flag

Many of the V86 thread's privileged instructions attempt to manipulate the machine's interrupt status, requiring the kernel to implement a "virtual" interrupt flag for each V86 thread. The kernel alone is allowed to enable or disable these interrupts, yet DOS programs all assume that they are in total control of the machine. Therefore it was necessary to modify the Mach kernel to provide a "virtual" interrupt state to the V86 thread. Since the V86 thread must perform a privileged instruction to inspect or modify the interrupt flag, the Mach kernel can always replace the true interrupt bit with this "virtual" interrupt flag. Space was made in the V86 thread's process control block to hold the flag, and the kernel was modified so that the `thread_get_state` and `thread_set_state` routines would access the "virtual" interrupt flag instead of the true one.

Additional kernel support is also required because of the peculiar way in which 80x86 processors delay the setting of the interrupt bit in the flag register after an STI instruction. The STI instruction doesn't set the interrupt flag during its execution, but delays the setting of that flag until after the *next* instruction finishes executing. Some DOS programs rely on this feature, so its emulation was critical. To solve this problem, the trace flag is set in the V86 thread's flag register during the STI instruction's emulation. Possibly by design, the trace exception then occurs at the same point that the interrupt flag bit should be set. When this trace exception occurs, the kernel sets the interrupt bit in the "virtual" flag register and clears the trace bit in the V86 thread's flags.

3.1.3. General Protection Fault Handling

The privileged instructions that are generated by the V86 thread and are not DOS or BIOS system calls are emulated in the Mach Kernel. These instructions are the set and clear interrupt (STI and CLI), push and pop of the flag register (PUSHF and POPF), and the return from interrupt (IRET) instructions. The instructions are emulated in the kernel as follows:

- CLI: Clears the "virtual" interrupt flag
- STI: Sets the "virtual" interrupt flag as described above
- PUSHF: Pushes the flag register onto the V86 thread's stack. The "virtual" interrupt flag bit is masked onto this value.

- POPF: Pops the flag register from the V86 thread's stack. The "virtual" interrupt flag bit is set to the value from the stack.
- IRET: The kernel pops the flag register from the stack like a POPF, and then pops the V86 thread's instruction pointer from the stack.

3.1.4. Simulating Interrupts Within the Virtual Machine

The Mach kernel is responsible for generating external interrupts within the V86 thread that have been queued by I/O threads in the Emulator task. Using the data structures in the V86 thread's pcb, the kernel determines whether or not there are any external interrupts that need to be delivered to the V86 thread.

To allow for smoother execution, the kernel interrupts the V86 thread sequentially. That is, it waits until the last kernel interrupt has completed before it generates another one. The kernel sets a flag when it generates an interrupt within the V86 thread and clears it when an interrupt return instruction is emulated. While the flag is set, the kernel does not generate any further external interrupts.

3.2. The Emulator Task

The Emulator task is a multithreaded Mach task that supplies input to the V86 thread and provides emulation for DOS and BIOS system calls. Five threads execute simultaneously in the Emulator task: the Monitor, V86, Timer, Mouse, and Keyboard threads. The Monitor thread initializes the DOS support and provides system call emulation. The V86 thread executes the DOS code in virtual 8086 mode. The Timer, Mouse and Keyboard threads all generate the input and interrupts that drive the DOS applications.

The first 1.15 Megabytes of the task's memory is set up by the Emulator to resemble a typical DOS machine. This is accomplished by setting up the interrupt table, scratch pad RAM, screen RAM, BIOS ROM, and high memory area as shown in figure 1. Several "snapshot" files are used to provide the DOS interrupt table and BIOS scratch pad RAM areas. These files are created during a one time setup session. To give fast access to the screen and BIOS ROM routines, the machine's physical memory is mapped onto the task's virtual address space using Mach system calls. Due to the segmented addressing scheme, the 8086 can generate addresses with up to 21 bits, 64 Kbytes above one megabyte. DOS programs expect the 64K High Memory Area above the one Megabyte limit to wrap back around to the beginning of physical memory. The Emulator maps both the High memory area and the first 64K bytes of low memory to the same place in the task's virtual address space.

