Michael Steil

Inside VMware

How VMware, VirtualPC and Parallels actually work

Abstract

Virtualization is complex. In cooperation with the host operating system, the Virtual Machine Monitor takes over complete control of the machine hundreds of times a second, handles pagetables completely manually, and may chose to wire (make-non-pageable) as much memory as it chooses. This paper explains why it still works.

Motivation

In 1999, VMware was the first virtualization solution for x86-based computers. Even 7 years later, there are only three competitors: Microsoft with VirtualPC (architected by Dynamic Recompilation pioneer Eric Traut who did Apple's 68K to PowerPC Recompiler) and that obscure Russian company that seems to have offered the same product under 3 different names (SVISTA, 2ON2 and now Parallels). Kevin Lawton, the creator of the x86-Emulator Bochs started an Open Source virtualization project called Plex86 (originally FreeMWare) - but it basically failed. Only recently, the Open Source QEMU project, which started as a recompiler, has been gaining execution speeds into the direction of VMware, by implementing some of VMware's methods.

This paper first summarizes some basic operating system features, like scheduling, managing page tables, and providing a system call interface, in order to have a common basis that can be talked about.

The main part is about the tricks a conventional virtualization solution has to apply to run the guest operating system as a user mode process: The virtual machine monitor (VMM) has to set up address spaces for guest code, handle nested page tables, switch between the host and the guest(s), trap I/O accesses, and help cooperate in memory management between the host and the guest(s).

The third part of the paper explains why the x86 architecture is not strictly virtualizable, what tricks VMware, VirtualPC and Parallels use to still make it possible, and in what way Intel VT (Vanderpool) and AMD SVN (Pacifica) help to make x86 virtualization easier or possibly more efficient.

Operating Systems

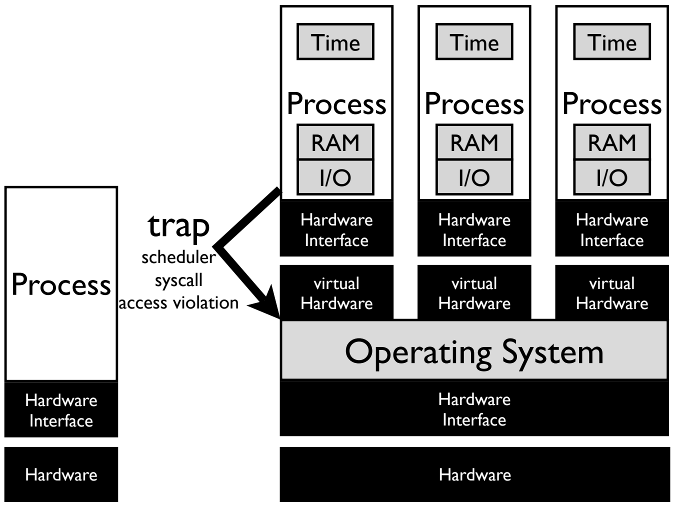
While "virtualization" might sound like a very modern and therefore "hot" piece of technology, it is actually quite old: Operating systems virtualize the machine by providing each of its processes a virtual CPU, virtual memory and virtual hardware (in the sense of kernel functions).

Multitasking

In the 1960s, computers usually did batch jobs that took hours, reading the input from tape, and writing the results back to tape. Since the tapes were slow, the CPU was basically wasted while the program was reading data from or writing data to tape and therefore waiting, as the CPU would have been independent to do other tasks in the meantime.

Attaching two tape drives and running two jobs at a time significantly optimized the load of the CPU: If one so-called process is blocked (because it is waiting for data from the tape drive), the CPU can be switched to the other process and perform calculations there. Also, if one process uses the CPU for too long, control can be migrated to the next process, in order to have a fair division of the CPU for the processes, regardless of the amount of I/O they perform.

Multitasking, which gives each process its own virtual CPU, can optimize the usage of the CPU if the processes do a lot of I/O.



CPU Modes

As different processes should not be able to influence each other, there must be an operating system to arbitrate. The CPU needs to have two modes: user mode and supervisor mode (kernel mode). Process run in user mode and therefore don't have full control of the machine. Only the operating system runs in kernel mode, it manages the processes and has therefore full control of the system. Each process can run as if it was the only one on the machine.

Syscalls

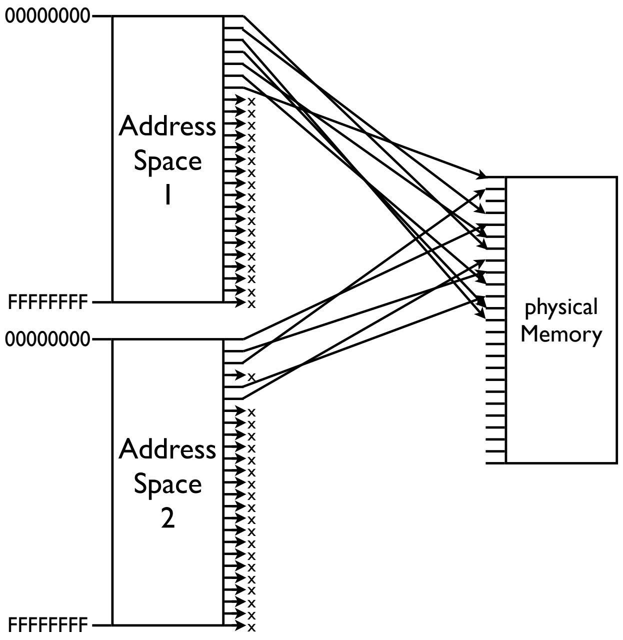
All instructions that change the overall state of the system (privileged instructions) do not work in user mode, they trap, i.e. they generate an exception, which means, the CPU will automatically switch into the kernel, which can then decide what to do. Usually user mode programs avoid execution of these privileged instructions; instead, they explicitly call the kernel for specific functionality, using syscalls: A syscall deliberately switches into kernel mode, calling a specific function of the kernel, which will then return to user mode.

Scheduling

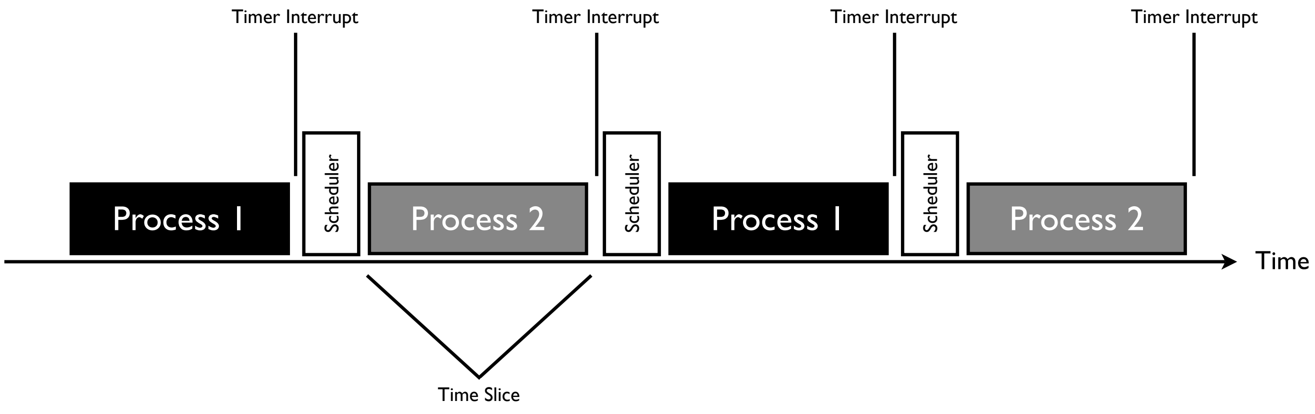
The kernel will set up to a timer that interrupts execution of user mode code typically about 100 times a second. Every time such an interrupt occurs, the CPU switches into kernel mode, and the scheduler, which is a part of the kernel, decides which process to run next and switches back to user mode. The time between two scheduler runs is called a time slice.

Virtual Memory

While in order to protect processes from each other, it would be enough to have a base and a limit address of memory that it can access for



each process, it is a lot more flexible to use paging: For every 4 KB "page" of memory, there is a mapping from virtual memory (the memory as the process sees it) to physical memory (as the address leaves the CPU for the



memory chips). This way, two processes can both have their code at (virtual) address 0x1234, but effectively use different parts of system RAM.

These mappings are stored in page tables. There is a set of page tables for every process, and on a context switch (as the scheduler switches from one process to the next one), the set of page tables gets switched.

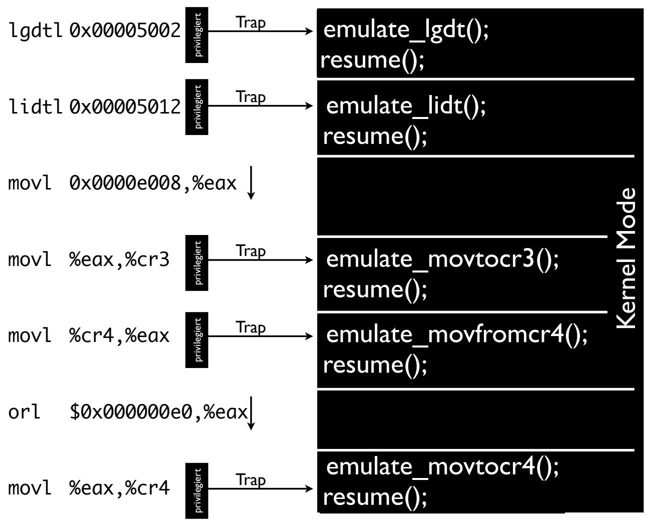
Virtualization

Although strictly speaking, operating systems already implement virtualization, in today's context, virtualization means running multiple operating systems at the same time. These operating systems are separated from each other, cannot influence each other and don't even know about each other. So an OS can no longer have full control of the CPU.

Hypervisor Mode

In order to disempower kernel mode (supervisor mode), some CPUs therefore introduce a third mode, called hypervisor mode: Hypervisor mode has full control of the CPU, and the Virtual Machine Monitor (VMM) runs in hypervisor mode and arbitrates between operating systems. If hypervisor mode is enabled, kernel mode will no longer have full control of the CPU.

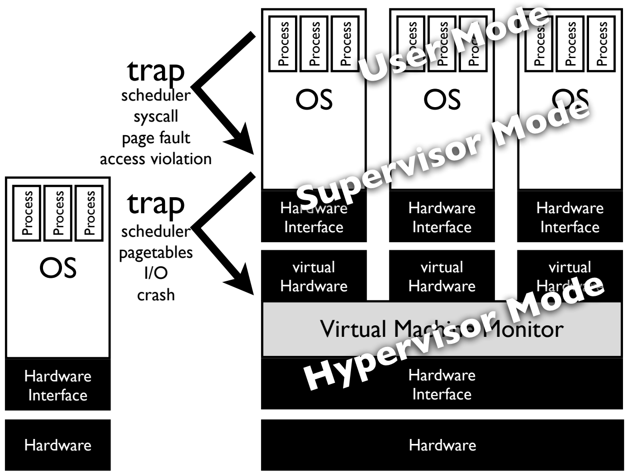
While the interface between user mode and kernel mode will still be the same, there is now a new interface between kernel mode and hypervisor mode: All privileged instructions issued by the kernel (in kernel mode) now trap into hypervisor mode, and the VMM can then emulate the desired behavior. This method is called "trap and emulate". All page table accesses, I/O accesses and (virtual) system crashes will be handled this way.

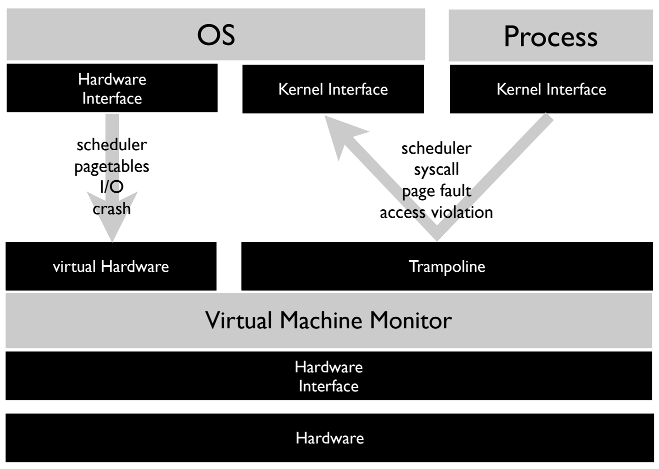

```
lgdtl 0x00005002
lidtl 0x00005012
movl  0x0000e008,%eax
movl  %eax,%cr3
movl  %cr4,%eax
orl   $0x000000e0,%eax
movl  %eax,%cr4
```
```
emulate_lgdt();
resume();
emulate_lidt();
resume();
emulate_movtocr3();
resume();
emulate_movfromcr4();
resume();
emulate_movtocr4();
resume();
```

An example of a CPU that implements a hypervisor mode is the IBM PowerPC 970, also known as G5. The architecture of Intel VT (Vanderpool) and AMD SVM (Pacifica) is also close to the hypervisor mode approach.

No Hypervisor Mode

But many architectures don't have a hypervisor mode, like the PowerPC G4 or the x86 line before VT/SVM. Every kind of protection in CPUs is always also possible with only two permission levels. The idea is to run the VMM in kernel mode, and push the guest kernel up into user mode. This way, only the VMM will run in the privileged mode of the CPU and have full control of the hardware.





Just like in the scenario with hypervisor mode, all privileged instructions in the guest kernel will trap into the VMM. For traps in user mode, it is different: All traps in user mode, including syscalls, will make the CPU switch into the VMM code in kernel mode. The VMM will then have to manually emulate this function, switch to the guest's kernel mode and move execution to the according handler.

Scheduler

The following paragraphs assume that the VMM is running on an existing OS, like VMware Workstation, Virtual PC and Parallels Workstation/Desktop do it, as this is the more interesting case. The bulk of the logic is in the user mode application (for example "vmware.exe"), but some operations can only be carried out in kernel mode. These reside in the kernel module that is shipping with each of these solutions (vmmain for VMware, vmmon for Parallels). For example, all decisions can be made by the user mode part, while all changes to the state of the CPU, like switches between the host and the guest, can only be carried out in kernel mode. The commercial virtualization solutions for Linux hosts also try to run as much code as possible in user mode, because they have to open source their Linux kernel module and thus all their kernel mode code.

Every guest OS will behave like an application on the host operating system and be scheduled against other apps; to the host kernel, the user mode application ("vmware.exe") is the representative of the virtual machine. This user mode application will donate its time slices to the virtual machine: Every time it is assigned a time slice, it will switch into the guest context, and at the end of the slice, a switch back will occur. These are called world switches.

For a world switch, the VMM user mode component calls the VMM kernel mode component, because a world switch can only be done in kernel mode. The kernel mode component will then

- switch the register set
- switch the interrupt/exception vector set
- switch the page tables

So the complete host context is saved, and the guest context is loaded. The kernel mode component will then switch into user mode and jump to the guest code.

A very small part of the VMM kernel mode component will remain mapped into the guest's address space, and all interrupt/exception vectors point to this code, so that it can catch all interrupts and exceptions.

The guest code will execute until the next interrupt or exception occurs. This can be a privileged instruction that the guest kernel tried to execute, a syscall of user mode code, or simply a timer interrupt indicating the end of the time slice. As all interrupts and exceptions will be caught by the VMM kernel mode component, it will switch the register set, the interrupt/exception vectors and the page tables back, and in case it was an exception caused by the VM, the reason will be passed up to the VMM user mode component, which will then decide how to handle it. If, on the other hand, it was a hardware interrupt, the VMM kernel mode component will pass this information to the host kernel. In case of the scheduler, control will be taken away from the current process (the VMM) and handed to the next process.

Virtual Memory

Since the virtual machine must behave like a real machine, memory must behave the same for all code running inside the VM, so both user mode and kernel mode must see memory as it is supposed to be mapped.

Just passing through page table entries from inside the VM to the physical system is not possible, because all operating systems think they can target physical pages starting with page 0, so they would map their virtual pages to the same physical page.

Therefore every page table access in the VM's kernel traps into the VMM, which will then create a mapping that has the same effect, but uses a different page - managed by the VMM. Also, if the guest kernel reads back the page table entry, there will also be a trap, and the VMM has to present the original (instead of the effective) value. The effective page tables are called "shadow page tables", and the whole method is called "two level paging".

When the guest wants to map a new page, the VMM kernel mode component will ask the host operating system for a page of memory, which the host kernel will then map into the VMM's address space. The VMM can read the page table entry that was just created by the host operating system and find out what physical page it was given. This physical page can then be used inside a VM.

The host kernel may never take this page away, because it is unaware of the fact that the page is still used inside the VM as well - after all, the VMM assigned its own page table mappings to this page for some VM. Therefore the VMM must mark the page as "wired", so that the host kernel will never take it away and put it into the paging file on disk.

Unfortunately, this would mean that paging out VMs isn't possible. While rarely used memory pages of ordinary applications will be moved from RAM into the paging file in order to free up some memory in case memory runs low, all the wired pages of a VM cannot be paged out. This can be solved by having a special "memory pressure" interface between the host kernel and the VMM: If system memory is low, the host kernel can tell the VMM, which will then look at its page usage statistics inside the VM and unmap a page from the VM. This page can then be "unwired" so that the host kernel can page it out. The next time the VM accesses this page, it will trap into the VMM, which will then be able to map the page back, according to its internal accounting structures.

But making the guest kernel trap on page table accesses might not be as easy: Page tables are normal data in memory, and while switching page tables, i.e. loading the pointer to the root page table is a privileged instruction that will trap when issued in user mode, accessing page table entries just means accessing memory, and it won't trap. The trick to still make these accesses trap is to mark the pages the page table entries reside on as invalid on the shadow page tables.

I/O

As all code inside the VM runs in user mode, all I/O accesses will trap into kernel mode, i.e. the VMM. The VMM will then find out the nature of the hardware access, fake the hardware and return to the guest. For example, if a guest asks the PS/2 mouse for its state, the access will trap to the VMM, and the VMM will communicate the current mouse state (in the PS/2 protocol encoding) to the guest, based on internal information about the emulated mouse.

If a virtual device is supposed to generate interrupts when new data is available, the VMM has to inject an interrupt into the VM by emulating exactly what would happen in case of an interrupt on a real machine: The next time the VM will be resumed, it will be in kernel mode, at the location specified by the (virtual) interrupt vector. This is also the way the scheduler inside the guest works: The guest programs the virtual timer and interrupt controller to generate periodic interrupts, and the VMM will inject interrupts into the guest from time to time.

Unfortunately, this method is quite slow for many devices, especially video, because the guest's driver and the VMM have to communicate in the language of the hardware protocol, which may be efficient for real hardware, but is not for interfaces between two pieces of software. Therefore, virtualization solutions typically provide special drivers to be installed inside the VM that either use special assembly instructions or I/O regions which are unused by a real machine to directly communicate with the VMM, using a very efficient high-level protocol.

x86

As pointed out above, having a hypervisor mode in a CPU is useful for virtualization but not strictly necessary. But there is one necessary requirement for a CPU to be "strictly" virtualizable: All privileged instructions must trap into kernel mode when used in user mode - they may not just behave differently. Otherwise the guest kernel cannot be run in user mode, as it could use assembly instructions that cannot be trapped and emulated, so the kernel would just behave differently when in user mode. The kernel would just not work.

Unfortunately, this is the case on the x86. There are several instructions that behave differently in user mode than in kernel mode, instead of causing a trap. For example, a user mode application could ask whether it is running in kernel mode or user mode, and it would get the answer “user mode” without any chance for the VMM to intercept this instruction and return the fake answer.

Therefore, the x86 CPU is not “strictly” virtualizable. But as every Turing complete machine can emulate every other Turing complete machine, running an operating system on top of another one is definitely possible. Bochs for example emulates an x86 PC by interpreting the x86 code instruction for instruction - at the expense of speed. Recompilers translate source assembly code to target assembly code, and are a lot faster.

The trick that is used by all x86 virtualization solutions is to recompile (“binary translate”) all potential sensitive code, i.e. translate all x86 assembly code that is problematic, because it does things that should trap but do not trap into code that has these instructions replaced with explicit traps into kernel mode.

All user mode code of the guest will also be executed in user mode, so all this code can be run natively. But all kernel code of the guest must be checked before it can be executed. As in dynamic recompilers, the code is split into “basic blocks” (a block of contiguous instructions up to the next control flow instruction, i.e. jump, branch, call, return), and these basic blocks are then verified. If they don’t contain problematic instructions, they can be executed verbatim, otherwise these instructions will be replaced by a call into the VMM.

All code that has been checked once does not need to be checked again, and a set of basic blocks that reference each other can be put together to a bigger block that can execute verbatim without future checks. Furthermode, the explicit traps can be translated into code that does the required function inline, without switching to the VMM: The instruction to find out the mode the CPU is in could be replaced by an instruction that loads the constant representing “kernel mode”.

But the disadvantage of having so much extra work to do (basic block accounting, translation, basic block linking etc.) leads to an advantage: Having all this infrastructure in place, the recompiler (on certain operating systems) can for example replace the instruction the guest kernel uses to return to user mode after a syscall (sysexit, sysret or iret) with code that just jumps to the guest user mode code. On a hypervisor mode trap-and-emulate solution, a sysexit would require a context switch (from kernel mode into user mode; just like on a real machine), and on a kernel mode trap-and-emulate solution, it would require two world switches (from the guest kernel into kernel mode, and back into guest user mode).

Another trick documented by VMware is this: There is usually no code block inside an operating system with a memory access instruction that sometimes accesses page table entries and sometimes accesses other data in memory. An instruction that writes a page table entry will always write page table entries. So as soon as such an instruction is identified (by making it trap using an invalid page), it can be translated inline to an explicit hypervisor call, which can be a lot cheaper than a trap.

VT and SVM

Both Intel and AMD understood the need for virtualization today, and that virtualizing the x86 is painful. Therefore they introduced very similar but incompatible technologies in all of their main-line x86 CPU starting in 2006. Intel calls its version of hardware-assisted virtualization “VT” (“Virtualization Technology”, formerly “Vanderpool”) while AMD calls it SVM (“Secure Virtual Machine”, formerly “Pacifica”).

The idea is to fix the x86 flaws that prevent “strict” virtualization by just adding a hypervisor mode to the CPU. Some sensitive instructions still don’t trap when in user mode, but this is no longer a concern, when the guest kernel can be run in kernel mode.

Intel calls its hypervisor mode “root mode”, and all code not in root mode executes in “non-root mode”, i.e. there is a non-root kernel mode and a non-root user mode. The host operating system runs in root mode (the host kernel in root kernel mode, the host processes in root guest mode), and the guest runs in non-root mode. So the machine is basically slit into two parts, and each part has its kernel and user mode.

The VMM can set up a complete virtual machine and then just issue the "vmenter" instruction. The CPU will save its complete state in a memory structure named "VMCS" ("virtual machine control structure") and load the state of the guest from the VMCS. It will execute the code inside the virtual machine, until there is a "VM exit". In this case, the CPU will save its state in the VMCS and load the state of the host from the VMCS. With VT and SVM, the "world switch" between the host and the guest is completely implemented in hardware.

The VMCS also contains a lot of bits, each of which specifies whether there will be a VM exit on a specific event. The VMM can make the CPU return to the VMM when there is a syscall - but it can also choose to ignore them, then they will just execute natively.

VT and SVM can make virtualization a lot easier: There is no need for a binary translator (recompiler) any more, the VMM can tell the CPU on which instructions to trap - but, as a second advantage, there are a lot less instructions that actually need to trap: On a hypervisor architecture, a switch between kernel mode and user mode can safely be executed natively.

But VT and SVM don't necessarily make virtualization faster. VMware for example claims that their binary translation solution is faster than a VT-based solution, because the binary translator can avoid many unnecessary traps, for example for page table accesses, that are still necessary on VT.

Furthermore, they state that the biggest speed boost would come from the implementation of nested paging in hardware, so that page table accesses won't need to trap, but instead, the non-root kernel mode code can write its own page tables, but the CPU will indirect all memory accesses through two sets of page tables: The guest's tables and the host's tables. So a guest's virtual address will be translated into a guest's physical address, and this address will then be translated into a host's physical address - all in hardware. Advanced versions of both VT and SVM will implement this.

Another enhancement of future hardware-assisted virtualization solutions will be the inclusion of I/O virtualization: Current (server level) virtualization solutions cannot exclusively assign physical hardware (like a network card) to one specific virtual machine by allowing the VM full access to the device, because the device is not aware of paging and only works with physical addresses, so it could read and overwrite all physical memory. I/O virtualization directs all memory accesses of devices through page tables.

References

I do not have any VMware/VirtualPC/Parallels inside knowledge; all my knowledge is derived from reverse engineering their kernel modules and reading public papers.

Lo, Jack: VMware and CPU Virtualization Technology,
http://download3.vmware.com/vmworld/2005/pac346.pdf

Adams, Keith; Agesen, Ole: A Comparison of Software and Hardware Techniques for x86 Virtualization,
http://www.vmware.com/pdf/asplos235_adams.pdf

Adams, Keith: "Blue Pill" is quasi-illiterate gibberish,
http://x86vmm.blogspot.com/2006/08/blue-pill-is-quasi-illiterate.html

VMware, Parallels, Plex86, QEMU/QVM86, XNU and Mac-on-Linux source code

Engel, Michael: Systemprogrammierung,
http://osg.informatik.tu-chemnitz.de/de/vorlesungen/ss06/systemprogrammierung.html

Steil, Michael: How retiring segmentation in AMD64 long mode broke VMware,
http://www.pagetable.com/?p=25

John Scott Robin; Cynthia E. Irvine: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor,
http://www.cs.nps.navy.mil/people/faculty/irvine/publications/2000/VMM-usenix00-0611.pdf