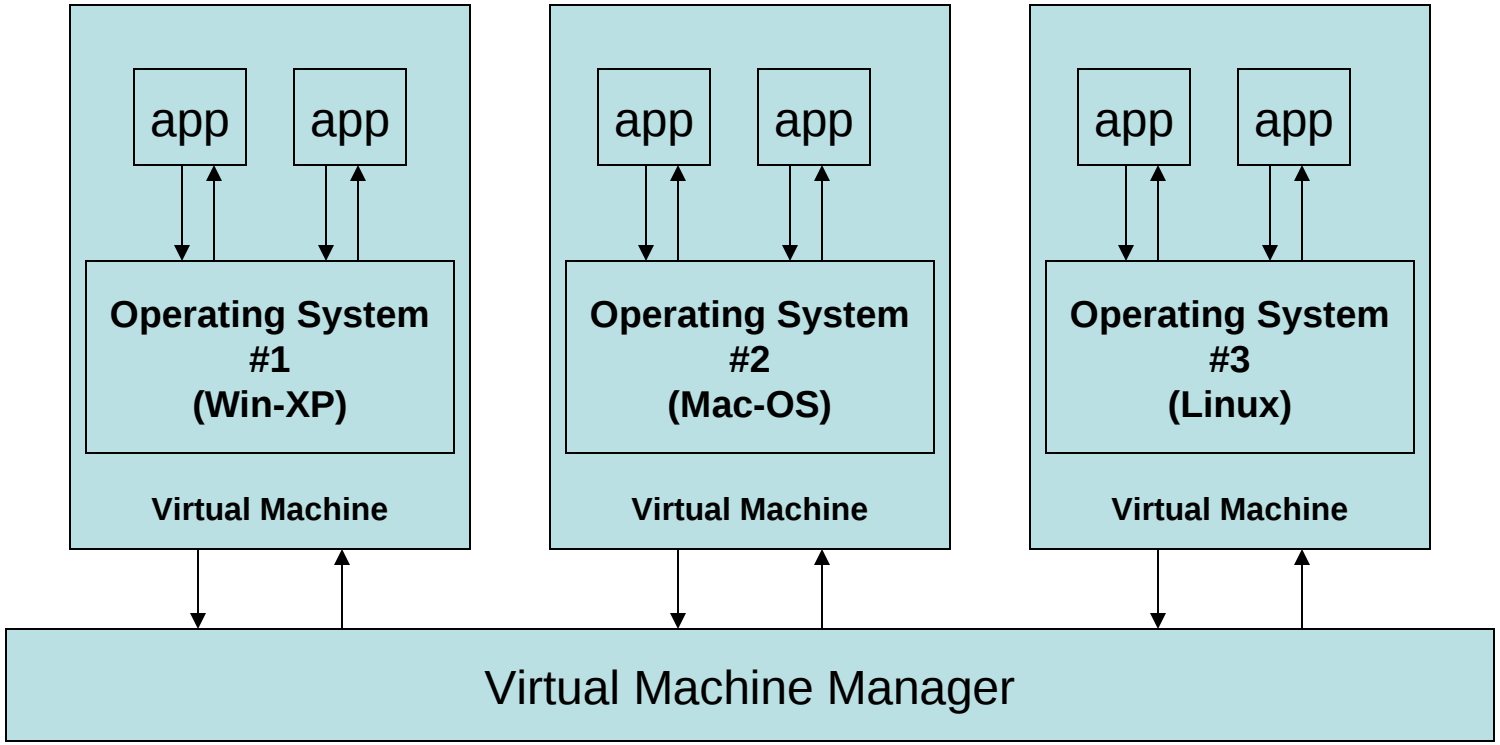
‘Trap-and-Emulate’

A look at one of the fundamental concepts for implementation of ‘virtual’ machines

What's VT for?



Each operating system was designed to be in total control of the system, which makes it impossible for two or more operating systems to be executing concurrently on the same platform – unless 'total control' is taken away from them by a new layer of control-software: the VMM

How to seize control?