3.2.1. The Monitor "INT" handler thread

The Monitor thread, formally the V86 thread's exception handler, provides emulation for DOS and BIOS system calls as well as support for privileged IN and OUT instructions. The system call emulation is broken down into groups depending on the type of service being emulated. Some of these major groups are the BIOS disk, video, mouse, and memory expansion routines as well as the DOS calls related to console I/O.

A secondary function of the monitor thread is to support `IN` and `OUT` instructions that attempt to access privileged I/O ports. To enable an I/O port for access, the emulator must use the `thread_set_state` call using the *port map* flavor described above. The majority of these "privileged" I/O instructions are executed by DOS interrupt service routines. By catching these privileged I/O instructions and supplying artificial return values, the Monitor thread is able to support several types of DOS device drivers including the Windows 3.0 Logitech Mouse driver and many types of DOS keyboard drivers.

- **Emulator Disk Support**

  The Emulator's disk support provides two main functions: emulation for BIOS disk system calls, and access to the unix file system as a Network DOS drive. By emulating the BIOS disk routines, we in effect emulate the DOS disk related system calls which all use BIOS routines to perform the actual disk I/O.

  Access to physical disk devices and to UFS files is provided by the Emulator. Physical devices and UFS files are both used as DOS file systems. The Emulator uses the Unix raw device files to access the hard disk's DOS partitions and DOS file systems on floppy disks. Unix files are also used by the Emulator as dos file system images.

  The Unix file system is also used as a Network DOS disk drive. The Emulator uses the DOS Network Redirector, a group of DOS system calls to achieve this. The Network Redirector is used by DOS to access file systems that do not use DOS file allocation tables. With the Network Redirector interface in place, the V86 thread can access any file within the Unix file system.

- **Emulator Video Support**

  The Emulator keeps track of the VGA video state, emulates BIOS video routines and redirects some video exceptions back into the V86 thread. The Emulator saves the VGA state at startup time and restores it upon exit. This video state consists of VGA register settings and the contents of the VGA's video RAM banks. Video state for each of the VGA's video modes is stored in a file. The Emulator uses the information in this file to change the VGA's state when a mode change is requested by a BIOS system call. This VGA information file is created during the setup session described above. A copy of the machine's default font is also kept in a file that is used with VGA text modes.

  To emulate the BIOS video routines, the emulator directly writes to the VGA registers and video RAM areas. If the emulator does not support a video routine, it passes the routine back into the VGA BIOS for the V86 thread to execute. These redirected routines generally do not require the VGA BIOS to know a previous state and will not crash the video subsystem.

- **Emulator Mouse support**

  The Emulator task supports all the functions of a DOS memory resident mouse device driver. It does this using both the Monitor and Mouse threads. The Emulator's mouse driver complies with version 6.0 of the Microsoft Mouse specification. This specification is comprised of both a set of Mouse related system calls, and hooks that allow DOS programs to supply their own mouse event handlers. The Monitor thread emulates these Mouse BIOS calls while the Mouse thread provides support for the user supplied mouse event handlers.

- **Emulator Expansion support**

  Several DOS expansion specifications are supported by the Emulator including the EMS 3.2 and XMS 2.0 specifications. Both of these specifications, like the Microsoft Mouse specification mentioned above, are collections of system calls that extend DOS and BIOS functionality. In 1985 Lotus, Intel and Microsoft agreed to support a standard that increased the amount of memory DOS applications could access. This standard, supported by the Emulator task with the Mach virtual memory subsystem, is known as the LIM EMS.

  The Expanded Memory Specification (EMS) allows DOS programs to access memory on special hardware boards by switching this memory in and out of certain banks in the DOS memory space. DOS programs access memory in this special hardware by gaining handles to certain expanded memory segments. The Emulator task emulates this behavior by `vm_allocating` memory in the task's address space to use as this special expanded memory and then moving this memory in and out of the special DOS memory banks whenever necessary.

