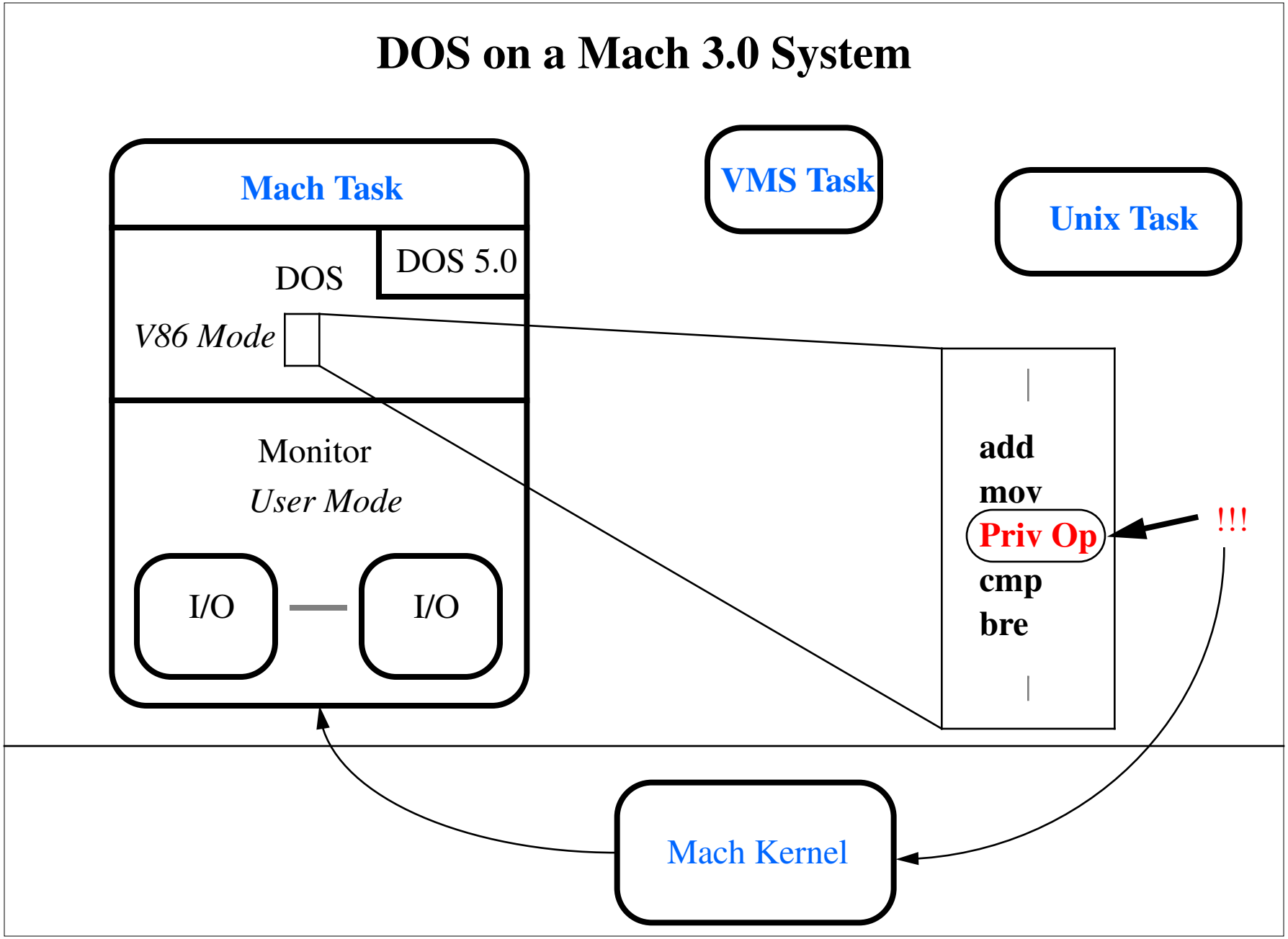
# DOS on a Mach 3.0 System



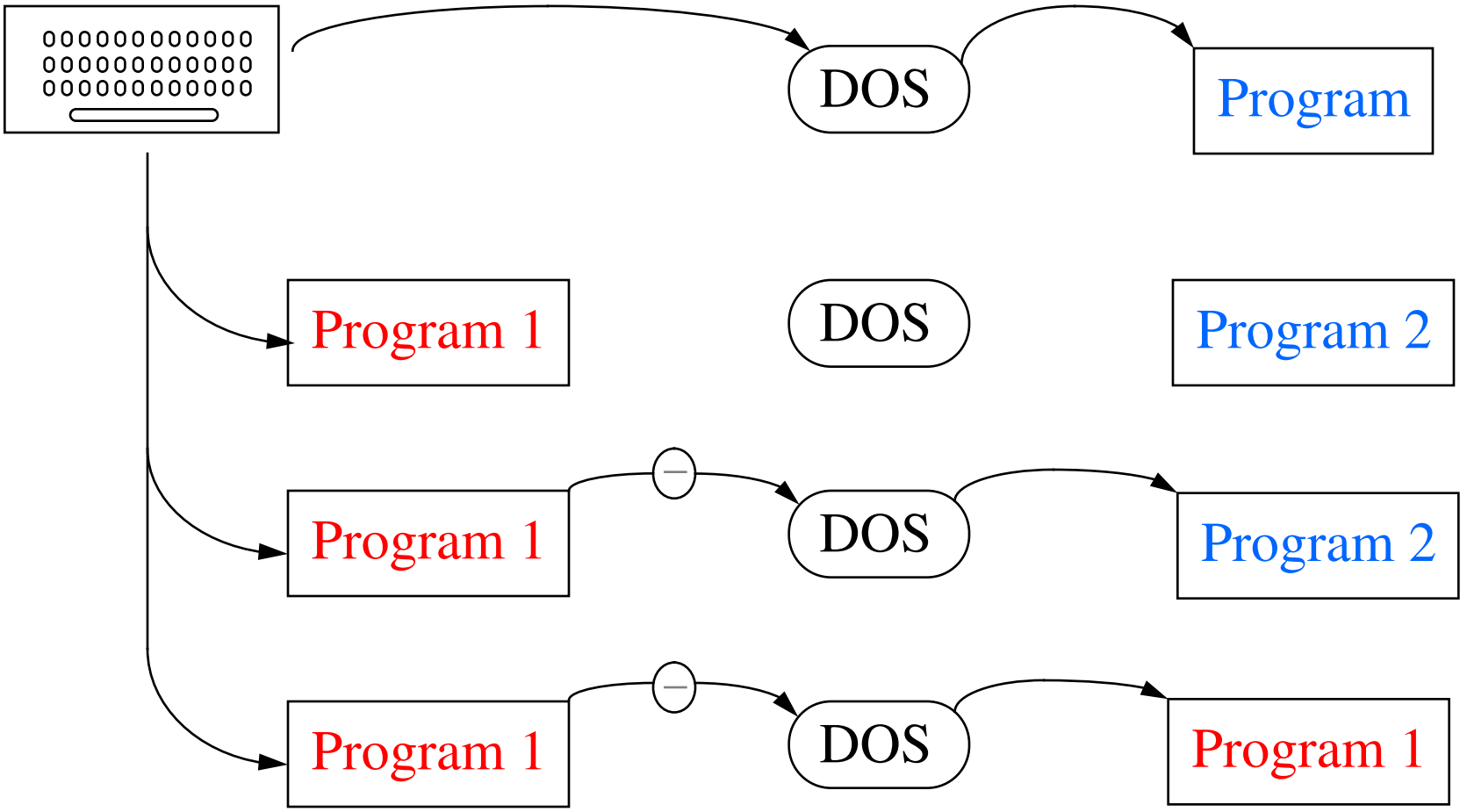
# Keyboard Interrupt Generation

* Like Mouse thread, must not be asynch, setup required

* Don’t let kernel do keyboard interrupts, only user level threads

* Keyboard interrupt:
  - Dequeue scan code
  - Store it in User space routine’s data area
  - Save it in keyboard_output_buffer variable
  - forces interrupt to occur