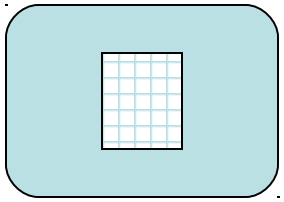
- The Virtual Machine Manager will have to be able to intervene whenever one OS is attempting to do something that conflicts with what another OS wants to do
- With the new VTX instructions, the CPU is able to ‘trap’ such attempts, and allow the VMM to ‘emulate’ the effect that is desired by one OS, but in a manner that does not interferw with any other OS

An example senario

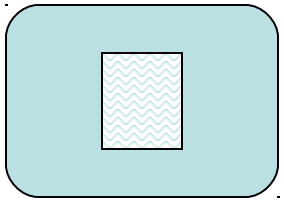
- Suppose a Win-XP application is drawing some text in a window on the screen, and a Mac-OS application also is drawing into a window on the same computer screen
- Neither application does its drawing itself, but only by asking its Operating System to draw to the screen on its behalf
- But the OS’s are unaware of each other

The VMM can sort this out

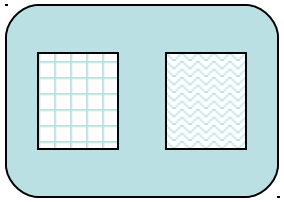
- The VMM can ‘trap’ all attempts to do any drawing to the screen, then can ‘redirect’ the two images to different screen regions whenever they might otherwise overlap



Win-XP tries to draw this



Mac-OS tries to draw this



VMM can be a mediator

VT-x controls

- As we shall soon see, Intel's Virtualization Technology instructions lets a 'Host' VMM specify numerous conditions under which the actions attempted by a 'Guest' VM will get 'trapped', allowing the 'Host' to seize control and take alternative actions when conflicts among OS's are about to arise

Details of an example

- We constructed a program-demo that can show you the full details for a fairly simple ‘trap-and-emulate’ example (‘vm86trap.s’)
- It doesn’t require the new VTX instructions (yet it does foreshadow their eventual role)
- It’s perhaps not a dramatic example, like redirecting graphical images would be, but it has the virtue of being more accessible

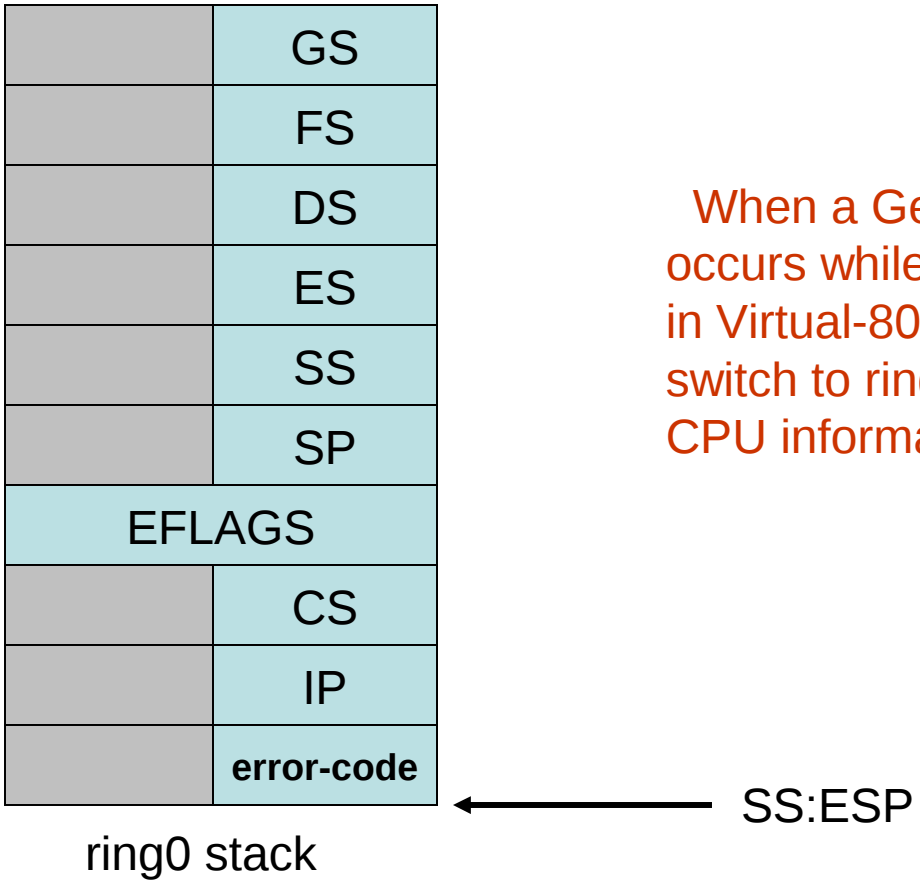
Control Register CR0

- The CPU can readily access register CR0 when it is executing in ‘real-mode’ (ring0), but it lacks the necessary privileges for executing ‘mov %cr0, %eax’ when it is running in Virtual-8086 Mode (ring3)
- The attempt by ring3 code to access any of the Control Registers will be ‘trapped’ by the CPU (a ‘General Protection’ Fault)

Our ‘fault-handler’

- We can design a ‘fault-handling’ procedure that will ‘emulate’ the ‘**mov %cr0, %eax**’ instruction when it’s encountered in ring3
- We just need to remember how the CPU responds to any General Profection fault
- Certain information gets automatically pushed onto the ring0 stack, where our fault-handler can find it -- and can use it!