  To allow DOS programs access to the conventional memory in the machine above the one megabyte boundary, the eXtended Memory Specification (XMS) was created. This specification also is a collection of BIOS system calls that switch blocks of extended memory with blocks of memory in the addressable DOS area. This is implemented on native DOS machines with special memory drivers that take the machine into and out of protected mode to gain access to the extended memory. The Emulator allocates virtual memory in the same way that it does for the EMS specification and copies this memory into and out of the DOS areas on demand.

3.2.2. The V86 thread

The V86 thread is the only thread that operates in virtual 8086 mode. This thread executes the DOS application and kernel code that generates the general protection faults. These faults are the privileged instructions intercepted by the Mach kernel.

3.2.3. I/O threads

Three I/O threads provide data and external interrupts for DOS programs executing in the V86 thread: the Timer, Mouse and Keyboard threads. These threads provide input for two types of DOS programs: *well-behaved* programs that get their information from BIOS data structures, and *ill-behaved* programs that bypass the BIOS and redirect external device interrupts into their own code. Well-behaved programs access the BIOS data structures through DOS or BIOS system calls. Ill-behaved programs, on the other hand, redirect the notification interrupts away from the BIOS device drivers and into their own ISRs. These Ill-behaved interrupt service routines directly access the input device's I/O ports to gather the input data. To determine which types of programs are executing within the V86 thread, the I/O threads check the interrupt vector table for redirected vectors.

The Monitor and the I/O threads use shared data structures to provide input data for well-behaved DOS programs. The input data for these programs is inserted into the shared data structures by the I/O threads, and is extracted by the monitor thread. In general, a DOS or BIOS system call will direct the monitor thread to place the input data into the V86 thread's registers and then exit to continue the DOS program's execution.

When providing data to an ill-behaved DOS program, an I/O thread queues a notification interrupt, and saves the device input data in an interthread data structure. After the ill-behaved program's ISR has been activated by the interrupt, it will read data from the input device using the privileged `IN` instruction on the device's I/O port. When the Monitor thread catches this attempt to access the privileged I/O port, it will return the data stored previously by the I/O thread as the data from the I/O port, completing the transaction.

The I/O threads use an *interrupt generation table*, shared with the Mach kernel, to generate the external interrupts within the V86 thread. During initialization, the Emulator passes the address of this table to the Mach kernel using the *v86 assist* flavor of the `thread_set_state` system call described above. This shared data structure contains a count and vector number for all the different types of external interrupts that the I/O threads can generate. When an I/O thread wants to generate an interrupt, it increments the interrupt's count field in this table, which causes the Mach kernel to generate the appropriate external interrupt.

Descriptions of the three I/O threads follow. Some of the most difficult problems encountered during the implementation of the DOS Emulator were associated with the Mouse and Keyboard threads. For this reason a greater depth of detail has been provided in these areas of the paper.

- **The Timer Thread**

  Two DOS timer interrupts are generated by the Timer thread: the Time of Day interrupt, and the DOS timer tick interrupt. On a native DOS machine, the Time of Day interrupt is generated by a clock chip connected to interrupt request line zero. The BIOS's Time of Day interrupt service routine generates 18.2 DOS timer tick interrupts per second. The DOS kernel keeps system time using these timer *ticks* in the BIOS scratch pad RAM area.

Timeouts on Mach ports periodically wake up the timer thread to check the V86 thread's timer interrupt status. When the timer thread wakes from this small timeout value, it sets the system time in the scratch pad ram area and checks the two timer interrupt vector table values. If either of these vectors have been redirected from the values set at the Emulator's initialization, the timer thread queues an appropriate interrupt in the interrupt generation table that triggers an external interrupt by the Mach kernel.

Many DOS programs require time of day interrupts in shorter time segments than the Mach kernel can provide. These interrupts are needed in small time slices to drive the PC speaker and other sound devices. To accomplish this, the Timer thread calculates and queues the number of timer interrupts that should have occurred during it's timeout period. All of these interrupts are delivered by the kernel in short succession, generating the correct sound output.

- **The Mouse Thread**

  Depending on the type of mouse input expected by the V86 thread, the Mouse thread emulates either a mouse device driver or a mouse connected to a serial line. When emulating a mouse device driver, the Mouse thread stores information in data structures shared with the Monitor thread, queues external interrupts, and updates the mouse pointer. To emulate a serial line mouse, the Mouse thread queues interrupts on the serial line for the V86 thread.

  Acting as the mouse device driver, the Mouse thread's main tasks are to keep track of the mouse's status, to update the mouse pointer in the VGA screen memory, and to provide interrupts for user supplied mouse event handlers. To keep track of the mouse's status, the Mouse thread updates internal variables when the mouse moves or generates button events. When the mouse pointer is enabled by Mouse BIOS system calls, the Mouse thread must display and track the mouse on the screen.

  By far the most interesting aspect of the Mouse thread's device driver support is the implementation of the DOS application-supplied mouse event handlers. The Mouse BIOS specification allows DOS applications to specify a routine that is called by the resident mouse driver when a user defined mouse event occurs. When this mouse event occurs, the mouse device driver loads the machine registers with mouse status information and then activates the application's routine using a `FAR CALL` instruction. Since the Mach kernel is used to asynchronously stop and redirect control flow within the V86 thread, the Mouse thread must use it to perform the switch to the user event handler. The Mouse thread uses an undefined interrupt vector as a special Mouse interrupt vector that calls the DOS event handler. This poses two problems for the Mouse thread:

  1. There must be a way to load the V86 registers with the correct calling parameters before control is given to the user event handler.
  2. There must be some routine used as an intermediary that will receive the user event handler's `FAR RET` call and will restore the previous program's state.

  To solve these problems, the Mouse thread uses two helper routines that are placed in the DOS application's address space during the Emulator's initialization. The first problem is solved by the first helper routine, which is invoked by the special Mouse interrupt generated by the Mach kernel. This first helper routine invokes the Monitor thread to load the calling parameters into the V86 thread's registers and to set up the stack for the DOS event handler's `FAR RET` instruction. The second helper routine, the address of which has been placed on the stack, is invoked when the user event handler finishes. This second helper routine restores the interrupted program's register state from the stack and returns control of the V86 thread back to the place where the Mach kernel's interrupt occurred.

  The mouse thread also supports the Windows 3.0 Logitech serial mouse driver by simulating the behavior of the Logitech mouse and the machine's COM port. Windows is the only program that bypasses the emulator's internal mouse driver and demands to use its own. To simulate the behavior of a Logitech serial mouse, the Mouse thread queues interrupts that the Window's device driver services by accessing privileged I/O ports. The Mouse thread keeps track of the serial mouse and its COM port by stepping through a state machine, the transitions of which occur in response to privileged I/O instructions on the COM device's I/O ports. Although it would be possible to support any type of mouse driver by expanding this state machine, no advantages would be gained over the internal mouse driver already present in the Emulator.

- **The Keyboard Thread**

  The function of the Keyboard thread is to gather raw keyboard data and to pass that data on to the V86 thread. Depending on the type of DOS program that is executing, the Keyboard thread either puts the keyboard data into a queue or passes the raw scan code to the DOS program's keyboard interrupt service routine. In order to understand the way the emulator handles keyboard data, a knowledge of the BIOS's default keyboard ISR, the scratch pad RAM's keyboard queue and the four general strategies DOS programs use to gather keyboard data is essential.

  In a DOS system with *well-behaved* DOS programs, the BIOS keyboard ISR intercepts keyboard interrupts and places decoded characters into the scratch pad RAM's keyboard queue. This *normal* DOS system is one in which the keyboard interrupt vector has not been redirected. Well-behaved DOS programs get their only keyboard input from this queue. They can either use BIOS system calls to indirectly manipulate the queue or bypass the BIOS and access the keyboard queue directly. Some of the DOS programs that perform keyboard input in this manner are the command.com parser, Borland's Turbo C editor, Lotus 123, edlin.com, and debug.com.