# Classes of Keyboard Programs



# Keyboard Thread

* Reads scan codes from /dev/kbd

* If well-behaved program running, decode scan code and insert into SPRAM

* If ill-behaved program, queue scan code and queue keyboard interrupt

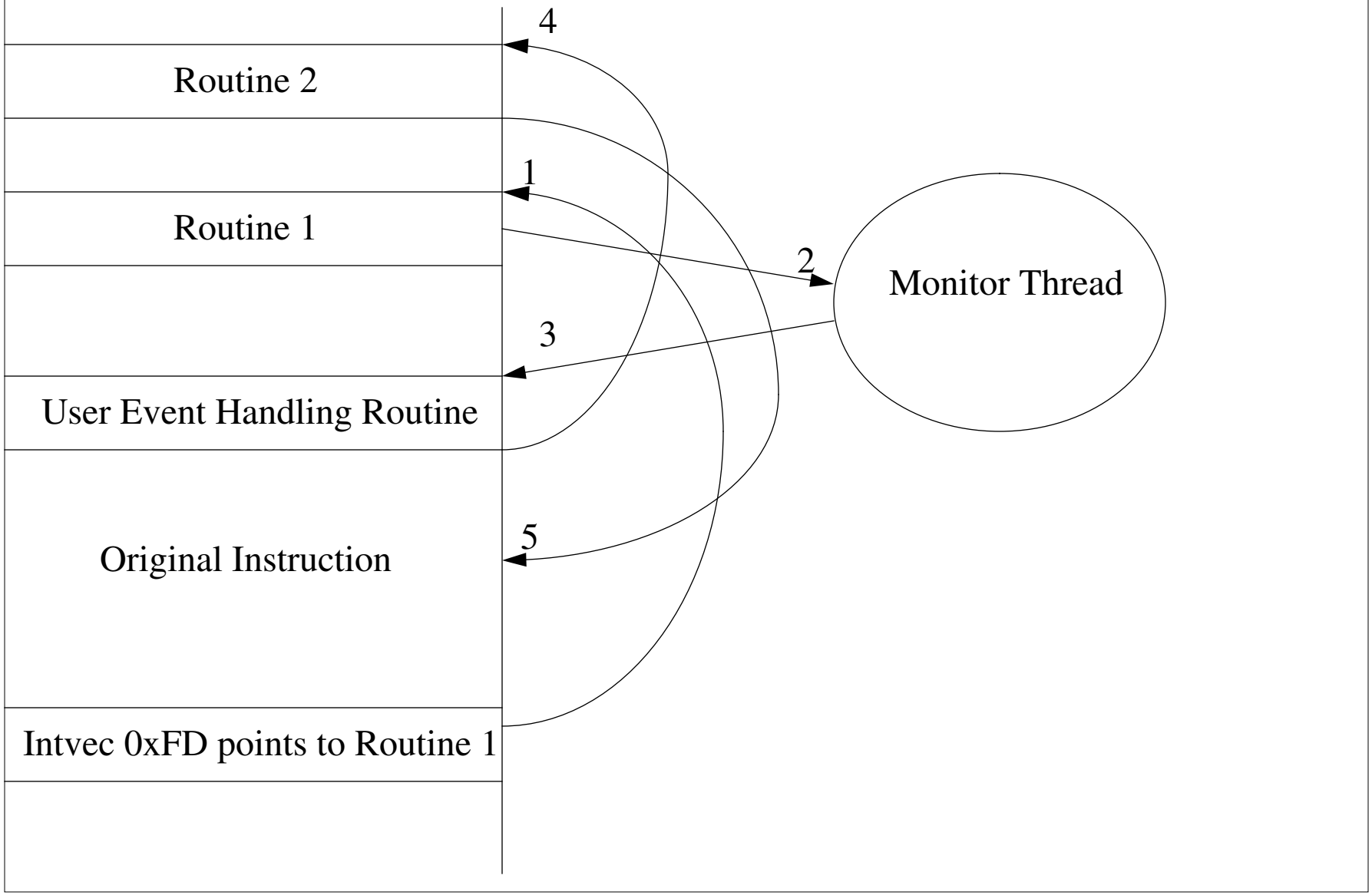
* Supports all three types of keyboard programs:
  - The Hog, takes all
  - The Chain, take what it wants and passes it on
  - The Split, takes what it want, and what everyone else doesn't

* Uses a user space routine like the mouse routines at the end of the chain.