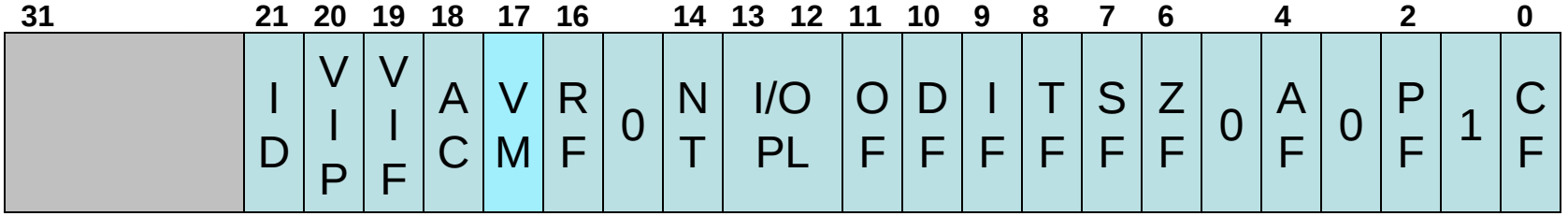
Stack-frame layout



When a General Protection fault occurs while the CPU is executing in Virtual-8086 mode, the CPU will switch to ring0 and will save all this CPU information on the ring0 stack

Steps in the ‘fault-handler’

- Step 1: Make sure the fault occurred while the CPU was in Virtual-8086 mode – since this affects what the stack’s values mean
- You confirm VM86-mode by testing the VM-bit (bit #17) in the EFLAGS image:




```
btl     $17, 12(%esp)     # Is VM-bit set to 1?
jc      inVM86
```

Preserve CPU registers

- Step 2: You must take care to preserve the values in all the CPU registers that your fault-handler might need to use, in case you do want to return to continue executing the interrupted VM86 task
- Simplest way to do it here is with ‘pushal’:

```
pushal          # preserves CPU’s general registers
```

- (All segment-registers already got saved)

Setup stack-frame access

- Step 3: You need to access the values on the stack, to locate the faulting instruction, to advance the IP-register's image past the faulting instruction, and to modify the value from a general register if you are going to emulate a '**mov %cr0, greg**'
- Best way to do this is with register %ebp:

```
mov     %esp, %ebp      # address the stack using EBP
```

The instruction-address

- For accessing the instruction that triggered the general protection fault, you'll need to compute its memory-address from images of CS and IP saved on the stack
- Algorithm: address = (CS * 16) + IP
  - Location of CS-image: 40(%ebp)
  - Location of IP-image: 36(%ebp)

Calculation details

```
# Here we compute in register EDI the memory-address of
# the instruction that caused the General Protection fault

        mov     40(%ebp), %di     # get image of CS
        movzx   %di, %edi         # extend to 32-bits
        shl     $4, %edi          # multiply by sixteen

        mov     36(%ebp), %ax     # get image of IP
        movzx   %ax, %eax         # extend to 32-bits
        add     %eax, %edi        # and add to EDI
```

The ‘flat’ address-space

- To examine the faulting-instruction, you’ll need the ability to address it, no matter where it may be in memory, which is why we constructed a segment-descriptor with base-address 0 that spans the full 4GB:

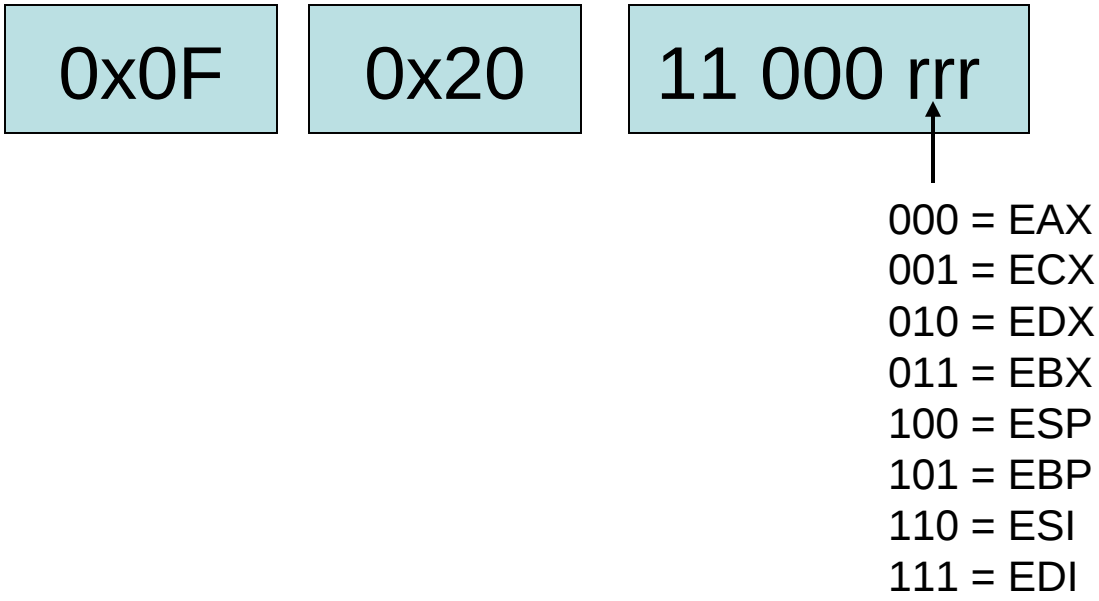
```
.quad       0x008F92000000FFFF
```

- Put a selector for this ‘flat’ segment in DS:

```
mov     $sel_fs, %ax
mov     %ax, %ds
```

Machine-code

- Step 4: See if the faulting instruction was indeed '**mov %CR0, greg**'
- The machine-code has this 3-byte format:



Fetch instruction and verify

```
# The instruction begins at address DS:EDI and is 3-bytes long
# (remember that Intel Architecture uses 'little-endian' storage)

        mov     %ds:(%edi), %eax        # get 4-bytes
        and     $0x00F8FFFF, %eax       # keep 21-bits
        cmp     $0x00C0200F, %eax       # is it 'mov %cr0,reg'?
        jne     depart                  # no, wrong opcode
        jmp     emulate                 # else we emulate it
```

Destination-register

- Step 5: Determine which general register is the destination-operand (determined by lowest 3-bits of the third instruction-byte), and locate that register's stack-image as the one to be overwritten by CR0's value

```
mov     %ds:2(%edi), %dl          # get instruction's 3rd byte
and     $0x00000007, %edx         # convert number to 32-bits
neg     %edx                      # get number's negative
add     $7, %edx                  # add 7 to get image-number
mov     %cr0, %eax                # read register CR0's value
mov     %eax, (%ebp, %edx, 4)     # overwrite operand's image
```

Skip past the fault

- Step 6: Now that we have performed the operation at ring0 which could not be done at ring3, we are ready to ‘return’ to resume the interrupted ring3-task – but NOT to the same instruction that caused the fault!
- We need to ‘skip’ that 3-byte insxtruction:

```
addw    $3, 36(%ebp)      # advance IP-image by 3-bytes

popal                     # restore images to general registers
add     $4, %esp          # discard error-code from the stack
iretl                     # return to the Virtual-8086 mode task
```

The ‘smsw’ instruction

- There is another CPU instruction that also returns the value in Control Register CR0
- It is NOT a privileged instruction – it can be directly executed in Virtual-8086 mode
- So you can use it to check on the validity of your ‘emulation’ for ‘**mov %cr0, reg**’

```
smsw    %eax            # reads CR0 into EAX
mov     %cr0, %edx      # reads CR0 into EDX
sub     %eax, %edx      # what's the difference?
jnz     emulation_flaw  # it ought to be zero!!
```

Flaw was intentional?

- There could be a valid reason for returning a value to Virtual-8086 mode that isn’t the processor’s actual value from register CR0
- The ring3 routine might be trying to detect whether or not it’s running in ‘real mode’, by examining the PE-bit from register CR0
- In a ‘virtual machine’ we might want code to behave as if it really was in real-mode!

Why we need VMX

- There are over a dozen x86 instructions, similar to ‘smsw’, that can execute without being ‘trapped’, and so reveal information about the CPU’s state that may interfere with attempts to build a ‘Virtual Machine’ that faithfully emulates the real machine
- But Intel’s VT extensions allow ‘trapping’ of all those problematic cases by a VMM

In-class exercise

- Can you modify our ‘vm86trap.s’ program so that it would allow a ‘Virtual-8086’ task to find out the physical address of its page directory (by reading from register CR3)?
- Machine-code for ‘**mov %cr3, reg**’ is: