Virtualization of the PC architecture components

Legacy PC components are virtualized by their interfaces with the IO, MMIO and interrupt subsystems.
IO instructions ([rep]in/ins/out/outs) are trapped by the VMX framework and delegated to their respective handlers (devices) in the vmx driver.

Memory-mapped IOs are complete regions of CPU physical address mapping, always page-aligned and multiple of page size (4K). They are also registered and routed to the respective handlers.

A virtual device driver may elect to send an interrupt into the guest.

Legacy devices resource map

8259A PIC, Programmable Interrupt Controller
- io = 0x20, 0x21 (master)
- io = 0xA0, 0xA1 (slave)

8253 PIT, Programmable Interval Timer
- io = 0x40 – 0x43 (primary) + IRQ 0
- io = 0x48 – 0x4B (secondary) NMI??

8237A DMA, Direct Memory Access controller
- io = 0x08 – 0x0F (primary) ???
- io = 0xD0 – 0xDE (secondary) ???

8255 PPE, Programmable Peripheral Interface, PS/2 Keyboard and Mouse
- io = 0x60 (keyboard + mouse)
- io = 0x61 (system speaker)
- io = 0x62 ???
- io = 0x64 (keyboard + mouse)

System CMOS + RTC, real-time clock
- io = 0x70 – 0x71

16550 UART, Serial communication ports
- io = 0x3F8 – 0x3FF (com1)
- io = 0x2F8 – 0x2FF (com2)
- io = 0x3E8 – 0x3EF (com3)
- io = 0x2E8 – 0x2EF (com4)
- sends IRQ: 3, 4
- mapped to a physical serial port, pipe, output log or a viewer

Parallel interface chip
- io = 0x378 – 0x37F (lpt1)
- io = 0x278 – 0x27F (lpt2)
- mapped to a physical parallel port, pipe, printer or output log or a viewer

Floppy controller
- io = 0x3F0 – 0x3FF (A:)
- io = 0x370 – 0x37F (B:)
- uses DMA 2 and IRQ 6
- mapped to a physical floppy drive, file image or an application container (what is that?? ☺)

IDE HDD
- io = 0x1F0 – 0x1F7
- io = 0x3F6 – 0x3F7 (drive 0:0, master + slave)
- io = 0x170 – 0x177
- io = 0x376 – 0x377 (drive 1:0, master + slave)
- uses IRQ 14
- mapped to a physical disk drive, file image, container or a host directory

VGA
- mmio = A0000 – BFFFF (display frame buffer, vga)
- io = 0x3C0 – 0x3DF
- io = 0x3B0 – 0x3BB

System BIOS image
- memio = 0xF000 – 0xFFFF

Video BIOS image
- memio = 0xC000 – 0xCFFF

Virtualization of the CPU

Upon the reset, the CPU starts at F000:FFF0 (0xFFFF0).

Since the real mode is very similar to the v86 mode, we start operation in v86 mode. The following is a list of instructions that need special consideration when running under the VMM.

System instructions (which tend to appear in operating system software) such as ARPL, are not listed unless their behavior in v86 mode differs from behavior in real mode, in which case they need to be emulated (example for that?)

clts	GP(0) *1		
hlt	VMX trap		
in/ins/out/outs	VMX trap		
lgdt/lidt	GP(0)		use VMCALL
lldt	#UD		use VMCALL
lmsw	VMX trap		
ltr	#UD		use VMCALL
mov to CR	VMX trap		
mov from CR	VMX shadow		
popf			
pushf			
cli/sti	Use vme86		use VMCALL
sgdt		problem	
sidt		problem	
sldt	#UD		use VMCALL
smsw	VMX shadow		
str	#UD		use VMCALL
verr/verw	#UD		use VMCALL
lar/lsl	#UD		use VMCALL
lds/les/lfs/lgs/lss			
mov to SEG			
mov from SEG			
push SEG			
pop SEG			
sysenter/sysexit			
iret			
wait			
enter/leave			
callf/jmpf			
int/into/bounds			
UD			
cupid	VMX trap		
invlpg	VMX trap		
monitor/mwait	VMX trap		
rdmsr/wrmsr	GP(0)		use VMCALL

*1 – These instructions can be executed only with a privilege level of 0, so they will GPF when executed within a v86 mode, and then can be emulated.