# DOS and the Keyboard

* How DOS programs aquire keyboard data:
  - Call keyboard BIOS routines with INT 0x16
  - Directly access the BIOS keyboard data structures in the SPRAM
  - Redirect hardware keyboard interrupts into the own ISR
  - Combinations of the above

# Picture of Mouse Interrupt



# How We Do User Event Handling

* Supply 2 small assembly routines in the DOS address space

* Use fake Int number when queuing the interrupt (Int 0xFD)

* At boot time, point intvec 0xFD to point to our first helper routine

* When first routine is called by kernel or I/O thread, it does a real INT 0xFD

* Monitor thread catches the INT 0xFD and knows it's a Mouse Interrupt

* Monitor thread sets up stack to return to second routine and loads regs

* V86 thread resumed at User Event Handler, returns with FAR RET

* Second routine restores registers and IRETS back to original instruction

# User Supplied Mouse Event Handlers

* Mouse BIOS allows DOS applications to specify an event handling routine

* When event occurs:
  - Mouse driver loads machine state with mouse info
  - Driver calls the user routine with a FAR CALL
  - User routine does its work
  - User routine returns to driver with a FAR RET
  - Driver exits with IRET

* Why this is difficult for Mach DOS:
  - Our interrupt mechanism only calls routines with pointers in Interrupt Vector Table
  - We must store state and load registers before call to user routine
  - FAR CALL is used to call routine, we only have INT mechanism
  - FAR RET can't return to driver, since no driver in memory

# Mouse Thread

* Emulate a DOS Mouse Device driver
  - Keep track of Mouse status
  - Update mouse pointer in VGA memory
  - Provide interrupts for user supplied event handlers

* Emulate a serial line (COM) Mouse
  - Logitech Serial Mouse emulation for Windows 3.0

# DOS and Mouse Usage

* How DOS programs use the Mouse:
  - At boot time, load mouse device driver
  - Utilize driver's BIOS syscall interface 0x33
  - DOS programs either:

    1. Poll the mouse driver for mouse status
    2. Provide a user address as a pseudo interrupt routine for mouse events
    3. Supplant the Mouse driver with a user supplied ISR

# How Are External Interrupts Delivered?

* In General, I/O Threads queue interrupt when input recieved.
    - Structure `interrupt_table` contains a `count` for each interrupt type:
        - keyboard interrupt
        - mouse interrupt
        - 2 timer interrupts
        - break interrupt
        - COM interrupt

* Interrupts are Delivered by:
    - Periodic checks by User threads after queueing
    - Kernel checks after each successful hw exception emulation

# Timer Thread

* Generates two types of Timer Interrupts:
  - Hardware Timer IRQ, Intvec 0x9
  - Timer Tick, Intvec 0x1c

* Updates SPRAM system time

* Timeout on Mach Port (timeoutval is user defined)

* Compensation for slow clock

# What Kind of DOS Program is Running?

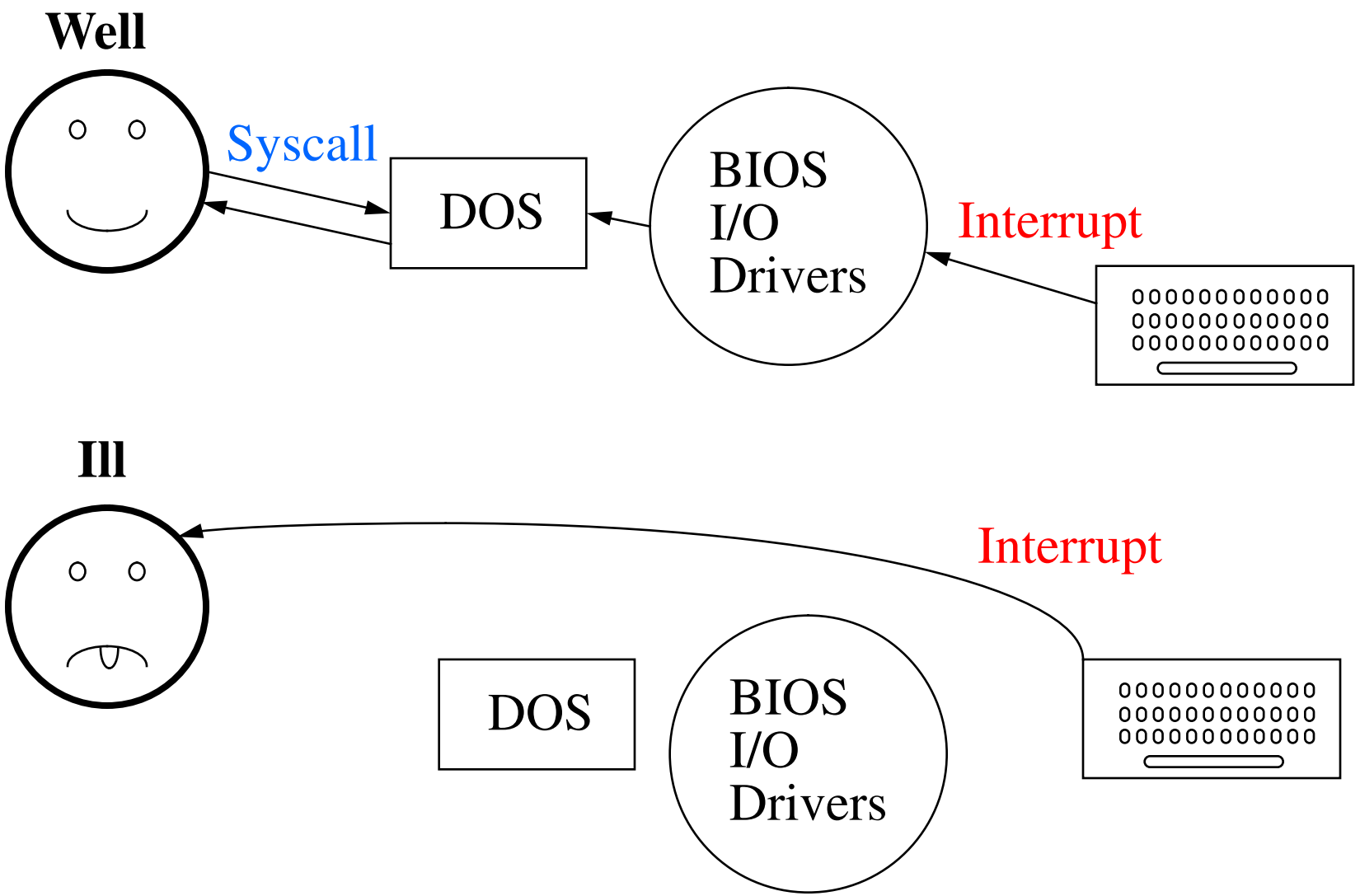
* We call `override_vectors()` at the start of execution

* Initialize the Interrupt Vector Table with Mdos values

* Store these values

* Compare Vector values during execution
  - Equal -> Well-behaved
  - Different -> Ill-behaved

# Well-Behaved & Ill-Behaved DOS Programs



# eXtended Memory Support

* History of XMS:
  - “Discovery” of the HMA
  - VDISK

* How it works:
  - Memory above 1 Meg Address Barrier
  - XMS Manager below 1 Meg supplies entry point
  - XMM transfers blocks of memory back and forth

* How we do it:
  - Support XMS calls in the Monitor thread
  - Use VM and Pager to remap VM into User Specified Areas

# Expanded Memory Support

* History of DOS memory expansion
    - 640K barrier

* How it works:
    - Holes in HMA
    - Banks of RAM on Expansion slot card
    - EMM manager supplies entry point INT 0x67

* How we do it:
    - Support INT 0x67 calls in the Monitor thread
    - Use VM and Pager to remap VM chunks into HMA hole

# Monitor Thread Video Support

* Redirect all video calls but console I/O

* Console I/O special case:
  - DOS programs are silly wrt tty output
  - Monitor thread intercepts all DOS strings commands
  - Incredibly slow

* SuperVGA support

# Monitor Thread Disk Support

* Monitor thread support:
  - DOS Hard Disk Support
    - Setup hard disk vtoc to point to DOS HD parition
    - Supports DOS HUGE file systems, systid = 1,4,6
    - Utilizes “/dev/dosdisk” file
  - Floppy Disks support
    - Raw device file access or floppy image support
    - Utilizes “ms.dos” file

# User Level Support

* Multithreaded Mach Task:
  - Exception Handler Thread (Monitor)
    - Disk Emulation
    - EMS, XMS Emulation
    - Video Support
  - I/O Threads
    - Timer Thread
    - Mouse Thread
    - Keyboard Thread

# Mach DOS Kernel Support

* V86 Thread Maintenance:
  - `thread_get_state, thread_set_state`
  - Thread flavors
    - `I386_THREAD_STATE`
    - `I386_ISA_PORT_MAP_STATE`
    - `I386_V86_ASSIST_STATE`
* Emulation of Hardware V86 Faults:
  - enable/disable interrupts (STI/CLI)
  - push or pop flag register (PUSHF/POPF)
  - return from interrupt (IRET)
* Simulation of External Interrupts Within the V86 Thread

# 4 Things to Remember:

1. We setup VM to look like DOS

2. We load the actual DOS boot code into the VM

3. Setup a special thread in V86 mode to point to boot code

4. Let V86 thread run and have the monitor thread catch any instructions that cause exceptions

# Run Time Narration Continued...

5. Load DOS Boot Image
   - Use DOS Hard Disk partition or Floppy Drive/Image
   - DOS boot address is 0x0070:0000

6. Setup V86 Thread
   - Use `thread_set_state` to create V86 mode thread
   - Set the exception port to handle GP faults
   - Setup the thread's registers
   - enable I/O ports for the thread

7. Create the I/O threads

8. Resume I/O threads and V86 thread

9. Service V86 thread's Exception Port

# What Happens at Run time?

1. Mach DOS (Mdos) code loaded high at 3 Meg

2. Checks to make sure running on console

3. Sets up task's VM:
   - VM allocates DOS address space from 0 to 3 Meg
   - Maps in the VGA RAM and BIOS ROM at 0xa0000
   - Uses External Pager to create High Memory Area Wrap at 1 Meg
   - Create EMS and XMS VM and initialize their data structures

4. Intialize the VGA
   - Saves the startup VGA register state (70 Registers)
   - Allocate VM for VGA RAM (256K currently)
   - Store VGA Fonts and Memory for Exit and Pausing

# How DOS Programs Operate

System Calls:

- Load Registers with parameters
- Call OS with the INT n instruction
- Return values in Registers

BIOS:

- ROM device drivers for resident hardware
- Interface exported to user via INT call

DOS:

- General Purpose OS code
- Interface exported to user via INT call

# 8086 Addressing and V86 Mode

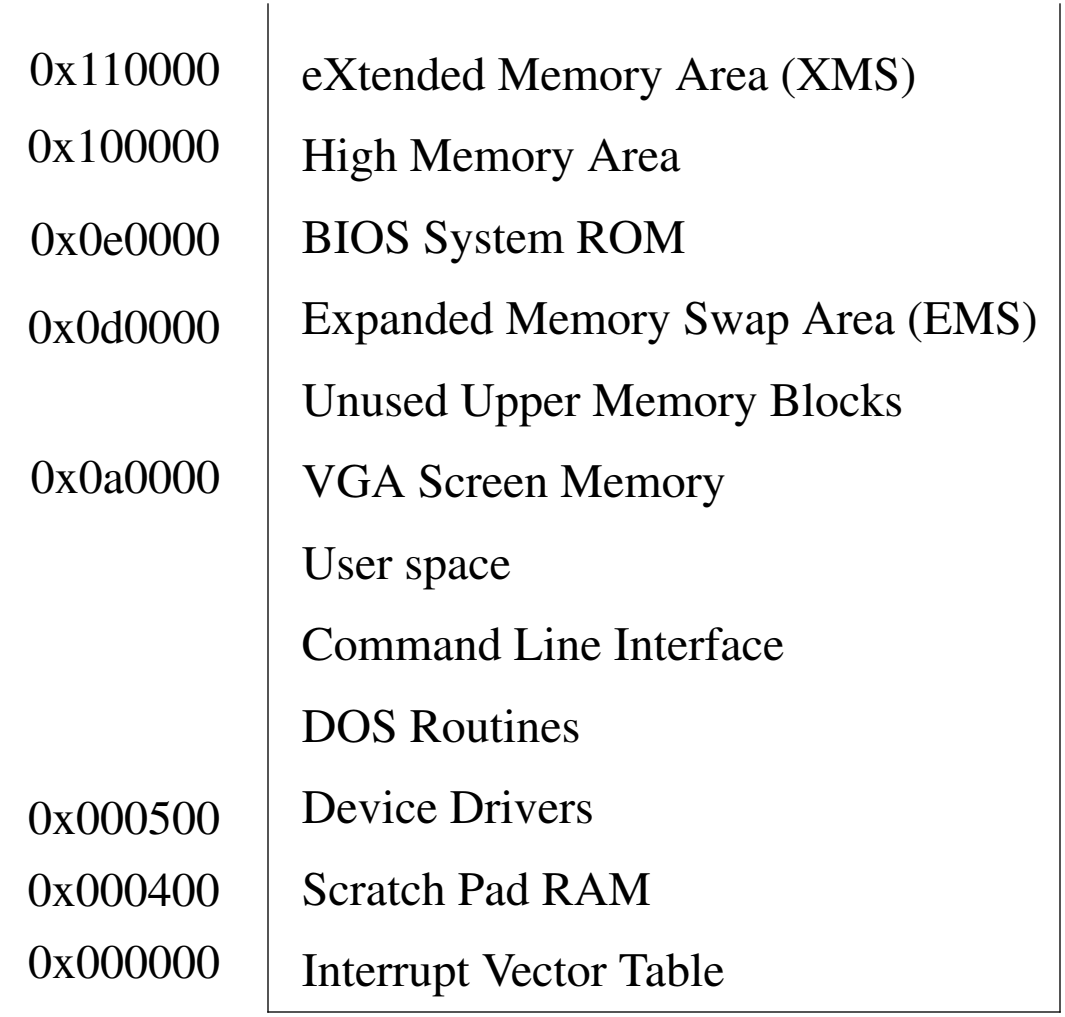
8086 Addressing:

- 16 bit Segment Register
- 16 bit Offset Register
- 20 bit Address = Segment << 4 + Offset
- Max address is 0x100000 (0x10FFEF)

Virtual 8086 Mode:

- Hardware mode operates with Virtual Memory
- Emulates 8086 instruction execution and addressing
- Privileged Operations cause General Protection Exceptions
- Exceptions handed to kernel

# DOS Machine's Memory Layout



# DOS Basics

* Memory layout of DOS machine

* 8086 Addressing Scheme

* Virtual 8086 (V86 Mode)

* How DOS programs operate

# Outline

* Overview of Mach DOS work

* Brief review of DOS

* Kernel Changes

* User Level Implementation

# Mach DOS Internals

Gerald Malan  Richard Rashid  David Golub  Robert Baron
School of Computer Science
Carnegie Mellon University