  Ill-behaved DOS programs perform keyboard input by redirecting keyboard interrupts into their own code. There are three types of programs that gather keyboard input in this way. These are:

  - A *Keyboard Hog* program redirects keyboard interrupts into its own code and does not share this information with other DOS programs. This type of program does not place character information into the scratch pad RAM keyboard queue and does not pass the interrupt to another interrupt service routine. In effect it captures the keyboard input all for itself. Epsilon, an Emacs-like text editor, is an example of this type of program.
  - A *Terminate and Stay Resident* (TSR) program redirects the keyboard interrupt into its own code like the hog, but after finishing with the scan code, it jumps to the ISR it replaced, in effect passing the interrupt down a chain of DOS interrupt service routines. The DOS program using this strategy would not use the scratch pad RAM's keyboard queue, but would allow the BIOS keyboard ISR to update the queue through the chaining mechanism.
  - The *Middleman* program uses both interrupt chaining and the scratch pad queue for keyboard input. This type of DOS program redirects the keyboard interrupt into its own ISR, chains the interrupt to the old keyboard ISR, and then uses BIOS system calls for character input. This allows DOS TSR programs to filter the character stream before it reaches the program, but allows the program to screen some keyboard events from the TSR programs. Examples of Middleman programs are Microsoft Word 5.0, Microsoft Windows 3.0, Wing Commander, Populous, and the Microsoft M text editor.

  The Keyboard thread supports all four types of DOS programs. The well-behaved DOS programs are easily supported by putting characters into the scratch pad's keyboard queue upon arrival. The ill-behaved DOS programs are supported with the aid of a helper routine used as a smart ISR located in the DOS application's address space. When executed, the helper routine puts a character located in its data area into the scratch pad's keyboard queue. At startup time the Emulator saves the address of this helper routine in the keyboard interrupt vector so that programs that chain the keyboard interrupt will call the helper routine.

  When the keyboard interrupt vector is redirected, the Keyboard thread queues the scan code in an emulator data structure, then queues a keyboard interrupt in the interrupt generation table. The keyboard interrupts are handled differently than other external interrupts in that they are generated by the Emulator task instead of the Mach kernel. This is done so that the emulator can pass the decoded character to the helper ISR in the DOS application space before the interrupt occurs. This way any programs that chain the keyboard interrupt will eventually place the decoded character into the keyboard queue.

3.2.4. Locking issues

There are many objects shared between the Emulator's threads that can only be manipulated by one thread at a time making the ability to lock these objects important. The objects that can only be accessed sequentially are the V86 thread's state, data structures shared between I/O threads and the Monitor thread, and the interrupt generation table that is shared between the I/O threads and the Mach kernel. To lock these objects, the emulator uses two mutual exclusion objects: a critical section lock and a mouse lock.

To insure that the V86 thread's state is not changed by more than one thread at a time, threads outside the kernel use the critical section lock and the the V86 thread's *resume* flag. The I/O threads take the critical section lock before attempting to suspend the V86 thread so that only one of them is able to modify the V86 thread's state. The emulator's I/O threads must then suspend the V86 thread with the Mach `thread_suspend` system call before they can modify its state. Once the I/O thread has the critical section lock and has suspended the V86 thread, it checks the Resume Flag bit in the V86 thread's flag register. When this bit is set the V86 thread has generated an exception and is being serviced by either the Mach kernel or the emulator's Monitor thread. Since the Mach kernel's V86 support routines are only invoked when the V86 thread is running there is no chance that one of the I/O threads is modifying the V86 thread's state when the Mach kernel's routines are running.

The Emulator's threads must acquire the critical section lock before they can access data structures shared between themselves. The mouse lock is used to guarantee the integrity of the data structures shared by the Monitor thread and the Mouse thread. The critical section lock is also used by the I/O threads to preserve the interrupt generation table's integrity.

4. Integrating DOS with the Unix server

As of this writing, the DOS support relies heavily upon features provided by the 4.3 BSD Unix server in areas including I/O device access, DOS file system implementation, and the management of multiple DOS machines. The DOS emulator task accesses the machine's keyboard, mouse and disk devices through Unix server system calls. Two approaches to accessing DOS file systems are implemented using the Unix server's system calls. Finally, the ability to create and schedule multiple DOS machines also depends on functionality provided by the Unix server. In the future, work will center on replacing the functionality provided by the 4.3 BSD Unix server with native Mach support.

4.1. Unix support for DOS I/O Devices

The Keyboard, mouse and disk devices are all accessed by the emulator task by using Unix system calls. The only I/O device that is not controlled with Unix calls is the display. In general, for input-only devices such as the keyboard and mouse, an I/O thread opens the Unix device file and enters a loop that performs a read select on the device's file descriptor before reading data from the device. The select is performed to check whether or not the Emulator has been resumed from a paused state (see below). The emulator's disk support code uses both Unix server raw disk device files and regular UFS files to implement DOS file systems.

The machine's VGA display is controlled by code in the Emulator task that directly manipulates the VGA's screen memory. Direct control over the VGA is needed because DOS programs use much more of the VGA's functionality than do any Unix programs, including X.

4.2. Unix Support for DOS File Systems

The DOS emulator uses Unix services to access two different types of DOS file systems: Real DOS file systems residing on the physical hard disk, and the Unix file system as a DOS network file system. A real DOS file system is one laid out using a DOS file system specification. The UFS as a DOS network file system is implemented using the DOS Network Redirector Interface.

4.2.1. Real DOS File Systems

The Real DOS file systems supported by the DOS Emulator are physical DOS file systems laid out on the disk that are pointed to by either a Unix device file or a regular Unix file. Unix device files are used to access resident DOS hard disk partitions or floppy DOS disks. A special device file is set up to point to the resident DOS hard disk partition by the Mach `diskutil` command. The Unix `dd` command is used to copy the image of a DOS floppy into a regular Unix file.

Since the Emulator does not contain a copy of DOS, one of these real DOS file system images must have a version of DOS on it for the Emulator to boot and begin operation. The Emulator will check for both a regular file image with DOS in it or for a special device file at run time. Run time flags can be used to specify boot files other than the defaults.

4.2.2. The UFS as a Network DOS File System

The Emulator implements the Unix file system as a DOS drive using the DOS Network Redirector Interface. Several DOS programs are executed by the V86 thread when DOS boots that initialize DOS version-specific variables in the Emulator task. Using these variables, the Emulator inserts the fake UFS drive into DOS data structures, making DOS believe that it has an extra disk drive. The Emulator supports this fake DOS disk drive by intercepting the DOS Network Redirector system calls.

There are several benefits to using the UFS as a disk drive under DOS. The large amount of space available under the UFS disk drive provides more room for DOS storage than do the early versions of DOS that limit Real DOS file system disk size. Since the Emulator uses Unix system calls to access files, those files stored directly in the UFS are accessed faster than those stored in the Real DOS file systems which must use the UFS as an intermediary. Finally, the Emulator can easily move data between the UFS and DOS file systems, making file transfers between the operating systems trivial.

The ability to use the UFS as a DOS disk enables a machine that does not have a DOS hard disk partition to effectively utilize the DOS emulator. The Emulator can boot from a floppy image and then use the UFS disk drive as its main drive for execution and storage.

4.3. Unix Support for Managing Multiple DOS Machines

The ability to pause the DOS Emulator and return to a Unix shell enables the user to execute multiple DOS Emulators running in separate DOS "machines." The Emulator can be paused or stopped by pressing the down the Control, Alt, and 'Z' keys simultaneously. By using the Unix server's ability to multitask, the user is able to execute as many DOS programs as he or she wants and to change back and forth between them at will.

The problem of sharing the keyboard and mouse devices between the different DOS Emulators is overcome by using the `select` system call to check for input on the devices before reading them. When a DOS

Emulator has been paused and then put into the foreground, the `select` call will return with a `Bad File Descriptor` error because the old file descriptor will no longer have a valid state. This error tells the Emulator to reinitialize the file descriptors for the input device before it continues.

5. Performance issues

The performance of most DOS programs is comparable to that of native DOS machines, in the sense that the Mach DOS system is quite useable as a day-to-day operating environment. In general, the Emulator has been tuned to provide good performance for the average DOS program. Optimizations for specific programs like Windows and Wing Commander were provided when the changes were beneficial to most DOS programs. DOS programs that rely on the fact that privileged instructions execute within one machine instruction period will not perform correctly. Although specific numbers comparing the performance of the Emulator and native DOS have not been calculated, some work has been done to analyze the performance of the Mach kernel's support. This performance is determined by two factors: fault handling overheads, and instruction emulation overheads.

5.1. General fault handling overheads

The fault handling overhead corresponds to the length of time spent in forwarding the V86 thread's exceptions to the appropriate emulation routines. This overhead lies mainly in two areas: the time it takes for the processor to switch back and forth between processor states, and the Mach Kernel fault handling code. Typically, an 80386 processor uses about 500 clock cycles for the switch from V86 mode to Protected mode and back. This roughly translates into about 100 machine instructions. Once the switch from V86 mode to Protected mode has been made, it takes about thirty machine instructions to get to the V86 mode support code in the Mach Kernel. If the V86 exception is handled by the Kernel's emulation code, it typically takes about forty to fifty more instructions to reach the proper code for the type of exception.

5.2. Instruction emulation overheads

The overhead related to the instruction emulation is large compared to native DOS code. The performance of privileged instructions emulated in the Mach Kernel depends on the type of instruction and whether the instruction generates more exceptions or write to the user space. The number of instructions used by the Mach Kernel's V86 support code for each of the following privileged instructions are as follows:

- CLI: 29 instructions to clear the "virtual" interrupt bit
- STI: 37 instructions for the General Protection fault plus another 200 instructions for the trace break point.
- PUSHF: 118 instructions to emulate, 46 instructions for the copyout to user space
- POPF: 121 instructions
- IRET: 131 instructions
- INT, IN, OUT: 21 instructions to leave the kernel V86 support.

Clearly there is a performance cost for programs that use these instructions in tight loops. This cost is one of the reasons why it was decided to move some of the emulation from user space into the kernel.

6. Implementation Lessons

Some lessons learned about Mach during the course of this project were: the suitability of the Mach Kernel's 386 machine dependent code to support a layered OS with intensive machine dependent applications, how to balance the exception handling code between the kernel and user space, how to provide a finer granularity of timing than the kernel supports to a Mach application, and how useful the Mach virtual memory system is for implementing different types of memory support.

Several features were added to the Mach Kernel's 386 Machine dependent code: the ability to take threads into and out of the V86 execution mode, a port-specific protection mechanism for the ISA I/O ports by implementing the bitmap in the task's TSS, new flavors of thread state, and a way to generate external interrupts within the V86 thread.

In order to achieve reasonable execution speeds, some V86 emulation had to be done in the kernel. The types of emulation performed by the kernel are related to hardware specific and non system call instruction emulation. Several different solutions were implemented both within the kernel and user space before the current balance described in this paper was decided upon.

Many DOS programs need time slices smaller than the smallest slice available from the Mach Kernel to perform precise sound output. To provide for this, the Emulator task has to compute the number of timer interrupts required to induce the DOS application into producing the right sounds.

The Mach Virtual Memory system was very successful in implementing the many special features associated with DOS memory and its specifications for memory extension. The Emulator was able to map two different address ranges to the same piece of virtual memory by using an external pager. The Mach VM was very useful for implementing the expanded and extended DOS memory specifications.

7. Conclusions

We have shown that it is possible to implement DOS as a Mach 3.0 application by using a combination of kernel and user level support. As of the writing of this paper, we have implemented DOS as a Mach task that uses the 4.3 BSD Unix server for many of its services. There were problems in this implementation in the form of deficiencies in the Mach Kernel's machine dependent code for the 386 processor which were overcome as described above. Over 100 DOS programs have been effectively run on the Mach DOS support, many of which demand very close integration with the machine hardware. In the future, work will center on severing the Unix dependencies and expanding the functionality of the DOS Emulator even further.