IN/OUT Instructions:
We set "activate IO bitmap" to 0 and "unconditional IO exiting" to 1, so any I/O will cause VM Exit. We don't need to use the IO Permission bitmap in the TSS. Note that we still need to set the address in the TSS since it is used to locate Software Interrupt Redirection Bitmap.

Software interrupts: *int n*, *into*, *bounds* instructions:
We don't need to handle software interrupts that are generated within the guest "real" mode code. We want them to be automatically reflected using real-mode IDT. Note that the first 32 interrupts are also used for CPU system use, which we *do* want to capture. However, we use VMX Exception Bitmap set to all 1's to force all CPU exceptions (Faults/Traps/Aborts) to unconditionally cause VM Exit. This way the Interrupt redirection bitmap is simply used to redirect all software interrupts back to the vm86 code.

We set the VME mode and use Software Interrupt Redirection Bitmap to indicate that an interrupt should be redirected back to the real-mode IDT located at linear address 0, by clearing the corresponding bit. We also set the IOPL to 0. This way our bitmap redirects interrupts accordingly, and also it handles VIF and VIP flags to support maskable hardware interrupts.

A special case is int3 (0xCC) where interrupt redirection does not happen when in VME mode; the interrupt is always handled by a protected-mode handler.

Special handling when changing modes

We define a mode changing section of the code as that within which a guest software switches from real mode into protected mode and back. Within that we can find most critical sections of the code in within which that transition happens at various instruction boundary.

Switching from real mode to protected mode

Guest executes in v86 mode and normally loads system registers such are GDT and IDT before turning on the protected mode by setting a bit in CR0 register. The critical section starts at the time of setting CR0.PE to the instruction that actually reloads the CS descriptor, at which point we can start using the guest selectors and VMX framework.

A potential problem can arise if a guest loads one of its data selectors from GDT. Since the code is still operating in v86 mode, any actual use of such selector would be invalid since the addressing is still done using the legacy segment:offset model.

In real mode, we trap on any attempt to load system registers (GDT/IDT/TR) and simply save the effective values that the guest code would want to load. This will be handled by the #GP (lgdt/lidt) or #UD (lldt/ltr) handlers.

At the point at which protected mode is enabled (application sets CR0.PE), VMX traps and we initiate single-step execution by the v86 monitor code. This single step will also quickly decode instruction to detect one which loads the CS by far jump, far call or far return. At that point we switch to the real VMX operation.

This single step engine can also decode instructions that load data selectors and flag them. We can see how often that actually happens and then decide what to do.

Switching from protected mode back to real mode

Guest executes under a VMX framework within full protected mode environment. The critical section starts when it disables protected mode by clearing CR0.PE. We trap this using a VMX trap and initiate mode transition.

We switch to v86 mode and single step instructions, quickly decoding them until we detect one that reloads CS by using a far jump, far call or far return. At that point we continue running under the v86 monitor.

The main problem is guest using data selectors which still have protected mode base and limit active. Executing them in v86 mode would be incorrect. The single step engine can decode and flag such instructions, then we will see how often they happen and what to do about that.

In the meantime, we will reload data segments with the segment form of the base address that each of those registers had at the time PE is turned off. This will ensure correct operation for at least data access within 64K limit.

Virtualization of the VGA subsystem

A standard VGA-class adapter is presented to the guest VM. It provides IO and memory-mapped frame buffer interface. The code that virtualizes it is split into low level layer, implemented in the VMX driver, and a presentation layer, implemented in the application.

VMX layer traps VGA IO accesses and maintains a “true register value” data structure which has the shared mapping with the application. The 128K frame buffer region mapped within the guest addresses 0xA0000-0xBFFFF is also shared with the application.

In addition, the shared data structure includes various control semaphores to signal the memory write to a FB region (so the app can update display only when it gets modified).

The application runs a thread that checks for updates and renders the frame buffer in a proper format.

Idea for future implementation: SVGA adapter with a custom VBIOS which would simply call into the VMX for fast INT10 operations.

Implementation Details

Host application: vmxSim.exe

The kernel system tick resolution is 10ms by default (on most systems). We need to set it to 1 ms. The application uses Media library to set the timer resolution. It does not revert it to what it was since some other app might start in the meantime (such as WinAmp) that require higher resolution.

Driver allocates and locks memory for the virtual machine. At this time it does not support more guest memory than it can lock. This is the order of calls:

SetProcessWorkingSetSize(GetCurrentProcess(), <size + extra>, [2Gb, max])
VirtualAlloc(<size>, MEM_RESERVE)
VirtualAlloc(<size>, MEM_COMMIT)
VirtualLock(ptr, <size>)

Kernel mode device driver: vmx.sys

The device driver that contains the following code:

VMX initialization, entry and exit handlers
VMCALL and interrupt handlers for simulation of vm86 run and transition states
Paging, virtual TLB implementation with shadow pages
Simulation code for all legacy PC architecture virtual devices
Communication with the application for extended device requests (file system, etc.) and control

Driver runs VMX execution control in a separate driver thread which has a particular CPU assigned to it by using a "KeSetTargetProcessorDpc". It is important to keep execution on a dedicated CPU since VMX operation depends on it. A call to "KeSetImportanceDpc" is used to set the DPC to "high" importance.

Driver has one global (using DeviceExtension structure) and (possibly several) VM instance data structures (per application) maintained in the driver instance data. Global structure holds information needed for management of the overall VMX state. VM Instance structures are created for each Virtual Machine opened by individual instances of the application. VMX is always using the CPU #0, and VM entries are interleaved using the VMX scheduler.

Global structure contains a pointer to a page allocated for VMXON CPU region.
Each VM Instance structure contains a pointer to its VMCS Task memory region (page).

A separate timer DPC thread (in the driver) is set to activate every OS tick (1 ms).

V86 mode monitor: vmm.bin

Monitor pages are mapped only when the guest is running in the real mode and during the mode transitions. In protected mode, they are mapped out of the address space since the effective guest code and page mappings are active under VMX operation.

We want the VMM to be as simple as possible. In fact, we don't want to have any code in it, just few necessary tables. Instead, we prefer to have all the code in the VMX device driver where it can be debugged easily. That brings some performance penalty, which is fine for the short duration of time while running in real mode. As soon as the guest turns the protected mode on, this VMM implementation changes into the VT-x framework.

Offset: 0x0000	VMCALL Table	

VMCALL Table

VMCALL instruction is (0x0F,0x01,0xC1); we add another byte to make it a DWORD: 0x90C1010F.
For 256 interrupts, we create an array of 256 calls for a total of 1024b for this table (0x400). A call will never return from the VMX driver; instead, we will always return into the v86 code being monitored.

VMCALL_TABLE + 0x00	0x90C1010F
VMCALL_TABLE + 0x04	0x90C1010F
	< … >
VMCALL_TABLE + 0xFC	0x90C1010F

Offset: 0x0400	IDT	

Interrupt Descriptor Table

IDT is compiled to contain pre-set offsets into the VMCALL Table, with each IDT entry addressing into corresponding VMCALL entry. The type is 0x0E, 32-bit Interrupt Gate. (The SEL_CODE's base address is set to the beginning of the VMCALL Table.)

IDT[0]	offset=0x00	sel=SEL_CODE, 0x08	type=0xE
IDT[1]	offset=0x04	sel=SEL_CODE, 0x08	type=0xE
…	< … >		
IDT[255]	offset=0xFC	sel=SEL_CODE, 0x08	type=0xE

Each IDT entry is 8 bytes long, making this table 2048 bytes in size (0x800)

Offset: 0x0C00	GDT	

Global Descriptor Table

VMX Guest GDTR is loaded with the linear base address of this table. The table describes the following selectors used by the VMM structures:

SEL_NULL 0x00 (Not used, set to 0.)
SEL_CODE 0x08 Base=<virtual address of the VMM>, Limit=max, RPL=0
SEL_DATA 0x10 Base=<virtual address of the VMM>, Limit=max, RPL=0
SEL_TSS 0x18 Base=<virtual address of the TSS Table>, Limit=(0x8C-1), RPL=0

Each entry is 8 bytes long, making this table 32 bytes in size (0x20).
Before the VMM is mapped in the guest address space, the base addresses are adjusted accordingly.

Offset: 0x0C20	TSS Table	

Task State Segment Table

VMX Guest Task Register (TR) is loaded with the SEL_TSS value before VM Entry.
TSS is compiled to contain the following values:

offset		
0x00		Previous Task Link
0x04	ESP0 = offset of the supervisor's stack **top**	
0x08		SS0 = SEL_DATA
0x0C	ESP1	
0x10		SS1
0x14	ESP2	
0x18		SS2
0x1C	CR3 (PDBR)	
…	EIP/EFLAGS/EAX…EDI/ES…GS	
0x60		LDT Segment Selector
0x64	I/O Map Base Address = 0x88	**T**
0x68 …	32 bytes of Software Interrupt Redirection Bitmap, set to 0	
0x88	0xFFFFFFFF	

Interrupt redirection bitmap is set to 32 bytes of zeroes to force all software interrupts occurring within the vm86 monitored code to be redirected back to the real-mode IDT. We still deploy IDT with VMM_CALL gates to be flexible should we want to redirect any interrupts by using that mechanism.

The total size of this structure is 140 bytes (0x8C). The SEL_TSS limit is set accordingly. The last value is simply a padding as we are not using IO Bitmap – any IO causes a VM Exit.

Offset: 0x0CAC	**ESP0 – Supervisor Stack**	

Supervisor Stack

The supervisor stack will be loaded with the stack frame on any software interrupt or CPU exception that we might end up allowing in vm86 mode.

The maximum stack frame used would be formatted this way:

top (from ESP0)	Unused	
-0x04		v86 GS
-0x08		v86 FS
-0x0C		v86 DS
-0x10		v86 ES
-0x14		v86 SS
-0x18	v86 ESP	
-0x1C	v86 FLAGS	
-0x20		v86 CS
-0x24	v86 EIP	
-0x28	Error Code	

When an interrupt is handled using the VMM IDT, the stack will be filled in, the code executed as part of the monitor interrupt handler will be a VMCALL which will neither push more data onto this stack nor modify it, but instead will cause VMX Exit which our driver will handle.

The size of this structure is 44 bytes (0x2C).

Ends at the offset of 0x0CD8.
New physical page.

Offset: 0x1000	**PD – Page Directory**	**Size = 4K (1 Page Frame)**

Page directory and page tables are set up to map 1:1 guest virtual space to what guest thinks physical frames are. We rely on the VMX paging framework to do the real paging. Complete range of physical memory assigned to a guest is mapped. Although only the first 1Mb is accessible to the real-mode code, all of it is accessible if the guest (usually the SBIOS) reloads a selector limit. Those instructions we simulate since they cannot be executed natively in the v86 mode.

Each PDE maps into a separate page table, mapping in the increments of 4Mb. So, for example, a guest with 256 Mb assigned RAM memory will use 256/4 = 64 PDE and that many separate pages of PT.

Offset: 0x2000	**PT – Page Tables**	**1 Frame for each 4Mb of RAM**

We map one page of PT for each 4 Mb of guest memory. Frames are mapped sequentially, starting at the frame #0 up to the allocated size / 4096.

The VMM pages are mapped as supervisor, and the frames are from a separate page pool, not the allocated guest pool. They are also mapped much higher in the linear address space, at the address of 0x80000000 (2Gb) to leave plenty of room for mapping of up to 2 Gb of guest memory (although the practical limit is much lower.)